Class

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such,the class forms the basis for object-oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class.

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class. Because an object is an instance of a class, you will often see the two words object and instance used interchangeably.

The general form of class

```
class classname {
        type instance-variable1;
        type instance-variable2;
        // ...
        type instance-variableN;
        type methodname1(parameter-list) {
        // body of method1
        }
        type methodname2(parameter-list) {
        // body of method2
        }
        // ...
        type methodnameN(parameter-list) {
        // body of methodN
        }
                }
```

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**.

Example of Class

```
class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
        double volume() {
        return width * height * depth;
        }
        // sets dimensions of box
```

```
        void setDim(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
        }
}
class BoxDemo {
        public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

Creating 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.

Box mybox = new Box();

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

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

Constructors

Java allows objects to initialize themselves when they are created. This automatic

initialization is performed through the use of a **constructor**. A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.

/* Here, Box uses a constructor to initialize the dimensions of a box.*/

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

        // This is the constructor for Box.
        Box() {
        System.out.println("Constructing Box");
        width = 10;
        height = 10;
        depth = 10;
        }

        // compute and return volume
        double volume() {
        return width * height * depth;
        }
        }
        class BoxDemo6 {
        public static void main(String args[]) {

        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
```

Volume is 1000.0
Volume is 1000.0

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be **overloaded**, and the process is referred to as **method overloading**. Method overloading is one of the ways that Java implements polymorphism.

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.

```
// Demonstrate method overloading.
class OverloadDemo {
        void test() {
        System.out.println("No parameters");
        }

        // Overload test for one integer parameter.
        void test(int a) {
        System.out.println("a: " + a);
        }

        // Overload test for two integer parameters.
        void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
        }

        // overload test for a double parameter
        double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
}
}
class Overload {
public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        double result;

        // call all versions of test()
```

```
        ob.test();
        ob.test(10);
        ob.test(10, 20);
        result = ob.test(123.25);
        System.out.println("Result of ob.test(123.25): " + result);
}
}
```

This program generates the following output:

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

Overloading Constructors

In addition to overloading normal methods, one can also overload constructors.

```
/* Here, Box defines three constructors to initializethe dimensions of a box various ways.*/
class Box {
        double width;
        double height;
        double depth;

        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
        }

        // constructor used when no dimensions specified
        Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
        }

        // constructor used when cube is created
        Box(double len) {
        width = height = depth = len;
        }

        // compute and return volume
        double volume() {
```

```
        return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {

        // create boxes using the various constructors
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;

        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);

        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);

        // get volume of cube
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0

Static Variables and Methods

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 call other static methods.
■ They must only access static data.
■ They cannot refer to this or super in any way. (The keyword super relates to inheritance.)
If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded.

```
// Demonstrate static variables, methods, and blocks.
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);
}
}
```

Here is the output of the program:
Static block initialized.
x = 42
a = 3
b = 12

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

Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.
classname.method( )

Example
Inside main( ), the static method callme( ) and the static variable b are accessed outside of their class.

```
class StaticDemo {
        static int a = 42;
        static int b = 99;
        static void callme() {
        System.out.println("a = " + a);
        }
}
class StaticByName {
        public static void main(String args[]) {
        StaticDemo.callme();
        System.out.println("b = " + StaticDemo.b);
}
}
```

Here is the output of this program:
a = 42
b = 99

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

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

Arrays
In Java Arrays are implemented as objects.

```
// This program demonstrates the length array member.

class Length {
public static void main(String args[ ]) {
int a1[ ] = new int[10];
int a2[ ] = {3, 5, 7, 1, 8, 99, 44, -10};
int a3[ ] = {4, 3, 2, 1};
System.out.println("length of a1 is " + a1.length);
System.out.println("length of a2 is " + a2.length);
System.out.println("length of a3 is " + a3.length);
```

```
}
}
```
This program displays the following output:
length of a1 is 10
length of a2 is 8
length of a3 is 4

Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a **superclass**. The class that does the inheriting is called a **subclass**. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

To inherit a class, you simply incorporate the definition of one class into another by using the **extends keyword**.

