ECS-503 Object Oriented Techniques

UNIT-4 Part-1

CHAPTER 1 The History and Evolution of Java

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs.

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn’t very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively.

Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles. Instead, they relied upon the GOTO as a primary means of program control. As a result, programs written using these languages tended to produce “spaghetti code”—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand.

On the eve of C’s creation, the stage was set for a quantum leap forward in computer languages. Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an older language called BCPL, developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s. For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in The C Programming Language by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is complexity. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need.

OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism. C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language “C with Classes.” However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C’s features, attributes, and benefits.

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of
the original prototype. Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and timeconsuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

Further, because (at that time) much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security.

The Java Buzzwords
No discussion of Java’s history 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 moulding 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

Security
As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.
Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached may have been the single most innovative aspect of Java.

**Portability**

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

**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.

**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 primitive types, such as integers, are kept as high performance non objects.

**Robust**

The multiplatformed 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. 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, runtime errors). Memory management can be a difficult, tedious task in traditional programming environments. For example, in C/C++, the programmer will often 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. At the time of Java’s creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would 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, anytime, 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 executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, 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.

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. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

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

CHAPTER
2 An Overview of Java

As in all other computer languages, the elements of Java do not exist in isolation. Rather, they work together to form the language as a whole.

Object-Oriented Programming
Object-oriented programming (OOP) is at the core of Java. In fact, all Java programs are to at least some extent object-oriented. OOP is so integral to Java that it is best to understand its basic principles before you begin writing even simple Java programs.

Two Paradigms
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. 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.

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. They can ignore the details of how the engine, transmission, and braking systems work. Instead, they are free to utilize the object as a whole. A powerful way to manage abstraction is through the use of hierarchical classifications.

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.

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! In Java, the basis of encapsulation is 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.

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.
Polymorphism

Polymorphism (from 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.

Extending the dog analogy, a dog’s sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog’s nose! This same general concept can be implemented in Java as it applies to methods within a Java program.

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. 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, and 4-, 6-, or 8-cylinder engines. Either way, you will still press the brake 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.

A First Simple Program

```java
/*
   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.");
    }
}
```

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`. In Java, a source file is officially called a compilation unit. It is a text file that contains (among other things) one or more class definitions. (For now, we will be using source files that contain only one class.) The Java compiler requires that a source file use the `.java` filename extension.

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 Virtual Machine 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 application launcher called `java`. To do so, pass the class name `Example` as a command-line argument, as shown here:

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

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 `java` as just shown, you are actually specifying the name of the class that you want 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.

```java
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.

    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( ).

The public keyword is an access modifier, 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 Virtual Machine before any objects are made. The keyword void simply tells the compiler that main( ) does not return a value.

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.

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

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 most real-world Java applications.

The Java Keywords
There are 50 keywords currently defined in the Java language (see Table 2-1).
CHAPTER

3 Data Types, Variables, and Arrays

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.

The Primitive Types
Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types. These can be put in four groups:

• Integers This group includes byte, short, int, and long, which are for whole-valued 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.

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.

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] ...];

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.
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 Scope and Lifetime of Variables
A block defines a scope. Thus, each time you start a new block, you are creating a new scope. A scope determines what objects are visible to other parts of your program. It also determines the lifetime of those objects.

Another code

// This fragment is wrong!
int count = 100; // oops! cannot use count before it is declared.
Lifetime Example

```java
// Demonstrate lifetime of a variable.

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);
    }
}
```

**Type Conversion and Casting**

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 automatic 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.

**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.

**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
```

```
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.

**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.

**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 one-dimensional array declaration is `type var-name[];` Here, `type` declares the element type (also called the base type) of the array. For example, the following declares an array named `month_days` with the type “array of int”:

```java
int month_days[];
```

The general form of `new` as it applies to one-dimensional arrays appears as follows:

```java
array-var = new type [size];
```

This example allocates a 12-element array of integers and links them to `month_days`:

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

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. 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`:

```java
month_days[1] = 28;
```

**Multidimensional Arrays**

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. 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`:

```java
int twoD[][] = new int[4][5];
```
Conceptually, this array will look like the one shown in Figure

![Array Diagram](image)

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

**Alternative Array Declaration Syntax**

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

```java
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:

```java
int a1[] = new int[3];
int[] a2 = new int[3];
```

**CHAPTER 4 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.

**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 (also unary plus)
- 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 Bitwise Operators
Java defines several bitwise operators that can be applied to the integer types: long, int, short, char, and byte.

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 zero fill assignment
<<= Shift left assignment

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. For example, –42, or 11010110 inverted, yields 00101001, or 41, so when you add 1 you get 42.

The Bitwise Logical Operators
The bitwise logical operators are &, |, ^, and ~.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A &amp; B</th>
<th>A ^ B</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>

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.

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.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Logical XOR (exclusive OR)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Short-circuit AND</td>
</tr>
<tr>
<td>!</td>
<td>Logical unary NOT</td>
</tr>
<tr>
<td>&amp;=</td>
<td>AND assignment</td>
</tr>
<tr>
<td></td>
<td>=</td>
</tr>
<tr>
<td>^=</td>
<td>XOR assignment</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>?:</td>
<td>Ternary if-then-else</td>
</tr>
</tbody>
</table>

The following table shows the effect of each logical operation:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>A &amp; B</th>
<th>A ^ B</th>
<th>!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

Operator Precedence
Table 4-1 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left).
CHAPTER 5
Control Statements

A programming 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.

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.

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.

Nested ifs
A nested if is an if statement that is the target of another if or else. When you nest ifs, 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.

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:

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

**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.

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

The `break` statement is optional. If you omit the `break`, execution will continue on into the next `case`.

**Iteration Statements**
Java’s iteration statements are for, while, and do-while. These statements create what we commonly call loops.

**while**
The `while` loop is Java’s most fundamental loop statement. It repeats a statement or block while its controlling expression is true.
do-while
As you just saw, if the conditional expression controlling a while loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```java
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.

for

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

Declaring Loop Control Variables Inside the for Loop

Often the variable that controls a for loop is needed only 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:

```java
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.)

The For-Each Version of the for Loop

A second form of for was defined that implements a “for-each” style loop. The advantage of this approach is that no new keyword is required, and no preexisting code is broken. The for-each style of for is also referred to as the enhanced for loop.
The general form of the for-each version of the for is shown here:

```java
for(type itr-var : collection) statement-block
```

To understand the motivation behind a for-each style loop, consider the type of for loop that it is designed to replace. The following fragment uses a traditional for loop to compute the sum of the values in an array:

```java
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int i=0; i < 10; i++) sum += nums[i];
```

To compute the sum, each element in `nums` is read, in order, from start to finish. Thus, the entire array is read in strictly sequential order. This is accomplished by manually indexing the `nums` array by `i`, the loop control variable. The for-each style for automates the preceding loop. Specifically, it eliminates the need to establish a loop counter, specify a starting and ending value, and manually index the array. Instead, it automatically cycles through the entire array, obtaining one element at a time, in sequence, from beginning to end. For example, here is the preceding fragment rewritten using a for-each version of the for:

```java
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int sum = 0;

for(int x : nums) sum += x;
```

### Jump Statements
Java supports three jump statements: `break`, `continue`, and `return`. These statements transfer control to another part of your program.

#### 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.

#### 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.

#### 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.
CHAPTER 6

Introducing 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. 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. A class is declared by use of the class keyword.

A simplified 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. 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.

