Mission MCA
THE SOURCE OF INNOVATION

JAVA PROGRAMMING

MCA – SEM 4

As per Mumbai University Syllabus

Prepared by
www.missionmca.com

For private circulation only

INDEX

1.
Java Fundamentals

1.1 Introduction

Java is a programming language originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere", meaning that code that runs on one platform does not need to be edited to run on another.

1.2 Features of Java

No discussion of the genesis of Java is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

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

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either left out of Java or implemented in a cleaner, more approachable manner. Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have surprising features. In Java, there are a small number of clearly defined ways to accomplish a given task.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance non objects.

Robust

The multi platformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java. To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments.

For example, in C/C++, the programmer must manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or "file not found," and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross platform solutions have done so at the expense of performance. Other interpreted systems, such as BASIC, Tcl, and PERL, suffer from almost insurmountable performance deficits. Java, however, was designed to perform well on very low-power CPUs. As explained earlier, while it is true that Java was engineered for interpretation, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. "High-performance cross-platform" is no longer an oxymoron.

Distributed

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra- address-space messaging. This 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 unparalleled level of abstraction to client/ server programming.

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 of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

1.3 The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). That is, in its standard form, the JVM is an interpreter for bytecode. This may come as a bit of a surprise. As you know, C++ is compiled to executable code. In fact, most modern languages are designed to be compiled, not interpreted—mostly because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet. Here is why.

Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all interpret the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs. The fact that a Java program is interpreted also helps to make it secure.

Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language. When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. The use of bytecode enables the Java run-time system to execute programs much faster than you might expect. Although Java was designed for interpretation, there is technically nothing about Java that prevents on-the-fly compilation of bytecode into native code.

Along these lines, Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release. When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not possible to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, the JIT compiles code as it is needed, during execution. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the run-time system (which performs the compilation) still is in charge of the execution environment. Whether your Java program is actually interpreted in the traditional way or compiled on-the-fly, its functionality is the same.

Object-Oriented Programming

Object-oriented programming is at the core of Java. In fact, all Java programs are object- oriented—this isn't an option the way that it is in C++, for example. OOP is so integral to Java that you must understand its basic principles before you can write even simple Java programs. Therefore, this chapter begins with a discussion of the theoretical aspects of OOP.

1.4 Two Paradigms

As you know, all computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around "what is happening" and others are written around "who is being affected." These are the two paradigms that govern how a program is constructed. The first way is called the process-oriented model. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. However, as mentioned in Chapter 1, problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called object-oriented programming, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

Abstraction

An essential element of object-oriented programming is abstraction. Humans manage complexity through abstraction. For example, people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well-defined object with its own unique behavior. This abstraction allows people to use a car to drive to the grocery store without being overwhelmed by the complexity of the parts that form the car.

can ignore the details of how the engine, transmission, and braking systems work. they are free to utilize the object as a whole. A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, the car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player.

The point is that you manage the complexity of the car (or any other complex system) through the use of hierarchical abstractions. Hierarchical abstractions of complex systems can also be applied to computer programs. The data from a traditional process-oriented program can be transformed by abstraction into its component objects. A sequence of process steps can become a collection of messages between these objects. Thus, each of these objects describes its own unique behavior. You can treat these objects as concrete entities that respond to messages telling them to do something. This is the essence of object-oriented programming. Object-oriented concepts form the heart of Java just as they form the basis for human understanding. It is important that you understand how these concepts translate into programs. As you will see, object-oriented programming is a powerful and natural paradigm for creating programs that survive the inevitable changes accompanying the life cycle of any major software project, including conception, growth, and aging. For example, once you have well-defined objects and clean, reliable interfaces to those objects, you can gracefully decommission or replace parts of an older system without fear.

1.5 The Three OOP Principles

All object-oriented programming languages provide mechanisms that help you implement the object oriented model. They are encapsulation, inheritance, and polymorphism. Let's take a look at these concepts now.

Encapsulation

Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper.

Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. To relate this to the real world, consider the automatic transmission on an automobile. It encapsulates hundreds of bits of information about your engine, such as how much you are accelerating, the pitch of the surface you are on, and the position of the shift lever.

You, as the user, have only one method of affecting this complex encapsulation: by moving the gear-shift lever. You can't affect the transmission by using the turn signal or windshield wipers, for example. Thus, the gear-shift lever is a well-defined (indeed, unique) interface to the transmission. Further, what occurs inside the transmission does not affect objects outside the transmission. For example, shifting gears does not turn on the headlights! Because an automatic transmission is encapsulated, dozens of car manufacturers can implement one in any way they please. However, from the driver's point of view, they all work the same. This same idea can be applied to programming. The power of encapsulated code is that everyone knows how to access it and thus can use it regardless of the implementation details—and without fear of unexpected side effects.

In Java the basis of encapsulation is the class. Although the class will be examined in great detail later in this book, the following brief discussion will be helpful now. A class defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as instances of a class. Thus, a class is a logical construct; an object has physical reality.

When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called members of the class. Specifically, the data defined by the class are referred to as member variables or instance variables. The code that operates on that data is referred to as member methods or just methods. (If you are familiar with C/C++, it may help to know that what a Java programmer calls a method, a C/C++ programmer calls a function.) In properly written Java programs, the methods define how the member variables can be used. This means that the behavior and interface of a class are defined by the methods that operate on its instance data. Since the purpose of a class is to encapsulate complexity, there are mechanisms for hiding the complexity of the implementation inside the class. Each method or variable in a class may be marked private or public. The public interface of a class represents everything that external users of the class need to know, or may know. The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable. Since the private members of a class may only be accessed by other parts of your program through the class' public methods, you can ensure that no improper actions take place. Of course, this means that the public interface should be carefully designed not to expose too much of the inner workings of a class (see Figure 2-1).

Inheritance

Inheritance is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. As mentioned earlier, most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification dog, which in turn is part of the mammal class, which is under the larger class animal.Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Let's take a closer look at this process.

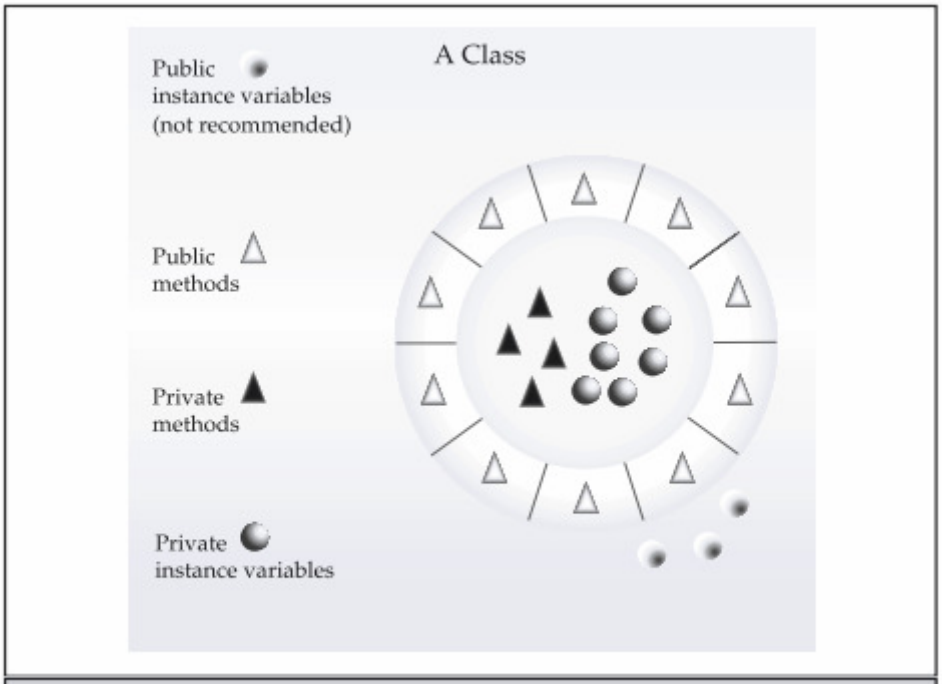
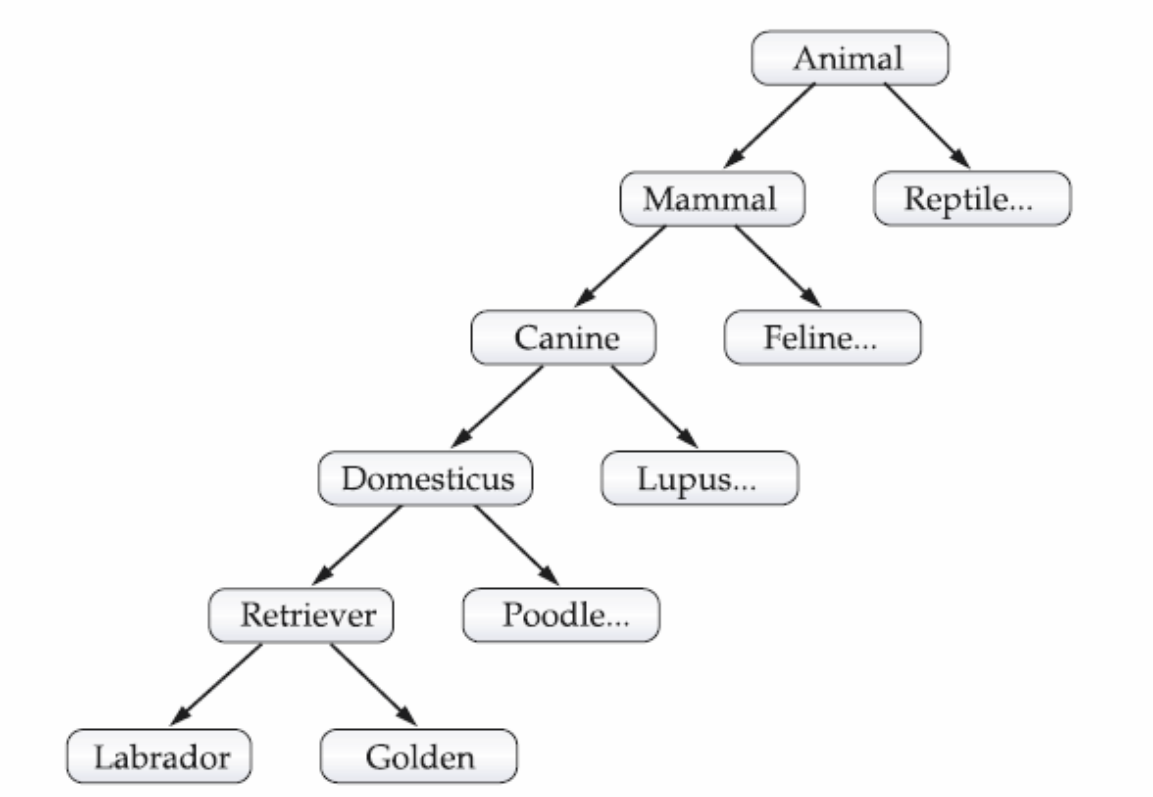


Figure 2-1. *Encapsulation: public methods can be used to protect private data*

Most people naturally view the world as made up of objects that are related to each other in a hierarchical way, such as animals, mammals, and dogs. If you wanted to describe animals in an abstract way, you would say they have some attributes, such as size, intelligence, and type of skeletal system. Animals also have certain behavioral aspects; they eat, breathe, and sleep. This description of attributes and behavior is the *class* definition for animals.

If you wanted to describe a more specific class of animals, such as mammals, they would have more specific attributes, such as type of teeth, and mammary glands. This is known as a *subclass* of animals, where animals are referred to as mammals' *superclass.* Since mammals are simply more precisely specified animals, they *inherit* all of the attributes from animals. A deeply inherited subclass inherits all of the attributes from each of its ancestors in the *class hierarchy.*

Inheritance interacts with encapsulation as well. If a given class encapsulates some attributes, then any subclass will have the same attributes *plus* any that it adds as part of its specialization (see Figure 2-2). This is a key concept which lets object-oriented programs grow in complexity linearly rather than geometrically. A new subclass inherits all of the attributes of all of its ancestors. It does not have unpredictable interactions with the majority of the rest of the code in the system.



Polymorphism

Polymorphism (from the Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks. One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non– object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However, because of polymorphism, in Java you can specify a general set of stack routines that all share the same names. More generally, the concept of polymorphism is often expressed by the phrase "one interface, multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.* It is the compiler's job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

1.6 Polymorphism, Encapsulation, and Inheritance Work Together

When properly applied, polymorphism, encapsulation, and inheritance combine to produce a programming environment that supports the development of far more robust and scaleable programs than does the process-oriented model. A well-designed hierarchy of classes is the basis for reusing the code in which you have invested time and effort developing and testing. Encapsulation allows you to migrate your implementations over time without breaking the code that depends on the public interface of your classes. Polymorphism allows you to create clean, sensible, readable, and resilient code. Of the two real-world examples, the automobile more completely illustrates the power of object-oriented design. Dogs are fun to think about from an inheritance standpoint, but cars are more like programs. All drivers rely on inheritance to drive different types (subclasses) of vehicles.

Whether the vehicle is a school bus, a Mercedes sedan, a Porsche, or the family minivan, drivers can all more or less find and operate the steering wheel, the brakes, and the accelerator. After a bit of gear grinding, most people can even manage the difference between a stick shift and an automatic, because they fundamentally understand their common superclass, the transmission. People interface with encapsulated features on cars all the time. The brake and gas pedals hide an incredible array of complexity with an interface so simple you can operate them with your feet! The implementation of the engine, the style of brakes, and the size of the tires have no effect on how you interface with the class definition of the pedals. The final attribute, polymorphism, is clearly reflected in the ability of car manufacturers to offer a wide array of options on basically the same vehicle.

For example, you can get an antilock braking system or traditional brakes, power or rack-and-pinion steering, 4-, 6-, or 8-cylinder engines. Either way, you will still press the break pedal to stop, turn the steering wheel to change direction, and press the accelerator when you want to move. The same interface can be used to control a number of different implementations. As you can see, it is through the application of encapsulation, inheritance, and polymorphism that the individual parts are transformed into the object known as a car. The same is also true of computer programs. By the application of object-oriented principles, the various parts of a complex program can be brought together to form a cohesive, robust, maintainable whole. As mentioned at the start of this section, every Java program is object-oriented. Or, put more precisely, every Java program involves encapsulation, inheritance, and polymorphism. Although the short example programs shown in the rest of this chapter and in the next few chapters may not seem to exhibit all of these features, they are nevertheless present. As you will see, many of the features supplied by Java are part of its built-in class libraries, which do make extensive use of encapsulation, inheritance, and polymorphism.

1.7 the First Sample Program

```
/*
This is a simple Java program.
Call this file "Example.java".
*/
class Example {
// Your program begins with a call to main().
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

Entering the Program

For most computer languages, the name of the file that holds the source code to a program is arbitrary. However, this is not the case with Java. The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**. Let's see why.

In Java, a source file is officially called a *compilation unit.* It is a text file that contains one or more class definitions. The Java compiler requires that a source file use the **.java** filename extension. Notice that the file extension is four characters long. As you might guess, your operating system must be capable of supporting long filenames. This means that DOS and Windows 3.1 are not capable of supporting Java. However, Windows 95/98 and Windows NT/2000/XP work just fine. As you can see by looking at the program, the name of the class defined by the program is also **Example**. This is not a coincidence. In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program. You should also make sure that the capitalization of the filename matches the class name. The reason for this is that Java is case-sensitive. At this point, the convention that filenames correspond to class names may seem arbitrary. However, this convention makes it easier to maintain and organize your programs.

Compiling the Program

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. As discussed earlier, the Java bytecode is the intermediate representation of your program that contains instructions the Java interpreter will execute. Thus, the output of **javac** is not code that can be directly executed. To actually run the program, you must use the Java interpreter, called **java**. To do so, pass the class name **Example** as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
This is a simple Java program.
```

When Java source code is compiled, each individual class is put into its own output file named after the class and using the **.class** extension. This is why it is a good idea to give your Java source files the same name as the class they contain—the name of the source file will match the name of the **.class** file. When you execute the Java interpreter as just shown, you are actually specifying the name of the class that you want the interpreter to execute. It will automatically search for a file by that name that has the **.class** extension. If it finds the file, it will execute the code contained in the specified class.

Although **Example.java** is quite short, it includes several key features which are common to all Java programs. Let's closely examine each part of the program. The program begins with the following lines:

```
/*This is a simple Java program. Call this file "Example.java".*/
```

This is a *comment.* Like most other programming languages, Java lets you enter a remark into a program's source file. The contents of a comment are ignored by the compiler. Instead, a comment describes or explains the operation of the program to anyone who is reading its source code. In this case, the comment describes the program and reminds you that the source file should be called **Example.java**. Of course, in real applications, Comments generally explain how some part of the program works or what a specific feature does. Java supports three styles of comments. The one shown at the top of the program is called a *multiline comment.* This type of comment must begin with **/*** and end with ***/**. Anything between these two comment symbols is ignored by the compiler. As the name suggests, a multiline comment may be several lines long. The next line of code in the program is shown here:


```
class Example {
```

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace ({) and the closing curly brace (}). The use of the curly braces in Java is identical to the way they are used in C, C++, and C#. For the moment, don't worry too much about the details of a class except to note that in Java, all program activity occurs within one. This is one reason why all Java programs are (at least a little bit) object-oriented. The next line in the program is the *single-line comment,* shown here:

```
// Your program begins with a call to main().
```

This is the second type of comment supported by Java. A *single-line comment* begins with a **//** and ends at the end of the line. As a general rule, programmers use multiline

comments for longer remarks and single-line comments for brief, line-by-line descriptions. The next line of code is shown here:

public static void main(String args[]) {

This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**. (This is just like C/C++.) The exact meaning of each part of this line cannot be given now, since it involves a detailed understanding of Java's approach to encapsulation. However, since most of the examples in the first part of this book will use this line of code, let's take a brief look at each part now.

The **public** keyword is an *access specifier,* which allows the programmer to control the visibility of class members. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (The opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.) In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.

The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made.

The keyword **void** simply tells the compiler that **main( )** does not return a value. As you will see, methods may also return values. If all this seems a bit confusing, don't worry. All of these concepts will be discussed in detail in subsequent chapters. As stated, **main( )** is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, **Main** is different from **main**. It is important to understand that the Java compiler will compile classes that do not contain a **main( )** method. But the Java interpreter has no way to run these classes. So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, the Java interpreter would report an error because it would be unable to find the **main( )** method.

Any information that you need to pass to a method is received by variables specified within the set of parentheses that follow the name of the method. These variables are called *parameters.* If there are no parameters required for a given method, you still need to include the empty parentheses. In **main( )**, there is only one parameter, albeit a complicated one. **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed. This program does not make use of this information, but other programs shown later in this book will.

The last character on the line is the **{**. This signals the start of **main( )**'s body. All of the code that comprises a method will occur between the method's opening curly brace and its closing curly brace. One other point: **main( )** is simply a starting place for your program. A complex program will have dozens of classes, only one of which will need to have a **main( )** method to get things started. When you begin creating applets—Java programs that are embedded in Web browsers—you won't use **main( )** at all, since the Web browser uses a different means of starting the execution of applets. The next line of code is shown here. Notice that it occurs inside **main( )**.

System.out.println("This is a simple Java program.");

This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. As you will see, **println( )** can be used to display other types of information, too. The line begins with **System.out**. While too complicated to explain in detail at this time, briefly, **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console. As you have probably guessed, console output (and input) is not used frequently in real Java programs and applets. Since most modern computing environments are windowed and graphical in nature, console I/O is used mostly for simple, utility programs and for demonstration programs. Later in this book,

you will learn other ways to generate output using Java. But for now, we will continue to use the console I/O methods. Notice that the **println( )** statement ends with a semicolon. All statements in Java end with a semicolon. The reason that the other lines in the program do not end in a semicolon is that they are not, technically, statements. The first **}** in the program ends **main( )**, and the last **}** ends the **Example** class definition.

1.8 Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the **Example** program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

1.9 Identifiers

Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**. Some examples of valid identifiers are: AvgTemp count a4 $test this_is_ok Invalid variable names include: 2count high-temp Not/ok

1.10 Literals

A constant value in Java is created by using a *literal* representation of it. For example, here are some literals: 100 98.6 'X' "This is a test" Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

1.11 Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a *documentation comment.* This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a **/**** and ends with a ***/**. Documentation comments are explained in Appendix A.

1.12 Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table: Symbol Name Purpose

( ) Parentheses Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types. { } Braces Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes. [ ] Brackets Used to declare array types. Also used when dereferencing array values. ; Semicolon Terminates statements. , Comma Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement. . Period Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

1.13 The Java Keywords

There are 49 reserved keywords currently defined in the Java language (see Table 2-1). These keywords, combined with the syntax of the operators and separators, form the definition of the Java language. These keywords cannot be used as names for a variable, class, or method.

abstract	continue	goto	package	synchronized
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const	for	new	switch	

Table 2-1. *Java Reserved Keywords*

The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use. However, the current specification for Java only defines the keywords shown in Table 2-1. The **assert** keyword was added by Java 2, version 1.4 In addition to the keywords, Java reserves the following: **true**, **false**, and **null**. These are values defined by Java. You may not use these words for the names of variables, classes, and so on.

1.14 Data Types

Java Is a Strongly Typed Language

It is important to state at the outset that Java is a strongly typed language. Indeed, part of Java's safety and robustness comes from this fact. Let's see what this means. First, every variable has a type, every expression has a type, and every type is strictly defined. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercions or conversions of conflicting types as in some languages. The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be corrected before the compiler will finish compiling the class. *If you come from a C or C++ background, keep in mind that Java is more strictly typed than either language. For example, in C/C++ you can assign a floating-point value to an integer. In Java, you cannot. Also, in C there is not necessarily strong type-checking between a parameter and an argument. In Java, there is. You might find Java's strong type-checking a bit tedious at first. But remember, in the long run it will help reduce the possibility of errors in your code.*

The Simple Types

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**. These can be put in four groups:

■ Integers This group includes **byte**, **short**, **int**, and **long**, which are for wholevalued signed numbers.

■ Floating-point numbers This group includes **float** and **double**, which represent numbers with fractional precision.

■ Characters This group includes **char**, which represents symbols in a character set, like letters and numbers.

■ Boolean This group includes **boolean**, which is a special type for representing true/false values. You can use these types as-is, or to construct arrays or your own class types. Thus, they form the basis for all other types of data that you can create.

The simple types represent single values—not complex objects. Although Java is otherwise completely object-oriented, the simple types are not. They are analogous to the simple types found in most other non–object-oriented languages. The reason for this is efficiency. Making the simple types into objects would have degraded performance too much. The simple types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment.

However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an **int** is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run *without porting* on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability. Let's look at each type of data in turn.

Integers

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. Java does not support unsigned, positive-only integers. Many other computer languages, including C/C++, support both signed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of *unsigned* was used mostly to specify the behavior of the *high-order bit,* which defined the *sign* of an **int** when expressed as a number. As you will see in Chapter 4, Java manages the meaning of the high-order bit differently, by adding a special "unsigned right shift" operator.

Thus, the need for an unsigned integer type was eliminated. The *width* of an integer type should not be thought of as the amount of storage it consumes, but rather as the *behavior* it defines for variables and expressions of that type. The Java run-time environment is free to use whatever size it wants, as long as the types behave as you declared them. In fact, at least one implementation stores **byte**s and **short**s as 32-bit (rather than 8- and 16-bit) values to improve performance, because that is the word size of most computers currently in use. The width and ranges of these integer types vary widely, as shown in this table:

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Let's look at each type of integer.

Byte

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127. Variables of type **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're working with raw binary data that may not be directly compatible with Java's other built-in types. Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called **b** and **c**: byte b, c;

Short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type, since it is defined as having its high byte first (called *big-endian* format). This type is mostly applicable to 16-bit computers, which are becoming increasingly scarce. Here are some examples of **short** variable declarations: short s; short t;
*"Endianness" describes how multibyte data types, such as **short**, **int**, and **long**, are stored in memory. If it takes 2 bytes to represent a **short**, then which one comes first, the most significant or the least significant? To say that a machine is big-endian, means that the most significant byte is first, followed by the least significant one. Machines such as the SPARC and PowerPC are big-endian, while the Intel x86 series is little-endian.*

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** are commonly employed to control loops and to index arrays. Any time you have an integer expression involving **byte**s, **short**s, **int**s, and literal numbers, the entire expression is *promoted* to **int** before the calculation is done. The **int** type is the most versatile and efficient type, and it should be used most of the time when you want to create a number for counting or indexing arrays or doing integer math. It may seem that using **short** or **byte** will save space, but there is no guarantee that Java won't promote those types to **int** internally anyway. Remember, type determines behavior, not size. (The only exception is arrays, where **byte** is guaranteed to use only one byte per array element, **short** will use two bytes, and **int** will use four.)

long

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large. This makes it useful when big, whole numbers are needed. For example, here is a program that computes the number of miles that light will travel in a specified number of days.

```
// Compute distance light travels using long variables.

class Light {
public static void main(String args[]) {
int lightspeed;
long days;
long seconds;
long distance;
// approximate speed of light in miles per second
lightspeed = 186000;
days = 1000; // specify number of days here
seconds = days * 24 * 60 * 60; // convert to seconds
distance = lightspeed * seconds; // compute distance
System.out.print("In " + days);
System.out.print(" days light will travel about ");
System.out.println(distance + " miles.");
}
}
```

This program generates the following output:

```
In 1000 days light will travel about 16070400000000 miles.
```

Clearly, the result could not have been held in an **int** variable.

Floating-Point Types

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type. Java implements the standard (IEEE–754) set of floating-point types and operators. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively. Their width and ranges are shown here:

Name	Width in Bits	Approximate Range
double	64 4.9e	–324 to 1.8e+308
float	32 1.4e	–045 to 3.4e+038

Each of these floating-point types is examined next.

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision. For example, **float** can be useful when representing dollars and cents. Here are some example **float** variable declarations: float hightemp, lowtemp;

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations, or are manipulating large-valued numbers, **double** is the best choice. Here is a short program that uses **double** variables to compute the area of a circle:

```
// Compute the area of a circle.
class Area {
public static void main(String args[]) {
double pi, r, a;
r = 10.8; // radius of circle
pi = 3.1416; // pi, approximately
a = pi * r * r; // compute area
```

Characters

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++. In C/C++, **char** is an integer type that is 8 bits wide. This is *not* the case in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fully international character set that can represent all of the characters found in all human languages. It is a unification of dozens of character sets, such as Latin, Greek, Arabic, Cyrillic, Hebrew, Katakana, Hangul, and many more. For this purpose, it requires 16 bits. Thus, in Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8-bit character set, ISO-Latin-1, ranges from 0 to 255. Since Java is designed to allow applets to be written for worldwide use, it makes sense that it would use Unicode to represent characters. Of course, the use of Unicode is somewhat inefficient for languages such as English, German, Spanish, or French, whose characters can easily be contained within 8 bits. But such is the price that must be paid for global portability. Here is a program that demonstrates **char** variables:

```
// Demonstrate char data type.
class CharDemo {
public static void main(String args[]) {
char ch1, ch2;
ch1 = 88; // code for X
ch2 = 'Y';
System.out.print("ch1 and ch2: ");
System.out.println(ch1 + " " + ch2);
}
}
```

This program displays the following output:

```
ch1 and ch2: X Y
```


Notice that **ch1** is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter *X*. As mentioned, the ASCII character set occupies the first 127 values in the Unicode character set. For this reason, all the "old tricks" that you have used with characters in the past will work in Java, too. Even though **char**s are not integers, in many cases you can operate on them as if they were integers. This allows you to add two characters together, or to increment the value of a character variable. For example, consider the following program:

```
// char variables behave like integers.
class CharDemo2 {
public static void main(String args[]) {
```

```
char ch1;
ch1 = 'X';
System.out.println("ch1 contains " + ch1);
ch1++; // increment ch1
System.out.println("ch1 is now " + ch1);
}
}
```

The output generated by this program is shown here:
ch1 contains X
ch1 is now Y

In the program, **ch1** is first given the value *X*. Next, **ch1** is incremented. This results in **ch1** containing *Y,* the next character in the ASCII (and Unicode) sequence.

Booleans

Java has a simple type, called **boolean**, for logical values. It can have only one of two possible values, **true** or **false**. This is the type returned by all relational operators, such as **a < b**. **boolean** is also the type *required* by the conditional expressions that govern the control statements such as **if** and **for**. Here is a program that demonstrates the **boolean** type:

```
// Demonstrate boolean values.
class BoolTest {
public static void main(String args[]) {
boolean b;
b = false;
System.out.println("b is " + b);
b = true;
System.out.println("b is " + b);
// a boolean value can control the if statement
if(b) System.out.println("This is executed.");
b = false;
if(b) System.out.println("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.println("10 > 9 is " + (10 > 9));
}
}
```

The output generated by this program is shown here:
b is false
b is true
This is executed.
10 > 9 is true

There are three interesting things to notice about this program. First, as you can see, when a **boolean** value is output by **println( )**, "true" or "false" is displayed. Second, the value of a **boolean** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this: if(b == true) ... Third, the outcome of a relational operator, such as **<**, is a **boolean** value. This is why the expression **10 > 9** displays the value "true." Further, the extra set of parentheses around **10 > 9** is necessary because the **+** operator has a higher precedence than the **>**.

1.15 A Closer Look at Literals

Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1, 2, 3, and 42. These are all decimal values, meaning they are describing a base 10 number. There are two other bases which can be used in integer literals, *octal* (base eight) and *hexadecimal* (base 16). Octal values are denoted in Java by a leading zero. Normal decimal numbers cannot have a leading zero. Thus, the seemingly valid value 09 will produce an error from the compiler, since 9 is outside of octal's 0 to 7 range. A more common base for numbers used by programmers is hexadecimal, which matches cleanly with modulo 8 word sizes, such as 8, 16, 32, and 64 bits.

You signify a hexadecimal constant with a leading zero-x, (**0x** or **0X**). The range of a hexadecimal digit is 0 to 15, so *A* through *F* (or *a* through *f)* are substituted for 10 through 15. Integer literals create an **int** value, which in Java is a 32-bit integer value. Since Java is strongly typed, you might be wondering how it is possible to assign an integer literal to one of Java's other integer types, such as **byte** or **long**, without causing a type mismatch error. Fortunately, such situations are easily handled. When a literal value is assigned to a **byte** or **short** variable, no error is generated if the literal value is within the range of the target type. Also, an integer literal can always be assigned to a **long** variable. However, to specify a **long** literal, you will need to explicitly tell the compiler that the literal value is of type **long**. You do this by appending an upper- or lowercase *L* to the literal. For example, 0x7fffffffffffffL or 9223372036854775807L is the largest **long**.

Floating-Point Literals

Floating-point numbers represent decimal values with a fractional component. They can be expressed in either standard or scientific notation. *Standard notation* consists of a whole number component followed by a decimal point followed by a fractional component. For example, 2.0, 3.14159, and 0.6667 represent valid standard-notation floating-point numbers. *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied. The exponent is indicated by an *E* or *e* followed by a decimal number, which can be positive or negative. Examples include 6.022E23, 314159E–05, and 2e+100. Floating-point literals in Java default to **double** precision. To specify a **float** literal, you must append an *F* or *f* to the constant. You can also explicitly specify a **double** literal by appending a *D* or *d.* Doing so is, of course, redundant. The default **double** type consumes 64 bits of storage, while the less-accurate **float** type requires only 32 bits.

Boolean Literals

Boolean literals are simple. There are only two logical values that a **boolean** value can have, **true** and **false**. The values of **true** and **false** do not convert into any numerical representation. The **true** literal in Java does not equal 1, nor does the **false** literal equal 0. In Java, they can only be assigned to variables declared as **boolean**, or used in expressions with Boolean operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. A literal character is represented inside a pair of single quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as *'a'*, *'z'*, and *'@'*. For characters that are impossible to enter directly, there are several escape sequences, which allow you to enter the character you need, such as '\'' for the single-quote character itself, and **'\n'** for the newline character. There is also a mechanism for directly entering the value of a character in octal or hexadecimal. For octal notation use the backslash followed by the three-digit number. For example, *'\141'* is the letter *'a'*. For hexadecimal, you enter a

backslash-u (**\u**), then exactly four hexadecimal digits. For example, *'\u0061'* is the ISO-Latin-1 *'a'* because the top byte is zero. *'\ua432'* is a Japanese Katakana character. Table 3-1 shows the character escape sequences.

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Table 3-1. *Character Escape Sequences*

String Literals

String literals in Java are specified like they are in most other languages—by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are "Hello World"

"two\nlines"
"\"This is in quotes\""

The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One important thing to note about Java strings is that they must begin and end on the same line. There is no line-continuation escape sequence as there is in other languages. *As you may know, in some other languages, including C/C++, strings are implemented as arrays of characters. However, this is not the case in Java. Strings are actually object types. As you will see later in this book, because Java implements strings as objects, Java includes extensive string-handling capabilities that are both powerful and easy to use.*

1.16 Variables

The variable is the basic unit of storage in a Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and a lifetime. These elements are examined next.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here: *type identifier* [= *value*][, *identifier* [= *value*] ...] ; The *type* is one of Java's atomic types, or

the name of a class or interface. (Class and interface types are discussed later in Part I of this book.) The *identifier* is the name of the variable. You can initialize the variable by specifying an equal sign and a value. Keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type, use a comma-separated list. Here are several examples of variable declarations of various types. Note that some include an initialization. int a, b, c;

```
 // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing
// d and f.
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Many readers will remember when FORTRAN predefined all identifiers from **I** through **N** to be of type **INTEGER** while all other identifiers were **REAL**. Java allows any properly formed identifier to have any declared type.

Dynamic Initialization

Although the preceding examples have used only constants as initializers, Java allows variables to be initialized dynamically, using any expression valid at the time the variable is declared.
For example, here is a short program that computes the length of the hypotenuse of a right triangle given the lengths of its two opposing sides:

```
// Demonstrate dynamic initialization.
class DynInit {
public static void main(String args[]) {
double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

Here, three local variables—**a**, **b**,and **c**—are declared. The first two, **a** and **b**, are initialized by constants. However, **c** is initialized dynamically to the length of the hypotenuse (using the Pythagorean theorem). The program uses another of Java's built-in methods, **sqrt( )**, which is a member of the **Math** class, to compute the square root of its argument. The key point here is that the initialization expression may use any element valid at the time of the initialization, including calls to methods, other variables, or literals.