```
// 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();
        System.out.println();

        // The subclass has access to all public members of its superclass.
        subOb.i = 7;
        subOb.j = 8;
        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.

Note: Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private.**

Super Keyword

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.

Using super to Call Superclass Constructors

A subclass can call a constructor method defined by its superclass by use of the following form of super:

super(parameter-list);

Here, parameter-list specifies any parameters needed by the constructor in the superclass. super( ) must always be the first statement executed inside a subclass' constructor.

Example

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

        // construct clone of an object
        Box(Box ob) {                   // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
        }
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
        }

        // constructor used when no dimensions specified
        Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
        }

        // constructor used when cube is created
        Box(double len) {
        width = height = depth = len;
        }

        // compute and return volume
        double volume() {
        return width * height * depth;
        }
}

// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
```

```
        double weight; // weight of box

        // construct clone of an object
        BoxWeight(BoxWeight ob) { // pass object to constructor
        super(ob);
        weight = ob.weight;
        }

        // constructor when all parameters are specified
        BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
        }

        // default constructor
        BoxWeight() {
        super();
        weight = -1;
        }

        // constructor used when cube is created
        BoxWeight(double len, double m) {
        super(len);
        weight = m;
        }
}
class DemoSuper {
        public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        System.out.println();
        vol = mybox3.volume();
        System.out.println("Volume of mybox3 is " + vol);
        System.out.println("Weight of mybox3 is " + mybox3.weight);
```

```
        System.out.println();
        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
        System.out.println("Weight of myclone is " + myclone.weight);
        System.out.println();
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
        System.out.println("Weight of mycube is " + mycube.weight);
        System.out.println();
        }
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
Volume of mybox3 is -1.0
Weight of mybox3 is -1.0
Volume of myclone is 3000.0
Weight of myclone is 34.3
Volume of mycube is 27.0
Weight of mycube is 2.0
```

A Second Use for super

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.

This usage has the following general form:

super.member

Here, member can be either a method or an instance variable.

Consider this simple

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

// 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,int c) {
        super.i = a; // i in A
        i = b; // i in B
```

```
        j=c;
        }

        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, 2,3);
        subOb.show();
        }
}
```

This program displays the following:
i in superclass: 1
i in subclass: 2

Package

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

The package is both a **naming** and a **visibility** control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a package

package pkg_name;

Here,package is a keyword and pkg_name is the name of the package. For example, the following statement creates a package called MyPackage.

package MyPackage;

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

package pkg1[.pkg2[.pkg3]];

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

package java.awt.image;
needs to be stored in java\awt\image on the Windows file system.

Access Modifiers

The three access specifiers, **private, public, and protected**, provide a variety of ways to produce the many levels of access required by these categories. Table below sums up the interactions

	Private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Table 9-1. *Class Member Access*

An Access Example

```
//This is file Protection.java:
package p1;
public class Protection {
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;

        public Protection() {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}

//This is file Derived.java:

package p1;
class Derived extends Protection {
        Derived() {
        System.out.println("derived constructor");
```

```
        System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}

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

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

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

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

// This is file Demo.java in  package p1.
package p1;
public class Demo {
        public static void main(String args[]) {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
        }
}

//This is file Demo.java in package p2.
package p2;
public class Demo {
        public static void main(String args[]) {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
        }
}
```

Importing Packages

Classes within packages must be fully qualified with their package name or names.It could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import statement** to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.

The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing.

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement:

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

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify

either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

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

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 {
}
```

NOTE: When a package is imported, only those items within the package declared as **public** will be available to **non-subclasses** in the importing code.

For example, if you want the Balance class of the package MyPack to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

```
package MyPack;
/* Now, the Balance class, its constructor, and its
show() method are public. This means that they can
be used by non-subclass code outside their package.
*/
public class Balance {
        String name;
        double bal;

        public Balance(String n, double b) {
        name = n;
        bal = b;
        }

        public void show() {
        if(bal<0)
        System.out.print("--> ");
        System.out.println(name + ": $" + bal);
        }
```

```
}
```

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

```
import MyPack.*;

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

Experiment- Remove the public specifier from the Balance class and then try compiling TestBalance. As explained, errors will result.