```
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. 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;
```
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:

```java
// 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();
        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
```

**Declaring Objects**

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.

```java
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`. At this point, `mybox` does not yet refer to an actual object. The next line allocates an 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, in essence, 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](image)

**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?

```java
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:

![Figure 6-1](image)
Introducing Methods

Classes usually consist of two things: instance variables and methods. This is the general form of a method:

\[
\text{type name(parameter-list)} \ |
// \text{body of method}
\]

Methods that have a return type other than \textbf{void} return a value to the calling routine using the following form of the \textbf{return} statement:

\[
\text{return value;}
\]

Here, \textit{value} is the value returned.

There are two important things to understand about returning values:

- 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 \textbf{boolean}, you could not return an integer.
- The variable receiving the value returned by a method (such as \textit{vol}, in this case) must also be compatible with the return type specified for the method.

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:

\[
\text{int square()}
\{
\text{ return 10 * 10;}
\}
\]

Constructors

A \textit{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 when the object is created, before the \textbf{new} operator completes. Constructors look a little strange because they have no return type, not even \textbf{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.

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the \textbf{this} keyword. \textbf{this} can be used inside any method to refer to the current object. That is, \textbf{this} is always a reference to the object on which the method was invoked. You can use \textbf{this} anywhere a reference to an object of the current class’ type is permitted. To better understand what \textbf{this} refers to, consider the following version of \textit{Box()}:
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, for example, 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 namespace collisions that might occur between instance variables and local variables.

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.

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 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:

```java
protected void finalize() {
    // finalization code here
}
```
CHAPTER
7 A Closer Look at Methods and Classes

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 supports 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:

```java
// Demonstrate method overloading.
public 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) {
        System.out.println("double a: " + a);
        return a*a;
    }
}

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);
}
```
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:

```java
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;
    }
}
```

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 to methods. For example, consider the following short program:

```java
// 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 equalTo(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.equalTo(ob2));
        System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
    }
}
```
This program generates the following output:

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

the `equalTo()` 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.

**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:

```java
// A simple example of recursion.
class Factorial {
    // this is a recursive method
    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
```

**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.

Java’s access modifiers are public, private, and protected. Java also defines a default access level. protected applies only when inheritance is involved. The other access modifiers are described
next. Let's begin by defining `public` and `private`. When a member of a class is modified by `public`, 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` modifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

```java
/* 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()`.

**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:
- They can only directly call other static methods.
- They can only directly access static data.
- 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 that 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:

```java
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, which prints a message and then initializes b 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.

Here is the output of the program:

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

Introducing final
A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

```java
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 fields, as this example shows. In addition to fields, both method parameters and local variables can be declared final. Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once.

The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

Exploring the String Class
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

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

the string "This is a String, too" is a String object.

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:

• If you need to change a string, you can always create a new one that contains the modifications.
• Java defines peer classes of String, called StringBuffer and StringBuilder, which allow strings to be altered, so all of the normal string manipulations are still available in Java. (StringBuffer and StringBuilder are described in Part II of this book.)

Strings can be constructed in a variety of ways. The easiest is to use a statement like this:

```java
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:

```java
System.out.println(myString);
```

Java defines one operator for String objects: +. It is used to concatenate two strings. For example, this statement

```java
String myString = "I" + " like " + "Java."
```
results in `myString` containing "I like Java."

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()`.

**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 a `String` array passed to the `args` parameter of `main()`. The first command-line argument is stored at `args[0]`, the second at `args[1]`, and so on. For example, the following program displays all of the command-line arguments that it is called with:

```java
// 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:

```bash
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
```

## CHAPTER 8

**Inheritance**

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications.

**Inheritance Basics**

To inherit a class, you simply incorporate the definition of one class into another by using the `extends` keyword. To see how, let’s begin with a short example. The following program creates a superclass called `A` and a subclass called `B`. Notice how the keyword `extends` is used to create a subclass of `A`. 

```java
class A {
    // Method implementation...
}

class B extends A {
    // Method implementation...
}
```
// A simple example of inheritance.

// Create a superclass.
class A {
    int i, j;

    void showij() {
        System.out.println("i and j: "+i+" "+j);
    }
}

// Create a subclass by extending class A.
class B extends A {
    int k;

    void showk() {
        System.out.println("k: "+k);
    }

    void sum() {
        System.out.println("i+j+k: "+(i+j+k));
    }
}

class SimpleInheritance {
    public static void main(String args []) {
        A superOb = new A();
        B subOb = new B();

        // The superclass may be used by itself.
        superOb.i = 10;
        superOb.j = 20;
        System.out.println("Contents of superOb:");
        superOb.showij();
        superOb.showk();
        System.out.println();

        /* The subclass has access to all public members of 
         * its superclass. */
        subOb.i = 7;
        subOb.j = 9;
        subOb.k = 9;
        System.out.println("Contents of subOb:");
        subOb.showij();
        subOb.showk();
        System.out.println();

        System.out.println("Sum of i, j and k in subOb: ");
        subOb.sum();
    }
}
The output from this program is shown here:

Contents of superOb:
i and j: 10 20

Contents of subOb:
i and j: 7 8
k: 9

Sum of i, j and k in subOb:
i+j+k: 24

As you can see, the subclass B includes all of the members of its superclass, A. This is why subOb can access i and j and call showij(). Also, inside sum(), i and j can be referred to directly, as if they were part of B. Even though A is a superclass for B, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, a subclass can be a superclass for another subclass. The general form of a class declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name {
    // body of class
}
```

A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situations. For example, consider the following:

```java
class RefDemo {
    public static void main(String args[]) {
        BoxWeight weightbox = new BoxWeight(3, 5, 7, 8.37);
        Box plainbox = new Box();
        double vol;

        vol = weightbox.volume();
        System.out.println("Volume of weightbox is "+vol);
        System.out.println("Weight of weightbox is "+weightbox.weight);
        System.out.println();
        // assign BoxWeight reference to Box reference
        plainbox = weightbox;

        vol = plainbox.volume(); // OK, volume() defined in Box
        System.out.println("Volume of plainbox is "+vol);

        /* The following statement is invalid because plainbox
        does not define a weight member. */
        System.out.println("Weight of plainbox is "+plainbox.weight);
    }
}
```

Here, weightbox is a reference to BoxWeight objects, and plainbox is a reference to Box objects. Since BoxWeight is a subclass of Box, it is permissible to assign plainbox a reference to the weightbox object.
Using super
In the preceding examples, classes derived from Box were not implemented as efficiently or as robustly as they could have been. For example, the constructor for BoxWeight explicitly initializes the width, height, and depth fields of Box. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

super has two general forms. The first calls the superclass’ constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors
A subclass can call a constructor defined by its superclass by use of the following form of super:

```
super(arg-list);
```

Here, arg-list specifies any arguments needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass’ constructor. To see how super( ) is used, consider this improved version of the BoxWeight class:

```java
// BoxWeight now uses super to initialize its Box attributes
class BoxWeight extends Box {
    double weight; // weight of box
    // initialize width, height, and depth using super()
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
}
```

Here, BoxWeight( ) calls super( ) with the arguments w, h, and d. This causes the Box constructor to be called, which initializes width, height, and depth using these values. BoxWeight no longer initializes these values itself. It only needs to initialize the value unique to it: weight. This leaves Box free to make these values private if desired. In the preceding example, super( ) was called with three arguments. Since constructors can be overloaded, super( ) can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments.

A Second Use for super
The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

```
super.member
```

Here, member can be either a method or an instance variable. This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:
Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclass and a subclass. However, you can build hierarchies that contain as many layers of inheritance as you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another. For example, given three classes called A, B, and C, C can be a subclass of B, which is a subclass of A. When this type of situation occurs, each subclass inherits all of the traits found in all of its superclasses. In this case, C inherits all aspects of B and A.

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```java
// Using super to overcome name hiding.
class A {
    int i;
}

// Create a subclass by extending class A.
class B extends A {
    int i; // this i hides the i in A
    B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
    }

    void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
    }
}

class UseSuper {
    public static void main(String args[]) {
        B subOb = new B(1, 2);

        subOb.show();
    }
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```
The output produced by this program is shown here:

```
k: 3
```

**Dynamic Method Dispatch**

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java’s most powerful concepts: *dynamic method dispatch*. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let’s begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how.

When an overridden method is called through a superclass reference, Java determines which version
of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

**Why Overridden Methods?**
As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

**Using Abstract Classes**
There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class *Figure* used in the preceding example. The definition of *area()* is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class *Triangle*. It has no meaning if *area()* is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java’s solution to this problem is the *abstract method*.

You can require that certain methods be overridden by subclasses by specifying the *abstract* type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the *abstract* keyword in front of the *class* keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the *new* operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared *abstract* itself.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:
Using final with Inheritance

The keyword `final` has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of `final` apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java’s most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify `final` as a modifier at the start of its declaration. Methods declared as `final` cannot be overridden. The following fragment illustrates `final`:

```java
public static void main(String args[]) {  
    B b = new B(); 
    b.meth(); 
    b.meth(); 
}
```

Because `meth()` is declared as `final`, it cannot be overridden in `B`. If you attempt to do so, a compile-time error will result.
Using final to Prevent Inheritance
Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations. Here is an example of a final class:

```java
final class A {
    //...
}

// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
    //...
}
```

CHAPTER
9 Packages and Interfaces

Packages are containers for classes. They 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.

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 exposed only 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.

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:

```java
package pkg;
```

Here, `pkg` is the name of the package. For example, the following statement creates a package called `MyPackage`:

```java
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 from being part of that same package. Most real-world packages are spread across many 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:

```java
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

```java
package java.awt.image;
```

needs to be stored in `java\.awt\image` in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

A short package Example

```java
// 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();
    }
}
```
Access Protection

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:

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

The three access modifiers, 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.

<table>
<thead>
<tr>
<th>Access Level</th>
<th>Private</th>
<th>No Modifier</th>
<th>Protected</th>
<th>Public</th>
</tr>
</thead>
<tbody>
<tr>
<td>Some class</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Some package subclass</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Some package non-subclass</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Different package subclass</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Different package non-subclass</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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:

```java
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:

```java
import java.util.Date;
import java.io.*;
```

It must be emphasized that the import statement is optional. 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 {
}

In this version, Date is fully-qualified.

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, as a general rule, their methods are declared without any body. In practice, this means that you can define interfaces that 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 provide the complete set of methods required 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 non-extensible 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.

Defining an Interface
An interface is defined much like a class. This is a simplified general form of an interface:

```java
   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 is an example of an interface definition. It declares a simple interface that contains one method called callback( ) that takes a single integer parameter.
interface Callback {
    void callback(int param);
}

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 required by the interface. The general form of a class that includes the `implements` clause looks like this:

```java
class classname [extends superclass] [implements interface [, interface...]] {
    // class-body
}
```

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`.

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

```java
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.

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.

```java
// 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:

```java
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
    private int stk[];
    private int tos;

    // allocate and initialize stack
    FixedStack(int size) {
        stk = new int[size];
        tos = -1;
    }

    // Push an item onto the stack
    public void push(int item) {
        if (tos == stk.length - 1) // use length member
            System.out.println("Stack is full.");
        else
            stk[++tos] = item;
    }

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

10 Exception Handling

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime 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.

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 runtime 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 after a try block completes is put in a finally block.

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

```java
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 after try block ends
}
```
Exception Types

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:

```java
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 exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
    at Exc0.main(Exc0.java: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 that 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

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 MultipleCatches {
        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.");
        }
    }

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, `Throwable` instance must be an object of type `Throwable` or a subclass of `Throwable`. Primitive 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 in 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 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.

```java
// 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("Recought: " + e);
        }
    }
}
```

`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:

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

`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:

```java
// Demonstrate finally.
class FinallyDemo {
    // Throw 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();
    }
}
```
Java’s Built-in Exceptions

Inside the standard package `java.lang`, Java defines several exception classes.

<table>
<thead>
<tr>
<th>Exception</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArithmeticException</td>
<td>Arithmetic error, such as divide-by-zero.</td>
</tr>
<tr>
<td>ArrayIndexOutOfBoundsException</td>
<td>Array index is out-of-bounds.</td>
</tr>
<tr>
<td>ArrayStoreException</td>
<td>Assignment to an array element of an incompatible type.</td>
</tr>
<tr>
<td>ClassCastException</td>
<td>Invalid cast.</td>
</tr>
<tr>
<td>EnumConstantNotPresentException</td>
<td>An attempt is made to use an undefined enumeration value.</td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td>Illegal argument used to invoke a method.</td>
</tr>
<tr>
<td>IllegalMonitorStateException</td>
<td>Illegal monitor operation, such as waiting on an unlocked thread.</td>
</tr>
<tr>
<td>IllegalStateException</td>
<td>Environment or application is in incorrect state.</td>
</tr>
<tr>
<td>IllegalThreadStateException</td>
<td>Requested operation not compatible with current thread state.</td>
</tr>
<tr>
<td>IndexOutOfBoundsException</td>
<td>Some type of index is out-of-bounds.</td>
</tr>
<tr>
<td>NegativeArraySizeException</td>
<td>Array created with a negative size.</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>Invalid use of a null reference.</td>
</tr>
<tr>
<td>NumberFormatException</td>
<td>Invalid conversion of a string to a numeric format.</td>
</tr>
<tr>
<td>SecurityException</td>
<td>Attempt to violate security.</td>
</tr>
<tr>
<td>StringIndexOutOfBoundsException</td>
<td>Attempt to index outside the bounds of a string.</td>
</tr>
<tr>
<td>TypeNotPresentException</td>
<td>Type not found.</td>
</tr>
<tr>
<td>UnsupportedOperationException</td>
<td>An unsupported operation was encountered.</td>
</tr>
</tbody>
</table>

**Table 10-1** Java’s Unchecked `RuntimeException` Subclasses Defined in `java.lang`

<table>
<thead>
<tr>
<th>Exception</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassNotFoundException</td>
<td>Class not found.</td>
</tr>
<tr>
<td>CloneNotSupportedException</td>
<td>Attempt to clone an object that does not implement the <code>Cloneable</code> interface.</td>
</tr>
<tr>
<td>IllegalAccessException</td>
<td>Access to a class is denied.</td>
</tr>
<tr>
<td>InstantiationException</td>
<td>Attempt to create an object of an abstract class or interface.</td>
</tr>
<tr>
<td>InterruptedException</td>
<td>One thread has been interrupted by another thread.</td>
</tr>
<tr>
<td>NoSuchFieldException</td>
<td>A requested field does not exist.</td>
</tr>
<tr>
<td>NoSuchMethodException</td>
<td>A requested method does not exist.</td>
</tr>
<tr>
<td>ReflectiveOperationException</td>
<td>Superclass of reflection-related exceptions.</td>
</tr>
</tbody>
</table>

**Table 10-2** Java’s Checked Exceptions Defined in `java.lang`
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>final void addSuppressed(Throwable ew)</td>
<td>Adds ew to the list of suppressed exceptions associated with the invoking exception. Primarily for use by the try-with-resources statement.</td>
</tr>
<tr>
<td>Throwable fillInStackTrace()</td>
<td>Returns a Throwable object that contains a completed stack trace. This object can be rethrown.</td>
</tr>
<tr>
<td>Throwable getCause()</td>
<td>Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.</td>
</tr>
<tr>
<td>String getLocalizedMessage()</td>
<td>Returns a localized description of the exception.</td>
</tr>
<tr>
<td>String getMessage()</td>
<td>Returns a description of the exception.</td>
</tr>
<tr>
<td>StackTraceElement[] getStackTrace()</td>
<td>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.</td>
</tr>
<tr>
<td>final Throwable[] getSuppressed()</td>
<td>Obtains the suppressed exceptions associated with the invoking exception and returns an array that contains the result. Suppressed exceptions are primarily generated by the try-with-resources statement.</td>
</tr>
<tr>
<td>Throwable initCause(Throwable causeEx)</td>
<td>Associates causeEx with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.</td>
</tr>
<tr>
<td>void printStackTrace()</td>
<td>Displays the stack trace.</td>
</tr>
<tr>
<td>void printStackTrace(PrintStream stream)</td>
<td>Sends the stack trace to the specified stream.</td>
</tr>
<tr>
<td>void printStackTrace(PrintWriter stream)</td>
<td>Sends the stack trace to the specified stream.</td>
</tr>
<tr>
<td>void setStackTrace(StackTraceElement elements[])</td>
<td>Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use.</td>
</tr>
<tr>
<td>String toString()</td>
<td>Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable object.</td>
</tr>
</tbody>
</table>

Table 10-3 The Methods Defined by Throwable
CHAPTER 11 Multithreaded Programming

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 many 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 or visiting a web site. In process-based multitasking, 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. Multithreading requires 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 lighter weight. 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 lower in cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under Java’s control. However, multithreaded multitasking is.

Multithreading enables you to write efficient programs that make maximum use of the processing power available in the system. One important way multithreading achieves this is by keeping idle time 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 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 most of the time the program is idle, waiting for input. Multithreading helps you reduce this idle time because another thread can run when one is waiting.

If you have programmed for operating systems such as Windows, 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.

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 program. 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 single-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 for 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. As most readers know, over the past few years, multi-core systems have become commonplace. Of course, single-core systems are still in widespread use. It is important to understand that Java’s multithreading features work in both types of systems. In a single core system, concurrently executing threads share the CPU, with each thread receiving a slice of CPU time. Therefore, in a single-core system, two or more threads do not actually run at the same time, but idle CPU time is utilized. However, in multi-core systems, it is possible for two or more threads to actually execute simultaneously. In many cases, this can further improve program efficiency and increase the speed of certain operations.

Threads exist in several states. Here is a general description. 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 halts 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.

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:

• 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.
• 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 pre-empted — 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, 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.

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.

In Java, there is no class “Monitor”; instead, each object has its own implicit monitor 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 into the language.

Messaging
After you divide your program into separate threads, you need to define how they will communicate with each other. When programming with some 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.

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. Several of those used in this chapter are shown here:

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>getName</td>
<td>Obtain a thread’s name.</td>
</tr>
<tr>
<td>getPriority</td>
<td>Obtain a thread’s priority.</td>
</tr>
<tr>
<td>isAlive</td>
<td>Determine if a thread is still running.</td>
</tr>
<tr>
<td>Join</td>
<td>Wait for a thread to terminate.</td>
</tr>
<tr>
<td>Run</td>
<td>Entry point for the thread.</td>
</tr>
<tr>
<td>Sleep</td>
<td>Suspend a thread for a period of time.</td>
</tr>
<tr>
<td>Start</td>
<td>Start a thread by calling its run method.</td>
</tr>
</tbody>
</table>

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:

• It is the thread from which other “child” threads will be spawned.
• 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:

```java
// 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. After the name of the thread is changed, \texttt{t} is again output. This time, the new name of the thread is displayed.

Let’s look more closely at the methods defined by \texttt{Thread} that are used in the program. The \texttt{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:

\begin{verbatim}
static void sleep(long milliseconds) throws InterruptedException
\end{verbatim}

The number of milliseconds to suspend is specified in \texttt{milliseconds}. This method may throw an \texttt{InterruptedException}. The \texttt{sleep()} method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:

\begin{verbatim}
static void sleep(long milliseconds, int nanoseconds) throws InterruptedException
\end{verbatim}

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 \texttt{setName(\texttt{)}. You can obtain the name of a thread by calling \texttt{getName()} (but note that this is not shown in the program). These methods are members of the \texttt{Thread} class and are declared like this:

\begin{verbatim}
final void setName(String threadName)
final String getName()
\end{verbatim}

Here, \texttt{threadName} specifies the name of the thread.

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type \texttt{Thread}. Java defines two ways in which this can be accomplished:

\begin{itemize}
  \item You can implement the \texttt{Runnable} interface.
  \item You can extend the \texttt{Thread} class, itself.
\end{itemize}

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the \texttt{Runnable} interface. \texttt{Runnable} abstracts a unit of executable code. You can construct a thread on any object that implements \texttt{Runnable}. To implement \texttt{Runnable}, a class need only implement a single method called \texttt{run(\texttt{), which is declared like this:

\begin{verbatim}
public void run()
\end{verbatim}

Inside \texttt{run(\texttt{, you will define the code that constitutes the new thread. It is important to understand that \texttt{run(\texttt{ can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that \texttt{run(\texttt{ establishes the entry point for another, concurrent thread of execution within your program. This thread will end when \texttt{run(\texttt{ returns.