The Scope and Lifetime of Variables

So far, all of the variables used have been declared at the start of the **main( )** method. However, Java allows variables to be declared within any block. As explained in Chapter 2, a block is begun with an opening curly brace and ended by a closing curly brace. A block defines a *scope.* Thus, each time you start a new block, you are creating a new scope. As you probably know from your previous programming experience, a scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Most other computer languages define two general categories of scopes: global and local. However, these traditional scopes do not fit well with Java's strict, objectoriented model. While it is possible to create what amounts to being a global scope, it is by far the exception, not the rule. In Java, the two major scopes are those defined by a class and those defined by a method. Even this distinction is somewhat artificial.

However, since the class scope has several unique properties and attributes that do not apply to the scope defined by a method, this distinction makes some sense. Because of the differences, a discussion of class scope (and variables declared within it) is deferred until Chapter 6, when classes are described. For now, we will only examine the scopes defined by or within a method.

The scope defined by a method begins with its opening curly brace. However, if that method has parameters, they too are included within the method's scope. Although this book will look more closely at parameters in Chapter 5, for the sake of this discussion, they work the same as any other method variable. As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.

Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification. Indeed, the scope rules provide the foundation for encapsulation. Scopes can be nested. For example, each time you create a block of code, you are creating a new, nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code within the inner scope.

However, the reverse is not true. Objects declared within the inner scope will not be visible outside it. To understand the effect of nested scopes, consider the following program: // Demonstrate block scope.

```
class Scope {
public static void main(String args[]) {
int x; // known to all code within main
x = 10;
if(x == 10) { // start new scope
int y = 20; // known only to this block
// x and y both known here.
System.out.println("x and y: " + x + " " + y);
x = y * 2;
}
// y = 100; // Error! y not known here
// x is still known here.
System.out.println("x is " + x);
}
}
```

As the comments indicate, the variable **x** is declared at the start of **main( )**'s scope and is accessible to all subsequent code within **main( )**. Within the **if** block, **y** is declared. Since a block defines a scope, **y** is only visible to other code within its block. This is why outside of its block, the line **y = 100;** is commented out. If you remove the leading comment symbol, a compile-time error will occur, because **y** is not visible outside of its block. Within the **if** block, **x** can be used because code within a block (that is, a nested scope) has access to variables declared by an enclosing scope. Within a block, variables can be declared at any point, but are valid only after they are declared. Thus, if you define a variable at the start of a method, it is available to all of the code within that method. Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it. For example, this fragment is invalid because **count** cannot be used prior to its declaration: // This fragment is wrong! count = 100; // oops! cannot use count before it is declared!

int count; Here is another important point to remember: variables are created when their scope is entered, and destroyed when their scope is left. This means that a variable will not hold its value once it has gone out of scope. Therefore, variables declared within a method will not hold their values between calls to that method. Also, a variable declared within a block will lose its value when the block is left. Thus, the lifetime of a variable is confined to its scope. If a variable declaration includes an initializer, then that variable will be reinitialized each time the block in which it is declared is entered. For example, consider the next program.

```
// Demonstrate lifetime of a variable.
class LifeTime {
public static void main(String args[]) {
int x;
for(x = 0; x < 3; x++) {
int y = -1; // y is initialized each time block is entered
System.out.println("y is: " + y); // this always prints -1
y = 100;
System.out.println("y is now: " + y);
}
}
}
```

The output generated by this program is shown here:

```
y is: -1
y is now: 100
y is: -1
y is now: 100
y is: -1
y is now: 100
```

As you can see, **y** is always reinitialized to –1 each time the inner **for** loop is entered. Even though it is subsequently assigned the value 100, this value is lost. One last point: Although blocks can be nested, you cannot declare a variable to have the same name as one in an outer scope. In this regard, Java differs from C and C++. Here is an example that tries to declare two separate variables with the same name. In Java, this is illegal. In C/C++, it would be legal and the two **bar**s would be separate.

```
// This program will not compile
class ScopeErr {
public static void main(String args[]) {
int bar = 1;
{ // creates a new scope
int bar = 2; // Compile-time error – bar already defined!
}
}
}
```

5.17 Type Conversion and Casting

If you have previous programming experience, then you already know that it is fairly common to assign a value of one type to a variable of another type. If the two types are compatible, then Java will perform the conversion automatically. For example, it is always possible to assign an **int** value to a **long** variable. However, not all types are compatible, and thus, not all type conversions are implicitly allowed. For instance, there is no conversion defined from **double** to **byte**. Fortunately, it is still possible to obtain a conversion between incompatible types. To do so, you must use a *cast,* which performs an explicit conversion between incompatible types. Let's look at both automatic type conversions and casting.

Java's Automatic Conversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

- The two types are compatible.
- The destination type is larger than the source type.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required. For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other. However, the numeric types are not compatible with **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other. As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, or **long**.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion,* since you are explicitly making the value narrower so that it will fit into the target type. To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion. It has this general form:

(*target-type*) *value* Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range. int a; byte b; // ... b = (byte) a; A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components.

Thus, when a floating-point value is assigned to an integer type, the fractional component is lost. For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1. The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range. The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
public static void main(String args[]) {
byte b;
int i = 257;
double d = 323.142;
System.out.println("\nConversion of int to byte.");
b = (byte) i;
System.out.println("i and b " + i + " " + b);
System.out.println("\nConversion of double to int.");
i = (int) d;
System.out.println("d and i " + d + " " + i);
System.out.println("\nConversion of double to byte.");
b = (byte) d;
System.out.println("d and b " + d + " " + b);
}
}
```

This program generates the following output:

```
Conversion of int to byte.
i and b 257 1
Conversion of double to int.
d and i 323.142 323
Conversion of double to byte.
d and b 323.142 67
```

Let's look at each conversion. When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case. When the **d** is converted to an **int**,

its fractional component is lost. When **d** is converted to a **byte**, its fractional component is lost, *and* the value is reduced modulo 256, which in this case is 67.

5.18 Automatic Type Promotion in Expressions

In addition to assignments, there is another place where certain type conversions may occur: in expressions. To see why, consider the following. In an expression, the precision required of an intermediate value will sometimes exceed the range of either operand. For example, examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte** operands. To handle this kind of problem, Java automatically promotes each **byte** or **short** operand to **int** when evaluating an expression. This means that the subexpression **a * b** is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression, **50 * 40**, is legal even though **a** and **b** are both specified as type **byte**. As useful as the automatic promotions are, they can cause confusing compile-time errors. For example, this seemingly correct code causes a problem: byte b = 50; b = b * 2; // Error! Cannot assign an int to a byte! The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable. However, because the operands were automatically promoted to **int** when the expression was evaluated, the result has also been promoted to **int**. Thus, the result of the expression is now of type **int**, which cannot be assigned to a **byte** without the use of a cast. This is true even if, as in this particular case, the value being assigned would still fit in the target type. In cases where you understand the consequences of overflow, you should use an
explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```
which yields the correct value of 100.

The Type Promotion Rules

In addition to the elevation of **byte**s and **short**s to **int**, Java defines several *type promotion rules* that apply to expressions. They are as follows. First, all **byte** and **short** values are promoted to **int**, as just described. Then, if one operand is a **long**, the whole expression is promoted to **long**. If one operand is a **float,** the entire expression is promoted to **float**. If any of the operands is **double**, the result is **double**. The following program demonstrates how each value in the expression gets promoted to match the second argument to each binary operator:


```
class Promote {
public static void main(String args[]) {
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) - (d * s);
System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
System.out.println("result = " + result);
}
}
```
Let’s look closely at the type promotions that occur in this line from the program:

double result = (f * b) + (i / c) - (d * s);

In the first subexpression, **f * b**, **b** is promoted to a **float** and the result of the subexpression is **float**. Next, in the subexpression **i / c**, **c** is promoted to **int**, and the result is of type **int**. Then, in **d * s**, the value of **s** is promoted to **double**, and the type of the subexpression is **double**. Finally, these three intermediate values, **float**, **int**, and **double**, are considered. The outcome of **float** plus an **int** is a **float**. Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression.

1.19 Arrays

An *array* is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information. *If you are familiar with C/C++, be careful. Arrays in Java work differently than they do in those languages.*

One-Dimensional Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type. The general form of a onedimensional array declaration is
type var-name[ ];

Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month_days** with the type "array of int":

int month_days[];

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory. You will look more closely at **new** in a later chapter, but you need to use it now to allocate memory for arrays. The general form of **new** as it applies to one-dimensional arrays appears as follows:

array-var = new *type*[*size*];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to **month_days**.
month_days = new int[12];

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero. Let's review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated. If the concept of dynamic allocation is unfamiliar to you, don't worry. It will be described at length later in this book. Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **month_days**. month_days[1] = 28; The next line displays the value stored at index 3. System.out.println(month_days[3]); Putting together all the pieces, here is a program that creates an array of the number of days in each month.

// Demonstrate a one-dimensional array.

```
class Array {
public static void main(String args[]) {
int month_days[];
month_days = new int[12];
month_days[0] = 31;
month_days[1] = 28;
month_days[2] = 31;
month_days[3] = 30;
month_days[4] = 31;
month_days[5] = 30;
month_days[6] = 31;
month_days[7] = 31;
month_days[8] = 30;
month_days[9] = 31;
month_days[10] = 30;
month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is **month_days[3]** or 30. It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
 int month_days[] = new int[12];
```

This is the way that you will normally see it done in professionally written Java programs. Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**. For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
public static void main(String args[]) {
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,
30, 31 };
System.out.println("April has " + month_days[3] + " days.");
}
}
```


When you run this program, you see the same output as that generated by the
previous version. Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. (In this regard, Java is fundamentally different from C/C++, which provide no run-time boundary checks.) For example, the run-time system will check the value of each index into **month_days** to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error. Here is one more example that uses a one-dimensional array. It finds the average of a set of numbers. // Average an array of values.

```
class Average {
public static void main(String args[]) {
double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
double result = 0;
```

```
int i;
for(i=0; i<5; i++)
result = result + nums[i];
System.out.println("Average is " + result / 5);
}
}
```

Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1. there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called **twoD**.

```
int twoD[][] = new int[4][5];
```

This allocates a 4 by 5 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**. Conceptually, this array will look like the one shown in Figure 3-1. Figure

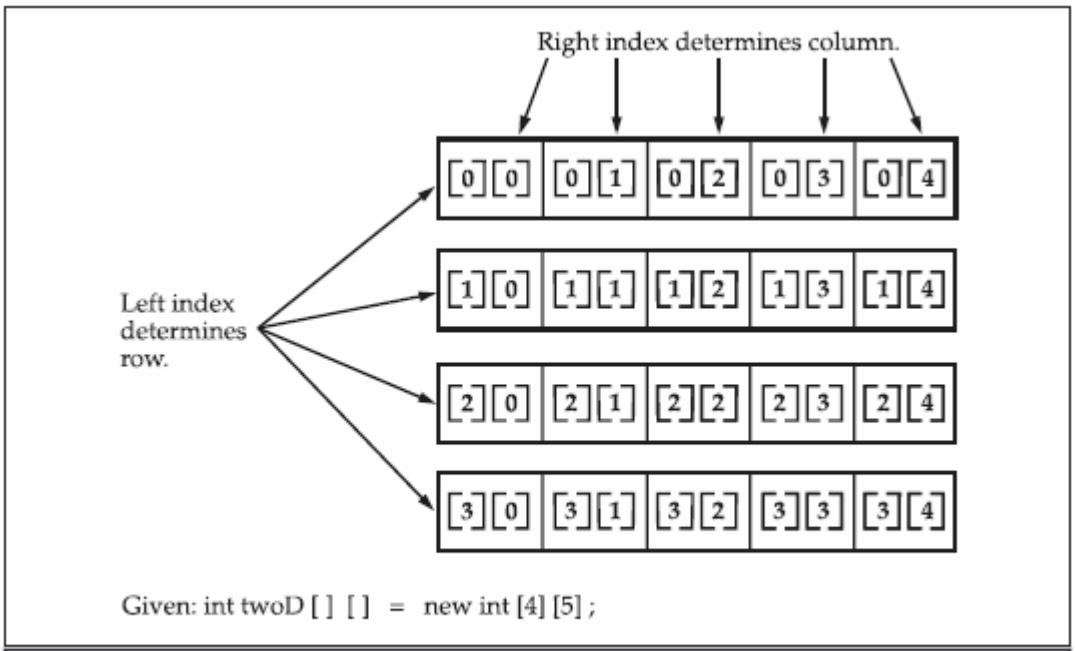


Figure 3-1. *A conceptual view of a 4 by 5, two-dimensional array*

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
```

```
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}
}
}
```

This program generates the following output:

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

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

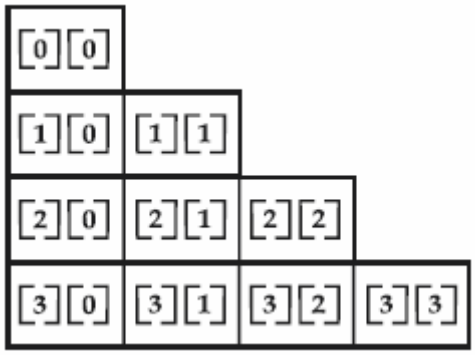
```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a twodimensional array in which the sizes of the second dimension are unequal.

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];
twoD[1] = new int[2];
twoD[2] = new int[3];
twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
```

```
System.out.println();
}
}
}
```
This program generates the following output:
```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



The use of uneven (or, irregular) multidimensional arrays is not recommended for most applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, it can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution. It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes. Also notice that you can use expressions as well as literal values inside of array initializers.

```
// Initialize a two-dimensional array.
class Matrix {
public static void main(String args[]) {
double m[][] = {
{ 0*0, 1*0, 2*0, 3*0 },
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println();
}
}
}
```


When you run this program, you will get the following output:
```
0.0     0.0     0.0     0.0
0.0     1.0     2.0     3.0
0.0     2.0     4.0     6.0
0.0     3.0     6.0     9.0
```

As you can see, each row in the array is initialized as specified in the initialization lists. Let's look at one more example that uses a multidimensional array. The following program creates a 3 by 4 by 5, three-dimensional array. It then loads each element with the product of its indexes. Finally, it displays these products.

```
// Demonstrate a three-dimensional array.

class threeDMatrix {
public static void main(String args[]) {
int threeD[][][] = new int[3][4][5];
int i, j, k;
for(i=0; i<3; i++)
for(j=0; j<4; j++)
for(k=0; k<5; k++)
threeD[i][j][k] = i * j * k;
for(i=0; i<3; i++) {
for(j=0; j<4; j++) {
for(k=0; k<5; k++)
System.out.print(threeD[i][j][k] + " ");
System.out.println();
}
System.out.println();
}
}
}
```

This program generates the following output:

```
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  1  2  3  4
0  2  4  6  8
0  3  6  9 12
0  0  0  0  0
0  2  4  6  8
0  4  8 12 16
0  6 12 18 24
```


Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

type[ ] *var-name;*

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form is included as a convenience, and is also useful when specifying an array as a return type for a method.

1.20 Operators

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. This chapter describes all of Java's operators except for the type comparison operator **instanceof**, which is examined in Chapter 12. *If you are familiar with C/C++/C#, then you will be pleased to know that most operators in Java work just like they do in those languages. However, there are some subtle differences, so a careful reading is advised.*

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:

Operator Result
+ Addition
– Subtraction (also unary minus)
* Multiplication
/ Division
% Modulus
++ Increment
+= Addition assignment
–= Subtraction assignment
*= Multiplication assignment
/= Division assignment
%= Modulus assignment
– – Decrement

The operands of the arithmetic operators must be of a numeric type. You cannot use them 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.
class BasicMath {
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;
double db = 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 = 2.0
db = 6.0
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 = 42.25;
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 = 2.25
```

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, any statement 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.
class OpEquals {
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 were introduced in Chapter 2. Here they will be discussed in detail. As you will see, they have some special properties that make them quite interesting. Let's begin by reviewing precisely what the increment and decrement operators do. 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 ++.
class IncDec {
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 following table:

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 20 at the rightmost bit. The next bit position to the left would be 21, 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 is represented 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, producing 100000000.

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.

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
11010101

after 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.
class BitLogic {
```

```
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 **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.
class ByteShift {
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 (0100 0000) twice results in **i** containing the value 256 (1 0000 0000). However, the value in **b** contains 0 because after the shift, the low-order byte is now zero. Its only 1 bit has been shifted 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.
class MultByTwo {
public static void main(String args[]) {
int i;
int num = 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 **a** being 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, is
11111000 –8
>>1
11111100 –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.
class HexByte {
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. This situation 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 its initial value was. This is known as an *unsigned shift.* To accomplish this, you will use Java'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 for 32- 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 a 32-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.
class ByteUShift {
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 the shift. 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 bit after 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 shorthandform of bitwise operator assignments to manipulate the variables:

```
class OpBitEquals {
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/C++/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.

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	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
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.
class BoolLogic {
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 many 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 the left 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, 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 is evaluated 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);
}
}
```

The output generated by the program is shown here:

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

Absolute value of -10 is 10

Operator Precedence

Table 4-1 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.

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

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

Table 4-1. *The Precedence of the Java Operators*

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.

1.21 control statements

Aprogramming language uses *control* statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration, and jump. *Selection* statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable. *Iteration* statements enable program 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 of Java's control statements are examined here.

If you know C/C++/C#, then Java's control statements will be familiar territory. In fact, Java's control statements are nearly identical to those in those languages. However, there are a few differences—especially in the ***break*** *and* ***continue*** *statements.*

Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time. You will be pleasantly surprised by the power and flexibility contained in these two statements.

if

The **if** statement was introduced in Chapter 2. It is examined in detail here. The **if** statement is Java's conditional branch statement. It can be used to route program execution 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 in curly 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. For example, 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 are they 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 single **boolean** 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 to include 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 than zero. Some programmers find it convenient to include the curly braces when using the **if**, even when there is only one statement in each clause. This makes it easy to add another statement 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 executed inside 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 was intended. 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 **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **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 not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else** is associated with **if(i==10)**. The inner **else** refers to **if(k>100)**, because it is the closest **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the

if-else-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 conditions controlling the **if** is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place. Here is a program that uses an **if-else-if** ladder to determine which season a particular month 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 to dispatch 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 in the **case** statements must be of a type compatible with the expression. Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The **switch** statement works like this: The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is 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. When a **break** statement is encountered, execution branches to the first line of code that follows the 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 **case** constant 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 the next **case**. It is sometimes desirable to have multiple **case**s without **break** statements between 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 of the **switch**) is reached. While the preceding example is, of course, contrived for the sake of illustration, omitting the **break** statement has many practical applications in real programs. To sample its more 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 is called a *ested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between he **case** constants in the inner **switch** and those in the outer **switch**. For example, the ollowing 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 n the outer switch. The **count** variable is only compared with the list of cases at the uter level. If **count** is 1, then **target** is compared with the inner list cases. n 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 **f** can evaluate any type of Boolean expression. That is, the **switch** looks only or a match between 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, **switch** statement enclosed by an outer **switch** can have **case** constants n common.

■ A **switch** statement is usually more efficient than a set of nested **if**s. he last point is particularly interesting because it gives insight into how the Java ompiler works. When it compiles a **switch** statement, the Java compiler will inspect ach of the **case** constants and create a "jump table" that it will use for selecting the ath of execution depending on the value of the expression. Therefore, if you need to elect among a large group of values, a **switch** statement will run much faster than the quivalent logic coded using a sequence of **if-else**s. The compiler can do this because it nows that the **case** constants are all the same type and simply must be compared for quality with the **switch** expression. The compiler has no such knowledge of a long listv **if** expressions.

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what e commonly call *loops.* As you probably know, a loop repeatedly executes the same et of instructions until a termination condition is met. As you will see, Java has a loop o fit any programming need.

while

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

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

The *condition* can be any Boolean expression. The body of the loop will be executed as ong as the conditional expression is true. When *condition* becomes false, control passes o the next line of code immediately following the loop. The curly braces are unnecessary f only a single statement is being repeated. ere 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
110 J a v a ™ 2 : T h e C o m p l e t e R e f e r e n c e
tick 3
tick 2
tick 1

Since the **while** loop evaluates its conditional expression at the top of the loop, the ody of the loop will not execute even once if the condition is false to begin with. For xample, 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 *null statement* (one that consists only of a semicolon) is syntactically valid in Java. For xample, 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: idpoint is 150 ere is how the **while** loop works. The value of **i** is incremented, and the value of **j** s decremented. These values are then compared with one another. If the new value of is still less than the new value of **j**, then the loop repeats. If **i** is equal to or greater than , the loop stops. Upon exit from the loop, **i** will hold a value that is midway between he original values of **i** and **j**. (Of course, this procedure only works when **i** is less than **j** o begin with.) As you can see, there is no need for a loop body; all of the action occurs ithin the conditional expression, itself. In professionally written Java code, short loops re frequently coded without bodies when the controlling expression can handle all of he details itself.

do-while

As you just saw, if the conditional expression controlling a **while** loop is initially false, hen the body of the loop will not be executed at all. However, sometimes it is desirable o execute the body of a **while** loop at least once, even if the conditional expression is alsoto begin with. In other words, there are times when you would like to test thecermination expression at the end of the loop rather than at the beginning. Fortunately, ava supplies a loop that does just that: the **do-while**. The **do-while** loop always executes ts body at least once, because its conditional expression is at the bottom of the loop. Its general form is

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

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop 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. It generates 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 more efficiently 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 for zero 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 you will 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 iteration statements:

// 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 valid choice. If not, then the user is reprompted. Since the menu must be displayed at least once, the **do-while** is the perfect loop to accomplish this. A few other points about this example: Notice that characters are read from the keyboard by calling **System.in.read( )**. This is one of Java's console input functions. Although Java's console I/O methods won't be discussed in detail until Chapter 12, **System.in.read( )** is used here to obtain the user's choice. It reads characters from standard input (returned as integers, which is why the return value was cast to **char**). By default, standard input is line buffered, so you must press ENTER before any characters that you type will be sent to your program. Java's console input is quite limited and awkward to work with. Further, most real-world Java programs and applets will be graphical and window-based. For these reasons, not much use of console input has been made in this book. However, it is useful in this context. One other point: Because **System.in.read( )** is being used, the program must specify the **throws java.io.IOException** clause. This line is necessary to handle input errors. It is part of Java's exception handling features, which are discussed in Later Chapter

for

You were introduced to a simple form of the **for** loop in Chapter 2. As you will see, it is a powerful 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 the loop is executed. Generally, this is an expression that sets the value of the *loop* c*control variable,* which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once. Next, *condition* is evaluated. Thiscmust be 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 loop terminates. Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, 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 the controlling 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 and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**. For example, here is the preceding program recoded so 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 the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop. When the loop control variable will not be needed elsewhere, most Java programmers declare it inside the **for**. For example, here is a simple program that tests for prime numbers. Notice that the loop control variable, **i**, is declared inside the **for** since it is not needed 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 the initialization and iteration portions of the **for** loop. For example, consider the loop in the following 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 is governed by two variables, it would be useful if both could be included in the **for** statement, itself, instead of **b** being handled manually. Fortunately, Java provides a way to accomplish this. To allow two or more variables to control a **for** loop, Java permits you to 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 comma-separated statements 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
```

*If you are familiar with C/C++, then you know that in those languages the comma is an operator that can be used in any valid expression. However, this is not the case with Java. In Java, the comma is a separator that applies only to the **for** loop.*

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, and the iteration, do not need to be used for only those purposes. In fact, the three sections of the **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, this expression does not need to test the loop control variable against some target value. In fact, 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 to **true**. It does not test the value of **i**. Here is another interesting **for** loop variation. Either the initialization or the iteration expression 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, it would be considered quite poor style—there can be times when this type of approach makes sense. For example, if the initial condition is set through a complex expression elsewhere in the program or if the loop control variable changes in a nonsequential manner determined by actions that occur within the body of the loop, it may be appropriate to leave these parts of the **for** empty. Here is one more **for** loop variation. You can intentionally create an infinite loop (a loop that 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, that require an infinite loop, most "infinite loops" are really just loops with special termination requirements. As you will soon see, there is a way to terminate a loop— even an infinite loop 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 loop may 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:

```
.........
........
.......
......
.....
....
...
..
.
```

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program. Each is examined here. *In addition to the jump statements discussed here, Java supports one other way that you can change your program's flow of execution: through exception handling. Exception handling provides a structured method by which run-time errors can be trapped and handled by your program. It is supported by the keywords **try**, **catch**, **throw**, **throws**, and **finally**. In essence, the exception handling mechanism allows your program to perform a nonlocal branch. Since exception handling is a large topic, it is discussed in its own chapter, Chapter 10.*

Using break

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as a "civilized" form of goto. The last two uses are explained here.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop. When a **break** statement is encountered inside a loop, the loop is terminated and program control resumes at the next statement following the loop. Here is a simple example:

```
// Using break to exit a loop.
class BreakLoop {
public static void main(String args[]) {
for(int i=0; i<100; i++) {
if(i == 10) break; // terminate loop if i is 10
```

```
System.out.println("i: " + i);
}
System.out.println("Loop complete.");
}
}
```

This program generates the following output:

```
i: 0
i: 1
i: 2
i: 3
i: 4
i: 5
i: 6
i: 7
i: 8
i: 9
Loop complete.
```

As you can see, although the **for** loop is designed to run from 0 to 99, the **break** statement causes it to terminate early, when **i** equals 10. The **break** statement can be used with any of Java's loops, including intentionally infinite loops. For example, here is the preceding program coded by use of a **while** loop. The output from this program is the same as just shown.

```
// Using break to exit a while loop.
class BreakLoop2 {
public static void main(String args[]) {
int i = 0;
while(i < 100) {
if(i == 10) break; // terminate loop if i is 10
System.out.println("i: " + i);
i++;
}
System.out.println("Loop complete.");
}
}
```

When used inside a set of nested loops, the **break** statement will only break out of the innermost loop. For example:


```
// Using break with nested loops.
class BreakLoop3 {
public static void main(String args[]) {
for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
for(int j=0; j<100; j++) {
if(j == 10) break; // terminate loop if j is 10
System.out.print(j + " ");
}
System.out.println();
}
System.out.println("Loops complete.");
}
}
```

This program generates the following output:

```
Pass 0: 0 1 2 3 4 5 6 7 8 9
```

Pass 1: 0 1 2 3 4 5 6 7 8 9
Pass 2: 0 1 2 3 4 5 6 7 8 9
Loops complete.

As you can see, the **break** statement in the inner loop only causes termination of that loop. The outer loop is unaffected. Here are two other points to remember about **break**. First, more than one **break** statement may appear in a loop. However, be careful. Too many **break** statements have the tendency to destructure your code. Second, the **break** that terminates a **switch** statement affects only that **switch** statement and not any enclosing loops. ***break*** *was not designed to provide the normal means by which a loop is terminated. The loop's conditional expression serves this purpose. The* ***break*** *statement should be used to cancel a loop only when some sort of special situation occurs.*

Using break as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also be employed by itself to provide a "civilized" form of the goto statement. Java does not have a goto statement, because it provides a way to branch in an arbitrary and unstructured manner. This usually makes goto-ridden code hard to understand and hard to maintain. It also prohibits certain compiler optimizations. There are, however, a few places where the goto is a valuable and legitimate construct for flow control. For example, the goto can be useful when you are exiting from a deeply nested set of loops. To handle such situations, Java defines an expanded form of the **break** statement. By using this form of **break**, you can break out of one or more blocks of code. These blocks need not be part of a loop or a **switch**. They can be any block. Further, you can specify precisely where execution will resume, because this form of **break** works with a label. As you will see, **break** gives you the benefits of a goto without its problems. The general form of the labeled **break** statement is shown here:

break *label*;

Here, *label* is the name of a label that identifies a block of code. When this form of **break** executes, control is transferred out of the named block of code. The labeled block of code must enclose the **break** statement, but it does not need to be the immediately enclosing block. This means that you can use a labeled **break** statement to exit from a set of nested blocks. But you cannot use **break** to transfer control to a block of code that does not enclose the **break** statement.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followed by a colon. Once you have labeled a block, you can then use this label as the target of a **break** statement. Doing so causes execution to resume at the *end* of the labeled block. For example, the following program shows three nested blocks, each with its own label. The **break** statement causes execution to jump forward, past the end of the block labeled **second**, skipping the two **println( )** statements


```
. // Using break as a civilized form of goto.
class Break {
public static void main(String args[]) {
boolean t = true;
first: {
second: {
third: {
System.out.println("Before the break.");
if(t) break second; // break out of second block
System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");
}
```

```
}
}
```

Running this program generates the following output:
Before the break.
This is after second block.

One of the most common uses for a labeled **break** statement is to exit from nested loops. For example, in the following program, the outer loop executes only once:

```
// Using break to exit from nested loops
class BreakLoop4 {
public static void main(String args[]) {
outer: for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
for(int j=0; j<100; j++) {
if(j == 10) break outer; // exit both loops
System.out.print(j + " ");
}
System.out.println("This will not print");
}
System.out.println("Loops complete.");
}
}
```

This program generates the following output:
Pass 0: 0 1 2 3 4 5 6 7 8 9 Loops complete.

As you can see, when the inner loop breaks to the outer loop, both loops have been terminated. Keep in mind that you cannot break to any label which is not defined for an enclosing block. For example, the following program is invalid and will not compile:

```
// This program contains an error.
class BreakErr {
public static void main(String args[]) {
one: for(int i=0; i<3; i++) {
System.out.print("Pass " + i + ": ");
}
for(int j=0; j<100; j++) {
if(j == 10) break one; // WRONG
System.out.print(j + " ");
}
}
}
```


Since the loop labeled **one** does not enclose the **break** statement, it is not possible to transfer control to that block.

Using continue

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The **continue** statement performs such an action. In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop. In a **for** loop, control goes first to the iteration portion of the **for**

statement and then to the conditional expression. For all three loops, any intermediate code is bypassed. Here is an example program that uses **continue** to cause two numbers to be printed on each line:

```
// Demonstrate continue.
class Continue {
public static void main(String args[]) {
for(int i=0; i<10; i++) {
System.out.print(i + " ");
if (i%2 == 0) continue;
System.out.println("");
}
}
}
```

This code uses the **%** operator to check if **i** is even. If it is, the loop continues without printing a newline. Here is the output from this program:

```
0 1
2 3
4 5
6 7
8 9
```

As with the **break** statement, **continue** may specify a label to describe which enclosing loop to continue. Here is an example program that uses **continue** to print a triangular multiplication table for 0 through 9.

```
// Using continue with a label.
class ContinueLabel {
public static void main(String args[]) {
outer: for (int i=0; i<10; i++) {
for(int j=0; j<10; j++) {
if(j > i) {
System.out.println();
continue outer;
}
System.out.print(" " + (i * j));
}
}
System.out.println();
}
}
```


The **continue** statement in this example terminates the loop counting **j** and continues with the next iteration of the loop counting **i**. Here is the output of this program:

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Good uses of **continue** are rare. One reason is that Java provides a rich set of loop statements which fit most applications. However, for those special circumstances in which early iteration is needed, the **continue** statement provides a structured way to accomplish it.

Return

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. Although a full discussion of **return** must wait until methods are discussed in Chapter 7, a brief look at **return** is presented here. At any time in a method the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main( )**.

```
// Demonstrate return.
class Return {
public static void main(String args[]) {
boolean t = true;
System.out.println("Before the return.");
if(t) return; // return to caller
System.out.println("This won't execute.");
}
}
```

The output from this program is shown here:
Before the return. As you can see, the final **println( )** statement is not executed. As soon as **return** is executed, control passes back to the caller. One last point: In the preceding program, the **if(t)** statement is necessary. Without it, the Java compiler would flag an "unreachable code" error, because the compiler would know that the last **println( )** statement would never be executed. To prevent this error, the **if** statement is used here to trick the compiler for the sake of this demonstration.

2.
Java Classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class. Because the class is so fundamental to Java, this and the next few chapters will be devoted to it. Here, you will be introduced to the basic elements of a class and learn how a class can be used to create objects. You will also learn about methods, constructors, and the **this** keyword.

2.1 Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been used. The classes created in the preceding chapters primarily exist simply to encapsulate the **main( )** method, which has been used to demonstrate the basics of the Java syntax. As you will see, classes are substantially more powerful than the limited ones presented so far. Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

2.2 The General Form of a Class

When you define a class, you declare its exact form and nature. You do this by specifying the data that it contains and the code that operates on that data.While very simple classes may contain only code or only data, most real-world classes contain both. As you will see, a class' code defines the interface to its data. A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. The general form of a **class** definition is shown here:

```
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 is contained within *methods.* Collectively, the methods and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class.

Thus, it is the methods that determine how a class' data can be used. Variables defined within a class are called instance variables because each instance of the 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 come back 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 general form of a class does not specify a **main( )** method. Java classes do not need to have a **main( )** method. You only specify one if that class is the starting point for your program. Further, applets don't require a **main( )** method at all.

*C++ programmers will notice that the class declaration and the implementation of the methods are stored in the same place and not defined separately. This sometimes makes for very large **.java** files, since any class must be entirely defined in a single source file. This design feature was built into Java because it was felt that in the long run, having specification, declaration, and implementation all in one place makes for code that is easier to maintain.*

2.3 A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {
double width;
double height;
double depth;
}
```

As stated, a class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a **class** declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have "physical" reality. For the moment, don't worry about the details of this statement. Again, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every **Box** object will contain its own copies of the instance variables **width**, **height**, and **depth**. To access these variables, you will use the *dot* (.) operator.