Interfaces

Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.

Using interface, you can specify what a class must do, but not how it does it.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the **"one interface, multiple methods"** aspect of polymorphism. Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.

Defining an Interface

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

***access* interface *name* {**

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

Here, **access** is either **public or not used**. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as public, the interface can be used by any other code. **name** is the name of the interface, and can be any valid identifier.

Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. **They must also be initialized with a constant value**. All methods and variables are implicitly public if the interface, itself, is declared as public.

Implementing Interface

```
interface Callback {
        void callback(int param);
}
--Save this file as Callback.java

class Client implements Callback {
        // Implement Callback's interface

        public void callback(int p) {
        System.out.println("callback called with " + p);
        }

        void nonIfaceMeth() {
        System.out.println("Classes that implement interfaces " +
        "may also define other members, too.");
        }
}

// Another implementation of Callback.
```

```
class AnotherClient implements Callback {
        // Implement Callback's interface

        public void callback(int p) {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
}
}

class TestIface {
        public static void main(String args[ ]) {
        Client ob1 = new Client();
        AnotherClient ob2 = new AnotherClient();
        ob1.callback(42);
        ob2.callback(42);
        }
}
```

The output from this program is shown here:

```
callback called with 42
Another version of callback
p squared is 1764
```

Exception-Handling

An **exception** is an abnormal condition that arises in a code sequence at **run time**. In other words, **an exception is a run-time error**. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome.Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

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

General form of an exception-handling block:

```
try {
```

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

Here, ExceptionType is the type of exception that has occurred.

Uncaught Exceptions

```
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.

Here is the output 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.
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.

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

This program generates the following **output:**
Division by zero.
After catch statement.

Notice that the call to println( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

Put differently, catch is not "called," so execution never "returns" to the try block from a catch. Thus, the line "This will not be printed." is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.
For example, in the next program each iteration of the for loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into a. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues.

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
        public static void main(String args[]) {
        int a=0, b=0, c=0;
        Random r = new Random();
        for(int i=0; i<32000; i++) {
        try {
        b = r.nextInt();
        c = r.nextInt();
        a = 12345 / (b/c);
```

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

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. **You can throw exceptions yourself by using the throw statement.** The general form of throw is :

throw ThrowableInstance;

The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

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

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

Here is the resulting output:
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo

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 the general form of a method declaration that includes a throws clause:

type method-name(parameter-list) throws exception-list
{
// body of method
}
Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Example

```
class ThrowsDemo {
        static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
}

public static void main(String args[]) {
        try {
        throwOne();
        } catch (IllegalAccessException e) {
        System.out.println("Caught " + e);
        }
}
}
```

Here is the output generated by running this example program:
inside throwOne
caught java.lang.IllegalAccessException: demo

finally

When exceptions are thrown, execution in a method takes a rather abrupt, 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.**
Each try statement requires at least one catch or a finally clause.

Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

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

```
        }
public static void main(String args[]) {
try {
procA();
} catch (Exception e) {
System.out.println("Exception caught");
}
procB();
procC();
}
}
```

In this example, procA( ) prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. procB( )'s try statement is exited via a return statement. The finally clause is executed before procB( ) returns. In procC( ), the try statement executes normally, without error. However, the finally block is still executed.
If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

Here is the **output** generated by the preceding program:

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

Multithreading

Unlike most other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains **two or more parts that can run concurrently.** Each part of such a program is called a **thread,** and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

Multitasking threads require less overhead than multitasking processes. Processes are **heavyweight** tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.

Multithreading enables to write very efficient programs that make maximumn use of the CPU, because idle time can be kept to a minimum.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread class**, **its methods, and its companion interface, Runnable. To create a new thread, program will either extend Thread or implement the Runnable interface**. The Thread class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

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

fig. The methods in the thread class

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 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. It 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.