After you create a class that implements \texttt{Runnable, you will instantiate an object of type \texttt{Thread from within that class. \texttt{Thread defines several constructors. The one that we will use is shown here:

\begin{verbatim}
Thread(Runnable threadOb, String threadName)
\end{verbatim}

In this constructor, \texttt{threadOb} is an instance of a class that implements the \texttt{Runnable} interface. This defines where execution of the thread will begin. The name of the new thread is specified by \texttt{threadName}.

After the new thread is created, it will not start running until you call its \texttt{start(\texttt{ method, which is declared within \texttt{Thread}. In essence, \texttt{start(\texttt{ executes a call to \texttt{run(\texttt{. The \texttt{start(\texttt{ method is shown here:

\begin{verbatim}
void start()
\end{verbatim}
Extending Thread

The second way to create a thread is to create a new class that extends \textit{Thread}, and then to create an instance of that class. The extending class must override the \texttt{run()} method, which is the entry point for the new thread. It must also call \texttt{start()} to begin execution of the new thread.

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 \texttt{Thread} class defines several methods that can be overridden by a derived class. Of these methods, the only one that \textit{must} be overridden is \texttt{run()}. This is, of course, the same method required when you implement \texttt{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 \texttt{Thread}'s other methods, it is probably best simply to implement \texttt{Runnable}. Also, by implementing \texttt{Runnable}, your thread class does not need to inherit \texttt{Thread}, making it free to inherit a different class. Ultimately, which approach to use is up to you. However, throughout the rest of this chapter, we will create threads by using classes that implement \texttt{Runnable}.

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.

Using \texttt{isAlive()} and \texttt{join()}

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling \texttt{sleep()} within \texttt{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, \texttt{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 \texttt{isAlive()} on the thread. This method is defined by \texttt{Thread}, and its general form is shown here:

\begin{verbatim}
final boolean isAlive()
\end{verbatim}

The \texttt{isAlive()} method returns \texttt{true} if the thread upon which it is called is still running. It returns \texttt{false} otherwise. While \texttt{isAlive()} is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called \texttt{join()}, shown here:

\begin{verbatim}
final void join() throws InterruptedException
\end{verbatim}

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 \texttt{joins} it. Additional forms of \texttt{join()} allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, over a given period of time, 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 system implements 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:

```java
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 `static final` variables within `Thread`.

You can obtain the current priority setting by calling the `getPriority()` method of `Thread`, shown here:

```java
final int getPriority()
```

Implementations of Java may have radically different behavior when it comes to scheduling. 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.

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. A monitor is an object that is used as a mutually exclusive lock. 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. You can synchronize your code in either of two ways. Both involve the use of the `synchronized` keyword, and both are examined here.

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`.

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:

```java
synchronized (objRef) {
    // statements to be synchronized
}
```

Here, `objRef` is a reference to the object being synchronized. A synchronized block ensures that a call to a synchronized method that is a member of `objRef`s class occurs only after the current thread has successfully entered `objRef`’s monitor.

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, 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:

- `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()` or `notifyAll()`.
- `notify()` wakes up a thread that called `wait()` on the same object.
• **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.
  These methods are declared within **Object**, as shown here:
  ```java
  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. Before working through an example that illustrates interthread communication, an important point needs to be made. Although **wait( )** normally waits until **notify( )** or **notifyAll( )** is called, there is a possibility that in very rare cases the waiting thread could be awakened due to a **spurious wakeup**. In this case, a waiting thread resumes without **notify( )** or **notifyAll( )** having been called. (In essence, the thread resumes for no apparent reason.) Because of this remote possibility, Oracle recommends that calls to **wait( )** should take place within a loop that checks the condition on which the thread is waiting.

**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:
• In general, it occurs only rarely, when the two threads time-slice in just the right way.
• 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.)

**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 early versions of Java, such as Java 1.0, and modern versions, beginning with Java 2. Prior to Java 2, a program used **suspend( ), resume( ), and stop( )**, which are methods defined by **Thread**, to pause, restart, and stop the execution of a thread. Although these methods 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 was deprecated by Java 2 several years ago. 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, was deprecated by 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. The trouble is that **stop( )** causes any lock the calling thread holds to be released. Thus, the corrupted data might be used by another thread that is waiting on the same lock.
Because you can’t now use the suspend(), resume(), or stop() methods 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.

Obtaining A Thread’s State
As mentioned earlier in this chapter, a thread can exist in a number of different states. You can obtain the current state of a thread by calling the getState() method defined by Thread. It is shown here:

Thread.State getState()

It returns a value of type Thread.State that indicates the state of the thread at the time at which the call was made. State is an enumeration defined by Thread. (An enumeration is a list of named constants. It is discussed in detail in Chapter 12.) Here are the values that can be returned by getState():

<table>
<thead>
<tr>
<th>Value</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLOCKED</td>
<td>A thread that has suspended execution because it is waiting to acquire a lock.</td>
</tr>
<tr>
<td>NEW</td>
<td>A thread that has not begun execution.</td>
</tr>
<tr>
<td>RUNNABLE</td>
<td>A thread that either is currently executing or will execute when it gains access to the CPU.</td>
</tr>
<tr>
<td>TERMINATED</td>
<td>A thread that has completed execution.</td>
</tr>
<tr>
<td>TIMED_WAITING</td>
<td>A thread that has suspended execution for a specified period of time, such as when it has called sleep(). This state is also entered when a timeout version of wait() or join() is called.</td>
</tr>
<tr>
<td>WAITING</td>
<td>A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of wait() or join().</td>
</tr>
</tbody>
</table>

![Thread State Diagram](image)
CHAPTER 13  I/O, Applets, and Other Topics

Streams
Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io package.

Byte Streams and Character Streams
Java defines two types of streams: byte and character. Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

Also, in some cases, character streams are more efficient than byte streams. The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. Although old code that doesn’t use character streams is becoming increasingly rare, it may still be encountered from time to time. As a general rule, old code should be updated to take advantage of character streams where appropriate.

One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters. An overview of both byte-oriented streams and character-oriented streams is presented in the following sections.

The Byte Stream Classes
Byte streams are defined by using two class hierarchies. At the top are two abstract classes: InputStream and OutputStream. Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers. The byte stream classes in java.io are shown in Table 13-1. A few of these classes are discussed later in this section. Others are described in Part II of this book. Remember, to use the stream classes, you must import java.io.
The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement. Two of the most important are read() and write(), which, respectively, read and write bytes of data. Each has a form that is abstract and must be overridden by derived stream classes.

The Character Stream Classes
Character streams are defined by using two class hierarchies. At the top are two abstract classes: Reader and Writer. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these. The character stream classes in java.io are shown in Table 13-2.