The dot operator links the name of the object with the name of an instance variable. For example, to assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

This statement tells the compiler to assign the copy of **width** that is contained within the **mybox** object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.

Here is a complete 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**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively. To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

```
Volume is 3000.0
```

As stated earlier, each object has its own copies of the instance variables. This means that if you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is important to understand that changes to the instance variables of one object have no effect on the instance variables of another. For example, the following program declares two **Box** objects:

```
// This program declares two Box objects.
class Box {
double width;
double height;
double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
```


C h a p t e r 6 : I n t r o d u c i n g C l a s s e s 133

```
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;
```

```
// compute volume of first box
vol = mybox1.width * mybox1.height * mybox1.depth;
System.out.println("Volume is " + vol);
// compute volume of second box
vol = mybox2.width * mybox2.height * mybox2.depth;
System.out.println("Volume is " + vol);
}
}
```

The output produced by this program is shown here:

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

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

2.4 Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can *refer* to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator. The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**.

This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure. In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object. The effect of these two lines of code is depicted in Figure 6-1.

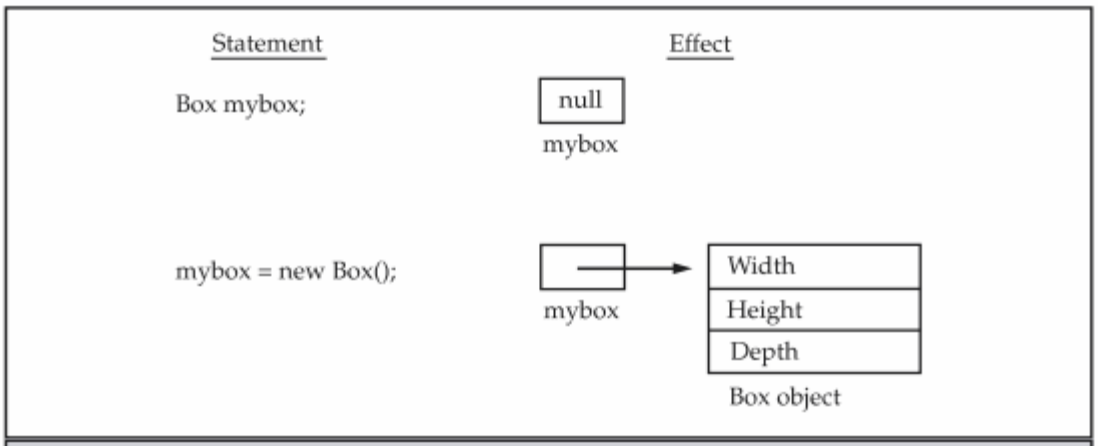


Figure 6-1. *Declaring an object of type* Box

2.5 A Closer Look at new

As just explained, the **new** operator dynamically allocates memory for an object. It has this general form:

class-var = new *classname*( );

Here, *class-var* is a variable of the class type being created. The *classname* is the name of the class that is being instantiated. The class name followed by parentheses specifies the *constructor* for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real-world classes explicitly define their own constructors within their class definition. However, if no explicit constructor is specified, then Java will automatically supply a default constructor. This is the case with **Box**. For now, we will use the default constructor. Soon, you will see how to define your own constructors. At this point, you might be wondering why you do not need to use **new** for such things as integers or characters. The answer is that Java's simple types are not implemented as objects. Rather, they are implemented as "normal" variables. This is done in the interest of efficiency. As you will see, objects have many features and attributes that require Java to treat themdifferently than it treats the simple types. By not applying the same overhead to the simple types that applies to objects, Java can implement the simple types more efficiently. Later, you will see object versions of the simple types that are available for your use in those situations in which complete objects of these types are needed. It is important to understand that **new** allocates memory for an object during run time.

The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that **new** will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur. (You will learn how to handle this and other exceptions in Chapter 10.) For the sample programs in this book, you won't need to worry about running out of memory, but you will need to consider this possibility in real-world programs that you write. Let's once again review the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.

2.6 Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object. This situation is depicted here:

Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.
When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.

2.7 Introducing Methods

As mentioned at the beginning of this chapter, classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes. 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 type must be **void**. The name of the method is specified by *name.* This can be any legal identifier other than those already used by other items within the current scope. 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 themethod when it is called. If the method has no parameters, then the parameter list will be empty. 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;
```

Here, *value* is the value returned.

In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

2.8 Adding a Method to the Box Class

Although it is perfectly fine to create a class that contains only data, it rarely happens.Most of the time you will use methods to access the instance variables defined by the class. In fact, methods define the interface to most classes. This allows the class implementor to hide the specific layout of internal data structures behind cleaner method abstractions. In addition to defining methods that provide access to data, you can

also define methods that are used internally by the class itself. Let's begin by adding a method to the **Box** class. It may have occurred to you while looking at the preceding programs that the computation of a box's volume was something that was best handled by the **Box** class rather than the **BoxDemo** class. After all, since the volume of a box is dependent upon the size of the box, it makes sense to have the **Box** class compute it. To do this, you must add a method to **Box**, as shown here:

```
// 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
```


Look closely at 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. If you are unfamiliar with the concept of calling a method, the following discussion will help clear things up.When **mybox1.volume( )** is executed, the Java run-time system transfers control to the code defined inside **volume( )**. After the statements inside **volume( )** have executed, control is returned to the calling routine, and execution resumes with the line of code following the call. In the most general sense, a method is Java's way of implementing subroutines. There is something very important to notice inside the **volume( )** method: the instance variables **width**, **height**, and **depth** are referred to directly, without preceding them with an object name or the dot operator.When a method uses an instance variable that is

defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A method is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within amethod, there is no need to specify the object a second time. This means that **width**, **height**, and **depth** inside **volume( )** implicitly refer to the copies of those variables found in the object that invokes **volume( )**.

Let's review: When an instance variable is accessed by code that is not part of the class in which that instance variable is defined, it must be done through an object, by use of the dot operator. However, when an instance variable is accessed by code that is part of the same class as the instance variable, that variable can be referred to directly. The same thing applies to methods.

2.9 Returning a Value

While the implementation of **volume( )** does move the computation of a box's volume inside the **Box** class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value? A better way to implement **volume( )** is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding 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 assignment statement. On the left is a variable, in this case **vol**, that will receive the value returned by **volume( )**. Thus, after
vol = 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:

n The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is

boolean, you could not return an integer. n The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method. One more point: The preceding program can be written a bit more efficiently because there is actually no need for the **vol** variable. The call to **volume( )** could have been used in 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 automatically and its value will be passed to **println( )**.

2.10 Adding a Method That Takes Parameters

While some methods don't need parameters, most do. Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
return 10 * 10;
}
```

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make **square( )** much more useful.

```
int square(int i)
{
return i * i;
}
```

Now, **square( )** will return the square of whatever value it is called with. That is, **square( )** is now a general-purpose method that can compute the square of any integer value, rather than just 10. Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
y = 2;
x = square(y); // x equals 4
```

In the first call to **square( )**, the value 5 will be passed into parameter **i**. In the second call, **i** will receive the value 9. The third invocation passes the value of **y**, which is 2 in this example. As these examples show, **square( )** is able to return the square of whatever data it is passed.

It is important to keep the two terms *parameter* and *argument* straight. A *parameter* is a variable defined by a method that receives a value when the method is called. For example, in **square( )**, **i** is a parameter. An *argument* is a value that is passed to a method when it is invoked. For example, **square(100)** passes 100 as an argument. Inside **square( )**, the parameter **i** receives that value.

You can use a parameterized method to improve the **Box** class. In the preceding examples, the dimensions of each box had to be set separately by use of a sequence

of statements, such as:
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;

While this code works, it is troubling for two reasons. First, it is clumsy and error prone. For example, it would be easy to forget to set a dimension. Second, in well-designed Java programs, instance variables should be accessed only through methods defined by their class. In the future, you can change the behavior of a method, but you can't change the behavior of an exposed instance variable.

Thus, a better approach to setting the dimensions of a box is to create a method that takes the dimension of a box in its parameters and sets each instance variable appropriately. This concept is implemented by the following program:

```
// This program uses a parameterized method.

class Box {
double width;
double height;
double depth;
// compute and return volume
double volume() {
return width * height * depth;
}
// sets dimensions of box
void setDim(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
}
class BoxDemo5 {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// initialize each box
mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 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, the **setDim( )** method is used to set the dimensions of each box. For example, when mybox1.setDim(10, 20, 15); is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively. For many readers, the concepts presented in the preceding sections will be familiar. However, if such things as method calls, arguments, and parameters are new to you, then you might want to

take some time to experiment before moving on. The concepts of the method invocation, parameters, and return values are fundamental to Java programming.

2.11 Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even when you add convenience functions like **setDim( )**, it would be simpler and more concise to have all of the setup done at the time the object is first created. Because the requirement for initialization is so common, Java allows objects to initialize themselves when they are created. This automatic initialization is performed through the use of a constructor.

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes. Constructors look a little strange because they have no return type, not even **void**. This is because the implicit return type of a class' constructor is the class type itself. It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately. You can rework the **Box** example so that the dimensions of a box are automatically initialized when an object is constructed. To do so, replace **setDim( )** with a constructor. Let's begin by defining a simple constructor that simply sets the dimensions of each box to the same values. This version is shown here:

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box {
double width;
double height;
C h a p t e r 6 : I n t r o d u c i n g C l a s s e s 145
V GUAGE

double depth;
// This is the constructor for Box.
Box() {
System.out.println("Constructing Box");
width = 10;
height = 10;
depth = 10;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;
// 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);
}
```

}

When this program is run, it generates the following results:
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0

As you can see, both **mybox1** and **mybox2** were initialized by the **Box( )** constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by 10 by 10, both **mybox1** and **mybox2** will have the same volume. The **println( )** statement inside **Box( )** is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object. Before moving on, let's reexamine the **new** operator. As you know, when you allocate an object, you use the following general form:

class-var = new *classname*( );

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

Box mybox1 = new Box();

new Box( ) is calling the **Box( )** constructor.When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding
line of code worked in earlier versions of **Box** that did not define a constructor. The default constructor automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

2.12 Parameterized Constructors

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct **Box** objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of **Box** defines a parameterized constructor which sets the dimensions of a box as specified by those parameters. Pay special attention to how **Box** objects are created.


```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
```

```
}
class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// 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);
}
}
```

The output from this program is shown here:
Volume is 3000.0
Volume is 162.0

As you can see, each object is initialized as specified in the parameters to its constructor. For example, in the following line,
Box mybox1 = new Box(10, 20, 15);

the values 10, 20, and 15 are passed to the **Box( )** constructor when **new** creates the object. Thus, **mybox1**'s copy of **width**, **height**, and **depth** will contain the values 10, 20, and 15, respectively.

2.13 The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside anymethod to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted. To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.

Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```


This version of **Box( )** operates exactly like the earlier version. The use of **this** is redundant, but perfectly correct. Inside **Box( )**, **this** will always refer to the invoking object. While it is redundant in this case, **this** is useful in other contexts, one of which is explained in the next section.

2.14 Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable. This is why **width**, **height**, and **depth** were not used as the names of the parameters to the **Box( )** constructor inside the **Box** class. If they had been, then **width** would have referred to the formal parameter, hiding the instance variable **width**.

While it is usually easier to simply use different names, there is another way around this situation. Because **this** lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables. For example, here is another version of **Box( )**, which uses **width**, **height**, and **depth** for parameter names and then uses **this** to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
}
```

A word of caution: The use of **this** in such a context can sometimes be confusing, and some programmers are careful not to use local variables and formal parameter names that hide instance variables. Of course, other programmers believe the contrary— that it is a good convention to use the same names for clarity, and use **this** to overcome the instance variable hiding. It is a matter of taste which approach you adopt. Although **this** is of no significant value in the examples just shown, it is very useful in certain situations.

2.15 Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection.* It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

2.16 The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called *finalization.* By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize( )** method on the object.

The **finalize( )** method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

Here, the keyword **protected** is a specifier that prevents access to **finalize( )** by code defined outside its class. This and the other access specifiers are explained in Chapter 7. It is important to understand that

finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—**finalize( )** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize( )** for normal program operation.

*If you are familiar with C++, then you know that C++ allows you to define a destructor for a class, which is called when an object goes out-of-scope. Java does not support this idea or provide for destructors. The **finalize( )** method only approximates the function of a destructor. As you get more experienced with Java, you will see that the need for destructor functions is minimal because of Java's garbage collection subsystem.*

2.17 A Stack Class

.While the **Box** class is useful to illustrate the essential elements of a class, it is of little practical value. To show the real power of classes, this chapter will conclude with a more sophisticated example. As you recall from the discussion of object-oriented programming (OOP) presented in Chapter 2, one of OOP's most important benefits is the encapsulation of data and the code that manipulates that data. As you have seen, the class is the mechanism by which encapsulation is achieved in Java. By creating a class, you are creating a new data type that defines both the nature of the data being manipulated and the routines used to manipulate it. Further, the methods define a consistent and controlled interface to the class' data. Thus, you can use the class through its methods without having to worry about the details of its implementation or how the data is actually managed within the class.

In a sense, a class is like a "data engine." No knowledge of what goes on inside the engine is required to use the engine through its controls. In fact, since the details are hidden, its inner workings can be changed as needed. As long as your code uses the class through its methods, internal details can change without causing side effects outside the class. To see a practical application of the preceding discussion, let's develop one of the archetypal examples of encapsulation: the stack. A *stack* stores data using first-in, last-out ordering. That is, a stack is like a stack of plates on a table—the first plate put down on the table is the last plate to be used. Stacks are controlled through two operations traditionally called *push* and *pop.* To put an item on top of the stack, you will use push. To take an item off the stack, you will use pop. As you will see, it is easy to encapsulate the entire stack mechanism. Here is a class called **Stack** that implements a stack for integers:

```
// This class defines an integer stack that can hold 10 values.
class Stack {
int stck[] = new int[10];
int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
```

```
else
return stck[tos--];
}
}
```

As you can see, the **Stack** class defines two data items and three methods. The stack of integers is held by the array **stck**. This array is indexed by the variable **tos**, which always contains the index of the top of the stack. The **Stack( )** constructor initializes **tos** to –1, which indicates an empty stack. The method **push( )** puts an item on the stack. To retrieve an item, call **pop( )**. Since access to the stack is through **push( )** and **pop( )**, the fact that the stack is held in an array is actually not relevant to using the stack. For example, the stack could be held in a more complicated data structure, such as a linked list, yet the interface defined by **push( )** and **pop( )** would remain the same. The class **TestStack**, shown here, demonstrates the **Stack** class. It creates two integer stacks, pushes some values onto each, and then pops them off.

```
class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();

// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
}
}
```

This program generates the following output:

```
Stack in mystack1:
9
8
7
6
5
4
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

As you can see, the contents of each stack are separate.

One last point about the **Stack** class. As it is currently implemented, it is possible for the array that holds the stack, **stck**, to be altered by code outside of the **Stack** class. This leaves **Stack** open to misuse or mischief. In the next chapter, you will see how to remedy this situation.

2.18 Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java implements polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.

class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for one integer parameter.
void test(int a) {
System.out.println("a: " + a);
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
double test(double a) {
156 J a v a ™ 2 : T h e C o m p l e t e R e f e r e n c e
System.out.println("double a: " + a);
return a*a;
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
double result;
// call all versions of test()
ob.test();
ob.test(10);
ob.test(10, 20);
result = ob.test(123.25);
System.out.println("Result of ob.test(123.25): " + result);
}
}
```


This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

As you can see, **test( )** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one **double** parameter. The fact that the fourth version of **test( )** also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution. When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```
// Automatic type conversions apply to overloading.
class OverloadDemo {
void test() {
System.out.println("No parameters");
}
// Overload test for two integer parameters.
void test(int a, int b) {
System.out.println("a and b: " + a + " " + b);
}
// overload test for a double parameter
void test(double a) {
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```


As you can see, this version of **OverloadDemo** does not define **test(int)**. Therefore, when **test( )** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**. Of course, if **test(int)** had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.

Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In

languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name.

For instance, in C, the function **abs( )** returns the absolute value of an integer, **labs( )** returns the absolute value of a long integer, and **fabs( )** returns the absolute value of a floating-point value. Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java's standard class library includes an absolute value method, called **abs( )**. This method is overloaded by Java's **Math** class to handle all numeric types. Java determines which version of **abs( )** to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name **abs** represents the *general action* which is being performed. It is left to the compiler to choose the right *specific* version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity.

When you overload a method, each version of that method can perform any activity you desire. There is no rule stating that overloaded methods must relate to one another. However, from a stylistic point of view, method overloading implies a relationship. Thus, while you can use the same name to overload unrelated methods, you should not. For example, you could use the name **sqr** to create methods that return the *square* of an integer and the *square root* of a floating-point value. But these two operations are fundamentally different. Applying method overloading in this manner defeats its original purpose. In practice, you should only overload closely related operations.

2.19 Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the **Box** class developed in the preceding chapter. Following is the latest version of **Box**:

```
class Box {
double width;
double height;
double depth;
// This is the constructor for Box.
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
```

As you can see, the **Box( )** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

Since **Box( )** requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know) what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the **Box** class is currently written, these other options are not available to you. Fortunately, the solution to these problems is quite easy: simply overload the **Box** constructor so that it handles the situations just described. Here is a program that contains an improved version of **Box** that does just that:

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
```

Volume of mybox2 is -1.0
Volume of mycube is 343.0
As you can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

2.20 Using Objects as Parameters

So far we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects tomethods. For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equals(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
```



```
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equals(ob2));
System.out.println("ob1 == ob3: " + ob1.equals(ob3));
}
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

As you can see, the **equals( )** method inside **Test** compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns **true**. Otherwise, it returns **false**. Notice that the parameter **o** in **equals**( ) specifies **Test** as its type. Although **Test** is a class type created by the program, it is used in just the same way as Java's built-in types.

One of the most common uses of object parameters involves constructors. Frequently you will want to construct a new object so that it is initially the same as some existing object. To do this, you must define a constructor that takes an object of its class as a parameter. For example, the following version of **Box** allows one object to initialize another:

```
// Here, Box allows one object to initialize another.

class Box {
double width;
double height;
double depth;
// construct clone of an object
```

```
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

As you will see when you begin to create your own classes, providing many forms of constructor methods is usually required to allow objects to be constructed in a convenient and efficient manner.

2.21 A Closer Look at Argument Passing

In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is *call-by-value*. This method copies the *value* of an argument into the formal parameter of the

subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument. The second way an argument can be passed is *call-by-reference.* In this method, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, Java uses both approaches, depending upon what is passed. In Java, when you pass a simple type to a method, it is passed by value. Thus, what occurs to the parameter that receives the argument has no effect outside the method.

For example, consider the following program:

```
// Simple types are passed by value.
class Test {
void meth(int i, int j) {
i *= 2;
j /= 2;
}
}

class CallByValue {
public static void main(String args[]) {
Test ob = new Test();
int a = 15, b = 20;
System.out.println("a and b before call: " +
a + " " + b);
ob.meth(a, b);
System.out.println("a and b after call: " +
a + " " + b);
}
}
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

As you can see, the operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10. When you pass an object to a method, the situation changes dramatically, because objects are passed by reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:


```
// Objects are passed by reference.
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// pass an object
void meth(Test o) {
o.a *= 2;
o.b /= 2;
}
}
class CallByRef {
```

```
public static void main(String args[]) {
Test ob = new Test(15, 20);
System.out.println("ob.a and ob.b before call: " +
ob.a + " " + ob.b);
ob.meth(ob);
System.out.println("ob.a and ob.b after call: " +
ob.a + " " + ob.b);
}
}
```

This program generates the following output:
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

As you can see, in this case, the actions inside **meth( )** have affected the object used as an argument. As a point of interest, when an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

When a simple type is passed to a method, it is done by use of call-by-value. Objects are passed by use of call-by-reference.

2.22 Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

```
// Returning an object.
class Test {
int a;
Test(int i) {
a = i;
}
Test incrByTen() {
Test temp = new Test(a+10);
return temp;
}
}
class RetOb {
public static void main(String args[]) {
Test ob1 = new Test(2);
Test ob2;
ob2 = ob1.incrByTen();
System.out.println("ob1.a: " + ob1.a);
System.out.println("ob2.a: " + ob2.a);
ob2 = ob2.incrByTen();
System.out.println("ob2.a after second increase: "
+ ob2.a);
}
}
```

The output generated by this program is shown here:
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22

As you can see, each time **incrByTen( )** is invoked, a new object is created, and a reference to it is returned to the calling routine. The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.

2.23 Recursion

Java supports *recursion.* Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive.* The classic example of recursion is the computation of the factorial of a number. The factorial of a number *N* is the product of all the whole numbers between 1 and *N.* For example, 3 factorial is 1 □2 □3, or 6. Here is how a factorial can be computed by use of a recursive method:

```
// A simple example of recursion.
class Factorial {
// this is a recursive function
int fact(int n) {
int result;
if(n==1) return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

The output from this program is shown here:

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```


If you are unfamiliar with recursive methods, then the operation of **fact( )** may seem a bit confusing. Here is how it works. When **fact( )** is called with an argument of 1, the function returns 1; otherwise it returns the product of **fact(n–1)*n**. To evaluate this expression, **fact( )** is called with **n–1**. This process repeats until **n** equals 1 and the calls to the method begin returning. To better understand how the **fact( )** method works, let's go through a short example. When you compute the factorial of 3, the first call to **fact( )** will cause a second call to be made with an argument of 2. This invocation will cause **fact( )** to be called a third time with an argument of 1. This call will return 1, which is then multiplied by 2 (the value of **n** in the second invocation). This result (which is 2) is then returned to the original invocation of **fact( )** and multiplied by 3 (the original value of **n**). This yields the answer, 6. You might find it interesting to insert **println( )** statements into **fact( )** which will show at what level each call is and what the intermediate answers are. When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start. A recursive call does not make a new copy of the method

Only the arguments are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method. Recursive

methods could be said to “telescope” out and back. Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional function calls. Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted. If this occurs, the Java run-time system will cause an exception. However, you probably will not have to worry about this unless a recursive routine runs wild. The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.

For example, the QuickSort sorting algorithm is quite difficult to implement in an iterative way. Some problems, especially AI-related ones, seem to lend themselves to recursive solutions. Finally, some people seem to think recursively more easily than iteratively. When writing recursive methods, you must have an **if** statement somewhere to force the method to return without the recursive call being executed. If you don’t do this, once you call the method, it will never return. This is a very common error in working with recursion. Use **println( )** statements liberally during development so that you can watch what is going on and abort execution if you see that you have made a mistake.

Here is one more example of recursion. The recursive method **printArray( )** prints the first **i** elements in the array **values**.

```
// Another example that uses recursion.
class RecTest {
int values[];
RecTest(int i) {
values = new int[i];
}
// display array -- recursively
void printArray(int i) {
if(i==0) return;
else printArray(i-1);
System.out.println("[" + (i-1) + "] " + values[i-1]);
}
}
class Recursion2 {
public static void main(String args[]) {
RecTest ob = new RecTest(10);
int i;
for(i=0; i<10; i++) ob.values[i] = i;
ob.printArray(10);
}
}
```


This program generates the following output:

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

2.24 Introducing Access Control

As you know, encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control.* Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a "black box" which may be used, but the inner workings of which are not open to tampering. However, the classes that were presented earlier do not completely meet this goal. For example, consider the **Stack** class shown at the end of Chapter 6. While it is true that the methods **push( )** and **pop( )** do provide a controlled interface to the stack, this interface is not enforced.

That is, it is possible for another part of the program to bypass these methods and access the stack directly. Of course, in the wrong hands, this could lead to trouble. In this section you will be introduced to the mechanism by which you can precisely control access to the various members of a class. How a member can be accessed is determined by the *access specifier* that modifies its declaration. Java supplies a rich set of access specifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.) These parts of Java's access control mechanism will be discussed later. Here, let's begin by examining access control as it applies to a single class. Once you understand the fundamentals of access control, the rest will be easy. Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level. **protected** applies only when inheritance is involved. The other access specifiers are described next. Let's begin by defining **public** and **private**. When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code.

When a member of a class is specified as **private**, then that member can only be accessed by other members of its class. Now you can understand why **main( )** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.) In the classes developed so far, all members of a class have used the default access mode, which is essentially public. However, this is not what you will typically want to be the case. Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods which are private to a class.

An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement. Here is an example:


```
public int i;
private double j;
private int myMethod(int a, char b) { // ...
```

To understand the effects of public and private access, consider the following program:

```
/* This program demonstrates the difference between
public and private.
*/
class Test {
int a; // default access
public int b; // public access
private int c; // private access
// methods to access c
void setc(int i) { // set c's value
c = i;
}
```

```
int getc() { // get c's value
return c;
}
}
class AccessTest {
public static void main(String args[]) {
Test ob = new Test();
// These are OK, a and b may be accessed directly
ob.a = 10;
ob.b = 20;
// This is not OK and will cause an error
// ob.c = 100; // Error!
// You must access c through its methods
ob.setc(100); // OK
System.out.println("a, b, and c: " + ob.a + " " +
ob.b + " " + ob.getc());
}
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**. If you were to remove the comment symbol from the beginning of the following line,

```
// ob.c = 100; // Error!
```

then you would not be able to compile this program because of the access violation. To see how access control can be applied to a more practical example, consider the following improved version of the **Stack** class shown at the end of Chapter 6.

```
// This class defines an integer stack that can hold 10 values.

class Stack {
/* Now, both stck and tos are private. This means
that they cannot be accidentally or maliciously
altered in a way that would be harmful to the stack.
*/
private int stck[] = new int[10];
private int tos;
// Initialize top-of-stack
Stack() {
tos = -1;
}
// Push an item onto the stack
void push(int item) {
if(tos==9)
System.out.println("Stack is full.");
else
stck[++tos] = item;
}

// Pop an item from the stack
int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
```

```
return 0;
}
else
return stck[tos--];
}
}
```

As you can see, now both **stck**, which holds the stack, and **tos**, which is the index of the top of the stack, are specified as **private**. This means that they cannot be accessed or altered except through **push( )** and **pop( )**. Making **tos** private, for example, prevents other parts of your program from inadvertently setting it to a value that is beyond the end of the **stck** array. The following program demonstrates the improved **Stack** class. Try removing the commented-out lines to prove to yourself that the **stck** and **tos** members are, indeed, inaccessible.

```
class TestStack {
public static void main(String args[]) {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();
// push some numbers onto the stack
for(int i=0; i<10; i++) mystack1.push(i);
for(int i=10; i<20; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
// these statements are not legal
// mystack1.tos = -2;
// mystack2.stck[3] = 100;
}
}
```

Although methods will usually provide access to the data defined by a class, this does not always have to be the case. It is perfectly proper to allow an instance variable to be public when there is good reason to do so. For example, most of the simple classes in this book were created with little concern about controlling access to instance variables for the sake of simplicity. However, in most real-world classes, you will need to allow operations on data only through methods. The next chapter will return to the topic of access control. As you will see, it is particularly important when inheritance is involved.

2.25 Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static**. When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object. You can declare both methods and variables to be **static**. The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist. Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable. Methods declared as **static** have several restrictions:

n They can only call other **static** methods.
n They must only access **static** data.
n They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance and is described in the next chapter.) If you need to do computation in order to initialize your **static** variables, you can declare a **static** block which gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block:

```
class UseStatic {
static int a = 3;
static int b;
static void meth(int x) {
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}
static {
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[]) {
meth(42);
}
}
```

As soon as the **UseStatic** class is loaded, all of the **static** statements are run. First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a * 4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

*It is illegal to refer to any instance variables inside of a **static** method.*
Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator. For example, if you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

Here, *classname* is the name of the class in which the **static** method is declared. As you can see, this format is similar to that used to call non-**static** methods through object- reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables. Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed outside of their class.

```
class StaticDemo {
static int a = 42;
static int b = 99;
static void callme() {
System.out.println("a = " + a);
}
}
```

```
class StaticByName {
public static void main(String args[]) {
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```
Here is the output of this program:
```
a = 42
b = 99
```

2.26 Introducing final

A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared. (In this usage, **final** is similar to **const** in C/C++/C#.) For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use **FILE_OPEN**, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for **final** variables. Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant. The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of **final** is described in the next chapter, when inheritance is described.

2.27 Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as *nested classes*. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B is known to A, but not outside of A. A nested class has access to themembers, including privatemembers, of the class in which it is nested. However, the enclosing class does not have access to themembers of the nested class. There are two types of nested classes: *static* and *non-static*. A static nested class is one which has the **static** modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used. The most important type of nested class is the *inner* class. An inner class is a non-static nested class.

It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class. The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test( )**, and defines one inner class called **Inner**.

```
// Demonstrate an inner class.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
```

```
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Output from this application is shown here:
display: outer_x = 100

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display( )** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called. It is important to realize that class **Inner** is known only within the scope of class **Outer**. The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**. Generalizing, a nested class is no different than any other program element: it is known only within its enclosing scope. As explained, an inner class has access to all of the members of its enclosing class, but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example,

```
// This program will not compile.
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
int y = 10; // y is local to Inner
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
void showy() {
System.out.println(y); // error, y not known here!
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Here, **y** is declared as an instance variable of **Inner**. Thus it is not known outside of that class and it cannot be used by **showy( )**. Although we have been focusing on nested classes declared within an outer class scope, it is possible to define inner classes within any block scope. For example, you can define a nested class within the block defined by a method or even within the body of a **for** loop, as this next program shows.

```
// Define an inner class within a for loop.

class Outer {
int outer_x = 100;
void test() {
for(int i=0; i<10; i++) {
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
Inner inner = new Inner();
inner.display();
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

The output from this version of the program is shown here.

```
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
display: outer_x = 100
```

While nested classes are not used in most day-to-day programming, they are particularly helpful when handling events in an applet. We will return to the topic of nested classes in Chapter 20. There you will see how inner classes can be used to simplify the code needed to handle certain types of events. You will also learn about *anonymous inner classes*, which are inner classes that don't have a name. One final point: Nested classes were not allowed by the original 1.0 specification for Java. They were added by Java 1.1.

2.28 Exploring the String Class

Although the **String** class will be examined in depth in Part II of this book, a short exploration of it is warranted now, because we will be using strings in some of the example programs shown toward the end of Part I. **String** is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming. The first thing to understand about strings is that every string you create is actually an object of type **String**. Even string constants are actually **String** objects. For example, in the statement

```
System.out.println("This is a String, too");
```

the string "This is a String, too" is a **String** constant. Fortunately, Java handles **String** constants in the same way that other computer languages handle "normal" strings, so you don't have to worry about this. The second thing to understand about strings is that objects of type **String** are immutable; once a **String**

object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not, for two reasons: n If you need to change a string, you can always create a new one that contains the modifications. n Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java. (**StringBuffer** is described in Part II of this book.) Strings can be constructed a variety of ways. The easiest is to use a statement like this:

String myString = "this is a test";

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString**: System.out.println(myString); Java defines one operator for **String** objects: **+**. It is used to concatenate two strings. For example, this statement

String myString = "I" + " like " + "Java.";

results in **myString** containing "I like Java."

The following program demonstrates the preceding concepts:

```
// Demonstrating Strings.
class StringDemo {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1 + " and " + strOb2;
System.out.println(strOb1);
System.out.println(strOb2);
System.out.println(strOb3);
}
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

The **String** class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals( )**. You can obtain the length of a string by calling the **length( )** method. You can obtain the character at a specified index within a string by calling **charAt( )**. The general forms of these three methods are shown here:


```
boolean equals(String object)
int length( )
char charAt(int index)
```

Here is a program that demonstrates these methods:

```
// Demonstrating some String methods.
class StringDemo2 {
public static void main(String args[]) {
String strOb1 = "First String";
String strOb2 = "Second String";
String strOb3 = strOb1;
System.out.println("Length of strOb1: " +
strOb1.length());
System.out.println("Char at index 3 in strOb1: " +
strOb1.charAt(3));
if(strOb1.equals(strOb2))
System.out.println("strOb1 == strOb2");
else
```

```
System.out.println("strOb1 != strOb2");
if(strOb1.equals(strOb3))
System.out.println("strOb1 == strOb3");
else
System.out.println("strOb1 != strOb3");
}
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
\
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
class StringDemo3 {
public static void main(String args[]) {
String str[] = { "one", "two", "three" };
for(int i=0; i<str.length; i++)
System.out.println("str[" + i + "]: " +
str[i]);
}
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

As you will see in the following section, string arrays play an important part in many Java programs.

2.29 Using Command-Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing *command-line arguments* to **main( )**. A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in the **String** array passed to **main( )**.

For example, the following program displays all of the command-line arguments that it is called with:

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
for(int i=0; i<args.length; i++)
System.out.println("args[" + i + "]: " +
args[i]);
}
}
```

Try executing this program, as shown here:

```
java CommandLine this is a test 100 -1
```

When you do, you will see the following output:

```
args[0]: this
```

```
args[1]: is
args[2]: a
args[3]: test
args[4]: 100
args[5]: -1
```

All command-line arguments are passed as strings. You must convert numeric values to their internal forms manually.

3.
Exception-Handling

An *exception* is an abnormal 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 support exception handling, errors must be checked and handled manually—typically through the 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-time error management into the object-oriented world. For the most part, exception handling has not changed since the original version of Java. However, Java 2, version 1.4 has added a new subsystem called the *chained exception facility*. This feature is described near the end of this chapter.

3.1 Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing 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, at some point, the exception is *caught* and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are 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 to monitor for exceptions are contained within a **try** block. If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner. System-generated exceptions are automatically thrown by the 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 **throws** clause.
Any code that absolutely must be executed before a method returns is put in a **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.