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

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
```

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.**

The **sleep( ) method** causes the thread from which it is called **to suspend** execution **for the specified period of milliseconds**. Its general form is shown here:

static void sleep(long milliseconds) throws InterruptedException

The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException.

Creating a Thread

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

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

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 Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called **run( ),** which is declared like this:

public void run( )

Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will **instantiate an object of type Thread** from within that class.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread. **In essence, start( ) executes a call to run( ).**

```
// Create a second thread.
class NewThread implements Runnable {
Thread t;
NewThread() {           // Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}

// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}

class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

The output produced by this program is as follows:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
```

```
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

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

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three **child threads:**

```
// Create multiple threads.
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;

        NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        }

        // This is the entry point for thread.
        public void run() {
        try {
        for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
        System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
        }
        }
```

```
class MultiThreadDemo {
    public static void main(String args[]) {
    new NewThread("One"); // start threads
    new NewThread("Two");
    new NewThread("Three");
    try {
    // wait for other threads to end
    Thread.sleep(10000);
    } catch (InterruptedException e) {
    System.out.println("Main thread Interrupted");
    }
    System.out.println("Main thread exiting.");
    }
    }
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

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

Using isAlive( ) and join( )

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling sleep( ) within main( ), with a long

enough delay to ensure that all child threads terminate prior to the main thread.However, this is hardly a satisfactory solution. Fortunately, Thread provides a means by which you can answer this question. Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread.

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

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

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates.

Here is an improved version of the preceding example that uses join( ) to ensure that the main thread is the last to stop. It also demonstrates the isAlive( ) method.

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;
        NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        }
        // This is the entry point for thread.
        public void run() {
        try {
        for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
        }
        } catch (InterruptedException e) {
        System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
}
}

class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
```

```
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
System.out.println("Main thread exiting.");
}
}
```

Sample output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
```

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

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

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.

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.

// This program is not synchronized.

```
class Callme {
        void  call(String msg) {
        System.out.print("[" + msg);
        try {
        Thread.sleep(1000);
        } catch(InterruptedException e) {
        System.out.println("Interrupted");
        }
        System.out.println("]");
        }
}
class Caller implements Runnable {
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
        }
```

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

Here is the output produced by this program:
Hello[Synchronized[World]
]

NOTE-Since the Call method is not synchronized more than one thread can access this method simultenously and result in the **Race Condition.**

To prevent from the race condition precede the call() with the keyword synchronized and analyze the output.

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

After synchronized has been added to call( ), the output of the program is as follows:
[Hello]
[Synchronized]
[World]

Suspending, Resuming, and Stopping Threads Using Java 2

wait()-method is invoked to suspend the execution of the thread.
notify()-method is invoked to wake up the thread.

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

```
            System.out.println("Suspending thread One");
            Thread.sleep(1000);
            ob1.myresume();
            System.out.println("Resuming thread One");
            ob2.mysuspend();
            System.out.println("Suspending thread Two");
            Thread.sleep(1000);
            ob2.myresume();
            System.out.println("Resuming thread Two");
            } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
            }
            // wait for threads to finish
            try {
            System.out.println("Waiting for threads to finish.");
            ob1.t.join();
            ob2.t.join();
            } catch (InterruptedException e) {
            System.out.println("Main thread Interrupted");
            }
            System.out.println("Main thread exiting.");
            }
}
```

Sample output from this program is shown here:

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

```
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

JAVA I/O STREAMS

Most programs use data in one form or another, whether it is as input, output, or both. The Java Development Kit (JDK) provides APIs for reading and writing streams of data. These APIs have been part of the core JDK since version 1.0, but are often overshadowed by the more well-known APIs, such as JavaBeans, RMI, JDBC, and so on. However, input and output streams are the backbone of the JDK APIs, and understanding them is not only crucial, but can also make programming with them a lot of fun.

To bring data into a program, a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. On the flip side, a program can open a stream to a data source and write to it in a serial fashion. Whether you are reading from a file or from a socket, the concept of serially reading from, and writing to different data sources is the same. For that very reason, once you understand the top level classes (java.io.Reader, java.io.Writer), the remaining classes are straightforward to work with.