<table>
<thead>
<tr>
<th>Stream Class</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BufferedReader</td>
<td>Buffered input character stream</td>
</tr>
<tr>
<td>BufferedWriter</td>
<td>Buffered output character stream</td>
</tr>
<tr>
<td>CharArrayReader</td>
<td>Input stream that reads from a char array</td>
</tr>
<tr>
<td>CharArrayWriter</td>
<td>Output stream that writes to a char array</td>
</tr>
<tr>
<td>FileReader</td>
<td>Input stream that reads from a file</td>
</tr>
<tr>
<td>FileWriter</td>
<td>Output stream that writes to a file</td>
</tr>
<tr>
<td>FilterReader</td>
<td>Filtered reader</td>
</tr>
<tr>
<td>FilterWriter</td>
<td>Filtered writer</td>
</tr>
<tr>
<td>InputStreamReader</td>
<td>Input stream that translates bytes to characters</td>
</tr>
<tr>
<td>LineNumberReader</td>
<td>Input stream that counts lines</td>
</tr>
<tr>
<td>OutputStreamWriter</td>
<td>Output stream that translates characters to bytes</td>
</tr>
<tr>
<td>PipedReader</td>
<td>Input pipe</td>
</tr>
<tr>
<td>PipedWriter</td>
<td>Output pipe</td>
</tr>
<tr>
<td>PrintReader</td>
<td>Output stream that contains println()</td>
</tr>
<tr>
<td>PrintWriter</td>
<td>Output stream that contains print() and println()</td>
</tr>
<tr>
<td>PushbackReader</td>
<td>Input stream that allows characters to be returned to the input stream</td>
</tr>
<tr>
<td>Reader</td>
<td>Abstract class that describes character stream input</td>
</tr>
<tr>
<td>StringReader</td>
<td>Input stream that reads from a string</td>
</tr>
<tr>
<td>StringWriter</td>
<td>Output stream that writes to a string</td>
</tr>
<tr>
<td>Writer</td>
<td>Abstract class that describes character stream output</td>
</tr>
</tbody>
</table>
The abstract classes Reader and Writer define several key methods that the other stream classes implement. Two of the most important methods are read( ) and write( ), which read and write characters of data, respectively. Each has a form that is abstract and must be overridden by derived stream classes.

The Predefined Streams
As you know, all Java programs automatically import the java.lang package. This package defines a class called System, which encapsulates several aspects of the run-time environment. For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system. System also contains three predefined stream variables: in, out, and err. These fields are declared as public, static, and final within System. This means that they can be used by any other part of your program and without reference to a specific System object.

System.out refers to the standard output stream. By default, this is the console. System.in refers to standard input, which is the keyboard by default. System.err refers to the standard error stream, which also is the console by default. However, these streams may be redirected to any compatible I/O device.

System.in is an object of type InputStream; System.out and System.err are objects of type PrintStream. These are byte streams, even though they are typically used to read and write characters from and to the console. As you will see, you can wrap these within character-based streams, if desired.

Reading Console Input
In Java 1.0, the only way to perform console input was to use a byte stream. Today, using a byte stream to read console input is still acceptable. However, for commercial applications, the preferred method of reading console input is to use a character-oriented stream. This makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from System.in. To obtain a character based stream that is attached to the console, wrap System.in in a BufferedReader object. BufferedReader supports a buffered input stream. A commonly used constructor is shown here:

BufferedReader(Reader inputReader)

Here, inputReader is the stream that is linked to the instance of BufferedReader that is being created. Reader is an abstract class. One of its concrete subclasses is InputStreamReader, which converts bytes to characters. To obtain an InputStreamReader object that is linked to System.in, use the following constructor:

InputStreamReader(InputStream inputStream)

Because System.in refers to an object of type InputStream, it can be used for inputStream. Putting it all together, the following line of code creates a BufferedReader that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

After this statement executes, br is a character-based stream that is linked to the console through System.in.

Reading Characters
To read a character from a BufferedReader, use read( ). The version of read( ) that we will be using is int read( ) throws IOException Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns –1 when the end of the stream is encountered. As you can see, it can throw an IOException.
The following program demonstrates `read()` by reading characters from the console until the user types a "q." Notice that any I/O exceptions that might be generated are simply thrown out of `main()`. Such an approach is common when reading from the console in simple example programs such as those shown in this book, but in more sophisticated applications, you can handle the exceptions explicitly.

```java
import java.io.*;

class BRRead {
    public static void main(String args[]) throws IOException {
        char c;
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter characters, 'q' to quit.");
        // read characters
        do {
            c = (char) br.read();
            System.out.println(c);
        } while(c != 'q');
    }
}
```

### Reading Strings

To read a string from the keyboard, use the version of `readLine()` that is a member of the `BufferedReader` class. Its general form is shown here:

```
String readLine() throws IOException
```

As you can see, it returns a `String` object.

### Writing Console Output

Console output is most easily accomplished with `print()` and `println()`, described earlier, which are used in most of the examples in this book. These methods are defined by the class `PrintStream` (which is the type of object referenced by `System.out`). Even though `System.out` is a byte stream, using it for simple program output is still acceptable.

However, a character-based alternative is described in the next section. Because `PrintStream` is an output stream derived from `OutputStream`, it also implements the low-level method `write()`. Thus, `write()` can be used to write to the console. The simplest form of `write()` defined by `PrintStream` is shown here:

```
void write(int byteval)
```

This method writes the byte specified by `byteval`. Although `byteval` is declared as an integer, only the low-order eight bits are written.

Because `PrintStream` is an output stream derived from `OutputStream`, it also implements the low-level method `write()`. Thus, `write()` can be used to write to the console. The simplest form of `write()` defined by `PrintStream` is shown here:

```
void write(int byteval)
```

This method writes the byte specified by `byteval`. Although `byteval` is declared as an integer, only the low-order eight bits are written. Here is a short example that uses `write()` to output the character "A" followed by a newline to the screen:
You will not often use `write()` to perform console output (although doing so might be useful in some situations) because `print()` and `println()` are substantially easier to use.

The `PrintWriter` Class
Although using `System.out` to write to the console is acceptable, its use is probably best for debugging purposes or for sample programs, such as those found in this book. For real world programs, the recommended method of writing to the console when using Java is through a `PrintWriter` stream. `PrintWriter` is one of the character-based classes. Using a character-based class for console output makes internationalizing your program easier. `PrintWriter` defines several constructors. The one we will use is shown here:

```java
PrintWriter(OutputStream outputStream, boolean flushingOn)
```

Here, `outputStream` is an object of type `OutputStream`, and `flushingOn` controls whether Java flushes the output stream every time a `println()` method (among others) is called. If `flushingOn` is `true`, flushing automatically takes place. If `false`, flushing is not automatic.

`PrintWriter` supports the `print()` and `println()` methods. Thus, you can use these methods in the same way as you used them with `System.out`. If an argument is not a simple type, the `PrintWriter` methods call the object's `toString()` method and then display the result.