3.2 Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.

There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing. The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type **Error**, because these are typically created in response 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 what happens when you don't handle them. This small program includes an expression that intentionally 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 a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: / by zero at 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 the type of the exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated. The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the 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 entire call 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 to **subroutine( )**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

3.4 Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this. To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a **try** block. Immediately following the **try** block, include a **catch** clause that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a **try** block and a **catch** clause which processes 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 an exception 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 from a **catch**. Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested **try** statements, described shortly). The statements that are protected by **try** must be surrounded by curly braces. (That is, they must be within a block.) You cannot use **try** on a single statement. The goal of most well-constructed **catch** clauses should be to resolve the exceptional 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 random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

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

3.5 Displaying a Description of an Exception

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** 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, each divide-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 description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

3.6 Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block. The following example traps two different exception 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 programwill cause a division-by-zero exception if it is started with no commandline parameters, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an**ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program 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 that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its 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 following program:

```
/* 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 that the second **catch** statement is unreachable because the exception has already been caught. Since **ArithmeticException** is a subclass of **Exception**, the first **catch** statement will handle all **Exception**-based errors, including **ArithmeticException**. This means that the second **catch** statement will never execute. To fix the problem, reverse the order of the **catch** statements.

3.7 Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**. Each time a **try** statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **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 program works as follows. When you execute the program with no command-line arguments, a divide-by-zero exception is generated by the outer **try** block. Execution of the program by one command-line argument generates a divide-by-zero exception from within the nested **try** block. Since the inner block does not catch this exception, it is passed on to the outer **try** block, where it is handled. If you execute the program with two command-line arguments, an array boundary exception is generated from within the 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 are involved. For example, you can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method. Here is the previous program recoded 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.

3.8 throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. 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 of **Throwable**. Simple types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions. There are two ways you can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator. The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace. Here is a sample program that creates and throws an exception. The handler that catches 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 an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception-handling context and immediately throws a new instance of **NullPointerException,** which is caught on the next line. The exception is then rethrown. 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's built-in run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**. It can also be obtained by a call to **getMessage( )**, which is defined by **Throwable**.

3.9 throws

If a method is capable of causing an exception that it does not handle, it must specify this 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** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, 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. If they 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 exception that it does not catch. Because the program does not specify a **throws** clause to declare this 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 to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **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

3.10 finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency. **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown.

If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement 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 an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block 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

3.11 Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since **java.lang** is implicitly imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available. Furthermore, they need not be included in any method's **throws** list. In the language of Java, these are called *unchecked exceptions* because the Compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table 10-1. Table 10-2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.* Java defines several other types of exceptions that relate to its various class libraries.

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 a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.

Table 10-1. *Java's Unchecked* RuntimeException *Subclasses*

Exception	Meaning
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1. *Java's Unchecked* RuntimeException *Subclasses* (continued)

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Table 10-2. *Java's Checked Exceptions Defined in* java.lang

3.11 Creating Your Own Exception Subclasses

Although Java's built-in exceptions handle most common errors, you will probably want to 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 subclass of **Throwable**). Your subclasses don't need to actually implement anything—it is their existence 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 those that you create, have the methods defined by **Throwable** available to them. They are shown in Table 10-3. 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 that you create.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. Added by Java 2, version 1.4.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name. Added by Java 2, version 1.4
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name. Added by Java 2, version 1.4
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception. Added by Java 2, version 1.4

Table 10-3. *The Methods Defined by* Throwable

Method	Description
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 passed in *elements*. This method is for specialized applications, not normal use. Added by Java 2, version 1.4
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

Table 10-3. *The Methods Defined by* Throwable *(continued)*

The following example declares a new subclass of **Exception** and then uses that subclass 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 subclass is quite simple: it has only a constructor plus an overloaded **toString( )** method that displays the value of the exception. The

ExceptionDemo class defines a method named **compute( )** that throws a **MyException** object. The exception is thrown when **compute( )**'s integer parameter is greater than 10. The **main( )** method sets up an exception handler for **MyException**, then calls **compute( )** with a legal value (less than 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]

3.12 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 a situation in which a method throws an **ArithmeticException** because of an attempt to divide 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 themethodmust certainly throw an **ArithmeticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let 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 two methods 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 allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

The chained exceptionmethods added to **Throwable** are **getCause( )** and **initCause( )**. Thesemethods are shown in Table 10-3, 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, you can associate a causewith an exception after the exception has been created.However, the cause exception can be set only once. Thus, you can call **initCause( )** only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause( )**. In general, **initCause( )** is used to set a cause for legacy exception classes which don't support the two additional constructors described earlier. At the time of this writing, most of Java's built-in exceptions, such as **ArithmeticException**, do not define the additional constructors. Thus, you will use **initCause( )** if you need to add an exception chain to these exceptions. When creating your own exception classes you will want to add the two chained-exception constructors if you will be using your exceptions 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 added a cause exception, **ArithmeticException**. When the exception is thrown out of **demoproc( )**, 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 necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design. Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

3.13 Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that 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 your program's logic. If you are like most programmers, then you probably are used to returning an error code when a method fails. When you are programming in Java, you should break this habit. When a method can fail, have it throw an exception. This is a cleaner way to handle failure modes. One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

4.
IO Package

This chapter examines two of Java's most innovative features: packages and interfaces. *Packages* are containers 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 some other class named **List** stored elsewhere.

Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.In previous chapters you have seen how methods define the interface to the data in a class. Through the use of the **interface** keyword, Java allows you to fully abstract the interface from its implementation. Using **interface**, you can specify a set of methods which can be implemented by one or more classes. The **interface**, itself, does not actually define any implementation. Although they are similar to abstract classes, **interface**s have an additional capability:

A class can implement more than one interface. By contrast, a class can only inherit a single superclass (abstract or otherwise). Packages and interfaces are two of the basic components of a Java program. In general, a Java source file can contain any (or all) of the following four internal parts: n A single package statement (optional) n Any number of import statements (optional) n A single public class declaration (required) n Any number of classes private to the package (optional) Only one of these—the single public class declaration—has been used in the examples so far. This chapter will explore the remaining parts.

4.1 Packages

In the preceding chapters, the name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name "Foobar" as a class name. Or, imagine the entire Internet community arguing over who first named a class "Espresso.") Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

4.2 Defining a Package

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 the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. 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 a package called **MyPackage**.

package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files frombeing part of that same package.Most real-world packages are spread acrossmany files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

package *pkg1*[.*pkg2*[.*pkg3*]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package java.awt.image;

needs to be stored in **java/awt/image**, **java\awt\image**, or **java:awt:image** on your UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

4.3 Finding Packages and CLASSPATH

As just explained, packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has two parts. First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found. Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable. For example, consider the following package specification.

package MyPack;

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include the path to **MyPack**. The first alternative is the easiest (and doesn't require a change to **CLASSPATH**), but the second alternative lets your program find **MyPack** no matter what directory the program is in. Ultimately, the choice is yours. The easiest way to try the examples shown in this book is to simply create the package directories below your current development directory, put the **.class** files into the appropriate directories and then execute the programs from the development directory. This is the approach assumed by the examples.

4.4 A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(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();
}
}
```

Call this file **AccountBalance.java**, and put it in a directory called **MyPack**. Next, compile the file.Make sure that the resulting **.class** file is also in the**MyPack** directory. Then try executing the **AccountBalance** class, using the following command line: java MyPack.AccountBalance Remember, you will need to be in the directory above **MyPack** when you execute this command, or to have your **CLASSPATH** environmental variable set appropriately. As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line: java AccountBalance **AccountBalance** must be qualified with its package name.

4.5 Access Protection

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access specifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members: n Subclasses in the same package n Non-subclasses in the same package n Subclasses in different packages n Classes that are neither in the same package nor subclasses The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Table 9-1 sums up the interactions.

While Java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

Table 9-1 applies only to members of classes. A class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

	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

Table 9-1. *Class Member Access*

4.6 An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**. The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**. Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out by use of the single-line comment **//**. Before each of these lines is a comment listing the places from which this level of protection would allow access. The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.
This is file **Protection.java**:

```
package p1;
public class Protection {
int n = 1;
private int n_pri = 2;
protected int n_pro = 3;
public int n_pub = 4;
public Protection() {
System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **Derived.java**:

```
package p1;
class Derived extends Protection {
Derived() {
System.out.println("derived constructor");
System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **SamePackage.java**:

```
package p1;
class SamePackage {
SamePackage() {
Protection p = new Protection();
System.out.println("same package constructor");
System.out.println("n = " + p.n);
// class only
// System.out.println("n_pri = " + p.n_pri);
System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

Following is the source code for the other package, **p2**. The two classes defined in **p2** cover the other two conditions which are affected by access control. The first class, **Protection2**, is a subclass of **p1.Protection**. This grants access to all of **p1.Protection**'s variables except for **n_pri** (because it is **private**) and **n**, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class **OtherPackage** has access to only one variable, **n_pub**, which was declared **public**.
This is file **Protection2.java**:

```
package p2;
class Protection2 extends p1.Protection {
Protection2() {
System.out.println("derived other package constructor");
// class or package only
// System.out.println("n = " + n);
// class only
// System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
}
}
```

This is file **OtherPackage.java**:

```
package p2;
class OtherPackage {
OtherPackage() {
p1.Protection p = new p1.Protection();
System.out.println("other package constructor");
// class or package only
// System.out.println("n = " + p.n);
// class only
```

```
// System.out.println("n_pri = " + p.n_pri);
// class, subclass or package only
// System.out.println("n_pro = " + p.n_pro);
System.out.println("n_pub = " + p.n_pub);
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo {
public static void main(String args[]) {
Protection ob1 = new Protection();
Derived ob2 = new Derived();
SamePackage ob3 = new SamePackage();
}
}
```

The test file for **p2** is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo {
public static void main(String args[]) {
Protection2 ob1 = new Protection2();
OtherPackage ob2 = new OtherPackage();
}
}
```

4.7 Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing. In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:

import *pkg1*[.*pkg2*].(*classname*|*);

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use: import java.util.Date; import java.io.*; *The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.* All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called

java.lang. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs: import java.lang.*; If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.
Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy. For example, this fragment uses an import statement:

```
 import java.util.*;
class MyDate extends Date {
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

As shown in Table 9-1, when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package MyPack;

/* Now, the Balance class, its constructor, and its
show() method are public. This means that they can
be used by non-subclass code outside their package.
*/
public class Balance {
String name;
double bal;
public Balance(String n, double b) {
name = n;
bal = b;
}
public void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
```


As you can see, the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;
class TestBalance {
public static void main(String args[]) {
/* Because Balance is public, you may use Balance
class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

As an experiment, remove the **public** specifier from the **Balance** class and then try compiling **TestBalance**. As explained, errors will result.

4.8 Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces. To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. Interfaces are designed to support dynamic method resolution at run time.Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized. *Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.*

4.9 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, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by 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 semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods. Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public**. Here is an example of an interface definition. It declares a simple interface which contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback {
void callback(int param);
}
```

4.10 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 then create the methods defined by the interface. The general form of a class that includes the **implements** 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 that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition. Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that **callback( )** is declared using the **public** access specifier. *When you implement an interface method, it must be declared as* ***public***. It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback( )** and adds the method

nonIfaceMeth( ):

```
class Client implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("callback called with " + p);
}
void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```


Accessing Implementations Through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee." This process is similar to using a superclass reference to access a subclass object

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the **callback( )** method via an interface reference variable:

```
class TestIface {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
}
```

The output of this program is shown here:
callback called with 42

Notice that variable **c** is declared to be of the interface type **Callback**, yet it was assigned an instance of **Client**. Although **c** can be used to access the **callback( )** method, it cannot access any other members of the **Client** class. An interface reference variable only has knowledge of the methods declared by its **interface** declaration. Thus, **c** could not be used to access **nonIfaceMeth( )** since it is defined by **Client** but not **Callback**. While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementationof **Callback**, shown here:

```
// Another implementation of Callback.
class AnotherClient implements Callback {
// Implement Callback's interface
public void callback(int p) {
System.out.println("Another version of callback");
System.out.println("p squared is " + (p*p));
}
}
```

Now, try the following class:

```
class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

The output from this program is shown here:
callback called with 42
Another version of callback
p squared is 1764

As you can see, the version of **callback( )** that is called is determined by the type of object that **c** refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

Partial Implementations

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as **abstract**. For example:

```
abstract class Incomplete implements Callback {
int a, b;
void show() {
System.out.println(a + " " + b);
}
// ...
}
```

Here, the class **Incomplete** does not implement **callback( )** and must be declared as abstract. Any class that inherits **Incomplete** must implement **callback( )** or be declared **abstract** itself.

4.11 Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters you developed a class called **Stack** that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples. First, here is the interface that defines an integer stack. Put this in a file called **IntStack.java**. This interface will be used by both stack implementations.

```
// Define an integer stack interface.
interface IntStack {
void push(int item); // store an item
int pop(); // retrieve an item
}
```

The following program creates a class called **FixedStack** that implements a fixed-length version of an integer stack:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
FixedStack(int size) {
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item) {
if(tos==stck.length-1) // use length member
System.out.println("Stack is full.");
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
```

```
return stck[tos--];
}
}

class IFTest {
public static void main(String args[]) {
FixedStack mystack1 = new FixedStack(5);
FixedStack mystack2 = new FixedStack(8);
// push some numbers onto the stack
for(int i=0; i<5; i++) mystack1.push(i);
for(int i=0; i<8; i++) mystack2.push(i);
// pop those numbers off the stack
System.out.println("Stack in mystack1:");
for(int i=0; i<5; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<8; i++)
System.out.println(mystack2.pop());
}
}
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
// Implement a "growable" stack.
class DynStack implements IntStack {
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size) {
stck = new int[size];
tos = -1;
}

// Push an item onto the stack
public void push(int item) {
// if stack is full, allocate a larger stack
if(tos==stck.length-1) {
int temp[] = new int[stck.length * 2]; // double size
for(int i=0; i<stck.length; i++) temp[i] = stck[i];

stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop() {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
```

```
return stck[tos--];
}
}
class IFTest2 {
public static void main(String args[]) {
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++) mystack1.push(i);
for(int i=0; i<20; i++) mystack2.push(i);
System.out.println("Stack in mystack1:");
for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
}
}
```

The following class uses both the **FixedStack** and **DynStack** implementations. It does so through an interface reference. This means that calls to **push( )** and **pop( )** are resolved at run time rather than at compile time.

```
/* Create an interface variable and access stacks through it.*/

class IFTest3 {
public static void main(String args[]) {
IntStack mystack; // create an interface reference variable
DynStack ds = new DynStack(5);
FixedStack fs = new FixedStack(8);
mystack = ds; // load dynamic stack
// push some numbers onto the stack
for(int i=0; i<12; i++) mystack.push(i);
mystack = fs; // load fixed stack
for(int i=0; i<8; i++) mystack.push(i);
mystack = ds;
System.out.println("Values in dynamic stack:");
for(int i=0; i<12; i++)
System.out.println(mystack.pop());
mystack = fs;
System.out.println("Values in fixed stack:");
for(int i=0; i<8; i++)
System.out.println(mystack.pop());
}
}
```

In this program, **mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push( )** and **pop( )** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push( )** and **pop( )** defined by **FixedStack**. As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

4.12 Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations. If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant variables into the class name space as **final** variables. The next example uses this technique to implement an automated "decision maker":

```
import java.util.Random;
interface SharedConstants {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants {
Random rand = new Random();
int ask() {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMe implements SharedConstants {
static void answer(int result) {
switch(result) {
case NO:
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
```

```
System.out.println("Never");
break;
}
}
public static void main(String args[]) {
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Notice that this program makes use of one of Java's standard classes: **Random**. This class provides pseudorandom numbers. It contains several methods which allow you to obtain random numbers in the form required by your program. In this example, the method **nextDouble( )** is used. It returns random numbers in the range 0.0 to 1.0. In this sample program, the two classes, **Question** and **AskMe**, both implement the **SharedConstants** interface where **NO**, **YES**, **MAYBE**, **SOON**, **LATER**, and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Later
Soon
No
Yes
```

4.13 Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A {
void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2() {
System.out.println("Implement meth2().");
}
public void meth3() {
System.out.println("Implement meth3().");
}
}
class IFExtend {
public static void main(String arg[]) {
```

```
MyClass ob = new MyClass();
ob.meth1();
ob.meth2();
ob.meth3();
}
}
```

As an experiment you might want to try removing the implementation for **meth1( )** in **MyClass**. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces. Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs and applets that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

5.
Multi threading

Unlike most other computer languages, 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. Thus, multithreading is a specialized form of multitasking. You are almost certainly acquainted with multitasking, because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form.

A *process* is, in essence, a program that is executing. Thus, *process-based* multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-basedmultitasking, a program is the smallest unit of code that can be dispatched by the scheduler. In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads. Thus, process-based multitasking deals with the "big picture," and thread-based multitasking handles the details. Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.

Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer.

In a traditional, single-threaded environment, your program has to wait for each of these tasks to finish before it can proceed to the next one—even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use. If you have programmed for operating systems such asWindows 98 orWindows 2000, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient, because many of the details are handled for you.

5.1 The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles. The value of a multithreaded environment is best understood in contrast to its counterpart. Single-threaded systems use an approach called an *event loop* with *polling*. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being

processed. In general, in a singled-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running. The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program.

.For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep or a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run. Threads exist in several states. A thread can be *running.* It can be *ready to run* as soon as it gets CPU time. A running thread can be *suspended,* which temporarily suspends its activity. A suspended thread can then be *resumed,* allowing it to pick up where it left off. A thread can be *blocked* when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

5.2 Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch.* The rules that determine when a context switch takes place are simple: n *A thread can voluntarily relinquish control.* This is done by explicitly yielding, sleeping, or blocking on pending I/O.

In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU. n *A thread can be preempted by a higher-priority thread.* In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing—by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called *preemptive multitasking.* In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run. *Problems can arise from the differences in the way that operating systems context-switch threads of equal priority.*

5.3 Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronization: the *monitor.* The monitor is a control mechanism first defined by C.A.R. Hoare. You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time. Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicitmonitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built in to the language.

5.4 Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.

5.5 The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface. The **Thread** class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

Method	**Meaning**
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and **Runnable** to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

5.6 The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

n It is the thread from which other "child" threads will be spawned.

n Often it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**. Its general form is shown here:


```
static Thread currentThread( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread. Let's begin by reviewing the following example:

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
```

```
System.out.println("After name change: " + t);
try {
for(int n = 5; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted");
}
}
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName( )** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep( )** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one.

This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Notice the output produced when **t** is used as an argument to **println( )**. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. A *thread group* is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment and is not discussed in detail here. After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed. Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here: static void sleep(long *milliseconds*) throws InterruptedException The number of milliseconds to suspend is specified in *milliseconds.* This method may throw an **InterruptedException**. The **sleep( )** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds: static void sleep(long *milliseconds,* int *nanoseconds*) throws InterruptedException This second form is useful only in environments that allow timing periods as short as nanoseconds. As the preceding program shows, you can set the name of a thread by using **setName( )**. You can obtain the name of a thread by calling **getName( )** (but note that this procedure is not shown in the program). These methods are members of the **Thread** class and are declared like this:

final void setName(String *threadName*)
final String getName( )
Here, *threadName* specifies the name of the thread.

5.7 Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished: n You can implement the **Runnable** interface. n You can extend the **Thread** class, itself. The following two sections look at each method, in turn.

5.8 Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this: public void run( ) Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns. After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread(Runnable *threadOb*, String *threadName*)

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*. After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:

void start( )

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.");
}
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run( )** method on **this** object. Next, **start( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's **for** loop to begin. After calling **start( )**, **NewThread**'s constructor returns to **main( )**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run-time system may "hang." The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

5.9 Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread**:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
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 ExtendThread {
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.");
}
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**. Notice the call to **super( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:

public Thread(String *threadName*)
Here, *threadName* specifies the name of the thread.

5.10 Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The **Thread** class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is **run( )**. This is, of course, the same method required when you implement **Runnable**. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**. This is up to you, of course. However, throughout the rest of this chapter, we will create threads by using classes that implement **Runnable**.

5.11 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, the following 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.

.As you can see, once started, all three child threads share the CPU. Notice the call to **sleep(10000)** in **main( )**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

5.12 Using isAlive( ) and join( )

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question. Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate. Here is an improved version of the preceding example that uses **join( )** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive( )** method.

```
// Using join() to wait for threads to finish.
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 DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.t.isAlive());
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "
+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "
+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "
+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Sample 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]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
```


288 J a v a ™ 2 : T h e C o m p l e t e R e f e r e n c e

```
Two exiting.
Three exiting.
```

One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.

As you can see, after the calls to **join( )** return, the threads have stopped executing.

5.13 Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower- priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating systemimplements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread. In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally, so that other threads can run. To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**. This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **final** variables within **Thread**.You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

final int getPriority( )

Implementations of Java may have radically different behavior when it comes to scheduling. TheWindows XP/98/NT/2000 version works, more or less, as you would expect. However, other versions may work quite differently.Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU. The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a nonpreemptive platform. One thread is set two levels above the normal priority, as defined by **Thread.NORM_PRIORITY**, and the other is set to two levels below it. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

```
// Demonstrate thread priorities.
class clicker implements Runnable {
int click = 0;
Thread t;
```

```
private volatile boolean running = true;
public clicker(int p) {
t = new Thread(this);
t.setPriority(p);
}
public void run() {
while (running) {
click++;
}
}
public void stop() {
running = false;
}
public void start() {
t.start();
}
}class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
hi.t.join();
lo.t.join();
} catch (InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Low-priority thread: " + lo.click);
System.out.println("High-priority thread: " + hi.click);
}
}
```


The output of this program, shown as follows when run under Windows 98, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got approximately 90 percent of the CPU time. Low-priority thread: 4408112 High-priority thread: 589626904 Of course, the exact output produced by this program depends on the speed of your CPU and the number of other tasks running in the system. When this same program is run under a nonpreemptive system, different results will be obtained. One other note about the preceding program. Notice that **running** is preceded by the keyword **volatile**. Although **volatile** is examined more carefully in the next chapter, it is used here to ensure that the value of **running** is examined each time the following loop iterates:

```
while (running) {
click++;
}
```

Without the use of **volatile**, Java is free to optimize the loop in such a way that a local copy of **running** is created. The use of **volatile** prevents this optimization, telling Java that **running** may change in ways not directly apparent in the immediate code.

5.14 Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization.* As you will see, Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. If you have worked with synchronization when using other languages, such as C or C++, you know that it can be a bit tricky to use. This is because most languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives. Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated. You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

5.15 Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method. To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call( )**. The **call( )** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call( )** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second. The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run( )** method. The thread is started immediately. The **run( )** method of **Caller** calls the **call( )** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.


```
// This program is not synchronized.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
```

```
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run() {
target.call(msg);
}
}
class Synch {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
}
}
```

Here is the output produced by this program:

```
Hello[Synchronized[World]
]
]
```

As you can see, by calling **sleep( )**, the **call( )** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition,* because the three threads are racing each other to complete the method. This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next. To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
synchronized void call(String msg) {
...
```

This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any

other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

5.16 The synchronized Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block. This is the general form of the **synchronized** statement:

```
synchronized(object) {
// statements to be synchronized
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor. Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method:

```
// This program uses a synchronized block.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
}
}
}
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
```

```
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
296 J a v a ™ 2 : T h e C o m p l e t e R e f e r e n c e
System.out.println("Interrupted");
}
}
}
```

Here, the **call( )** method is not modified by **synchronized**. Instead, the **synchronized** statement is used inside **Caller**'s **run( )** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

5.17 Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. As you will see, this is especially easy in Java. As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete and logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable. To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

n **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**. n **notify( )** wakes up the first thread that called **wait( )** on the same object. n **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first.

These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Additional forms of **wait( )** exist that allow you to specify a period of time to wait. The following sample program incorrectly implements a simple form of the producer/consumer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC**, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

```
// An incorrect implementation of a producer and consumer.
```

```
class Q {
int n;
synchronized int get() {
System.out.println("Got: " + n);
return n;
}
synchronized void put(int n) {
this.n = n;
System.out.println("Put: " + n);
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PC {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```


Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
```

```
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them. The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions, as shown here:

```
// A correct implementation of a producer and consumer.
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
if(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
```

```
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PCFixed {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
302 J a v a ™ 2 : T h e C o m p l e t e R e f e r e n c e
System.out.println("Press Control-C to stop.");
}
}
```

Inside **get( )**, **wait( )** is called. This causes its execution to suspend until the **Producer** notifies you that some data is ready. When this happens, execution inside **get( )** resumes. After the data has been obtained, **get( )** calls **notify( )**. This tells **Producer** that it is okay to put more data in the queue. Inside **put( )**, **wait( )** suspends execution until the **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called. This tells the **Consumer** that it should now remove it. Here is some output fromthis program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```


5.18 Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is *deadlock,* which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

n In general, it occurs only rarely, when the two threads time-slice in just the right way. n It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods **foo( )** and **bar( )**, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an **A** and a **B** instance, and then starts a second thread to set up the deadlock condition. The **foo( )** and **bar( )** methods use **sleep( )** as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A {
synchronized void foo(B b) {
String name = Thread.currentThread().getName();
System.out.println(name + " entered A.foo");
try {
Thread.sleep(1000);
} catch(Exception e) {
System.out.println("A Interrupted");
}
System.out.println(name + " trying to call B.last()");
b.last();
}
synchronized void last() {
System.out.println("Inside A.last");
}
}
class B {
synchronized void bar(A a) {
String name = Thread.currentThread().getName();
System.out.println(name + " entered B.bar");
try {
Thread.sleep(1000);
} catch(Exception e) {
System.out.println("B Interrupted");
}
System.out.println(name + " trying to call A.last()");
a.last();
304 J a v a ™ 2 : T h e C o m p l e t e R e f e r e n c e
}
synchronized void last() {
System.out.println("Inside A.last");
}
}
class Deadlock implements Runnable {
A a = new A();
B b = new B();
Deadlock() {
Thread.currentThread().setName("MainThread");
Thread t = new Thread(this, "RacingThread");
t.start();
a.foo(b); // get lock on a in this thread.
System.out.println("Back in main thread");
}
public void run() {
b.bar(a); // get lock on b in other thread.
System.out.println("Back in other thread");
}
public static void main(String args[]) {
new Deadlock();
```

```
}
}
```

When you run this program, you will see the output shown here:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC . You will see that **RacingThread** owns the monitor on **b**, while it is waiting for the monitor on **a**. At the same time, **MainThread** owns **a** and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

5.19 Suspending, Resuming, and Stopping Threads

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter. The mechanisms to suspend, stop, and resume threads differ between Java 2 and earlier versions. Although you should use the Java 2 approach for all new code, you still need to understand how these operations were accomplished for earlier Java environments. For example, you may need to update or maintain older, legacy code. You also need to understand why a change was made for Java 2. For these reasons, the next section describes the original way that the execution of a thread was controlled, followed by a section that describes the approach required for Java 2.

5.20 Suspending, Resuming, and Stopping Threads Using Java 1.1 and Earlier

Prior to Java 2, a program used **suspend( )** and **resume( )**, which are methods defined by **Thread**, to pause and restart the execution of a thread. They have the form shown below:

```
final void suspend( )
final void resume( )
```

The following program demonstrates these methods:

```
// Using suspend() and resume().
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 = 15; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(200);
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
```

```
System.out.println(name + " exiting.");
}
}
class SuspendResume {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try {
Thread.sleep(1000);
ob1.t.suspend();
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.t.resume();
System.out.println("Resuming thread One");
ob2.t.suspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
ob2.t.resume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
```

```
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

The **Thread** class also defines a method called **stop( )** that stops a thread. Its signature is shown here:

```
final void stop( )
```

Once a thread has been stopped, it cannot be restarted using **resume( )**.

5.21 Suspending, Resuming, and Stopping Threads Using Java 2

While the **suspend( )**, **resume( )**, and **stop( )** methods defined by **Thread** seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The **suspend( )** method of the **Thread** class is deprecated in Java 2. This was done because **suspend( )** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked. The **resume( )** method is also deprecated. It does not cause problems, but cannot be used without the **suspend( )** method as its counterpart. The **stop( )** method of the **Thread** class, too, is deprecated in Java 2. This was done because this method can sometimes cause serious system failures.

Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state. Because you can't use the **suspend( )**, **resume( )**, or **stop( )** methods in Java 2 to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run( )** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution.

Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running," the **run( )** method must continue to let the thread execute. If this variable is set to "suspend," the thread must pause. If it is set to "stop," the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the **wait( )** and **notify( )** methods that are inherited from **Object** can be used to control the execution of a thread. This example is similar to the program in the previous section. However, the deprecated method calls have been removed. Let us consider the operation of this program. The **NewThread** class contains a **boolean** instance variable named **suspendFlag**, which is used to control the execution of the thread. It is initialized to **false** by the constructor. The **run( )** method contains a

synchronized statement block that checks **suspendFlag**. If that variable is **true**, the **wait( )** method is invoked to suspend the execution of the thread. The **mysuspend( )** method sets **suspendFlag** to **true**. The **myresume( )** method sets **suspendFlag** to **false** and invokes **notify( )** to wake up the thread. Finally, the **main( )** method has been modified to invoke the **mysuspend( )** and **myresume( )** methods.

```
// Suspending and resuming a thread for Java 2
class NewThread implements Runnable {
String name; // name of thread
Thread t;
boolean suspendFlag;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
suspendFlag = false;
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 15; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(200);
synchronized(this) {
while(suspendFlag) {
wait();
}
}
}
} catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
}
void mysuspend() {
suspendFlag = true;
}
synchronized void myresume() {
suspendFlag = false;
notify();
}
}
class SuspendResume {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
try {
Thread.sleep(1000);
ob1.mysuspend();
System.out.println("Suspending thread One");
Thread.sleep(1000);
ob1.myresume();
System.out.println("Resuming thread One");
ob2.mysuspend();
System.out.println("Suspending thread Two");
Thread.sleep(1000);
```

```
ob2.myresume();
System.out.println("Resuming thread Two");
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

The output from this program is identical to that shown in the previous section. Later in this book, you will see more examples that use the Java 2 mechanism of thread control. Although this mechanism isn't as "clean" as the old way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that *must* be used for all new code.

5.22 Using Multithreading

If you are like most programmers, having multithreaded support built into the language will be new to you. The key to utilizing this support effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!

6.
Graphic User Interface (GUI)

6.1 What Is AWT?

The GUI (Graphical User Interface) in java can be done with AWT and also with Swing APIs. The major difference is that AWT depends on the Operating System classes for the components whereas Swing is 100% pure java components. This would mean that any component developed using Swing would look exactly the same in all operating systems, whereas AWT components tend to look little bit different in different operating systems.

The Java Abstract Windowing Toolkit (AWT) provides numerous classes that support window program development. These classes are used to create and organize windows, implement GUI components, handle events, draw text and graphics, perform image processing, and obtain access to the native Windows implementation.

The basic idea behind the AWT is that a java window is a set of nested components starting from the outermost window all the way down to the smallest User Interface component. The AWT classes are contained in the java.awt package. It is one of the java's largest packages.

The AWT classes provide the following:

A full set of UI widgets and other components, including windows, menus, buttons, checkboxes etc.
Support for UI containers, which can contain other embedded containers and widgets.
Provision of Layout Managers whose duty is to handle the laying out of components on Containers.

An event system for managing system and user events between and among parts of the AWT
The Component class is the super class of the set of AWT classes that implement graphical user interface controls. The Component class provides a common set of methods that are used by all these subclasses. These methods include methods for handling events and working with images, fonts, and colors.

Java Components are implemented by the many subclasses of the java.awt.Component and java.awt.MenuComponent super class. The general way to categories them is to divide them into the following categories:

Visual Components
Container Components
Menu Components

The fundamental visual controls in java are:

1. Label – A simple label
2. Button – A simple push button
3. Checkbox – A combination of check box and radio (option) button
4. Choice – A drop down list control
5. List – A list box
6. Scroll bar – Horizontal and Vertical Scroll bar
7. Text Field – A single line text entry field
8. Text Area – A multiple line text entry field

The fundamental Container Components in java are:

1. Frame
2. Window
3. Applet
4. Panel

The fundamental Menu Components are:

1. MenuBar
2. Menu
3. MenuItem
4. CheckboxMenuItem

The superclass of all Non-Menu items in java is the Component class and for all Menu Items the superclass is the MenuComponent class.

Component class

1. A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

1. The Component class is the abstract superclass of the nonmenu-related Abstract Window Toolkit components. Class Component can also be extended directly to create a lightweight component. A lightweight component is a component that is not associated with a native opaque window.

Container Classes

- The **Container** class is a subclass of the Component class that is used to define components that have the capability to contain other components. It provides methods for adding, retrieving, displaying, counting, and removing the components that it contains. The Container class also provides methods for working with layouts. The layout classes control the layout of components within a container.

- The Container class has two major subclasses: Window and Panel. Window provides a common super class for application main windows (Frame objects) and Dialog windows. The Panel class is a generic container that can be displayed within a window. It is sub classed by the java.applet.Applet class as the base class for all Java applets.

- The containers contain individual components inside them.
- The two important things done with the container is establishing the container's layout manager and adding components to the container. Container is the abstract subclass of component, which allows other components to be nested inside it. These components can also be containers allowing other components to be nested inside it.
    - **Window –** It is a freestanding native window on the display. Window has Frame and Dialog as its subclasses
    - **Frame –** A frame is a window with a title and resizing corners.
    - **Dialog –** A Dialog does not have a menu bar. Although u can move it, you cannot resize it.
    - **Panel –** It is contained inside another container of inside a web browser's window. Panel identifies a rectangular area into which you must place other components. You must place Panel into a Window or subclass of window to be displayed.
- Incase you are not using the default layout managers associated with each container, we must ourselves place the components using the 3 methods – setLocation (), setSize () and setBounds ().