Character Streams versus Byte Streams

Prior to JDK 1.1, the input and output classes (mostly found in the java.io package) only supported **8-bit byte streams**. The concept of **16-bit Unicode character** streams was introduced in JDK 1.1. While **byte streams** were supported via the **java.io.InputStream** and **java.io.OutputStream classes** and their subclasses, **character streams** are implemented by the **java.io.Reader and java.io.Writer classes** and their subclasses.

Most of the functionality available for byte streams is also provided for character streams. The

methods for character streams generally accept parameters of data type char parameters, while byte streams, you guessed it, work with byte data types. The names of the methods in both sets of classes are almost identical except for the suffix, that is, character-stream classes end with the suffix Reader or Writer and byte-stream classes end with the suffix InputStream and OutputStream. **For example, to read files using character streams, you would use the java.io.FileReader class; for reading it using byte streams you would use java.io.FileInputStream.**

Unless you are working with binary data, such as image and sound files, you should use readers and writers (character streams) to read and write information for the following reasons:

- They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
- They are easier to internationalize because they are not dependent upon a specific character encoding.
- They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

The Byte Stream Classes

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains **print( )** and **println( )**
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Table 12-1. *The Byte Stream Classes*

The Character Stream Classes

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains **print( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

Table 12-2. *The Character Stream I/O Classes*

Reading Console Input

In Java, console input is accomplished by reading from **System.in.** To obtain a character-based stream that is attached to the console, you wrap **System.in** in a **BufferedReader object**, to create a character stream.

BufferedReader br = new BufferedReader(new InputStreamReader(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":

```
// Use a BufferedReader to read characters from the console.
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');
        }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

This output may look a little different from what you expected, because **System.in is line buffered, by default**. **This means that no input is actually passed to the program until you press ENTER.** As you can guess, this does not make read( ) particularly valuable for interactive, console input.

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.

The following program demonstrates BufferedReader and the readLine( ) method to read from the console.It creates an array of String objects and then reads in lines of text, storing each line in the array. It will read up to 100 lines or until you enter "stop".

```
import java.io.*;
class TinyEdit {
```

```
        public static void main(String args[])
        throws IOException
        {
        // create a BufferedReader using System.in
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str[] = new String[100];
        System.out.println("Enter lines of text.");
        System.out.println("Enter 'stop' to quit.");
        for(int i=0; i<100; i++) {
        str[i] = br.readLine();
        if(str[i].equals("stop")) break;
        }
        System.out.println("\nHere is your file:");
        // display the lines
        for(int i=0; i<100; i++) {
        if(str[i].equals("stop")) break;
        System.out.println(str[i]);
        }
        }
}
```

Here is a sample run:

```
Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
stop
Here is your file:
This is line one.
This is line two.
Java makes working with strings easy.
Just create String objects.
```

File I/O

The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. It contains two main abstract classess **InputStream and OutputStream**.

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. All of the methods in this class will throw an IOException on error conditions. Below are the list of the methods in InputStream :

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an IOException.
void mark(int numBytes)	Places a mark at the current point in the input stream that will remain valid until numBytes bytes are read.
boolean markSupported( )	Returns true if mark( )/reset( ) are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ])	Attempts to read up to buffer.length bytes into buffer and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte buffer[ ], int offset, int numBytes)	Attempts to read up to numBytes bytes into buffer starting at buffer[offset], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long numBytes)	Ignores (that is, skips) numBytes bytes of input, returning the number of bytes actually ignored.

OutputStream

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

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an IOException.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int b)	Writes a single byte to an output stream. Note that the parameter is an int, which allows you to call write( ) with expressions without having to cast them back to byte.
void write(byte buffer[ ])	Writes a complete array of bytes to an output stream.

void write(byte buffer[ ], int offset, int numBytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].

FileInputStream

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

FileInputStream(String filepath)
FileInputStream(File fileObj)

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

The following example creates two FileInputStreams that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("D:\abc.txt")
File f = new File("D:\abc.txt");
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allows us to closely examine the file using the File methods, before we attach it to an input stream. When a FileInputStream is created, it is also opened for reading.

The example below shows how to read a single byte, an array of bytes, and a subrange array of bytes. It also illustrates how to use **available( )** to determine the number of bytes remaining, and how to use the **skip( )** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory.