To write to the console by using a `PrintWriter`, specify `System.out` for the output stream and automatic flushing. For example, this line of code creates a `PrintWriter` that is connected to console output:

```java
PrintWriter pw = new PrintWriter(System.out, true);
```

The following application illustrates using a `PrintWriter` to handle console output:

```java
// Demonstrate PrintWriter
import java.io.*;

class PrintWriterDemo {
  public static void main(String args[]) {
    PrintWriter pw = new PrintWriter(System.out, true);
    pw.println("This is a string");
    int i = -7;
    pw.println(i);
    double d = 4.5e-7;
    pw.println(d);
  }
}
```

Reading and Writing Files
Java provides a number of classes and methods that allow you to read and write files. Before we begin, it is important to state that the topic of file I/O is quite large and file I/O is examined in detail in Part II. The purpose of this section is to introduce the basic techniques that read from and write to a file. Although bytes streams are used, these techniques can be adapted to the character-based streams.

Two of the most often-used stream classes are `FileInputStream` and `FileOutputStream`, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. Although both classes support additional constructors, the following are the forms that we will be using:
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException

Here, fileName specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then FileNotFoundException is thrown. For output streams, if the file cannot be opened or created, then FileNotFoundException is thrown. FileNotFoundException is a subclass of IOException. When an output file is opened, any preexisting file by the same name is destroyed.

When you are done with a file, you must close it. This is done by calling the close() method, which is implemented by both FileInputStream and FileOutputStream. It is shown here:

```java
void close() throws IOException
```

Closing a file releases the system resources allocated to the file, allowing them to be used by another file. Failure to close a file can result in “memory leaks” because of unused resources remaining allocated.

Before moving on, it is important to point out that there are two basic approaches that you can use to close a file when you are done with it. The first is the traditional approach, in which close() is called explicitly when the file is no longer needed. This is the approach used by all versions of Java prior to JDK 7 and is, therefore, found in all pre-JDK 7 legacy code. The second is to use the try-with-resources statement added by JDK 7, which automatically closes a file when it is no longer needed. In this approach, no explicit call to close() is executed. Since there is a large amount of pre-JDK 7 legacy code that is still being used and maintained, it is important that you know and understand the traditional approach. Therefore, we will begin with it. The new automated approach is described in the following section.

To read from a file, you can use a version of read() that is defined within FileInputStream. The one that we will use is shown here:

```java
int read() throws IOException
```

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. read() returns -1 when the end of the file is encountered. It can throw an IOException.

To write to a file, you can use the write() method defined by FileOutputStream. Its simplest form is shown here:

```java
void write(int byteval) throws IOException
```

This method writes the byte specified by byteval to the file. Although byteval is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing, an IOException is thrown.

**Automatically Closing a File**

In the preceding section, the example programs have made explicit calls to close() to close a file once it is no longer needed. As mentioned, this is the way files were closed when using versions of Java prior to JDK 7. Although this approach is still valid and useful, JDK 7 added a new feature that offers another way to manage resources, such as file streams, by automating the closing process. This feature, sometimes referred to as automatic resource management, or ARM for short, is based on an expanded version of the try statement. The principal advantage of automatic resource management is that it prevents situations in which a file (or other resource) is inadvertently not released after it is no longer needed. As explained, forgetting to close a file can result in memory leaks, and could lead to other problems.

Automatic resource management is based on an expanded form of the try statement. Here is its general form:
try (resource-specification) {
    // use the resource
}

Here, resource-specification is a statement that declares and initializes a resource, such as a file stream. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the try block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. (Thus, there is no need to call close( ) explicitly.) Of course, this form of try can also include catch and finally clauses. This new form of try is called the try-with-resources statement.

The try-with-resources statement can be used only with those resources that implement the AutoCloseable interface defined by java.lang. This interface defines the close( ) method. AutoCloseable is inherited by the Closeable interface in java.io. Both interfaces are implemented by the stream classes. Thus, try-with-resources can be used when working with streams, including file streams.

Applet Fundamentals
All of the preceding examples in this book have been Java console-based applications. However, these types of applications constitute only one class of Java programs. Another type of program is the applet. As mentioned in Chapter 1, applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document. After an applet arrives on the client, it has limited access to resources so that it can produce a graphical user interface and run various computations without introducing the risk of viruses or breaching data integrity.

Many of the issues connected with the creation and use of applets are found in Part II, when the applet package is examined, and also when Swing is described in Part III. However, the fundamentals connected to the creation of an applet are presented here, because applets are not structured in the same way as the programs that have been used thus far. As you will see, applets differ from console-based applications in several key areas. Let’s begin with the simple applet shown here:

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

public class SimpleApplet extends Applet {
    public void paint(Graphics g) {
        g.drawString("A Simple Applet", 20, 20);
    }
}
```

This applet begins with two import statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through a GUI framework, not through the console-based I/O classes. One of these frameworks is the AWT, and that is the framework used here to introduce applet programming. The AWT contains very basic support for a window-based, graphical user interface. As you might expect, the AWT is quite large, and a detailed discussion of it is found in Part II of this book.

Fortunately, this simple applet makes very limited use of the AWT. (Another commonly used GUI for applets is Swing, but this approach is described later in this book.) The second import
statement imports the `applet` package, which contains the class `Applet`. Every AWT-based applet that you create must be a subclass (either directly or indirectly) of `Applet`.

The next line in the program declares the class `SimpleApplet`. This class must be declared as `public`, because it will be accessed by code that is outside the program. Inside `SimpleApplet`, `paint()` is declared. This method is defined by the AWT and must be overridden by the applet. `paint()` is called each time that the applet must redisplay its output. This situation can occur for several reasons. For example, the window in which the applet is running can be overwritten by another window and then uncovered. Or, the applet window can 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 contains 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.

Inside `paint()` is a call to `drawString()`, which is a member of the `Graphics` class. This method outputs a string beginning at the specified X,Y location. It has the following general form:

```java
void drawString(String message, int x, int y)
```

Here, `message` is the string to be output beginning at `x, y`. In a Java window, the upper-left corner is location `0,0`. The call to `drawString()` in the applet causes the message "A Simple Applet" to be displayed beginning at location `20,20`. Notice that the applet does not have a `main()` method. Unlike Java programs, applets do not begin execution at `main()`. In fact, most applets don't even have a `main()` method. Instead, an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

After you enter the source code for `SimpleApplet`, compile in the same way that you have been compiling programs. However, running `SimpleApplet` involves a different process. In fact, there are two ways in which you can run an applet:

- **Executing the applet within a Java-compatible web browser.**
- **Using an applet viewer, such as the standard tool, `appletviewer`.** An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

Each of these methods is described next.

One way to execute an applet in a web browser is to write a short HTML text file that contains a tag that loads the applet. At the time of this writing, Oracle recommends using the `APPLET` tag for this purpose. (The `OBJECT` tag can also be used. See Chapter 23 for further information regarding applet deployment strategies.) Using `APPLET`, here is the HTML file that executes `SimpleApplet`:

```html
<applet code="SimpleApplet" width=200 height=60>
</applet>
```

The `width` and `height` statements specify the dimensions of the display area used by the applet. (The `APPLET` tag contains several other options that are examined more closely in Part II.) After you create this file, you can use it to execute the applet. To execute `SimpleApplet` with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called `RunApp.html`, then the following command line will run `SimpleApplet`:

```
C:\>appletviewer RunApp.html
```

While the subject of applets is more fully discussed later in this book, here are the key points that you should remember now:

- Applets do not need a `main()` method.
- Applets must be run under an applet viewer or a Java-compatible browser.
- User I/O is not accomplished with Java’s stream I/O classes. Instead, applets use the interface provided by a GUI framework.
The transient and volatile Modifiers
Java defines two interesting type modifiers: transient and volatile. These modifiers are used to handle somewhat specialized situations. When an instance variable is declared as transient, then its value need not persist when an object is stored. For example:

class T {
    transient int a; // will not persist
    int b; // will persist
}

Here, if an object of type T is written to a persistent storage area, the contents of a would not be saved, but the contents of b would. The volatile modifier tells the compiler that the variable modified by volatile can be changed unexpectedly by other parts of your program. One of these situations involves multithreaded programs. In a multithreaded program, sometimes two or more threads share the same variable. For efficiency considerations, each thread can keep its own, private copy of such a shared variable. The real (or master) copy of the variable is updated at various times, such as when a synchronized method is entered. While this approach works fine, it may be inefficient at times. In some cases, all that really matters is that the master copy of a variable always reflects its current state. To ensure this, simply specify the variable as volatile, which tells the compiler that it must always use the master copy of a volatile variable (or, at least, always keep any private copies up-to-date with the master copy, and vice versa). Also, accesses to the master variable must be executed in the precise order in which they are executed on any private copy.

Using instanceof
Sometimes, knowing the type of an object during run time is useful. For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects. In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object’s type at run time is important involves casting. In Java, an invalid cast causes a runtime error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts that can be detected only at run time. For example, a superclass called A can produce two subclasses, called B and C. Thus, casting a B object into type A or casting a C object into type A is legal, but casting a B object into type C (or vice versa) isn’t legal. Because an object of type A can refer to objects of either B or C, how can you know, at run time, what type of object is actually being referred to before attempting the cast to type C? It could be an object of type A, B, or C. If it is an object of type B, a run-time exception will be thrown. Java provides the run-time operator instanceof to answer this question.

The instanceof operator has this general form:

`objref instanceof type`

Here, objref is a reference to an instance of a class, and type is a class type. If objref is of the specified type or can be cast into the specified type, then the instanceof operator evaluates to true. Otherwise, its result is false. Thus, instanceof is the means by which your program can obtain run-time type information about an object.

strictfp
With the creation of Java 2, the floating-point computation model was relaxed slightly. Specifically, the new model does not require the truncation of certain intermediate values that occur during a computation. This prevents overflow or underflow in some cases. By modifying a class, a method, or interface with strictfp, you ensure that floating-point calculations (and thus all truncations) take
place precisely as they did in earlier versions of Java. When a class is modified by strictfp, all the methods in the class are also modified by strictfp automatically.

Native Methods
Although it is rare, occasionally you may want to call a subroutine that is written in a language other than Java. Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code. For example, you may want to call a native code subroutine to achieve faster execution time. Or, you may want to use a specialized, third-party library, such as a statistical package. However, because Java programs are compiled to bytecode, which is then interpreted (or compiled on-the-fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the native keyword, which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the native modifier, but do not define any body for the method. For example:

```java
public native int meth();
```

After you declare a native method, you must write the native method and follow a rather complex series of steps to link it with your Java code. Most native methods are written in C. The mechanism used to integrate C code with a Java program is called the Java Native Interface (JNI). A detailed description of the JNI is beyond the scope of this book, but the approach described here provides sufficient information for simple applications.

Problems with Native Methods
Native methods seem to offer great promise, because they enable you to gain access to an existing base of library routines, and they offer the possibility of faster run-time execution. But native methods also introduce two significant problems:

- **Potential security risk** Because a native method executes actual machine code, it can gain access to any part of the host system. That is, native code is not confined to the Java execution environment. This could allow a virus infection, for example. For this reason, unsigned applets cannot use native methods. Also, the loading of DLLs can be restricted, and their loading is subject to the approval of the security manager.

- **Loss of portability** Because the native code is contained in a DLL, it must be present on the machine that is executing the Java program. Further, because each native method is CPU- and operating system–dependent, each DLL is inherently nonportable. Thus, a Java application that uses native methods will be able to run only on a machine for which a compatible DLL has been installed. The use of native methods should be restricted, because they render your Java programs nonportable and pose significant security risks.

Using assert
Another relatively new addition to Java is the keyword assert. It is used during program development to create an assertion, which is a condition that should be true during the execution of the program. For example, you might have a method that should always return a positive integer value. You might test this by asserting that the return value is greater than zero using an assert statement. At run time, if the condition is true, no other action takes place. However, if the condition is false, then an AssertionError is thrown. Assertions are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.

The assert keyword has two forms. The first is shown here:
assert condition;
Here, condition is an expression that must evaluate to a Boolean result. If the result is true, then the assertion is true and no other action takes place. If the condition is false, then the assertion fails and a default AssertionError object is thrown.
The second form of assert is shown here:
assert condition: expr;
In this version, expr is a value that is passed to the AssertionError constructor. This value is converted to its string format and displayed if an assertion fails. Typically, you will specify a string for expr, but any non-void expression is allowed as long as it defines a reasonable string conversion.

Assertion Enabling and Disabling Options
When executing code, you can disable all assertions by using the -da option. You can enable or disable a specific package (and all of its subpackages) by specifying its name followed by three periods after the -ea or -da option. For example, to enable assertions in a package called MyPack, use
- ea: MyPack...
To disable assertions in MyPack, use
- da: MyPack...
You can also specify a class with the -ea or -da option. For example, this enables AssertDemo individually:
- ea: AssertDemo

Static Import
Java includes a feature called static import that expands the capabilities of the import keyword. By following import with the keyword static, an import statement can be used to import the static members of a class or interface. When using static import, it is possible to refer to static members directly by their names, without having to qualify them with the name of their class. This simplifies and shortens the syntax required to use a static member. To understand the usefulness of static import, let’s begin with an example that does not use it. The following program computes the hypotenuse of a right triangle. It uses two static methods from Java’s built-in math class Math, which is part of java.lang. The first is Math.pow(), which returns a value raised to a specified power. The second is Math.sqrt(), which returns the square root of its argument.

Invoking Overloaded Constructors Through this() 
When working with overloaded constructors, it is sometimes useful for one constructor to invoke another. In Java, this is accomplished by using another form of the this keyword. The general form is shown here:
this(arg-list)
When this() is executed, the overloaded constructor that matches the parameter list specified by arg-list is executed first. Then, if there are any statements inside the original constructor, they are executed. The call to this() must be the first statement within the constructor.