- The layout manager can override your decision. If you must control the size or position of components in such a way that cannot be done using the standard layout managers, you can disable the layout manager by issuing the following method call in your container

- Each Container has a default Layout Manager and to override that default layout manager, we use the method setLayout (new xxxLayout) where xxx is the name of the Layout which we want to set the container to.

- Incase we do not want to use the Layout Manger at all then we should use the method setLayout(null) and then it is the duty of the programmer to manually place the components.

The most common methods in the Component classes are:

1. **setBounds-**
   - Syntax:- public void **setBounds**(int x, int y, int width, int height)
   - Moves and resizes this component. The new location of the top-left corner is specified by x and y, and the new size is specified by width and height.
   - **Parameters:**

x - the new *x*-coordinate of this component
y - the new *y*-coordinate of this component
width - the new width of this component
height - the new height of this component

1. **setSize-**

- Syntax:-public void **setSize**(Dimension d)
- Resizes this component so that it has width d.width and height d.height.
- **Parameters:**

d - the dimension specifying the new size of this component

6.2 AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table lists some of the many AWT classes.

Class	**Description**
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.
CardLayout	The card layout manager. Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list. Color Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of **Component** that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in **width**, and the height is stored in **height**.
Event	Encapsulates events.
EventQueue	Queues events.
FileDialog	Creates a window from which a file can be selected.
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Font	Encapsulates a type font.
FontMetrics	Encapsulates various information related to a font. This information helps you display text in a window.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.

Graphics	Encapsulates the graphics context. This context is used by the various output methods to display output in a window.
GraphicsDevice	Describes a graphics device such as a screen or printer.
GraphicsEnvironment	Describes the collection of available **Font** and **GraphicsDevice** objects.
GridBagConstraints	Defines various constraints relating to the **GridBagLayout** class.
GridBagLayout	The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by **GridBagConstraints**.
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid. Image Encapsulates graphical images.
Insets	Encapsulates the borders of a container.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
MediaTracker	Manages media objects.
Menu	Creates a pull-down menu.
MenuBar	Creates a menu bar.
MenuComponent	An abstract class implemented by various menu classes.
MenuItem	Creates a menu item.
MenuShortcut	Encapsulates a keyboard shortcut for a menu item.
Panel	The simplest concrete subclass of **Container**.
Point	Encapsulates a Cartesian coordinate pair, stored in **x** and **y**.
Polygon	Encapsulates a polygon.
PopupMenu	Encapsulates a pop-up menu.
PrintJob	An abstract class that represents a print job.
Rectangle	Encapsulates a rectangle.
Robot	Supports automated testing ofAWT- based applications. (Added by Java 2, vl.3) Scrollbar Creates a scroll bar control.

Although the basic structure of the AWT has been the same since Java 1.0, some of the originalmethods were deprecated and replaced by new ones when Java 1.1 was released. For backward-compatibility, Java 2 still supports all the original 1.0methods. However, because thesemethods are not for use with new code, this book does not describe them.

6.3 Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets, and those derived from **Frame**, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 21-1 shows the class hierarchy for **Panel** and **Frame**. Let's look at each of these classes now.

6.4 Component

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. (You already used many of these methods when you created applets in Chapters 19 and 20.) A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Class	**Description**

ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
TextArea	Creates a multiline edit control.
TextComponent	A superclass for **TextArea** and **TextField**.
TextField	Creates a single-line edit control.
Toolkit	Abstract class implemented by the AWT.
Window	Creates a window with no frame, no menu bar, and no title.

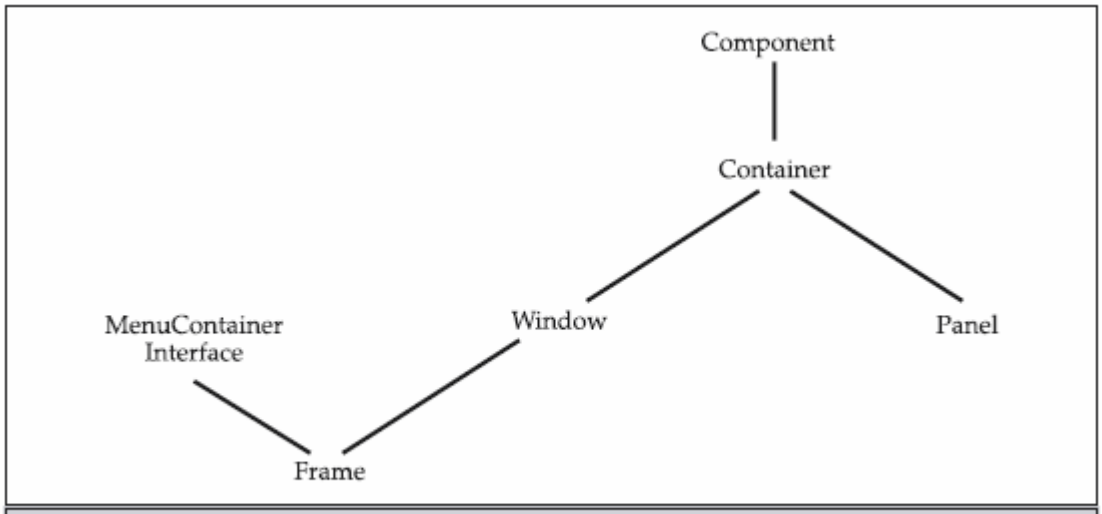


Figure 21-1. *The class hierarchy for* Panel *and* Frame

6.5 Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers.

6.6 Panel

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods defined by **Component**.

6.7 Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

6.8 Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of

Window and has a title bar, menu bar, borders, and resizing corners. If you create a

Frame object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a **Frame** window is created by a program rather than an applet, a normal window is created.

6.9 Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

6.9 Working with Frame Windows

After the applet, the type of window you will most often create is derived from **Frame**. You will use it to create child windows within applets, and top-level or child windows for applications. As mentioned, it creates a standard-style window. Here are two of **Frame**'s constructors:

Frame( )
Frame(String *title*)

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title*. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created.

6.10 Layout Managers

The duty of the Layout Manager is to layout the component automatically once the container is resized. Each of the containers has a default layout manager and we can change them with the use of the setLayout(new LayoutManager) method.

Incase we set the layout manager to null, then we have to manually reposition the components using the methods setLayout(), setSize() and setBounds() methods.
Every component has a preferred size (default size) and some of the Layout Managers fully honors and other does not honor the preferred size of the components.

The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout( )** method has the following general form:

void setLayout(LayoutManager layoutObj)

6.11 The types of Layout Managers are

- BorderLayout
- FlowLayout
- GridLayout
- GridbagLayout

FlowLayout

FlowLayout is the default layout manager. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor.

Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right.

Here are the constructors for **FlowLayout**:

- FlowLayout( )
- FlowLayout(int how)
- FlowLayout(int how, int horz, int vert)

- The first form creates the default layout, which centers components and leaves five pixels of space between each component.
- The second form lets you specify how each line is
- aligned. Valid values for *how* are as follows:
- FlowLayout.LEFT
- FlowLayout.CENTER
- FlowLayout.RIGHT
- The third form allowsyou to specify the horizontal and vertical space left between components in *horz* and *vert,*

Code

```
import java.awt.*;
class Test104
{
 Frame f;
 Button b1;
 Button b2;
        public void go()
        {
        f=new Frame("FlowLayout Example");
        f.setLayout(new FlowLayout());
        b1 = new Button("First Button");
        b2 = new Button("Second Button");
        f.add(b1);
        f.add(b2);
        f.pack();
        f.setVisible(true);
        }
        public static void main(String args[])
        {
        Test104 a = new Test104();
        a.go();
}
}
```

Applet Code:

```
import java.awt.*;
import java.applet.Applet;
public class Test105 extends Applet
{
        Button button1, button2, button3;
        public void init()
        {
        button1 = new Button("Ok");
        button2 = new Button("Open");
        button3 = new Button("Close");
        add(button1);
        add(button2);
        add(button3);
        }
}
<Applet code=Test105.class height = 400 width=400></Applet>
```

BorderLayout

- This is the default layout manager for Frame and Window. It has five distinct areas: North, South, East, West and Center. It is indicated by BorderLayout.NORTH, BorderLayout.SOUTH, BorderLayout.EAST, BorderLayout.WEST, and BorderLayout.CENTER.
- Also incase we do not specify the position mandatorily while adding the components, all the components, would be added to the Center position of the Container.
- The first way to add components to a container having a Border Layout is **add (Component c, int Position).** The c over is the component to be added and int position is one of the 5 styles as described above. (e.g. BorderLayout.EAST)
- Incase we do not specify the position; the component would be added to the center.
- The second way to add components in BorderLayout is by using the **add (String position e.g. "North", Component c)..**
- When adding a component to a container with a border layout, use one of these five constants, for example:

```
Panel p = new Panel();
p.setLayout(new BorderLayout());
p.add(new Button("Okay"), BorderLayout.SOUTH);
```


- As a convenience, BorderLayout interprets the absence of a string specification the same as the constant CENTER:

```
Panel p2 = new Panel();
p2.setLayout(new BorderLayout());
p2.add(new TextArea());  /
// Same as p.add(new TextArea(),
```

Code

```
import java.awt.*;
public class Test106 extends Frame
{
        Test106()
        {
```

```
                Button b = new Button("North");
                Button b1 = new Button("South");
                Button b2 = new Button("East");
                Button b3 = new Button("West");
                Button b4 = new Button("Center");
        add(b, BorderLayout.NORTH);
        add(b1, BorderLayout.SOUTH);
        add(b2, BorderLayout.EAST);
        add(b3, BorderLayout.WEST);
        add(b4, BorderLayout.CENTER);
        setSize(400,400);
        setVisible(true);
        }
public static void main(String arg[])
        {
                new Test106();
        }
}
```

Output

- Each of the components would be added their respective positions. For the North and South position the preferred height of the components to be added is taken and then it occupies the whole length of the North and South position.
- For the East and West position, the preferred width of the components to be added is taken and then it occupies the whole height of the East and West position.
- Once the four corner positions are taken, the balance space is taken by the Center position.
- Incase we do not have any of the 4 corner position, that is no components are added to the 4 corner position, then the Center position would take the balance space also.

Code

```
import java.awt.*;
public class Test106A extends Frame
{
        Test106A()
        {
                Button b = new Button("OK");
                Button b1 = new Button("CANCEL");
                Button b2 = new Button("CENTER");
        add("North",b);
        add("West",b1);
        add(b2);
        setSize(400,400);
        setVisible(true);
        }
        public static void main(String arg[])
        {
                new Test106A();
        }
}
```


Output

Here we have only two positions occupied and that is North and West and the Center Button takes the remaining position in the size of the Container.

You can add only a single component to each of the 5 regions of the BorderLayout manager and if you add more than one, only the one added last is visible.

Code

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

```
public class Test106B extends Frame
{           Test106B()
    {
     Button b = new Button("Hi");
     Button b1 = new Button("Hello");
     Button b2 = new Button("Welcome");
      add(b); add(b1); add(b2);
      setSize(400,400);
      setVisible(true);
           }
           public static void main(String arg[])
           {
           new Test106B();
           }
}
```

OutputOnly the Welcome button is shown and it takes the whole container object, this is because we have not specified the position where to place the component objects and all the components have been added to the center and only the last component added is shown.

GridLayout

- **GridLayout** lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns.
- The constructors supported by **GridLayout** are shown here:
    1. **GridLayout( )**
    2. **GridLayout(int numRows, int numColumns)**
    3. **GridLayout(int numRows, int numColumns, int horz, int vert)**
- The first form creates a single-column grid layout.
- The second form creates a grid layout with the specified number of rows and columns.
- The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.
- Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Code

```
import java.awt.*;
public class Test107 extends Frame
{
           Test107()
           {
           setLayout(new GridLayout(3,2));
           add(new Button("1"));
           add(new Button("2"));
           add(new Button("3"));
           add(new Button("4"));
           add(new Button("5"));
           add(new Button("6"));
           setSize(200,200);
           setVisible(true);
           }
public static void main(String arg[])
           {
                      new Test107();
           }
```

}

- When both the number of rows and the number of columns have been set to non-zero values, either by a constructor or by the setRows and setColumns methods, the number of columns specified is ignored. Instead, the number of columns is determined from the specified number or rows and the total number of components in the layout.
- So, for example, if three rows and two columns have been specified and nine components are added to the layout, then they will be displayed as three rows of three columns. Specifying the number of columns affects the layout only when the number of rows is set to zero.

GridbagLayout

- Whenever we need a complex layout and that existing layout managers do not meet our requirements, we will be using a Gridbag layout. It is important over here with, we will be only studying the conceptual basis and no coding is required.
- The GridBagLayout can be used to layout components in a grid like container layout. Unlike the GridLayout, the sizes of the grids do not need to be constant, and a component can occupy more (or less) than one row or column.

- Before we go ahead with building the GridbagLayout the following information would be needed :
    1. The number of rows and columns required
    2. Each component is associated with one area which may be one or more grid rectangles.
    3. The number of rows and columns in the grid is determined by the largest values of gridx and gridy.
    4. The anchor attribute indicates where in the grid the component will appear in its area if it does not exactly fill its area.
    5. The fill attribute determines whether the components should be stretched to fill the entire area.
    6. The weight attributes determine the various sizes of the grid areas.
    - The information's like the weight, fill etc attributes are stored in the GridBagContstraints class and it can be associated with a component using "setContraints (Component, GridBagContraints)"
- This causes the layout manager to make a copy of the constraints and associate them with the object. Hence we would need only one of these GridBagContraints objects.

- **The following is a complete list of all of the constraints:**
    - Anchor determines position in the display area
    - Fill determines if a component is stretched to fill the area
    - Gridheight and gridwidth determine the number of rows and columns in the component's area
    - Gridx and gridy determine the position of the component's area.
    - Insets determine a border around a component's area.
    - Ipadx and ipady allows the minimum or preferred size of a component to be adjusted.
    - Weightx and weighty determine the sizes of the rows and columns of the grids.
    - The weights determine how the extra space is distributed after the components are laid out using their preferred sizes.

- The algorithm for doing this is simple in the case in which all components have gridwidth and gridheight equal to 1. In this case the weight of a grid column or row is the maximal weight of any component in that column or row.
- The extra space is divided based on these weights. For example, if the total weight of all of the columns is 10, then a column with weight 4 would get 4/10 of the total extra space.
- If there are components which span more than one row or column, the weights are first determined by those components spanning only one row and column. Then each additional component is handled in the order it was added.

- If its weight is less than the total weights of the rows or columns it spans, it has no effect. Otherwise, its extra weight is distributed based on the weights of those rows or columns it spans.
- If the extra space is negative, the layout manager squeezes things and you have not control over how this is done.

6.12 Anchors

There are nine anchor types specifying 8 basic compass positions and the center position. NORTH, SOUTH, EAST, WEST, NORTHWEST, NORTHEAST, SOUTHWEST, SOUTHEAST, CENTER

Fill
The four fill types are NONE (the default), VERTICAL, HORIZONTAL, and BOTH. They affect components whose preferred size does not completely fill their grid area.
gridx, gridy
The gridx and gridy fields determine the positioning of the components. The default is to put the object in the position after the last one which was added.

Internal Padding
The fields' ipadx and ipady can be used to modify the minimum or preferred size of a component. For a button, the preferred width is determined by the length of its label and the preferred height is determined by the font used for the label.

Gridwidth and gridheight
The gridwidth and gridheight fields determine the number of cells that are included in a component's area.

The possible values are a (small) positive integer, or one of the special values REMAINDER or RELATIVE.

REMAINDER means the component will occupy all of the cells to the right or below it.

RELATIVE means the component will occupy all of the cells to the right or below it except for the last cell.By default, a cell is placed right after the previous one. By setting a width or height to REMAINDER, the component will occupy all cells to the right or below it.

By setting the width or height to Relative, the component will occupy all cells to the right or below, except for the last one.

6.13 Event Handling

event handling is at the core of successful applet programming. Most events to which your applet will respond are generated by the user. These events are passed to your applet in a variety of ways, with the specific method depending upon the actual event. There are several types of events.

The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button. Events are supported by the **java.awt.event** package.
The chapter begins with an overview of Java's event handling mechanism.

It then examines the main event classes and interfaces, and develops several examples that demonstrate the fundamentals of event processing. This chapter also explains how to use adapter classes, inner classes, and anonymous inner classes to streamline event handling code. The examples provided in the remainder of this book make frequent use of these techniques.

6.14 Two Event Handling Mechanisms

Before beginning our discussion of event handling, an important point must be made: The way in which events are handled by an applet changed significantly between the original version of Java (1.0) and modern versions of Java, beginning with version 1.1. The 1.0 method of event handling is still supported, but it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs, including those written for Java 2, and thus is the method employed by programs in this book.

6.15 The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns.

The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

Java also allows you to process events without using the delegation event model. This can be done by extending an AWT component. This technique is discussed at the end of Chapter 22. However, the delegation event model is the preferred design for the reasons just cited.

The following sections define events and describe the roles of sources and listeners.

6.16 Events

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

6.17 Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void add*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example,
the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such
a method is this:

public void add*Type*Listener(*Type*Listener *el*)
throws java.util.TooManyListenersException

Here, *Type* is the name of the event and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, *Type* is the name of the event and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**. The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

6.18 Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Many other listener interfaces are discussed later in this and other chapters.

6.19 Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events. At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

EventObject(Object *src*)
Here, *src* is the object that generates this event.

EventObject contains two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

Object getSource( )

As expected, **toString( )** returns the string equivalent of the event. The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:
int getID( )

Additional details about **AWTEvent** are provided at the end of Chapter 22. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**. To summarize:

■ **EventObject** is a superclass of all events.
■ **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The package **java.awt.event** defines several types of events that are generated by various user interface elements. Table 20-1 enumerates the most important of these event classes and provides a brief description of when they are generated. The most commonly used constructors and methods in each class are described in the following sections.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.

Table 20-1. *Main Event Classes in* java.awt.event

Event Class	Description
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 20-1. *Main Event Classes in* java.awt.event (continued)

The ActionEvent Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

ActionEvent has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
ActionEvent(Object *src*, int *type,* String *cmd,* long *when*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred. The third constructor was added by Java 2, version 1.4. You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

String getActionCommand( )

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button. The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its formis shown here:

Event Class	Description
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

int getModifiers( )
Java 2, version 1.4 added the method **getWhen( )** that returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen( )** method is shown here. long getWhen( ) Timestamps were added by **ActionEvent** to help support the improved input focus subsystem implemented by Java 2, version 1.4.

The AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here: BLOCK_DECREMENT The user clicked inside the scroll bar to decrease its value. BLOCK_INCREMENT The user clicked inside the scroll bar to increase its value.

TRACK The slider was dragged. UNIT_DECREMENT The button at the end of the scroll bar was clicked to decrease its value. UNIT_INCREMENT The button at the end of the scroll bar was clicked to increase its value. In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred. Here is one **AdjustmentEvent** constructor:

AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *data*)

Here, *src* is a reference to the object that generated this event. The *id* equals

ADJUSTMENT_VALUE_CHANGED. The type of the event is specified by *type*,

and its associated data is *data*.

The **getAdjustable( )** method returns the object that generated the event. Its form is shown here:

Adjustable getAdjustable( )

The type of the adjustment event may be obtained by the **getAdjustmentType( )** method. It returns one of the constants defined by **AdjustmentEvent**. The general form is shown here:

int getAdjustmentType( )
The amount of the adjustment can be obtained from the **getValue( )** method, shown here:

int getValue( )

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

The ComponentEvent Class

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

COMPONENT_HIDDEN The component was hidden.
COMPONENT_MOVED The component was moved.
COMPONENT_RESIZED The component was resized.
COMPONENT_SHOWN The component became visible.

ComponentEvent has this constructor:

ComponentEvent(Component *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**.

The **getComponent( )** method returns the component that generated the event. It is shown here:
Component getComponent( )

The ContainerEvent Class

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**. They indicate that a component has been added to or removed from the container. **ContainerEvent** is a subclass of **ComponentEvent** and has this constructor:

ContainerEvent(Component *src*, int *type*, Component *comp*)

Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*. You can obtain a reference to the container that generated this event by using the **getContainer( )** method, shown here:

Container getContainer( )
The **getChild( )** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:
Component getChild( )

The FocusEvent Class

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**. **FocusEvent** is a subclass of **ComponentEvent** and has these constructors:
FocusEvent(Component *src*, int *type*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
Focus Event(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.) The other component involved in the focus change, called the *opposite component*, is passed in *other*.

Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus. Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus. The third constructor was added by Java 2, version 1.4. You can determine the other component by calling **getOppositeComponent( )**, shown here. Component getOppositeComponent( ) The opposite component is returned. This method was added by Java 2, version 1.4. The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary( )

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

The InputEvent Class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers.

ALT_MASK BUTTON2_MASK META_MASK
ALT_GRAPH_MASK BUTTON3_MASK SHIFT_MASK
BUTTON1_MASK CTRL_MASK

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, Java 2, version 1.4 added the following extended modifier values.

ALT_DOWN_MASK ALT_GRAPH_DOWN_MASK BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK BUTTON3_DOWN_MASK CTRL_DOWN_MASK
META_DOWN_MASK SHIFT_DOWN_MASK

When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers. To test if a modifier was pressed at the time an event is generated, use the

isAltDown( ), **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:

boolean isAltDown( )
boolean isAltGraphDown( )

boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
You can obtain a value that contains all of the original modifier flags by calling the **getModifiers( )** method. It is shown here:
int getModifiers( )
You can obtain the extended modifiers by called **getModifiersEx( )**, which is shown here.
int getModifiersEx( )
This method was also added by Java 2, version 1.4.

The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this book.) There are two types of item events, which are identified by the following integer constants:

DESELECTED The user deselected an item.
SELECTED The user selected an item.

In addition, **ItemEvent** defines one integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state. **ItemEvent** has this constructor:

ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.

The **getItem( )** method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

Object getItem( )

The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

ItemSelectable getItemSelectable( )

Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.The **getStateChange( )** method returns the state change (i.e., **SELECTED** or **DESELECTED**) for the event. It is shown here:

int getStateChange( )

The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all key presses result in characters. For example, pressing the SHIFT key does not generate a character. There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ENTER VK_ESCAPE VK_CANCEL VK_UP
VK_DOWN VK_LEFT VK_RIGHT VK_PAGE_DOWN
VK_PAGE_UP VK_SHIFT VK_ALT VK_CONTROL

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt. **KeyEvent** is a subclass of **InputEvent**. Here are two of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*)
KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains

CHAR_UNDEFINED. For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**. The **KeyEvent** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

char getKeyChar( )
int getKeyCode( )

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED The user clicked the mouse.
MOUSE_DRAGGED The user dragged the mouse.
MOUSE_ENTERED The mouse entered a component.
MOUSE_EXITED The mouse exited from a component.
MOUSE_MOVED The mouse moved.
MOUSE_PRESSED The mouse was pressed.
MOUSE_RELEASED The mouse was released.
MOUSE_WHEEL The mouse wheel was moved (Java 2, v1.4).

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors. MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *x*, int *y*, int *clicks*, boolean *triggersPopup*) Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*.

The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. Java 2, version 1.4 adds a second constructor which also allows the button that caused the event to be specified. The most commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse when the event occurred. Their forms are shown here:

int getX( )
int getY( )

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:
Point getPoint( )
It returns a **Point** object that contains the X, Y coordinates in its integer members: **x** and **y**.
The **translatePoint( )** method changes the location of the event. Its form is shown here:
void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event. The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

int getClickCount( )
The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

boolean isPopupTrigger( )
Java 2, version 1.4 added the **getButton( )** method, shown here.
int getButton( )

It returns a value that represents the button that caused the event. The return value will be one of these constants defined by **MouseEvent**.
NOBUTTON BUTTON1 BUTTON2 BUTTON3
The **NOBUTTON** value indicates that no button was pressed or released.

The MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent** and was added by Java 2, version 1.4. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants.

WHEEL_BLOCK_SCROLL A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL A line-up or line-down scroll event occurred.
MouseWheelEvent defines the following constructor.

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*,
int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event.

To obtain the number of rotational units, call **getWheelRotation( )**, shown here.
int getWheelRotation( )

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. To obtain the type of scroll, call **getScrollType( )**, shown next.

int getScrollType( )

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount( )**. It is shown here.

int getScrollAmount( )

The TextEvent Class

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**. The one constructor for this class is shown here:

TextEvent(Object *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the **TextEvent** class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED The window was activated.
WINDOW_CLOSED The window has been closed.
WINDOW_CLOSING The user requested that the window be closed.
WINDOW_DEACTIVATED The window was deactivated.
WINDOW_DEICONIFIED The window was deiconified.
WINDOW_GAINED_FOCUS The window gained input focus.
WINDOW_ICONIFIED The window was iconified.
WINDOW_LOST_FOCUS The window lost input focus.
WINDOW_OPENED The window was opened.
WINDOW_STATE_CHANGED The state of the window changed. (Added by Java 2, version 1.4.)

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors.

The first is WindowEvent(Window *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is *type.* Java 2, version 1.4 adds the next three constructors. WindowEvent(Window *src*, int *type*, Window *other*) WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*) WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Here, *other* specifies the opposite window when a focus event occurs. The *fromState* specifies the prior state of the window and *toState* specifies the new state that the window will have when a window state change occurs. The most commonly used method in this class is **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:

Window getWindow( )

Java 2, version 1.4, adds methods that return the opposite window (when a focus event has occurred), the previous window state, and the current window state. These methodsare shown here:

Window getOppositeWindow()
int getOldState()
int getNewState()

6.20 Sources of Events

Table 20-2 lists some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, other components, such as an applet, can generate events. For example, you receive key and mouse events from an applet. (You may also build your own components that generate events.) In this chapter we will be handling only mouse and keyboard events, but the following two chapters will be handling events from the sources shown in Table 20-2.

Event Source	**Description**
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 20-2. Event Source Examples

6.21 Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 20-3 lists commonly used listener interfaces and provides a brief description of the methods that they define. The following sections examine the specific methods that are contained in each interface.

Interface	**Description**
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.

MouseWheelListener	Defines one method to recognize when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus. (Added by Java 2, version 1.4)
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 20-3. Commonly Used Event Listener Interfaces

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:
void actionPerformed(ActionEvent *ae*)

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:
void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:
void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

*The AWT processes the resize and move events. The **componentResized( )** and **componentMoved( )** methods are provided for notification purposes only.*

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered. For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:
void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively. The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here.

void mouseWheelMoved(MouseWheelEvent *mwe*)
MouseWheelListener was added by Java 2, version 1.4.

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or losses input focus. Their general forms are shown here.

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)
WindowFocusListener was added by Java 2, version 1.4.

The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent *we*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

6.22 Using the Delegation Event Model

Now that you have learned the theory behind the delegation event model and have had an overview of its various components, it is time to see it in practice. Applet programming using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications. Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events. To see how the delegation model works in practice, we will look at examples that handle the two most commonly used event generators: the mouse and keyboard.

6.23 Handling Mouse Events

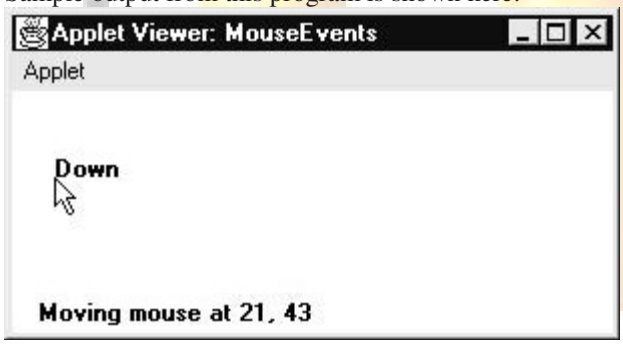
To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. (You may also want to implement **MouseWheelListener**, but we won't be doing so, here.) The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area. As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When

dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener {
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init() {
addMouseListener(this);
addMouseMotionListener(this);
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
// Handle mouse exited.
public void mouseExited(MouseEvent me) {
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse exited.";
repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "Down";
repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me) {
// save coordinates
```

```
mouseX = me.getX();
mouseY = me.getY();
msg = "Up";
repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
// save coordinates
mouseX = me.getX();
mouseY = me.getY();
msg = "*";
showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me) {
// show status
showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
}
// Display msg in applet window at current X,Y location.
public void paint(Graphics g) {
g.drawString(msg, mouseX, mouseY);
}
}
```

Sample output from this program is shown here:



Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events.

This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here: void addMouseListener(MouseListener *ml*) void addMouseMotionListener(MouseMotionListener *mml*)Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both. The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

6.24 Handling Keyboard Events

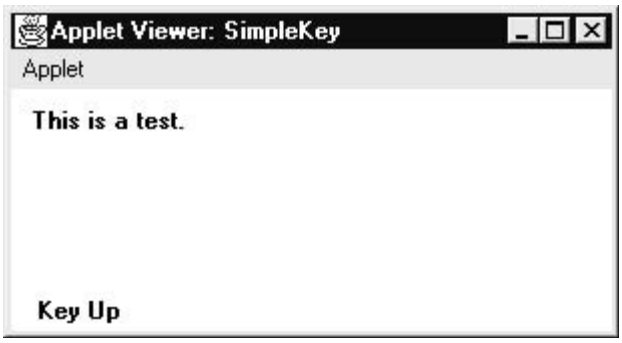
To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface. Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated.

If all you care about are actual characters, then you can ignore the information passed by the key press and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed( )** handler. There is one other requirement that your program must meet before it can process keyboard events: it must request input focus. To do this, call **requestFocus( )**, which is defined by **Component**. If you don't, then your program will not receive any keyboard events.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
requestFocus(); // request input focus
}
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

Sample output is shown here:



If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed( )** handler. They are not available through **keyTyped( )**. To identify the keys, you use their virtual key codes. For example, the next applet outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/
public class KeyEvents extends Applet
implements KeyListener {
String msg = "";
int X = 10, Y = 20; // output coordinates
public void init() {
addKeyListener(this);
requestFocus(); // request input focus
}
public void keyPressed(KeyEvent ke) {
showStatus("Key Down");
int key = ke.getKeyCode();
switch(key) {
case KeyEvent.VK_F1:
msg += "<F1>";
break;
case KeyEvent.VK_F2:
msg += "<F2>";
break;
case KeyEvent.VK_F3:
msg += "<F3>";
break;
case KeyEvent.VK_PAGE_DOWN:
msg += "<PgDn>";
break;
case KeyEvent.VK_PAGE_UP:
msg += "<PgUp>";
break;
case KeyEvent.VK_LEFT:
msg += "<Left Arrow>";
break;
case KeyEvent.VK_RIGHT:
msg += "<Right Arrow>";
```

```
break;
}
repaint();
}
public void keyReleased(KeyEvent ke) {
showStatus("Key Up");
}
public void keyTyped(KeyEvent ke) {
msg += ke.getKeyChar();
repaint();
}
// Display keystrokes.
public void paint(Graphics g) {
g.drawString(msg, X, Y);
}
}
```

Sample output is shown here:

The procedures shown in the preceding keyboard and mouse event examples can be generalized to any type of event handling, including those events generated by controls. In later chapters, you will see many examples that handle other types of events, but they will all follow the same basic structure as the programs just described.

6.25 Adapter Classes

Java provides a special feature, called an *adapter class*, that can simplify the creation ofevent handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested. For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**. The signatures of these empty methods are exactly as defined in the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and implement **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

Table 20-4 lists the commonly used adapter classes in **java.awt.event** and notes the interface that each implements. The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init( )** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument. **MyMouseAdapter**

implements the **mouseClicked( )** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** implements the **mouseDragged( )** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

Note that both of our event listener classes save a reference to the applet. This information is provided as an argument to their constructors and is used later to invoke the **showStatus( )** method.

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo) {
```

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Table 20-4. Commonly Used Listener Interfaces Implemented by Adapter Classes

```
this.adapterDemo = adapterDemo;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me) {
adapterDemo.showStatus("Mouse clicked");
}
}
class MyMouseMotionAdapter extends MouseMotionAdapter {
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
this.adapterDemo = adapterDemo;
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me) {
adapterDemo.showStatus("Mouse dragged");
}
}
```

As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener** and **MouseListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods. As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a **KeyAdapter**.

6.26 Inner Classes

In Chapter 7, the basics of inner classes were explained. Here you will see why they are important. Recall that an *inner class* is a class defined within other class, or even within an expression. This section illustrates how inner classes can be used to simplify the code when using event adapter classes.
To understand the benefit provided by inner classes, consider the applet shown in the following listing. It *does not* use an inner class. Its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

There are two top-level classes in this program. **MousePressedDemo** extends **Applet**, and **MyMouseAdapter** extends **MouseAdapter**. The **init( )** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener( )** method. Notice that a reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor. This reference is stored in an instance variable for later use by the **mousePressed( )** method.

When the mouse is pressed, it invokes the **showStatus( )** method of the applet
through the stored applet reference. In other words, **showStatus( )** is invoked relative to the applet reference stored by **MyMouseAdapter**.

```
// This applet does NOT use an inner class.

import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter {
MousePressedDemo mousePressedDemo;
public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
this.mousePressedDemo = mousePressedDemo;
}
public void mousePressed(MouseEvent me) {
mousePressedDemo.showStatus("Mouse Pressed.");
}
}
```


The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**. Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed( )** method can call the **showStatus( )** method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass **MyMouseAdapter( )** a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
```

```
</applet>
*/
public class InnerClassDemo extends Applet {
public void init() {
addMouseListener(new MyMouseAdapter());
}
class MyMouseAdapter extends MouseAdapter {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
}
}
```

6.27 Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. This section illustrates how an anonymous inner class can facilitate the writing of event handlers. Consider the applet shown in the following listing. As before, its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/
public class AnonymousInnerClassDemo extends Applet {
public void init() {
addMouseListener(new MouseAdapter() {
public void mousePressed(MouseEvent me) {
showStatus("Mouse Pressed");
}
});
}
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init( )** method calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully. The syntax **new MouseAdapter( ) { ... }** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**.