```
// Demonstrate FileInputStream.
import java.io.*;
class FileInputStreamDemo {
        public static void main(String args[]) throws Exception {
        int size;
        InputStream f = new FileInputStream("FileInputStreamDemo.java");
        System.out.println("Total Available Bytes: " + (size = f.available()));
        int n = size/40;
        System.out.println("First " + n + " bytes of the file one read() at a time");
        for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
        }
        System.out.println("\nStill Available: " + f.available());
        System.out.println("Reading the next " + n + " with one read(b[ ])");
        byte b[] = new byte[n];
        if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
        }
        System.out.println(new String(b, 0, n));
        System.out.println("\nStill Available: " + (size = f.available()));
```

```
        System.out.println("Skipping half of remaining bytes with skip()");
        f.skip(size/2);
        System.out.println("Still Available: " + f.available());
        System.out.println("Reading " + n/2 + " into the end of array");
        if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("couldn't read " + n/2 + " bytes.");
        }
        System.out.println(new String(b, 0, b.length));
        System.out.println("\nStill Available: " + f.available());
        f.close();
        }
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;
class FileInputS
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S
Still Available: 665
```

FileOutputStream

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

FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)

They can throw a FileNotFoundException or a SecurityException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, the file is opened in append mode. The fourth constructor was added by Java 2, version 1.4.

Creation of a FileOutputStream is not dependent on the file already existing. FileOutputStream will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an IOException will be thrown.

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

// Demonstrate FileOutputStream.

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

Here are the contents of each file after running this program.

First, file1.txt:

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

Next, file2.txt:

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

Finally, file3.txt:

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

The Character Streams

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with **Unicode characters**. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters.At the top of the **character stream hierarchies** are the **Reader**

and **Writer** abstract classes.

Reader

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

Method	Description
abstract void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numChars*)	Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported on this stream.
int read( )	Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered.
int read(char *buffer*[ ])	Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered.
abstract int read(char *buffer*[ ], int *offset*, int *numChars*)	Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when the end of the file is encountered.
boolean ready( )	Returns **true** if the next input request will not wait. Otherwise, it returns **false**.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numChars*)	Skips over *numChars* characters of input, returning the number of characters actually skipped.

Table 17-3. *The Methods Defined by* Reader

FileReader

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

FileReader(String filePath)

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

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

// Demonstrate FileReader.

```
import java.io.*;
class FileReaderDemo {
public static void main(String args[]) throws Exception {
FileReader fr = new FileReader("FileReaderDemo.java");
BufferedReader br = new BufferedReader(fr);
String s;
while((s = br.readLine()) != null) {
System.out.println(s);
}
fr.close();
}
}
```

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

Method	Description
abstract void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
abstract void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *ch*)	Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with expressions without having to cast them back to **char**.
void write(char *buffer*[ ])	Writes a complete array of characters to the invoking output stream.
abstract void write(char *buffer*[ ], int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream.
void write(String *str*)	Writes *str* to the invoking output stream.
void write(String *str*, int *offset*, int *numChars*)	Writes a subrange of *numChars* characters from the array *str*, beginning at the specified *offset*.

Table 17-4. *The Methods Defined by* Writer

FileWriter

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

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

They can throw an IOException. Here, filePath is the full path name of a file, and fileObj is a File object that describes the file. If append is true, then output is appended to the end of the file.

Creation of a FileWriter is not dependent on the file already existing. FileWriter will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an IOException will be thrown.The following example is a character stream version of an example shown earlier when FileOutputStream was discussed.

This version creates a sample buffer of characters by first making a String and then using the **getChars( )** method to extract the character array equivalent. It then creates three files. The first, **file1.txt**, will contain every other character from the sample. The second, **file2.txt**, will contain the entire set of characters. Finally, the third, **file3.txt**, will contain only the last quarter.

// **Demonstrate FileWriter.**

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