This new class is not named, but it is automatically instantiated when this expression is executed. Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus( )** method directly. As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.

6.28 Applet Basics

All applets are subclasses of **Applet**. Thus, all applets must import **java.applet**. Applets must 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, they are executed by either a Web browser or an applet viewer. The figures shown in this chapter were created with the standard applet viewer, called **appletviewer**, provided by 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 have **main( )** methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by **System.out.println( )**.

Rather, it is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location. Input is also handled differently than in an application. Once an applet has been compiled, it is included in an HTML file using the APPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is 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 a window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

6.29 The Applet Class

The **Applet** class defines the methods shown in Table 19-1. **Applet** provides all necessary support for applet execution, such as starting and stopping. It also provides methods 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 extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities. (The AWT is described in detail in following chapters.)

Method	Description
void destroy( )	Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction.
AccessibleContext getAccessibleContext( )	Returns the accessibilty context for the invoking 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 that encapsulates the audio clip found at the location specified by *url*.
AudioClip getAudioClip(URL *url*, String *clipName*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url* and having the name specified by *clipName*.
URL getCodeBase( )	Returns the URL associated with the invoking applet.
URL getDocumentBase( )	Returns the URL of the HTML document that invokes the applet.

Table 19-1. *The Methods Defined by* Applet

Method	Description
Image getImage(URL *url*)	Returns an **Image** object that encapsulates the image found at the location specified by *url*.
Image getImage(URL *url*, String *imageName*)	Returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.
Locale getLocale( )	Returns a **Locale** object that is used by various locale-sensitive classes and methods.
String getParameter(String *paramName*)	Returns the parameter associated with *paramName*. **null** is returned if the specified parameter is not found.
String[ ] [ ] getParameterInfo( )	Returns a **String** table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose.
void init( )	Called when an applet begins execution. It is the first method called for any applet.
boolean isActive( )	Returns **true** if the applet has been started. It returns **false** if the applet has been stopped.
static final AudioClip newAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. This method is similar to **getAudioClip( )** except that it is static and can be executed without the need for an **Applet** object. (Added by Java 2)

Table 19-1. *The Methods Defined by* Applet (continued)

Method	Description
void play(URL *url*)	If an audio clip is found at the location specified by *url*, the clip is played.
void play(URL *url*, String *clipName*)	If an audio clip is found at the location specified by *url* with the name specified by *clipName*, the clip is played.
void resize(Dimension *dim*)	Resizes the applet according to the dimensions specified by *dim*. **Dimension** is a class stored inside **java.awt**. It contains two integer fields: **width** and **height**.
void resize(int *width*, int *height*)	Resizes the applet according to the 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-time system and is not usually called by your applet. A *stub* is a small piece of code that provides the linkage between your applet and the browser.
void showStatus(String *str*)	Displays *str* in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place.
void start( )	Called by the browser when an applet should start (or resume) execution. It is automatically called after **init( )** when an applet first begins.
void stop( )	Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls **start( )**.

Table 19-1. *The Methods Defined by* Applet (continued)

6.30 Applet Architecture

An applet is a window-based program. As such, its architecture is different from the so-called normal, console-based programs shown in the first part of this book. If you are 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 until the following chapter, it is important to understand in a general way how the event-driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler

that has been provided by the applet. Once this happens, the applet must take appropriate 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 it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the AWT run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional 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. As you know, in a nonwindowed program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( )**.

This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must 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 window has input focus, a keypress event is generated.

As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated. While the architecture of an applet is not as easy to understand as that of a console-based program, Java's AWT makes it as simple as possible. If you have written programs for Windows, you know how intimidating that environment can be. Fortunately, Java's AWT provides a much cleaner approach that is more quickly mastered.

6.31 An Applet Skeleton

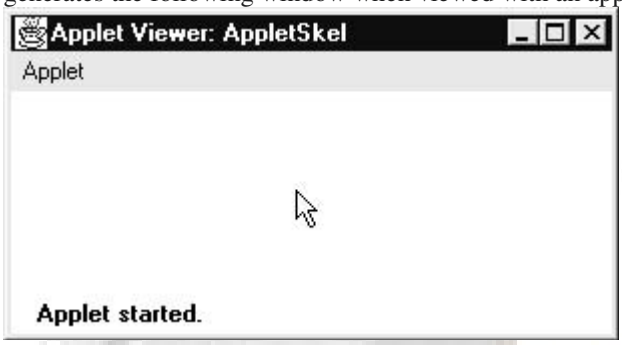
All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods—**init( )**, **start( )**, **stop( )**, and **destroy( )**—are defined by **Applet**. Another, **paint( )**, is defined by the AWT **Component** class. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. These five methods can be assembled into the skeleton shown 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, it generates the following window when viewed with an applet viewer:



6.32 Applet Initialization and Termination (Applet Life Cycle)

It is important to understand the order in which the various methods shown in the skeleton 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 initialize variables. 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 has been 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 user leaves 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. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also

called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

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

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

6.33 Overriding update( )

In some situations, your applet may need to override another method defined by the AWT, called **update( )**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** first fills an applet with the default background color and then calls **paint( )**. If you fill the background using a different color in **paint( )**, the user will experience a flash of the default background 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 performs all necessary display activities. Then have **paint( )** simply call **update( )**. Thus, for some applications, 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.

6.34 The HTML APPLET Tag

The APPLET tag is used to start an applet from both an HTML document and from an applet 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 allow many applets on a single page. So far, we have been using only a simplified form of the APPLET 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 the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read. CODE CODE is a required attribute that gives the name of the file containing your applet'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 by CODEBASE if set.

ALT The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that 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 by name 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 the size (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 each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes. PARAM NAME AND VALUE The PARAM tag allows you to specify appletspecific arguments in an HTML page. Applets access their attributes with the **getParameter( )** method.

HANDLING OLDER BROWSERS Some very old web browsers can't execute applets and 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 them occasionally. The best way to design your HTML page to deal with such browsers is to include HTML text and markup within your <applet></applet> tags. If the applet tags are not recognized by your browser, you will see the alternate markup. If Java is available, it will consume all of the markup between the <applet></applet> tags and disregard the alternate markup. Here's the HTML to start an applet called **SampleApplet** in Java and to display a message 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; here.<p>
</applet>

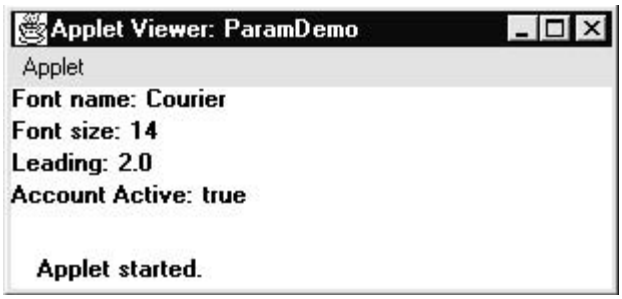
6.35 Passing Parameters to Applets

As just discussed, the APPLET tag in HTML allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter( )** method. It returns the value the 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 demonstrates 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:



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

6.36 getDocumentBase( ) and getCodeBase( )

Often, you will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet (the *document base*) and the directory from which the applet's class file was loaded (the *code base*). These directories are returned as **URL** objects (described in Chapter 18) by **getDocumentBase( )** and **getCodeBase( )**. They can be concatenated with a string that names the file you want to load. To actually load another file, you will use the **showDocument( )** method defined by the **AppletContext** 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:

6.37 AppletContext and showDocument( )

One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than the underlined blue words 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 information from the applet's execution environment. The methods defined by **AppletContext** are shown in Table 19-2. The context 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 bring another document into view by calling **showDocument( )**. This method has no return value and throws no exception if it fails, so use it carefully. There are two **showDocument( )** methods. The method **showDocument(URL)** displays the document

Method	Description
Applet getApplet(String *appletName*)	Returns the applet specified by *appletName* if it is within the current applet context. Otherwise, **null** is returned.
Enumeration getApplets( )	Returns an enumeration that contains all of the applets within the current applet context.
AudioClip getAudioClip(URL *url*)	Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url.*
Image getImage(URL *url*)	Returns an **Image** object that encapsulates the image found at the location specified by *url.*
InputStream getStream(String *key*)	Returns the stream linked to *key*. Keys are linked to streams by using the **setStream( )** method. A null reference is returned if no stream is linked to *key*. (Added by Java 2, version 1.4)
Iterator getStreamKeys( )	Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See **getStream( )** and **setStream( )**. (Added by Java 2, version 1.4)
void setStream(String *key*,InputStream *strm*)	Links the stream specified by *strm* to the key passed in *key*. The *key* is deleted from the invoking object if *strm* is null. (Added by Java 2, version 1.4)
void showDocument(URL *url*)	Brings the document at the **URL** specified by *url* into view. This method may not be supported by applet viewers.
void showDocument(URL *url*, String *where*	Brings the document at the **URL** specified by *url* into view. This method may not be supported by applet viewers. The placement of the document is specified by *where* as described in the text. void showStatus(String *str*) Displays *str* in the status window.

Table The Abstract Methods Defined by the AppletContext Interface

at the specified **URL**. The method **showDocument(URL, where)** displays the specified document 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 also specify 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 to transfer control to a file called **Test.html**. This file must be in the same directory as 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");
}
}
}
```

6.38 Swing

Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.
Swing provides a set of "lightweight" (all-Java language) components that, to the maximum degree possible, work the same on all platforms.
The term *lightweight* is used to describe such elements. The Swing component classes that are used are shown here:

Class	Description
Abstract Button	Abstract superclass for Swing buttons.
Button Group	Encapsulates a mutually exclusive set of buttons.
ImageIcon	Encapsulates an icon.
Applet	The Swing version of Applet.
Button	The Swing push button class.
Checkbox	The Swing check box class.
JComboBox	Encapsulates a combo box (an combination of a drop-down list and text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
Textile	The Swing version of a text field.
Tree	Encapsulates a tree-based control.

- **Advantages of swing component:**
  - In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.
  - Components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.
  - Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent.
  - You can easily add or change the borders drawn around most Swing components. For example, it's easy to put a box around the outside of a container or label.
  - You can easily change the behavior or appearance of a Swing component by either invoking methods on it or creating a subclass of it.
  - Swing components don't have to be rectangular. Buttons, for example, can be round.
  - Assistive technologies such as screen readers can easily get information from Swing components. For example, a tool can easily get the text that's displayed on a button or label.

○

6.39 Difference between AWT and Swing

The biggest difference between the AWT components and Swing components is that the Swing components are implemented with absolutely no native code. Since Swing components aren't restricted to the least common denominator -- the features that are present on every platform -- they can have more functionality than AWT components.

- Swing lets you specify which look and feel your program's GUI uses. By contrast, AWT components always have the look and feel of the native platform.
- Major difference between swing and AWT components:

Sr. no.	AWT	SWING
1.	Heavy weight	Light weight
2.	Native component	Pure java Component
3.	Native look and feel	Better look and feel
4.	Does not have complex component	Has additional components like JTree, JTable, JProgressBar, Slider, etc
5.	Applet can not have menu	Applet can have menu.
6.	List has scrollbar	JList does not support scrolling but this can be done using scroll Pane
7.	Components can be added directly on the Frame or window	While adding components on Frame or window, they have to be added on its content pane.
8.	Does not have slidePane or TabbedPane	Has slide Pane or TabbedPane
9.	Does not support MDI window	MDI can be achived using InternalFrame Object
10.	Menu item cannot have images or radio buttons or checkboxes.	Menu item can have images or radio buttons or checkboxes.
11.	They do not have JMV(Java Model Viewport)	All swing components have JMV
12.	Default layout is flowLayout	Default layout is BorderLayout.

6.40 JApplet

- Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that use Swing must be subclasses of **JApplet**.
- **JApplet** are specialized components that provide a place for other Swing components to paint themselves.
- **JApplet** supports various "panes," such as the content pane, the glass pane, and the root pane.
- When adding a component to an instance of **JApplet**, do not invoke the **add()** method of the applet. Instead, call **add()** for the *content pane* of the **JApplet** object.
- The content pane can be obtained via the method shown here:

  Container getContentPane( )

  The **add( )** method of **Container** can be used to add a component to a content pane. Its form is shown here:

  void add(comp)

  Here, *comp* is the component to be added to the content pane.
- **Example:-**
- **Source code:-**

```
// JApplet Example Code
import java.awt.*;
import javax.swing.*;
public class JAppletExample extends JApplet
{
public void init()
{
Container content = getContentPane();
content.setBackground(Color.white);
content.setLayout(new FlowLayout());
content.add(new JButton("Button 1"));
content.add(new JButton("Button 2"));
content.add(new JButton("Button 3"));
}
}
```

6.41 Icons and Labels

Icons:-

- In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image. An icon is a small fixed size picture, typically used to decorate components.
- ImageIcon is an implementation of the Icon interface that paints Icons from Image.

- Two of its constructors are shown here:
    1. **ImageIcon(String filename):-** Creates an ImageIcon from the specified file. The specified String can be a file name or a file path.
    2. **ImageIcon(URL url):-** Creates an ImageIcon from the specified URL. The icon's description is initialized to be a string representation of the URL.
- The **ImageIcon** class declares following method:
- **Method**

int getIconHeight( )- Returns the height of the icon in pixels.
int getIconWidth( )- Returns the width of the icon in pixels.
void paintIcon(Component *comp*,Graphics *g*,int *x*, int *y*)- Paints the icon at position *x*, *y* on the graphics context *g*. Additional information about the paint operation can be provided in *comp*.
public Image getImage():-Returns this icon's `Image`.
public void setImage(Image image):-Sets the image displayed by this icon.
public String getDescription():-Gets the description of the image.

Labels:-

- Swing labels are instances of the **JLabel** class, which extends **JComponent**. It can display text and/or an icon. A label does not react to input events.
- A `JLabel` object can display either text, an image, or both. You can specify where in the label's display area the label's contents are aligned by setting the vertical and horizontal alignment. By default, labels are vertically centered in their display area.
- Some of its constructors are shown here:
    1. **JLabel(Icon i):-** Creates a `JLabel` instance with the specified image.
    2. **JLabel(String s):-** Creates a `JLabel` instance with the specified text.
    3. **JLabel(String s, Icon i, int align):-** Creates a `JLabel` instance with the specified text, image, and horizontal alignment. The *align* argument is either **LEFT**, **RIGHT**, or

CENTER. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

- The icon and text associated with the label can be read and written by the following methods:
  1. **Icon getIcon( ):-** Returns the graphic image (glyph, icon) that the label displays.
  2. **String getText( ):-** Returns the text string that the label displays.
  3. **void setIcon(Icon i):-** Defines the icon this component will display.
  4. **void setText(String s):-** Defines the single line of text this component will display.
  5. **public void setHorizontalTextPosition**(int textPosition)**:-**Sets the horizontal position of the label's text, relative to its image. `textPosition`'s values are: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, or `TRAILING` (the default).
  6. **public void setVerticalTextPosition**(int textPosition)**:-**Sets the vertical position of the label's text, relative to its image. The default value of this property is CENTER.

- **Example-1:-**

- The following program uses various constructors of JLabel and displays three different labels on the panel. The first label is with text and icon, second is with test only and third is with icon only.
- **Source code:-**

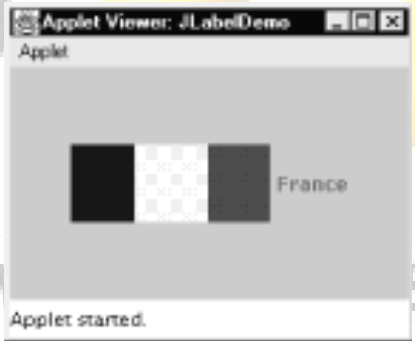
```
import java.awt.*;
import javax.swing.*;

public class JlabelDemo extends JPanel
{
   JLabel jlb1, jlb2, jlb3;
   public JlabelDemo()
   {
        ImageIcon icon = new ImageIcon("image1.jpg"); // Creating an Icon
       setLayout(new GridLayout(3, 1));
        // 3 rows, 1 column Panel having Grid Layout
        jlb1 = new JLabel("Image with Text", icon, JLabel.CENTER);
        jlb1.setVerticalTextPosition(JLabel.BOTTOM);
        jlb1.setHorizontalTextPosition(JLabel.CENTER);
        jlb2 = new JLabel("Text Only Label");
       jlb3 = new JLabel(icon); // Label of Icon Only
        // Add labels to the Panel
        add(jlb1);
        add(jlb2);
        add(jlb3);
   }
    public static void main(String[] args)
  {
       JFrame frame = new JFrame("JLabel Usage Demo");
       frame.addWindowListener(new WindowAdapter()
      {
                // Shows code to Add Window Listener
                public void windowClosing(WindowEvent e)
            {
                        System.exit(0);
                }
        });
   frame.setContentPane(new JlabelDemo());
   frame.pack();
   frame.setVisible(true);
 }
}
```

- **Example-2:-**
- The following example illustrates how to create and display a label containing both an icon and a string. The applet begins by getting its content pane. Next, an **ImageIcon** object is created for the file **france.gif**. This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.
- **Source code:-**

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JLabelDemo" width=250 height=150>
</applet>
*/
public class JLabelDemo extends JApplet
{
  public void init()
  {
    // Get content pane
     Container contentPane = getContentPane();
   // Create an icon
     ImageIcon ii = new ImageIcon("france.gif");
   // Create a label
     JLabel jl = new JLabel("France", ii, JLabel.CENTER);
   // Add label to the content pane
     contentPane.add(jl);
  }
}
```

Output from this applet is shown here:



6.42 Text Fields

- The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components.
- One of its subclasses is **JTextField**. `JTextField` is a lightweight component that allows the editing of a single line of text.
- `JTextField` components generate `ActionEvents` whenever the user presses the Return key within the field. To get these events, implement the `ActionListener` interface and register your listener using the `addActionListener()` method.
- Some of its constructors are shown here:
    1. **JTextField( ):-** Constructs a new `TextField`.

2. **JTextField(int cols):-** Constructs a new empty `TextField` with the specified number of columns.
3. **JTextField(String s, int cols):-** Constructs a new `TextField` initialized with the specified text and columns.
4. **JTextField(String s):-** Constructs a new `TextField` initialized with the specified text.

- **Method:-**

1. **public Action getAction():-**Returns the currently set Action for this ActionEvent source, or null if no Action is set.
2. **public int getHorizontalAlignment():-**Returns the horizontal alignment of the text. Valid keys are:
   `JTextField.LEFT`
   `JTextField.CENTER`
   `JTextField.RIGHT`
   `JTextField.LEADING`
   `JTextField.TRAILING`
3. **public void setHorizontalAlignment(int alignment):-**Sets the horizontal alignment of the text.
4. **public void setActionCommand(String command):-**Sets the command string used for action events.

- **Example:-**
- The following swing application which displays two text fields among which one is editable and one is disabled. And whatever text you will enter into text fields that will get display on the message box.
- **Source code:-**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class JTextFieldDemo extends JFrame
{
  JTextField jtf1, jtfText;
  String disp = "";
  //Constructor
  public JTextFieldDemo()
  {
            super("TextField Test Demo");
            Container container = getContentPane();
            container.setLayout(new FlowLayout());
            jtf1 = new JTextField(10);
            jtfText = new JTextField("Uneditable text field", 20);
            jtfText.setEditable(false);
            container.add(jtf1);
            container.add(jtfText);
            jtf1.addActionListener(this);
            jtfText.addActionListener(this);
            setSize(325,100);
            setVisible(true);
  }
  public void actionPerformed(ActionEvent e)
  {
     if (e.getSource() == jtf1)
     {
       disp = "text1 : " + e.getActionCommand();
```

```
            }
            else
            if (e.getSource() == jtfText)
            {
                disp = "text3 : " + e.getActionCommand();
            }
            JOptionPane.showMessageDialog(null, disp);
        }
        public static void main(String args[])
        {
            JTextFieldDemo test = new JTextFieldDemo();
            test.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

- **Output**

- **Example:-2 -**
  The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a **JTextField** object is created and is added to the content pane.

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet
{
JTextField jtf;
public void init()
{
 // Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add text field to content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
}
```

- Output from this applet is shown here:

6.43 JTextArea

- A JTextArea is used to accept multi-line input from the user.
- The java.awt.TextArea internally handles scrolling. JTextArea is different in that it doesn't manage scrolling, but implements the swing Scrollable interface. This allows it to be placed inside a JScrollPane if scrolling behavior is desired, and used directly if scrolling is not desired.
- The java.awt.TextArea has the ability to do line wrapping. This was controlled by the horizontal scrolling policy.
- JTextArea has a bound property for line wrapping that controls whether or not it will wrap lines. By default, the line wrapping property is set to false (not wrapped).
- java.awt.TextArea has two properties rows and columns that are used to determine the preferred size. If the value for rows or columns is equal to zero, the preferred size along that axis is used for the viewport preferred size along the same axis.
- The java.awt.TextArea could be monitored for changes by adding a TextListener for TextEvents.
- **Constructors:-**
    1. **JTextArea():-**Constructs a new TextArea.
    2. **JTextArea(Document doc)** :-Constructs a new JTextArea with the given document model, and defaults for all of the other arguments (null, 0, 0).
    3. **JTextArea(Document doc, String text, int rows, int columns) :-**Constructs a new JTextArea with the specified number of rows and columns, and the given model.
    4. **JTextArea(int rows, int columns):-**Constructs a new empty TextArea with the specified number of rows and columns.
    5. **JTextArea(String text):-**Constructs a new TextArea with the specified text displayed.
    6. **JTextArea(String text, int rows, int columns):-**Constructs a new TextArea with the specified text and number of rows and columns.

6.44 Buttons

- Swing buttons provide features that are not found in the **Button** class defined by the AWT.
- For example, you can associate an icon with a Swing button. Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent**. **AbstractButton** contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons.
- For example, you can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over that component.
- The following are the methods that control this behavior:
    1. **void setDisabledIcon(Icon di):-**Sets the disabled icon for the button.
    2. **void setPressedIcon(Icon pi):-** Sets the pressed icon for the button.
    3. **void setSelectedIcon(Icon si):-** Sets the selected icon for the button.
    4. **void setRolloverIcon(Icon ri):-** Sets the rollover icon for the button
- The text associated with a button can be read and written via the following methods:
    1. **String getText( ):-** Returns the button's text.
    2. **void setText(String s):-** Sets the button's text.

- Concrete subclasses of **AbstractButton** generate action events when they are pressed. Listeners register and unregistered for these events via the methods shown here:
  void addActionListener(ActionListener al)
  void removeActionListener(ActionListener al)
  Here, *al* is the action listener.

6.45 The JButton Class

The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button.

- Some of its constructors are shown here:
  1. **`JButton`** ( ) :-Creates a button with no set text or icon.
  2. **JButton(Icon i):-** Creates a button with an icon.
  3. **JButton(String s):-** Creates a button with text.
  4. **JButton(String s, Icon i):-** Creates a button with initial text and an icon.

Example:-1 -The following program helps you to change the label of the button. In this program, addActionListener() method has been added to the button to register the action listener and then if you click on the button, the generated action event is captured in the actionPerformed(ActionEvent e) method. In the actionPerformed(ActionEvent e) we check the label of the button. If the label of the button is "TYIT", it will change the button label to "Click Me" otherwise it will change the label to "TYIT".

Source code:-

```
import javax.swing.*;
import java.awt.event.*;

public class ChangeButtonLabel
{
        JButton button;
        public static void main(String[] args)
        {
                ChangeButtonLabel cl = new ChangeButtonLabel();
        }
        public ChangeButtonLabel()
        {
                JFrame frame = new JFrame("Change JButton Lebel");
                button = new JButton("Click Me");
                button.addActionListener(this);
                frame.add(button);
                frame.setSize(400, 400);
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }

        public void actionPerformed(ActionEvent e)
                {
                        String text = (String)e.getActionCommand();
                        if (text.equals("Click Me"))
                        {
                                button.setText("TYIT");
                        }
```

```
                    else
                    {
                        button.setText("Click Me");
                    }
                }
            }
```

- **Example-2:-**
- The following example displays four push buttons. First button displays only text and other three buttons display text along with the icon.

```
import java.awt.*;
import javax.swing.*;
public class JButtons extends JFrame {
public static void main(String[] args) {
new JButtons();
}
public JButtons() {
super("Using JButton");
//WindowUtilities.setNativeLookAndFeel();
//addWindowListener(new ExitListener());
Container content = getContentPane();
content.setBackground(Color.white);
content.setLayout(new FlowLayout());
JButton button1 = new JButton("Java");
content.add(button1);
ImageIcon cup = new ImageIcon("iso.gif");
JButton button2 = new JButton(cup);
content.add(button2);
JButton button3 = new JButton("Java", cup);
content.add(button3);
JButton button4 = new JButton("Java", cup);
button4.setHorizontalTextPosition
(SwingConstants.LEFT);
content.add(button4);
pack();
setVisible(true);
}
}
```

6.46 Check Boxes

- The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**.
- This is an implementation of a check box, which is an item that can be selected or deselected, and which displays its state to the user.
- Some of its constructors are shown here:
    1. **JcheckBox(Icon i):-** Creates an initially unselected check box with an icon.
    2. **JcheckBox(Icon I, □oolean state):-** Creates a check box with an icon and specifies whether or not it is initially selected.
    3. **JcheckBox(String s):-** Creates an initially unselected check box with text.
    4. **JcheckBox(String s, □oolean state):-** Creates a check box with text and specifies whether or not it is initially selected.
    5. **JcheckBox(String s, Icon i):-** Creates an initially unselected check box with the specified text and icon.
    6. **JcheckBox(String s, Icon I, □oolean state):-** Creates a check box with text and icon, and specifies whether or not it is initially selected.

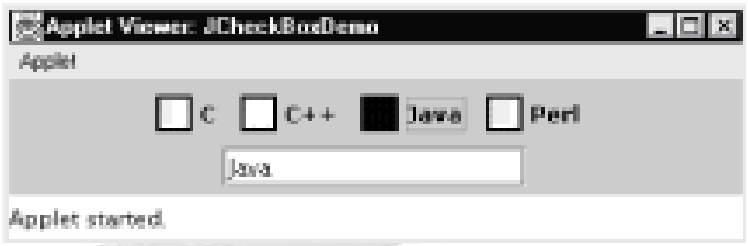
- The state of the check box can be changed via the following method:
    1. **void setSelected(□oolean state):-** Sets the state of the button. Here, *state* is **true** if the check box should be checked.

- **Example:-** The following example illustrates how to create an applet that displays four check boxes and a text field. When a check box is pressed, its text is displayed in the text field. When a check box is selected or deselected, an item event is generated. This is handled by **itemStateChanged( )**. Inside **itemStateChanged( )**, the **getItem( )** method gets the **JcheckBox** object that generated the event. The **getText( )** method gets the text for that check box and uses it to set the text inside the text field.

```
Import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JcheckBoxDemo" width=400 height=50>
</applet>
*/
public class JcheckBoxDemo extends Japplet
implements ItemListener
{
JtextField jtf;
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create icons
ImageIcon normal = new ImageIcon("normal.gif");
ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane
JcheckBox cb = new JcheckBox("C", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JcheckBox("C++", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JcheckBox("Java", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JcheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JtextField(15);
contentPane.add(jtf);
}
```

```
public void itemStateChanged(ItemEvent ie)
{
JcheckBox cb = (JcheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```

- Output from this applet is shown here:



6.47 adio Buttons

Radio buttons are supported by the **JradioButton** class, which is a concrete implementation of **AbstractButton**.

- Some of its constructors are shown here:
  1. **JradioButton(Icon i):-** Creates an initially unselected radio button with the specified image but no text.
  2. **JradioButton(Icon I, Boolean state):-** Creates a radio button with the specified image and selection state, but no text.
  3. **JradioButton(String s):-** Creates an unselected radio button with the specified text.
  4. **JradioButton(String s, oolean state):-** Creates a radio button with the specified text and selection state.
  5. **JradioButton(String s, Icon i):-** Creates a radio button that has the specified text and image, and that is initially unselected.
  6. **JradioButton(String s, Icon I, oolean state):-** Creates a radio button that has the specified text, image, and selection state.
- Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time.
- The **ButtonGroup** class is instantiated to create a button group. Elements are then added to the button group via the following method:

  void add(AbstractButton ab)

  Here, *ab* is a reference to the button to be added to the group.

Example-1 :-

The following swing application displays three radio buttons and on clicking on that radio button the respective image will display in label.

Source code:-

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class JRadioButtonDemo extends JPanel
{
   static JFrame frame;
   JLabel jlbPicture;
   public JRadioButtonDemo()
  {
                // Create the radio buttons and assign Keyboard shortcuts using
                // Mnemonics
```

```
                JRadioButton jrbNum = new JRadioButton("Numbers");
                jrbNum.setMnemonic(KeyEvent.VK_N);
                jrbNum.setActionCommand("numbers");
                jrbNum.setSelected(true);
                JRadioButton jrbAlph = new JRadioButton("Alphabets");
                jrbAlpha.setMnemonic(KeyEvent.VK_A);
                jrbAlpha.setActionCommand("alphabets");
                JRadioButton jrbSymb = new JRadioButton("Symbols");
                jrbSymb.setMnemonic(KeyEvent.VK_S);
                jrbSymb.setActionCommand("symbols");
                // Group the radio buttons.
                ButtonGroup group = new ButtonGroup();
                group.add(jrbNum);
                group.add(jrbAlpha);
                group.add(jrbSymb);
                // Register an action listener for the radio buttons.
                jrbNum.addActionListener(this);
                jrbAlpha.addActionListener(this);
                jrbSymb.addActionListener(this);
                // Set up the picture label
                jlbPicture = new JLabel(new ImageIcon("" + "numbers" + ".jpg"));
                // Set the Default Image
                jlbPicture.setPreferredSize(new Dimension(177, 122));
                // Put the radio buttons in a column in a panel
                JPanel jplRadio = new JPanel();
                jplRadio.setLayout(new GridLayout(0, 1));
                jplRadio.add(jrbNum);
                jplRadio.add(jrbAlpha);
                jplRadio.add(jrbSymb);
                setLayout(new BorderLayout());
                add(jplRadio, BorderLayout.WEST);
                add(jlbPicture, BorderLayout.CENTER);
                setBorder(BorderFactory.createEmptyBorder(20, 20, 20, 20));
    }
    public void actionPerformed(ActionEvent e)
    {
                jlbPicture.setIcon(new ImageIcon("" + e.getActionCommand()
                                                    + ".jpg"));
    }
    public static void main(String s[])
    {
                frame = new JFrame("JRadioButton Usage Demo");
                frame.addWindowListener(new WindowAdapter()
                {
                    public void windowClosing(WindowEvent e)
                {
                            System.exit(0);
                    }
                  });
           frame.getContentPane().add(new JRadioButtonDemo(),
           BorderLayout.CENTER);
           frame.pack();
           frame.setVisible(true);
    }
}
```

- **Example-2:-** The following example displays three radio buttons and one text field are created. When a radio button is pressed, its text is displayed in the text field. Radio button presses generate action events that are handled by **actionPerformed( )**. The **getActionCommand( )** method gets the text that is associated with a radio button and uses it to set the text field.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet
implements ActionListener
{
JTextField tf;
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add radio buttons to content pane
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
contentPane.add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
contentPane.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);
// Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae)
{
tf.setText(ae.getActionCommand());
}
}
```


- Output from this applet is shown here:

6.48 Container

- A container is a kind of component that holds and manages other components. `JComponent` objects can be containers because the `JComponent` class descends from the `Container` class. However, you wouldn't normally add components directly to specialized components such as buttons or lists.
- Three of the most useful general container types are `JFrame`, `JPanel`, and `JApplet`.
- A `JFrame` is a top-level window on your display. `JFrame` is derived from `JWindow`, which is pretty much the same but lacks a border.
- A `JPanel` is a generic container element that groups components inside `JFrames` and other `JPanels`.
- The `JApplet` class is a kind of container that provides the foundation for applets that run inside web browsers. Like other `JComponents`, a `JApplet` can contain other user-interface components.
- You can also use the `JComponent` class directly, like a `JPanel`, to hold components inside another container. With the exception of `JFrame` and `JWindow`, all the components and containers in Swing are lightweight.
- The `add()` method of the `Container` class adds a component to the container. Thereafter, this component can be displayed in the container's display area and positioned by its layout manager. You can remove a component from a container with the `remove()` method.

Windows and Frames

- Windows and frames are the top-level containers for Java components.
- A JWindow is simply a plain, graphical screen that displays in your windowing system. Windows are mainly suitable for making "splash" screens and pop-up windows.
- JFrame, on the other hand, is a subclass of JWindow that has a border and can hold a menu bar. You can drag a frame around on the screen and resize it, using the ordinary controls for your windowing environment.
- All other Swing components and containers must be held, at some level, inside a JWindow or JFrame.
- JFrames and JWindows are the only components that can be displayed without being added or attached to another Container. After creating a JFrame or JWindow, you can call the setVisible() method to display it.
- **Crating a JFrame Window**
  - Step 1: Construct an object of the JFrame class.
  - Step 2: Set the size of the Jframe.
  - Step 3: Set the title of the Jframe to appear in the title bar (title bar will be blank if no title is set).
  - Step 4: Set the default close operation. When the user clicks the close button, the program stops running.
  - Step 5: Make the Jframe visible.
- **Constructors of JFrame:-**
  - **JFrame()**:-Constructs a new frame that is initially invisible.
  - **JFrame(String title):-**Creates a new, initially invisible Frame with the specified title.
  - **JFrame(String title, GraphicsConfiguration gc):-**Creates a JFrame with the specified title and the specified GraphicsConfiguration of a screen device.
- **Example:-**

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class JFrameDemo
{
   public static void main(String s[])
   {
      JFrame frame = new JFrame("JFrame Source Demo");
      frame.addWindowListener(new WindowAdapter()
{
   public void windowClosing(WindowEvent e)
   {
      System.exit(0);
   }
});
// This is label to add it on the frame
JLabel jlb = new JLabel("Hello");
jlb.setPreferredSize(new Dimension(175, 100));
frame.getContentPane().add(jlb, BorderLayout.CENTER);
frame.pack();
frame.setVisible(true);
   }
}
```

- **Constructors of JWindow:-**
  - **JWindow**() :-Creates a window with no specified owner.
  - **JWindow**(Frame owner):-Creates a window with the specified owner frame.
  - **JWindow**(Window owner):-Creates a window with the specified owner window.
  - **JWindow**(Window owner, GraphicsConfiguration gc):-Creates a window with the specified owner window and GraphicsConfiguration of a screen device
  - **Example:-**

```
import javax.swing.*;

  public class TopTest
{
 public static void main(String[] args)
  {
    JFrame frame = new JFrame("The Frame");
    frame.setSize(300, 300);
    frame.setLocation(100, 100);

    JWindow window = new JWindow( );
    window.setSize(300, 300);
    window.setLocation(500, 100);

    frame.setVisible(true);
    window.setVisible(true);
  }
}
```

- The JFrame constructor can take a String argument that supplies a title, displayed in the JFrame's titlebar. After creating the JFrame, we create a JWindow in almost exactly the same way. The JWindow doesn't have a titlebar, so there are no arguments to the JWindow constructor.

Using Content Panes

- Windows and frames don't behave exactly like regular containers. With other containers, you can add child components with the add() method.
- JFrame and JWindow have some extra stuff in them (mostly to support Swing's peerless components), so you can't just add() components directly. Instead, you need to add the components to the associated content pane.
- The content pane is just a Container that covers the visible area of the JFrame or JWindow. Whenever you create a new JFrame or JWindow, a content pane is automatically created for you. You can retrieve it with getContentPane().
- Here's another example that creates a JFrame and adds some components to its content pane:

```
import java.awt.*;
import javax.swing.*;

public class Test11
{
   public static void main(String[] args)
  {
   JFrame frame = new JFrame("The Frame");
   frame.setLocation(100, 100);

   Container content = frame.getContentPane( );
   content.setLayout(new FlowLayout( ));
   content.add(new JLabel("Label1"));
   content.add(new JButton("Label2"));

   frame.pack( );
   frame.setVisible(true);
  }
}
```

4.49 JPanel

- The `JPanel` class provides general-purpose containers for lightweight components.
- By default, panels don't paint anything except for their background; however, you can easily add borders to them and otherwise customize their painting.
- In many look and feels, panels are opaque by default. Opaque panels work well as content panes and can help painting efficiency.
- You can change a panel's transparency by invoking `setOpaque`. A transparent panel draws no background.

- **Example:-**

```
public class Paneldemo extends JPanel
{
public Paneldemo(String title, String[] buttonLabels)
{
super(new GridLayout(3, 2));
setBackground(Color.lightGray);
setBorder(BorderFactory.createTitledBorder(title));
ButtonGroup group = new ButtonGroup();
JRadioButton option;
int halfLength = buttonLabels.length/2;
for(int i=0; i<halfLength; i++) {
option = new JRadioButton(buttonLabels[i]);
group.add(option);
```

```
add(option);
option = new JRadioButton(buttonLabels[i+halfLength]);
group.add(option);
add(option);
}
}
}
```

6.50 Combo Boxes

- Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.
- A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry.
- Two of **JComboBox**'s constructors are shown here:

  JComboBox( )

  JComboBox(Vector v)

  Here, *v* is a vector that initializes the combo box.
- Items are added to the list of choices via the **addItem( )** method, whose signature is shown here:

  void addItem (Object ob)

  Here, *obj* is the object to be added to the combo box.
- The following example contains a combo box and a label. The label displays an icon.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet
implements ItemListener
{
JLabel jl;
ImageIcon france, germany, italy, japan;
public void init()
{
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
JComboBox jc = new JComboBox();
jc.addItem("Mumbai");
jc.addItem("Goa");
jc.addItem("Bihar");
jc.addItem("Gujrat");
jc.addItemListener(this);
contentPane.add(jc);
jl = new JLabel(new ImageIcon("Mumbai.gif"));
contentPane.add(jl);
}
public void itemStateChanged(ItemEvent ie)
{
String s = (String)ie.getItem();
jl.setIcon(new ImageIcon(s + ".gif"));
}
}
```

6.50 Tabbed Panes

- A **tabbed pane** is a component that appears as a group of folders in a file cabinet. Each folder has a title.
- Tabbed panes are commonly used for setting configuration options.
- Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**.
- **Constructor:**
    - **JTabbedPane**() :-Creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP
    - **JTabbedPane(int tabPlacement) :-**Creates an empty TabbedPane with the specified tab placement of either: JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT, or JTabbedPane.RIGHT.
    - **JTabbedPane(int tabPlacement, int tabLayoutPolicy**):-Creates an empty TabbedPane with the specified tab placement and tab layout policy.
- Tabs are defined via the following method:
    - **void addTab(String str, Component comp):-** Adds a component represented by a title and no icon.

      Parameters:
        - **title -** the title to be displayed in this tab
        - **component -** the component to be displayed when this tab is clicked
- The general procedure to use a tabbed pane in an applet is outlined here:
    1. Create a **JTabbedPane** object.
    2. Call **addTab( )** to add a tab to the pane.
    3. Repeat step 2 for each tab.
    4. Add the tabbed pane to the content pane of the applet.
- **Example:-**
- The following example illustrates how to create a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color. The third tab is titled "Flavors" and contains one combo box. This enables the user to select one of three flavors.

- **Source code:-**

```
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400 height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet
{
public void init()
{
JTabbedPane jtp = new JTabbedPane();
jtp.addTab("Cities", new CitiesPanel());
jtp.addTab("Colors", new ColorsPanel());
jtp.addTab("Flavors", new FlavorsPanel());
getContentPane().add(jtp);
}
}
class CitiesPanel extends JPanel
{
public CitiesPanel()
{
JButton b1 = new JButton("New York");
add(b1);
JButton b2 = new JButton("London");
```

```
add(b2);
JButton b3 = new JButton("Hong Kong");
add(b3);
JButton b4 = new JButton("Tokyo");
add(b4);
}
}
class ColorsPanel extends JPanel
{
public ColorsPanel()
{
JCheckBox cb1 = new JCheckBox("Red");
add(cb1);
JCheckBox cb2 = new JCheckBox("Green");
add(cb2);
JCheckBox cb3 = new JCheckBox("Blue");
add(cb3);
}
}
class FlavorsPanel extends JPanel
{
public FlavorsPanel()
{
JComboBox jcb = new JComboBox();
jcb.addItem("Vanilla");
jcb.addItem("Chocolate");
jcb.addItem("Strawberry");
add(jcb);
}
}
```

- **Output is:**

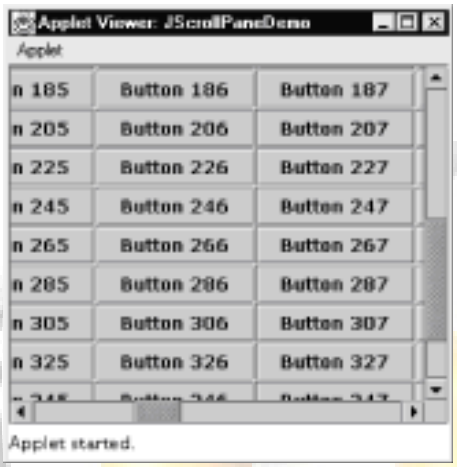
6.51 Scroll Panes

- A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary.
- Scrollpanes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**.
- Some of its constructors are shown here:
    1. **JScrollPane(Component comp):-** Creates a `JScrollPane` that displays the contents of the specified component, where both horizontal and vertical scrollbars appear whenever the component's contents are larger than the view.
    2. **JScrollPane(int vsb, int hsb):-** Creates an empty (no viewport view) `JScrollPane` with specified scrollbar constants.
    3. **JScrollPane(Component comp, int vsb, int hsb):-** Creates a `JScrollPane` that displays the view component in a viewport whose view position can be controlled with a pair of scrollbars.
- These constants are defined by the **ScrollPaneConstants** interface. Some examples of these constants are described as follows:
    1. HORIZONTAL_SCROLLBAR_ALWAYS:- Always provide horizontal scroll bar
    2. HORIZONTAL_SCROLLBAR_AS_NEEDED:- Provide horizontal scroll bar, if needed
    3. VERTICAL_SCROLLBAR_ALWAYS:- Always provide vertical scroll bar
    4. VERTICAL_SCROLLBAR_AS_NEEDED:-Provide vertical scroll bar, if needed
- Following steps that you should follow to use a scroll pane in an applet:
    1. Create a **JComponent** object.
    2. Create a **JScrollPane** object.
    3. Add the scroll pane to the content pane of the applet.
- **Example:-**
- The following example illustrates a scroll pane. First, the content pane of the **JApplet** object is obtained and a border layout is assigned as its layout manager. Next, a **JPanel** object is created and four hundred buttons are added to it, arranged into twenty columns. The panel is then added to a scroll pane, and the scroll pane is added to the content pane. This causes vertical and horizontal scroll bars to appear. You can use the scrollbars to scroll the buttons into view.
- **Source code:-**

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JScrollPaneDemo" width=300 height=250>
</applet>
*/
public class JScrollPaneDemo extends JApplet
{
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());
// Add 400 buttons to a panel
JPanel jp = new JPanel();
jp.setLayout(new GridLayout(20, 20));
int b = 0;
for(int i = 0; i < 20; i++)
{
for(int j = 0; j < 20; j++)
{
jp.add(new JButton("Button " + b));
++b;
}
}
```

```
// Add panel to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(jp, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

- Output from this applet is shown here:



6.52 Tables

- A *table* is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position.
- Tables are implemented by the **JTable** class, which extends **JComponent**.
- Constructors
  - **JTable**():-Constructs a default JTable that is initialized with a default data model, a default column model, and a default selection model.
  - **JTable(Object data[ ][ ], Object colHeads[ ]):-** Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.
  - **JTable(int numRows, int numColumns):-**Constructs a JTable with numRows and numColumns of empty cells using DefaultTableModel.
  - **`JTable(Vector rowData, Vector columnNames)`:-**Constructs a `JTable` to display the values in the `Vector` of `Vectors`, `rowData`, with column names, `columnNames`.
- Here are the steps for using a table in an applet:
  1. Create a **JTable** object.
  2. Create a **JScrollPane** object.
  3. Add the table to the scroll pane.
  4. Add the scroll pane to the content pane of the applet.
- **Example:-**
- The following example illustrates how to create and use a table. The content pane of the **JApplet** object is obtained and a border layout is assigned as its layout manager. This table has three columns. A two-dimensional array of strings is created for the table cells. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane and then the scroll pane is added to the content pane.
- **Source code:-**

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTableDemo" width=400 height=200>
```

```
</applet>
*/
public class JTableDemo extends JApplet
{
public void init()
{
// Get content pane
Container contentPane = getContentPane();
// Set layout manager
contentPane.setLayout(new BorderLayout());
// Initialize column headings
final String[] colHeads = { "Name", "Roll No", "Address", "Percentage" };
// Initialize data
final Object[][] data = {
{ "Ramesh Shetty","1","kalina","67%" },
{ "Kapil Varma","2","bandra","45%" },
{ "Ritu Singh", "3","malad","56%" },
{ "Amita Jadhav","4","miraroad","89%" },
{ "Sumit Sahani", "5","mahim","79%" },

};
// Create the table
JTable table = new JTable(data, colHeads);
// Add tree to a scroll pane
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);
// Add scroll pane to the content pane
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

6.53 Trees

- One of Swing's advanced components is `JTree`. Trees are good for representing hierarchical information, like the contents of a disk drive or a company's organizational chart.
- The classes that support `JTree` have their own package, `javax.swing.tree`.
- A tree's data model is made up of interconnected nodes. A node has a name, typically, a parent, and some number of children (possibly 0).
- In Swing, a node is represented by the `TreeNode` interface. Nodes that can be modified are represented by `MutableTreeNode`. A concrete implementation of this interface is `DefaultMutableTreeNode`. One node, called the root node, usually resides at the top of the hierarchy.
- A tree's data model is represented by the `TreeModel` interface. Swing provides an implementation of this interface called `DefaultTreeModel`. You can create a `DefaultTreeModel` by passing a root `TreeNode` to its constructor.
- You could create a `TreeMdel` with just one node like this:

  ```
  TreeNode root = new DefaultMutableTreeNode("Root node");
  TreeModel model = new DefaultTreeModel(root);
  ```
- Here's another example with a real hierarchy. The root node contains two nodes, Node 1 and Group. The Group node contains Node 2 and Node 3 as subnodes.

  ```
  MutableTreeNode root = new DefaultMutableTreeNode("Root node");
  MutableTreeNode group = new DefaultMutableTreeNode("Group");
  ```

```
root.insert(group, 0);
root.insert(new DefaultMutableTreeNode("Node 1"), 1);
group.insert(new DefaultMutableTreeNode("Node 2"), 0);
group.insert(new DefaultMutableTreeNode("Node 3"), 1);
```

- The second parameter to the `insert()` method is the index of the node in the parent. Once you've got your nodes organized, you can create a `TreeModel` in the same way as before:

```
TreeModel model = new DefaultTreeModel(root);
```

- Once you have a tree model, creating a `JTree` is simple:

```
JTree tree = new JTree(model);
```

- **Tree Events**
  - A tree fires several events. You can find out when nodes have been expanded and collapsed, and when selections occur. Two distinct event listener interfaces handle this information.

TreeExpansionListener
TreeSelectionListener

- The following example registers an event listener that prints out the last selected node:

```
tree.addTreeSelectionListener(new TreeSelectionListener( ) {
  public void valueChanged(TreeSelectionEvent e) {
    TreePath tp = e.getNewLeadSelectionPath( );
    System.out.println(tp.getLastPathComponent( ));
  }
});
```

- **Constructors:-**
  - **`JTree()`:-**Returns a `JTree` with a sample model.
  - **`JTree(TreeModel newModel)`**:-Returns an instance of `JTree` which displays the root node -- the tree is created using the specified data model.
  - **`JTree(TreeNode root)`:-**Returns a `JTree` with the specified `TreeNode` as its root, which displays the root node.
  - **`JTree(Object [] value)`:-** Returns a `JTree` with each element of the specified array as the child of a new root node which is not displayed.
- **Interfaces** associated with the trees are:-
  - **`MutableTreeNode`:-**Defines the requirements for a tree node object that can change -- by adding or removing child nodes, or by changing the contents of a user object stored in the node.
  - **TreeExpansionListener**:- The listener that's notified when a tree expands or collapses a node.
- **Event class:-**
  - **TreeExpansionEvent**:-An event used to identify a single path in a tree. The source returned by **getSource** will be an instance of JTree.
- **Other Classes** associated with the trees are:-
  - **`DefaultMutableTreeNode` :-**DefaultMutableTreeNode implements the MutableTreeNode interface A `DefaultMutableTreeNode` is a general-purpose node in a tree data structure. A tree node may have at most one parent and 0 or more children. `DefaultMutableTreeNode` provides operations for examining and modifying a node's parent and children and also operations for examining the tree that the node is a part of.
  - **TreeModel:-** A simple tree data model that uses TreeNodes

- **TreePath:-** Represents a path to a node. A TreePath is an array of Objects that are vended from a TreeModel. The elements of the array are ordered such that the root is always the first element (index 0) of the array.

- **Methods associated with trees are:-**
  - **public void add(MutableTreeNode newChild):-**Removes newChild from its parent and makes it a child of this node by adding it to the end of this node's child array.
  - **public TreeNode[] getPath():-**Returns the path from the root, to get to this node. The last element in the path is this node.
  - **public Object[] getPath():-**Returns an ordered array of Objects containing the components of this TreePath. The first element (index 0) is the root.
  - **public void insert(MutableTreeNode newChild, int childIndex):-**Removes newChild from its present parent (if it has a parent), sets the child's parent to this node, and then adds the child to this node's child array at index childIndex. newChild must not be null and must not be an ancestor of this node.
- Here are the steps that you should follow to use a tree:-
  - Create a JTree object.
  - Create a JScrollPane Object.
  - Add the tree to the scroll panes.
  - Add the scroll to the content pane of the applet.
- **Example:-**
- The following swing application displays a tree structure of subjects in a department of information technology.
- **Source code:-**

```
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
public class JTreeExample extends JFrame
{
  JTreeExample()
  {
   MutableTreeNode it = new DefaultMutableTreeNode("IT Department");
   MutableTreeNode fyit = new DefaultMutableTreeNode("FYIT");
   MutableTreeNode syit = new DefaultMutableTreeNode("SYIT");
   it.insert(fyit,0);
   it.insert(syit,1);
   fyit.insert(new DefaultMutableTreeNode("Maths-2"), 0);
   fyit.insert(new DefaultMutableTreeNode("DAA"), 1);
   fyit.insert(new DefaultMutableTreeNode("ETC"), 2);
   fyit.insert(new DefaultMutableTreeNode("PSD"), 3);
   fyit.insert(new DefaultMutableTreeNode("CG"), 4);
   syit.insert(new DefaultMutableTreeNode("DBMS"), 1);
   syit.insert(new DefaultMutableTreeNode("OS"), 2);
   syit.insert(new DefaultMutableTreeNode("C++,JAVA"), 3);
   syit.insert(new DefaultMutableTreeNode("EComm"), 4);
   syit.insert(new DefaultMutableTreeNode("SE"), 5);

   DefaultTreeModel model = new DefaultTreeModel(it);
   JTree tree = new JTree(model);
   getContentPane().add(tree);
    setSize(300,300);
   setVisible(true);
}
public static void main(String[] args)
{
```

```
            new JTreeExample();
        }
    }
```

- This section contains an example that showcases the following tree techniques:

    - Construction of a tree model, using `DefaultMutableTreeNode`.
    - Creation and display of a `JTree`
    - Listening for tree selection events
    - Modifying the tree's data model while the `JTree` is showing

- **Source code**:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

public class PartsTree
{
  public static void main(String[] args)
  {
    // create a hierarchy of nodes
    MutableTreeNode root = new DefaultMutableTreeNode("Parts");
    MutableTreeNode beams = new DefaultMutableTreeNode("Beams");
    MutableTreeNode gears = new DefaultMutableTreeNode("Gears");
    root.insert(beams, 0);
    root.insert(gears, 1);
    beams.insert(new DefaultMutableTreeNode("1x4 black"), 0);
    beams.insert(new DefaultMutableTreeNode("1x6 black"), 1);
    beams.insert(new DefaultMutableTreeNode("1x8 black"), 2);
    beams.insert(new DefaultMutableTreeNode("1x12 black"), 3);
    gears.insert(new DefaultMutableTreeNode("8t"), 0);
    gears.insert(new DefaultMutableTreeNode("24t"), 1);
    gears.insert(new DefaultMutableTreeNode("40t"), 2);
    gears.insert(new DefaultMutableTreeNode("worm"), 3);
    gears.insert(new DefaultMutableTreeNode("crown"), 4);

    // create the JTree
    final DefaultTreeModel model = new DefaultTreeModel(root);
    final JTree tree = new JTree(model);

    // create a text field and button to modify the data model
    final JTextField nameField = new JTextField("16t");
    final JButton button = new JButton("Add a part");
    button.setEnabled(false);
    button.addActionListener(new ActionListener( )
    {
      public void actionPerformed(ActionEvent e)
      {
        TreePath tp = tree.getSelectionPath( );
        MutableTreeNode insertNode
        (MutableTreeNode)tp.getLastPathComponent( );
```

```
            int insertIndex = 0;
            if (insertNode.getParent( ) != null)
            {
              MutableTreeNode parent =
                  (MutableTreeNode)insertNode.getParent( );
              insertIndex = parent.getIndex(insertNode) + 1;
              insertNode = parent;
            }
            MutableTreeNode node =
            new DefaultMutableTreeNode(nameField.getText( ));
            model.insertNodeInto(node, insertNode, insertIndex);
          }
        });
        JPanel addPanel = new JPanel(new GridLayout(2, 1));
        addPanel.add(nameField);
        addPanel.add(button);

        // listen for selections
        tree.addTreeSelectionListener(new TreeSelectionListener( )
        {
          public void valueChanged(TreeSelectionEvent e)
          {
            TreePath tp = e.getNewLeadSelectionPath();
            button.setEnabled(tp != null);
          }
        });

        // create a JFrame to hold the tree
        JFrame frame = new JFrame("PartsTree v1.0");
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        frame.setSize(200, 200);
        frame.getContentPane( ).add(new JScrollPane(tree));
        frame.getContentPane( ).add(addPanel, BorderLayout.SOUTH);
        frame.setVisible(true);
      }
    }
```

6.54 Custom Rendering of JList Cells

JList

- JList is a component that allows the user to select one or more objects from a list. A separate model, `ListModel`, represents the contents of the list.
- It's easy to display an array or vector of objects, using a `JList` constructor that builds a `ListModel` instance for you.
- Build JList:
    - Create a JList that displays the strings in data[]

```
String[] data = {"one", "two", "three", "four"};

JList dataList = new JList(data);
```

- JList doesn't support scrolling directly. To create a scrolling list you make the JList the viewport view of a JScrollPane. For example:

```
JScrollPane scrollPane = new JScrollPane(dataList);
```

Custom Rendering of JList Cells

- Every element within JList is called cell.
- Every Jlist has an installed cell renderer that draws every cell when the Jlist needs to be drawn.
- **DefaultListCellRenderer** is subclass of JLabel which means you can use either text or an icon as the graphical description for the cell.
- Because every JList can have atmost one renderer installed, customization requires you to replace the existing renderder.
- When its time to draw each cell, Jlist uses the rendering to draw the element.
- **JList** uses a java.awt.Component, provided by a delegate called the cellRendere, to paint the visible cells in the list.
- The cell renderer component is used like a "rubber stamp" to paint each visible row.
- Each time the JList needs to paint a cell it asks the cell renderer for the component, moves it into place using setBounds() and then draws it by calling its paint method.
- The default cell renderer uses a JLabel component to render the string value of each component. You can substitute your own cell renderer, using code like this:

```
// Imports
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class RenderListExample extends          JFrame
 {
        // Instance attributes used in this example
        private JPanel topPanel;
        private JList listbox;

        // Constructor of main frame
        public RenderListExample()
        {
                // Set the frame characteristics
                setTitle( "Rendered ListBox Application" );
                setSize( 300, 160 );
                setBackground( Color.gray );

                // Create a panel to hold all other components
                topPanel = new JPanel();
                topPanel.setLayout( new BorderLayout() );
                getContentPane().add( topPanel );

                // Create some items to add to the list
                String listData[] =
                {
                        "Circle",
                        "Bubbles",
                        "Thatch",
                        "Pinstripes"
                };
```

```
                // Create a new listbox control
                listbox = new JList( listData );
                listbox.setCellRenderer( new CustomCellRenderer() );
                topPanel.add( listbox, BorderLayout.CENTER );
        }

        // Main entry point for this example
        public static void main( String args[] )
        {
                // Create an instance of the test application
                RenderListExample mainFrame    = new RenderListExample();
                mainFrame.setVisible( true );
        }
}
// Display an icon and a string for each object in the list.
 class CustomCellRenderer extends JLabel implements ListCellRenderer
 {
    final static ImageIcon longIcon = new ImageIcon("iso.gif");
    final static ImageIcon shortIcon = new ImageIcon("aa.gif");

    // This is the only method defined by ListCellRenderer.
    // We just reconfigure the JLabel each time we're called.

    public Component getListCellRendererComponent(
      JList list,
      Object value,            // value to display
      int index,               // cell index
      boolean isSelected,      // is the cell selected
      boolean cellHasFocus)    // the list and the cell have the focus
     {
      String s = value.toString();
      setText(s);
      setIcon((s.length() > 7) ? longIcon : shortIcon);
        if (isSelected)
        {
            setBackground(list.getSelectionBackground());
            setForeground(list.getSelectionForeground());
        }
       else
       {
            setBackground(list.getBackground());
            setForeground(list.getForeground());
        }
       setEnabled(list.isEnabled());
       setFont(list.getFont());
      setOpaque(true);
      return this;
    }
 }
```


Example of custom cell rendering

- The RenderListExample.java and CustomCellRenderer.java classes make up an application that implements a simple list like the example which also provides provides a custom cell renderer, which has the responsibility of drawing each element in the list, as shown above:

- Items can be drawn differently depending on whether or not they are selected, including the foreground and background colors and the font.
- This example shown here displays selected item in a 24-point font and changes the background color of the selection to red.
- This example includes images for each item to help the user better identify the style associated with each selection, greatly improving the user experience with this application.

- The main points of the application are:

  - The CustomCellRenderer class first loads the required images in its constructor, then waits for the list box to request a rendering operation.
  - The getListCellRenderer() method performs the actual rendering, determining the colors and font required to display the item.

- Additionally, this method assigns the correct image to the item being drawn.
- That the above example extended a JLabel class to custom render the list items. Other components can be used in place of the JLabel ranging JButton, JCheckBox through to JTextArea.
- The code listing for the RenderListExample.java class is as follows:

```
// Imports
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class RenderListExample extends JFrame
 {
        // Instance attributes used in this example
        private JPanel topPanel;
        private JList listbox;

        // Constructor of main frame
        public RenderListExample()
        {
                // Set the frame characteristics
                setTitle( "Rendered ListBox Application" );
                setSize( 300, 160 );
                setBackground( Color.gray );

                // Create a panel to hold all other components
                topPanel = new JPanel();
                topPanel.setLayout( new BorderLayout() );
                getContentPane().add( topPanel );

                // Create some items to add to the list
                String      listData[] =
                {
                        "Circle",
                        "Bubbles",
                        "Thatch",
                        "Pinstripes"
                };

                // Create a new listbox control
                listbox = new JList( listData );
```

```
                    listbox.setCellRenderer( new CustomCellRenderer() );
                    topPanel.add( listbox, BorderLayout.CENTER );
                }

                // Main entry point for this example
                public static void main( String args[] )
                {
                    // Create an instance of the test application
                    RenderListExample mainFrame    = new RenderListExample();
                    mainFrame.setVisible( true );
                }
        }
```

// The code listing for the CustomCellRenderer.java class is as follows:

```
        // Imports
        import java.awt.*;
        import javax.swing.*;
        import javax.swing.ListCellRenderer.*;

        class CustomCellRenderer extends JLabel implements ListCellRenderer
        {
                private ImageIcon image[];

                public CustomCellRenderer()
                {
                    setOpaque(true);

                    // Pre-load the graphics images to save time
                    image = new ImageIcon[4];
                    image[0] = new ImageIcon( "circles.gif" );
                    image[1] = new ImageIcon( "bubbles.gif" );
                    image[2] = new ImageIcon( "thatch.gif" );
                    image[3] = new ImageIcon( "pinstripe.gif" );
                }

                public Component getListCellRendererComponent(
                                        JList list, Object value, int index,
                                        boolean isSelected, boolean cellHasFocus )
                {
                    // Display the text for this item
                    setText(value.toString());

                    // Set the correct image
                    setIcon( image[index] );

                    // Draw the correct colors and font
                    if( isSelected )
                    {
                        // Set the color and font for a selected item
                        setBackground( Color.red );
                        setForeground( Color.white );
                        setFont( new Font( "Roman", Font.BOLD, 24 ) );
                    }
                    else
```

```
                {
                        // Set the color and font for an unselected item
                        setBackground( Color.white );
                        setForeground( Color.black );
                        setFont( new Font( "Roman", Font.PLAIN, 12 ) );
                }
                return this;
            }
        }
```

Interface used:-

- **ListCellRenderer:-**
  - Identifies components that can be used as "rubber stamps" to paint the cells in a JList.
  - For example, to use a JLabel as a ListCellRenderer, you would write something like this:

```
class MyCellRenderer extends JLabel implements ListCellRenderer {
    public MyCellRenderer() {
        setOpaque(true);
    }
    public Component getListCellRendererComponent(
        JList list,
        Object value,
        int index,
        boolean isSelected,
        boolean cellHasFocus)
    {
        setText(value.toString());
        setBackground(isSelected ? Color.red : Color.white);
        setForeground(isSelected ? Color.white : Color.black);
        return this;
    }
}
```

Methods Used:-

- Component **getListCellRendererComponent**(JList list,
  Object value,
  int index,
  boolean isSelected,
  boolean cellHasFocus)

  :-Return a component that has been configured to display the specified value. That component's `paint` method is then called to "render" the cell. If it is necessary to compute the dimensions of a list because the list cells do not have a fixed size, this method is called to generate a component on which `getPreferredSize` can be invoked.

7.

Database Connectivity

7.1 Database Connectivity through drivers

A programmer interacts with DBMS by writing SQL queries which are accepted and executed by DB Engine. But when applications are required to interact with databases, then one needs a program which acts as an interface between the application and DB Engine. The program acts as an interface and is known as Call-Level Interface or Program Interface. The interface is in the form of driver written in (native) languages such as c/c++. The driver contains basic function such as connecting to the database, executing queries, processing result-sets etc. But this driver is DB specific. So if you change the database, the application fails.

To avoid this, Microsoft introduced ODBC(Open Database Connectivity) drivers. It's a standard way of accessing database services. ODBC acts as an interface between applications and native drivers. So even if the database is changed, application programs are not required to change because they interact with ODBC drivers and not with native drivers. But due to ODBC, special services of the specific database cannot be accessed.

What Is JDBC?

JDBC is a Java API for executing SQL statements. (JDBC is a trademarked name and is not an acronym; nevertheless, JDBC is popularly known as Java Database Connectivity.") It consists of a set of classes and interfaces written in the Java programming language. JDBC provides a standard API for tool/database developers and makes it possible to write database applications using a pure Java API. JDBC provides a pure Java API for database access along with a driver manager to allow third party drivers to connect to specific databases. Database vendors provide their own drivers to plug into the driver manager. Thus, using JDBC, it is easy to send SQL statements to virtually any relational database. With the JDBC API, it isn't necessary to write different programs for different databases. JDBC makes it possible to do following main things:

1. Load JDBC driver
2. Establish a connection to the database,
3. Execute one or more SQL statements,
4. Process the results,
5. Execute transactions, call stored procedures and access metadata,
6. Disconnect from the database.

JDBC versus ODBC and other APIs

Microsoft's ODBC (Open DataBase Connectivity) API is one of the programming interface used for accessing relational databases. It offers the ability to connect to almost all databases on almost all platforms. But using only ODBC has following problems -

1. ODBC is not appropriate for direct use from Java because it uses a C interface. Calls from Java to native C code have a number of drawbacks in the security, implementation, robustness, and automatic portability of applications.

2. A literal translation of the ODBC C API into a Java API would not be desirable. For example, Java has no pointers, and ODBC makes copious use of them, including the notoriously error-prone generic pointer void *. You can think of JDBC as ODBC translated into an object-oriented interface that is natural for Java programmers.

2. ODBC is hard to learn. It mixes simple and advanced features together, and it has complex options even for simple queries. JDBC, on the other hand, was designed to keep simple things simple while allowing more advanced capabilities where required.

4. A Java API like JDBC is needed in order to enable a "pure Java" solution. When ODBC is used, the ODBC driver manager and drivers must be manually installed on every client machine. When the JDBC driver is written completely in Java, however, JDBC code is automatically installable, portable, and secure on all Java platforms from network computers to mainframes.

7.2 JDBC Model

Two-tier Model

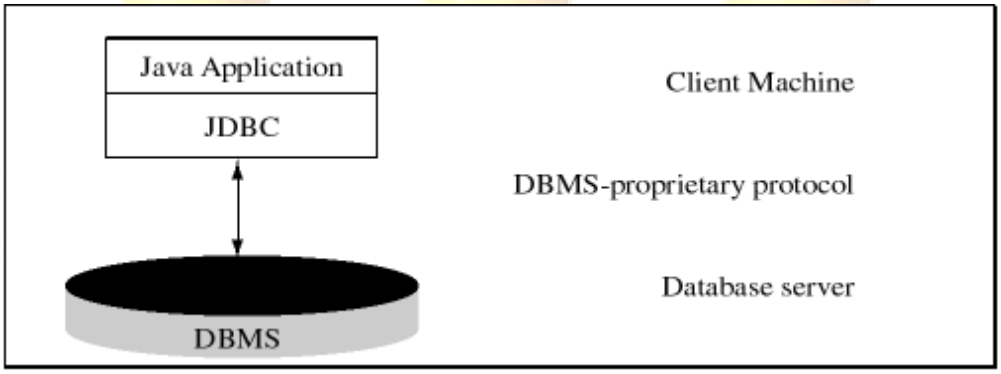
The JDBC API supports both two-tier and three-tier models for database access. In the two-tier model, the client/database application acts as the first tier and the database server acts as the second tier. A Java applet or application talks directly to the database. This requires a JDBC driver that can communicate with the particular database management system being accessed. A user's SQL statements are delivered to the database, and the results of those statements are sent back to the user. The database may be located on another machine to which the user is connected via a network. This is referred to as a *client/server* configuration, with the user's machine as the client, and the machine housing the database as the server. The client communicates to the server without the help of another server or server process. The network can be an intranet, which, for example, connects employees within a corporation, or it can be the Internet.

Advantages

i) It is easy to implement.
ii) It maintains a persistent connection between the client and the database thereby eliminating overhead associated with opening and closing connections.
iii) It is faster than three-tier system.

Disadvantages

i) Native libraries must be loaded on each client machine.
ii) Applets can only open up connections to the server from which they were downloaded. This requires that web server and database server should be present on the same machine.



JDBC Two-Tier Model

Three-Tier Model

In the three-tier model, commands are sent to a "middle tier" of services, which then sends SQL statements to the database. The database processes the SQL statements and sends the results back to the middle tier, which then sends them to the user. The three-tier model is advantageous because the middle

tier makes it possible to maintain control over access and the kinds of updates that can be made to corporate data. Another advantage is that when there is a middle tier, the user can employ an easy-to-use higher-level API, which is translated by the middle tier into the appropriate low-level calls.

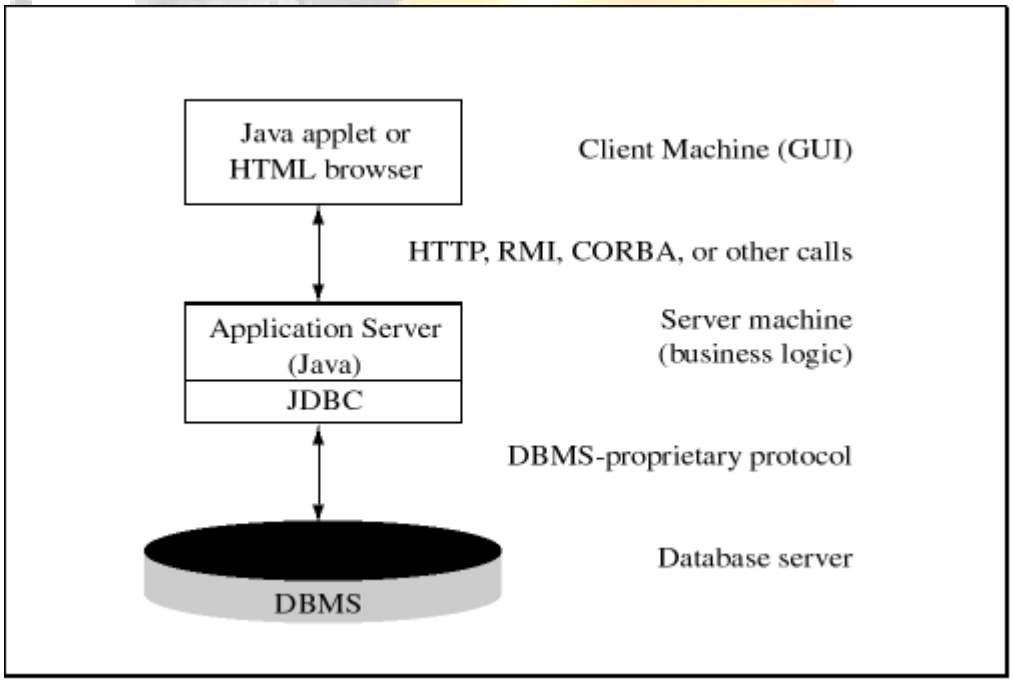
Until now the middle tier has typically been written in languages such as C or C++, which offer fast performance. But now, it can also be developed in Java to take advantage of Java's robustness, multithreading, and security features. Of course, JDBC is important in allowing database access from a Java middle tier. Using Three-tier model, it is not necessary to keep the Web server and the Database server on the same machine. It also eliminates the need for applets to download the driver to the clients.

Advantages

i) Clients do not need to have native libraries loaded locally,
ii) Drivers can be managed centrally,
iii) Database server is not directly visible to the internet.

Disadvantages

i) The client does not maintain a persistent database connection.
ii) A separate proxy server may be required.
iii) The communication between the clients and the server is slower because the client calls must be translated into network protocol and then they must be translated into specific DB calls.



JDBC Three-tier Model

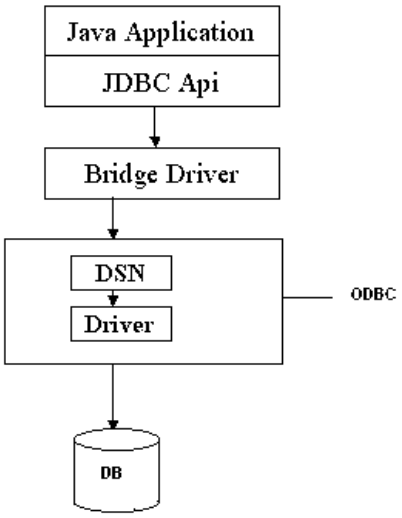
7.3 JDBC Driver Types

JDBC driver is responsible for ensuring that an application has consistent and uniform access to the database. It receives the client applications request, translates it into a format that the database can

understand and then presents the request to the database. The response is received by the JDBC driver, translated back into Java data format and presented to the client application.

The various types of JDBC drivers are as follows -

1. **JDBC-ODBC bridge plus ODBC driver(Type I)***:* This driver provides JDBC access via ODBC drivers. Note that ODBC binary code, and in many cases database client code, must be loaded on each client machine that uses this driver. When used in the applications, the JDBC driver translates the request to an ODBC call which then translates the same call for the use with the database



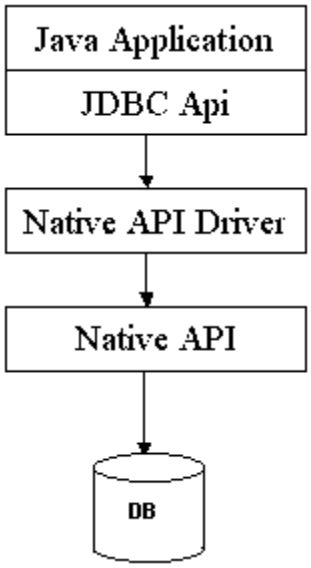
Advantages

i) This type of driver is cheaper.
ii) This kind of driver is most appropriate on a corporate network or in LANs where client installations are not a major problem, or for application server code written in Java in a three-tier architecture.

Disadvantages

i) This driver is slower because of double translation (JDBC to ODBC, ODBC to DBMS)
ii) If the driver is downloaded as an applet, the security manager of applets will not allow access to local files. This means that the downloaded driver will not be able to access the ODBC driver files installed locally on the machine. For these situations, this driver is not a viable solution.

2. **Native-API partly-Java driver(Type II)***:* This driver directly interacts with native driver. It is partly written in Java and partly in native code. The native code is known as Call Level Interface(CLI) . The CLI libraries are responsible for the actual communications with the database server. This kind of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, DB2, or other DBMS. Note that, like the bridge driver, this style of driver requires that some binary code be loaded on each client machine.



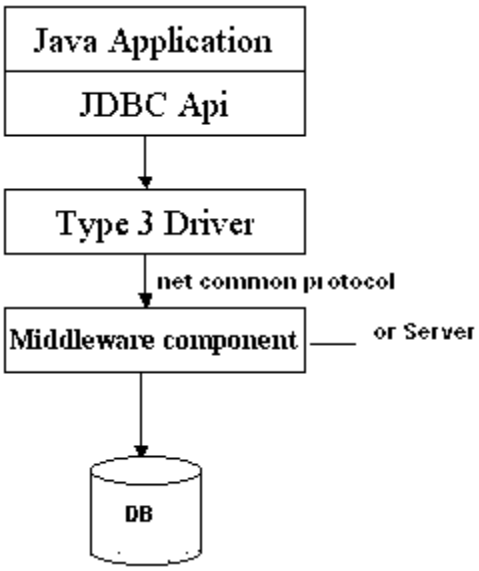
Advantages

i) This type of driver is faster than ODBC because there is no double translation.

Disadvantages

i) It cannot be used with Applets.
ii) Requires client-side installations.

3. **JDBC-Net pure Java driver(Type III)**: This driver translates JDBC calls into a DBMS-independent net protocol, which are then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to different databases. In Type I and Type II drivers, the drivers are kept on the client side. In type III, the driver is divided into two parts – one part is kept on the client side and the other part is kept on the server side. The server side part contains the CLI libraries that actually interact with the database. The client part just converts the JDBC requests to driver specific network protocol. It then sends the request to a listener process(CLI) on the server. It is the responsibility of the server process to present the request to the DB server. So the communication between the applications and the DB server is 100% Java to Java.



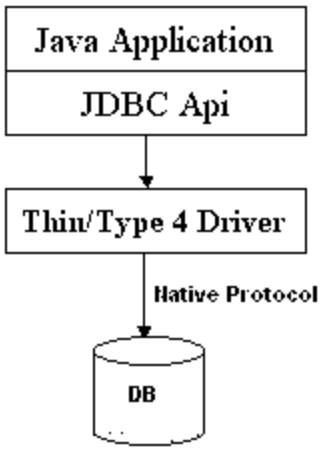
Advantages

i) It can be used with applets.
ii) It can be used in 3-tier architecture.

Disadvantages

i) The clients use different types of network protocols. So the client part should send the JDBC calls through network specific protocol. As different clients use different types of network, the development of client part becomes tedious.

4. **Native-protocol pure Java driver(Type IV)***:* In this, JDBC itself is a native driver. This kind of driver converts JDBC calls directly into the network protocol used by DBMS. This allows a direct call from the client machine to the DBMS server and is an excellent solution for intranet access.



Advantage

i) This is the fastest way of accessing databases using JDBC driver.

Disadvantages

i) It is database specific.

7.4 Loading the Driver

The first thing you need to do is establish a connection with the DBMS you want to use. This involves two steps: (1) loading the driver and (2) making the connection. To do this, you must first import all the classes from *java.sql* package.

Loading the driver or drivers you want to use is very simple and involves just one line of code. If, for example, you want to use the JDBC-ODBC Bridge driver, the following code will load it:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

Your driver documentation will give you the class name to use. For instance, if the class name is `jdbc.DriverXYZ`, you would load the driver with the following line of code:

Class.forName("jdbc.DriverXYZ");

JdbcOdbcDriver is a driver that comes with JDK. *sun.jdbc.odbc* represents the hierarchy of packages in which it is located. It can be found of in *rt.jar* located in <jdk>\jre\lib directory.
class `Class` is used to access classes and interfaces in a running Java application. *forName* method returns the `Class` object associated with the class or interface with the given string name. It throws *ClassNotFoundException* if the driver is not found. When you have loaded a driver, it is available for making a connection with a DBMS.

Establishing the Connection

The second step in establishing a connection is to have the appropriate driver connect to the DBMS. The following line of code illustrates the general idea:

Connection con = DriverManager.getConnection(url, "myLogin", "myPassword");

Where *url* specifies the location to send/obtain data. A JDBC URL provides a way of identifying a database so that the appropriate driver will recognize it and establish a connection with it. `Driver` writers are the ones who actually determine what the JDBC URL that identifies their particular driver will be. The jdbc url has three parts -

jdbc:<subprotocol>:<subname>

The three parts of a JDBC URL are broken down as follows:

(i) `jdbc` - the protocol. The protocol in a JDBC URL is always `dbc`.

(ii) `<subprotocol>`--the name of the driver or the name of a database connectivity mechanism, which may be supported by one or more drivers. A prominent example of a subprotocol name is `odbc`, which has been reserved for URLs that specify ODBC-style data source names. For example, to access a database through a JDBC-ODBC bridge, one might use a URL such as the following:

jdbc:odbc:k

In this example, the subprotocol is `odbc`, and the subname `k` is a local ODBC data source. If one wants to use a network name service (so that the database name in the JDBC URL does not have

to be its actual name), the naming service can be the subprotocol. So, for example, one might have a URL like:

jdbc:krb:students

In this example, the URL specifies that the local service should resolve the database name `students` into a more specific name that can be used to connect to the real database.

(iii) `<subname>`--a way to identify the database. The point of a subname is to give enough information to locate the database. In the previous example, k is enough because ODBC provides the remainder of the information. A database on a remote server requires more information, however. If the database is to be accessed over the Internet, for example, the network address should be included in the JDBC URL as part of the subname and should follow the standard URL naming convention of

//hostname:port/subsubname

Supposing that `dbnet` is a protocol for connecting to a host on the Internet, a JDBC URL might look like this:

jdbc:dbnet://vesadmin:123/k

In place of "`myLogin`" you put the name you use to log in to the DBMS; in place of "`myPassword`" you put your password for the DBMS. So if you log in to your DBMS with a login name of "rahul" and a password of "krb," just these two lines of code will establish a connection:

```
String url = "jdbc:odbc:k";
Connection con = DriverManager.getConnection(url, "rahul", "krb");
```

If one of the drivers you loaded recognizes the JDBC URL supplied to the method `DriverManager.getConnection`, that driver will establish a connection to the DBMS specified in the JDBC URL. The `DriverManager` class selects an appropriate driver from the set of registered JDBC drivers and manages all of the details of establishing the connection for you behind the scenes. The connection returned by the method `DriverManager.getConnection` is an open connection you can use to create JDBC statements that pass your SQL statements to the DBMS. *Connection* is an interface that represents connection with a specific database. Within the context of a Connection, SQL statements are executed and results are returned.

A Connection's database is able to provide information describing its tables, its supported SQL grammar, its stored procedures, the capabilities of this connection, and so on.
With respect to java program, connection object is like a DB Engine. By default the Connection automatically commits changes after executing each statement. If auto commit feature has been disabled, the method `commit` must be called explicitly; otherwise, database changes will not be saved.

Creating Statement Objects

After establishing connection to the database, we are ready to execute SQL statements that perform some kind of work. Before executing an SQL statement, we need to create a statement object that provides an interface to an underlying database engine. The three Statements objects are explained as follows –

1. Statement : This is the base statement object that provides methods to execute SQL statements directly against the database. The statement object is useful in creating one-time queries and DDL statements such as create, drop, select etc.

2. PreparedStatement : This statement object is created using an SQL statement that will be used many times, replacing only the data values to be used. The main part of the query is kept ready on the DB Server side to improve the overall performance. JDBC offers methods to specify the input parameters used by the statements.

3. CallableStatement : This statement object is used to access stored procedures in the database. Methods exist to specify the input and output parameters used by the statement.

Actually all of the above represent interfaces in Java. *Statement* is the topmost interface. *PreparedStatement* is subclass of *Statement* and *CallableStatement* is a subclass of *PreparedStatement*.

Executing SQL statements using Statement

In order to execute SQL statements, we need to create a statement object first. It takes an instance of an active connection to create a `Statement` object. In the following example, we use our `Connection` object *`con`* to create the `Statement` object *`stmt`*:

```
Statement stmt = con.createStatement();
```

createStatement creates a `Statement` object for sending SQL statements to the database. SQL statements without parameters are normally executed using Statement objects. At this point *`stmt`* exists, but it does not have an SQL statement to pass on to the DBMS. We need to supply that to the method we use to execute *stmt*. For a `SELECT` statement, the method to use is `executeQuery`. For statements that create or modify tables, the method to use is `executeUpdate`. `You can use generic execute method also if the type of the statement is not known.`

Here is a program -

```
import java.sql.*;

class j1
{
   public static void main(String args[])
   {
      try
      {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection con = DriverManager.getConnection("jdbc:odbc:kamlakar");
         Statement st = con.createStatement();
         boolean b = st.execute("create table students( roll int, name varchar(20))");
           System.out.println(b);
         b = st.execute("insert into students values (101, 'rahul')");
           System.out.println(b);
         String s = "update students set name='xyz' where roll=101";
           int i = st.executeUpdate(s);
         System.out.println(i);
```

```
        st.close();  con.close();
    }
    catch(ClassNotFoundException e1)    {    System.out.println(e1);    }
    catch(SQLException e2)    {    System.out.println(e2);    }
}}
```

The different types of execute statements accept the query in the form of string. The *execute* method returns true if the statement produces one or more records or else if it results an update count. The *executeUpdate* method returns an integer representing the number of rows that got changed. It can also be used to execute statements (such as insert, update, delete) that return nothing. The above methods throw an *SQLException* in case of any error. After executing all the statements, it is necessary to close the connection. *close* method releases a Connection's database and JDBC resources immediately instead of waiting for them to be automatically released.

Processing the Results

Suppose we write a query as follows –

```
select * from student;
```

It returns a set of rows. In Java applications, we may need to retrieve the results and work with them. This is done using *ResultSet* object. For example,

```
ResultSet rs = st.executeQuery("select * from students");
```

`Actually ResultSet` is an interface that contains different methods to access the data. The object returned by the query contains the results an SQL query which is stored in *rs*. A `ResultSet` object maintains a cursor, which points to its current row of data. The cursor moves down one row each time the method `next` is called. Initially it is positioned before the first row, so that the first call to `next` puts the cursor on the first row, making it the current row. `ResultSet` rows are retrieved in sequence from the top row down as the cursor moves down one row with each successive call to `next`. A cursor remains valid until the `ResultSet` object or its parent `Statement` object is closed.

The `getXXX` methods provide the means for retrieving column values from the current row. Within each row, column values may be retrieved in any order, but can be read only once. Either the column name or the column number can be used to designate the column from which to retrieve data. For example, if the second column of a `ResultSet` object `rs` is named "roll," and it stores values as strings, either of the following will retrieve the value stored in that column:

```
String name= rs.getString(2);
int i = rs.getInt("roll");
```


Note that columns are numbered from left to right starting with column 1. Also, column names used as input to `getXXX` methods are case insensitive.

```
import java.sql.*;

class j2
{
   public static void main(String args[])
   {
      try
      {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection con = DriverManager.getConnection("jdbc:odbc:kamlakar");
```

```
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery("select * from studnets");
        while(rs.next())
        {
                int roll = rs.getInt(1);
                String name = rs.getString("name");
                System.out.println("Roll : " + roll + ",Name :" + name);
          }
          rs.close();
          st.close();
        con.close();
      }
      catch(Exception e)
      {
        System.out.println(e);
      }
   }
}
```

Normally, nothing needs to be done to close a `ResultSet` object; it is automatically closed by the `Statement` object that generated it when that `Statement` object is closed, is re-executed, or is used to retrieve the next result from a sequence of multiple results. The method `close` is provided so that a `ResultSet` object can be closed explicitly, thereby immediately releasing the resources held by the `ResultSet` object. This could be necessary when several statements are being used and the automatic close does not occur soon enough to prevent database resource conflicts.

If you don't know the type of the query, you can use *execute( )* method. *execute* method returns *true* if the query produces number or rows. To get the corresponding resultset, *getResultSet()* method is used. To get the number of rows affected by the query, you can use *getUpdateCount()* method. Consider a portion of a Java code –

```
if(st.execute(sql)
{       ResultSet rs  =  st.getResultSet(); }
else
        int I = st.getUpdateCount();
```

7.5 About ResultSetMetaData

ResultSetMetaData provides information about the types and properties of the columns in a *ResultSet* object. An instance of ResultSetMetaData actually contains the information and ResultSetMetaData methods give access to that information. Normally, we use three basic methods - (i) *getColumnCount()* which returns the number of columns in the result set, (ii) *getColumnLable(int n)* which returns the name of a particular column in a result set, and (iii)*getColumnTypeName(int n)* which returns the type-name of the column.

```
        ResultSet rs = st.executeQuery("select * from data1");
        ResultSetMetaData  rsmd = rs.getMetaData();
        int n = rsmd.getColumnCount();
        for(int i=1; i<=n; i++)
         System.out.println(rsmd.getColumnLabel(i) + " : " + rsmd.getColumnTypeName(i));
```

7.6 About PreparedStatement

The `PreparedStatement` interface inherits from `Statement` and differs from it in two ways, first Instances of `PreparedStatement` contain an SQL statement that has already been compiled. This is what makes a statement "prepared." Secondly, the SQL statement contained in a `PreparedStatement` object may have one or more IN parameters. An IN parameter is a parameter whose value is not specified when the SQL statement is created. Instead the statement has a question mark (?) as a placeholder for each IN parameter. A value for each question mark must be supplied by the appropriate `setXXX` method before the statement is executed.

Because `PreparedStatement` objects are precompiled, their execution can be faster than that of `Statement` objects. Consequently, an SQL statement that is executed many times is often created as a `PreparedStatement` object to increase efficiency.

Being a subclass of `Statement`, `PreparedStatement` inherits all the functionality of `Statement`. In addition, it adds a set of methods that are needed for setting the values to be sent to the database in place of the placeholders for IN parameters. Also, the three methods `execute`, `executeQuery`, and `executeUpdate` are modified so that they take no argument. The `Statement` forms of these methods (the forms that take an SQL statement parameter) should never be used with a `PreparedStatement` object.

Before a `PreparedStatement` object is executed, the value of each ? parameter must be set. This is done by calling a `setXXX` method, where `XXX` is the appropriate type for the parameter. The first argument to the `setXXX` methods is the *ordinal position* of the parameter to be set, with numbering starting at 1. The second argument is the *value* to which the parameter is to be set. Once a parameter value has been set for a given statement, it can be used for multiple executions of that statement until it is cleared by a call to the method *`clearParameters`* or until a new value is set.

```
import java.io.*;
import java.sql.*;

class j2
{
   public static void main(String args[]) throws Exception
   {
      try
      {
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
         Connection con = DriverManager.getConnection("jdbc:odbc:kamlakar");
         PreparedStatement st = con.prepareStatement("insert into students values(?,?)");
         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         String c = "yes";
         while(!c.equals("STOP"))
         {
            System.out.println("enter roll number :");
            String r = br.readLine();
            System.out.println("enter name :");
            String s = br.readLine();
            st.clearParameters();
            st.setInt(1,Integer.parseInt(r));
            st.setString(2,s);
            st.executeUpdate();
            System.out.println("Continue?");
            c = br.readLine();
         }
```

```
        con.close();
      }
      catch(SQLException e)                 {       System.out.println(e);      }
      catch(ClassNotFoundException e)       {       System.out.println(e);      }
   }
}
```

7.7 About CallableStatement

A `CallableStatement` object provides a way to call stored procedures in a standard way for all RDBMSs. A stored procedure is stored in a database; the *call* to the stored procedure is what a `CallableStatement` object contains. This call is written in an escape syntax that may take one of two forms: one form with a result parameter, and the other without one. A result parameter, a kind of OUT parameter, is the return value for the stored procedure. Both forms may have a variable number of parameters used for input (IN parameters), output (OUT parameters), or both (INOUT parameters). A question mark serves as a placeholder for a parameter. The syntax for invoking a stored procedure in JDBC is shown here. Note that the square brackets indicate that what is between them is optional; they are not themselves part of the syntax.

{call procedure_name[(?, ?, ...)]}

The syntax for a procedure that returns a result parameter is:

{? = call procedure_name[(?, ?, ...)]}

The syntax for a stored procedure with no parameters would look like this:

{call procedure_name}

`CallableStatement` inherits `Statement` methods, which deal with SQL statements in general, and it also inherits `PreparedStatement` methods, which deal with IN parameters. All of the methods defined in `CallableStatement` deal with OUT parameters or the output aspect of INOUT parameters: registering the JDBC types of the OUT parameters, retrieving values from them, or checking whether a returned value was JDBC `NULL`. Whereas the `getXXX` methods defined in `ResultSet` retrieve values from a result set, the `getXXX` methods in `CallableStatement` retrieve values from the OUT parameters and/or return value of a stored procedure.

`CallableStatement` objects are created with the `Connection` method `prepareCall`. The following example, in which *`con`* is an active JDBC `Connection` object, creates an instance of `CallableStatement`:

CallableStatement cstmt = con.prepareCall("{call getTestData(?, ?)}");

The variable *`cstmt`* contains a call to the stored procedure `getTestData`, which has two argument parameters and no result parameter. Whether the ? placeholders are IN, OUT, or INOUT parameters depends on the stored procedure `getTestData`.

Passing in any IN parameter values to a `CallableStatement` object is done using the `setXXX` methods inherited from `PreparedStatement`. The type of the value being passed in determines which `setXXX` method to use (`setFloat` to pass in a `float` value, `setBoolean` to pass in a `boolean`, and so on).

If the stored procedure returns OUT parameters, the JDBC type of each OUT parameter must be registered before the `CallableStatement` object can be executed. This is necessary because some DBMSs require the SQL type. Registering the JDBC type is done with the method `registerOutParameter`. Then after the statement has been executed, `CallableStatement`'s `getXXX` methods can be used to retrieve OUT parameter values. The correct `CallableStatement.getXXX` method to use is the Java type that corresponds to the JDBC type registered for that parameter.
The method `executeQuery` is used to execute *`cstmt`* because the stored procedure that it calls returns a result set.

```
CallableStatement cstmt = con.prepareCall("{call getTestData(?, ?)}");
cstmt.registerOutParameter(1, INT);
```

cstmt.registerOutParameter(2, FLOAT, 3);
ResultSet rs = cstmt.executeQuery();
int x = cstmt.getInt(1);
float n = cstmt.getFLOAT (2, 3);

A parameter that supplies input as well as accepts output (an INOUT parameter) requires a call to the appropriate `setXXX` method (inherited from `PreparedStatement`) in addition to a call to the method `registerOutParameter`.

```
create or replace procedure proc1 is
begin
delete  from students;
end;
.
/
[ at sql prompt →
set serveroutput on;
exec proc1;
]

create or replace procedure proc2 ( a in number, b out number ) is
begin
select count(*) into b from emp where sal>=a;
end;
.
/
 [ at sql prompt –
var a number;
exec proc2(3000, :a);
print :a;
]

create or replace procedure proc3 ( a in out number) is
begin
select sal into a from emp where empno=a;
end;
.
/
 [at sql prompt→
exec :a := 100;
exec proc3(:a);
print :a;
]

create or replace function fun1( a in number ) return number is
b number;
begin
select count(*) into b from emp where sal>=a;
return b;
end;
.
/
---
[at sql prompt→
exec :a := fun1(9000);
print :a;]
import java.sql.*;
```

```
class j4 {
   public static void main(String args[]) throws Exception    {
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con = DriverManager.getConnection("jdbc:odbc:rahul","scott","tiger");
      CallableStatement st1 = con.prepareCall("{ call proc1}");
      CallableStatement st2 = con.prepareCall("{ call proc2(?,?)}");
      CallableStatement st3 = con.prepareCall("{ call proc3(?)}");
      CallableStatement st4 = con.prepareCall("{ ? = call fun1(?)}");

      st1.execute();

      st2.setInt(1,3000);
      st2.registerOutParameter(2,Types.INTEGER);
      st2.execute();
      System.out.println("Proc2:" + st2.getInt(2));

      st3.registerOutParameter(1,Types.VARCHAR);
      st3.setString(1,"JAMES");
      st3.execute();
      System.out.println("proc3 :" + st3.getString(1));

      st4.setInt(2,3000);
      st4.registerOutParameter(1,Types.INTEGER);
      st4.execute();
      System.out.println("Fun1 :" + st4.getInt(1));    } }
```

7.8 Using Transactions with JDBC

A transaction is a set of one or more statements that are executed together as a unit, so either all of the statements are executed or none of the statements is executed. Transaction management with JDBC takes place via the Connection object. By default, new connections start out in auto-commit mode. This means that every SQL statement is executed as an individual transaction that is automatically committed to the database. For controlling the commitment ourselves, thereby allowing the group SQL statement into transactions, call
SetAutoCommit(false) on the connection object. After completing all SQL statements, call *commit()* on connection object to permanently record the transaction in the database or call *rollback()* on the same to undo the transaction. *getAutoCommit()* is used to obtain the current state of AutoCommit.

```
con.setAutoCmmit(false);

try
{
        st.execute("update order set quantity=15 where orderid=3 and itemcode=5");
        st.execute("update item_master set quantity=quantity-15 where itemcode=5");
        con.cmmit();
}
catch(Exception e)
{
        con.rollback();
```

```
}
```

7.9 Making Batch Updates

A batch update is a set of multiple update statements that is submitted to the database for processing as a batch. Sending multiple update statements to the database together as a unit can, in some situations, be much more efficient than sending each update statement separately. This ability to send updates as a unit, referred to as the batch update facility, is one of the features provided with the JDBC 2.0 API.

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();
stmt.addBatch("insert into students values (101, 'amit', 99)");
stmt.addBatch("insert into students values (102, 'ketan', 87)");
stmt.addBatch("insert into students values (103, 'rohit', 59)");
stmt.addBatch("insert into students values (104, 'stimit', 77)");
int [] updateCounts = stmt.executeBatch();
```

There are two exceptions that can be thrown during a batch update operation: `SQLException` and `BatchUpdateException`.

7.10 Additional Methods

clearWarnings() / SQLWarnings getWarnings() available with Connection
boolean isClosed()available with Connection
boolean isReadOnly() / setReadOnly(boolean) available with Connection

7.11 About DatabaseMetaData

DatabaseMetaData is used to obtain the information about the entire database like schemas, tables, procedures, constraints etc.


```
import java.sql.*;
class dmetadata
{       public static void main(String args[])
   {          try          {
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
           Connection con = DriverManager.getConnection("jdbc:odbc:ora","scott", "tiger");
          DatabaseMetaData d = con.getMetaData();
          ResultSet rs = d.getTables(null,null,null,null);
          while(rs.next()) { System.out.println(rs.getString(3));   }
          con.close();
       }
      catch(Exception e)          {             System.out.println(e);    }       }    }
```

7.12 Scrollable Result Sets

One of the new features in the JDBC 2.0 API is the ability to move a result set's cursor backward as well as forward. There are also methods that let you move the cursor to a particular row and update it. The following line of code illustrates one way to create a scrollable `ResultSet` object:

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);

ResultSet srs = stmt.executeQuery("select name, percent
from students");
```

The first argument to createStatement is one of three constants added to the `ResultSet` API to indicate the type of a `ResultSet` object: `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE` , and `TYPE_SCROLL_SENSITIVE` .

Specifying the constant `TYPE_FORWARD_ONLY` creates a non-scrollable result set, that is, one in which the cursor moves only forward. You will get a scrollable `ResultSet` object if you specify `TYPE_SCROLL_INSENSITIVE` or `TYPE_SCROLL_SENSITIVE`. The difference in last two is that a result set that is `TYPE_SCROLL_INSENSITIVE` does not reflect changes made while it is still open and one that is `TYPE_SCROLL_SENSITIVE` does. All three types of result sets will make changes visible if they are closed and then reopened. The second argument is one of two `ResultSet` constants for specifying whether a result set is read-only or updatable: `CONCUR_READ_ONLY` and `CONCUR_UPDATABLE`. If you do not specify any constants for the type and updatability of a `ResultSet` object, you will automatically get one that is `TYPE_FORWARD_ONLY` and `CONCUR_READ_ONLY`. `TYPE_SCROLL_INSENSITIVE` and `CONCUR_READ_ONLY` are used together in order to scroll without making and changes to the database and `TYPE_SCROLL_SENSITIVE` and `CONCUR_UPDATABLE` are used together in order to scroll and update in the database.

You can move the cursor to a particular row in a `ResultSet` object. The methods *`first`*, *`last`*, *`beforeFirst`*, and *`afterLast`* move the cursor to the row indicated in their names. The method *`absolute`* will move the cursor to the row number indicated in the argument passed to it. If the number is positive, the cursor moves the given number from the beginning, and if the number is negative, the cursor moves the given number from the end. With the method *`relative`*, you can specify how many rows to move from the current row and also the direction in which to move. A positive number moves the cursor forward the given number of rows; a negative number moves the cursor backward the given number of rows. The method *`getRow`* lets you check the number of the row where the cursor is positioned. Four additional methods let you verify whether the cursor is at a particular position. The position is stated in their names: *`isFirst`* , *`isLast`* , *`isBeforeFirst`* , *`isAfterLast`* that return a Boolean value.

Connection con =
DriverManager.getConnection ("jdbc:odbc:rahul");

Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery("select name, percent from students");

Updating through the scrollable result-sets

Continuing from previous example, following statements how to update data in a scrollable result set.

```
uprs.last();
uprs.updateFloat("percent", 90.19);
```

The first method navigates the cursor to the last record. The `ResultSet.updateXXX` methods take two parameters: the column name/number to update and the new value to put in that column. The data in the result set gets updated. But this change is not reflected in the database. To make the necessary changes one has to call another method as follows –

```
uprs.last();
uprs.updateFloat("percent", 90.19);
uprs.updateRow();
```

If you move the cursor to a different row before calling the method `updateRow` , the update will be lost. To cancel all the updates in a row, one can call *cancelRowUpdates* method before calling the *updateRow* method.

```
uprs.last();
uprs.updateFloat("percent", 90.19);
uprs.cancelRowUpdates();
uprs.updateFloat("percent", 90.90);
uprs.updateRow();
```

7.13 Inserting and Deleting Rows Programmatically

In the previous section you saw how to modify a column value using methods in the JDBC 2.0 API rather than having to use SQL commands. With the JDBC 2.0 API, you can also insert a new row into a table or delete an existing row programmatically.

Your first step will be to move the cursor to the insert row, which you do by invoking the method `moveToInsertRow`. The next step is to set a value for each column in the row. You do this by calling the appropriate `updateXXX` method for each value. Note that these are the same `updateXXX` methods you used in the previous section for changing a column value. Finally, you call the method `insertRow` to insert the row you have just populated with values into the result set. This one method simultaneously inserts the row into both the `ResultSet` object and the database table from which the result set was selected.

The following code fragment creates the scrollable and updatable `ResultSet` object `uprs` , which contains all of the rows and columns in the table students:

Connection con = DriverManager.getConnection("jdbc:odbc:k");
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
*ResultSet uprs = stmt.executeQuery("select * from students");*

The next code fragment uses the `ResultSet` object `uprs` to insert the row. It moves the cursor to the insert row, sets the three column values, and inserts the new row into `uprs`:

```
uprs.moveToInsertRow();
uprs.updateInt("roll", 150);
uprs.updateString("name", "pqr");
uprs.updateFloat(3, 90.99);
uprs.insertRow();
```

In both updates and insertions, calling an `updateXXX` method does not affect the underlying database table. The method `updateRow` must be called to have updates occur in the database. For insertions, the method `insertRow` inserts the new row into the result set and the database at the same time.

So far, you have seen how to update a column value and how to insert a new row. Deleting a row is the third way to modify a `ResultSet` object, and it is the simplest. All you do is move the cursor to the row you want to delete and then call the method `deleteRow` . For example, if you want to delete the fourth row in the `ResultSet` `uprs`, your code will look like this:

```
uprs.absolute(4);
uprs.deleteRow();
```