UNIT – II

JAVA

Introduction to object-oriented programming-Features of Java – Data types, variables and arrays –Operators – Control statements – Classes and Methods – Inheritance. Packages and Interfaces – Exception Handling – Multithreaded Programming – Input/Output – Files – Utility Classes – String Handling.

Introduction to object-oriented programming

All computer programs consist of two elements: code and data. These are the two paradigms that govern how a program is constructed. The first way is called the *process-oriented model.* The process-oriented model can be thought of as *code acting on data.*
Procedural languages such as C employ this model to considerable success. , problems with this approach appear as programs grow larger and more complex.

To manage increasing complexity, the second approach, called *object-oriented programming*, was conceived. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as *data controlling access to code.*

Abstraction

in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

Encapsulation

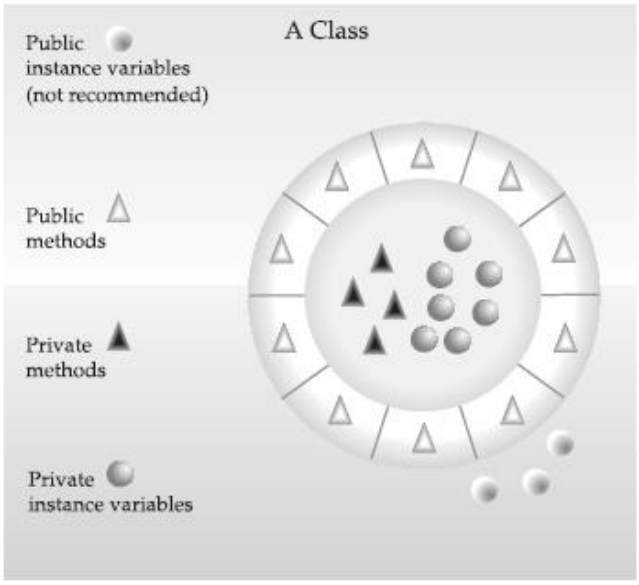
Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse.

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* the methods define how the member variables can be used.

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.

FIGURE 2-1
Encapsulation: public methods can be used to protect private data



Polymorphism
Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions.

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.

Features of Java

Java is −

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

Simple

Java was designed to be easy for the professional programmer to learn and use effectively.

If you already understand the basic concepts of object-oriented programming, learning Java will be even easier.

Object Oriented

In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

Security

In order for Java to enable applets to be downloaded and executed on the client computer safely, Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will
be done.

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.

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.

Platform Independent − Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

- **Secure** − With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architecture-neutral** − Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.
- **Portable** − Being architecture-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary, which is a POSIX subset.
- **Robust** − Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multithreaded** − With Java's multithreaded feature it is possible to write programs that can perform many tasks simultaneously. This design feature allows the developers to construct interactive applications that can run smoothly.

- **Interpreted** − Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light-weight process.
- **High Performance** − With the use of Just-In-Time compilers, Java enables high performance.
- **Distributed** − Java is designed for the distributed environment of the internet.
- **Dynamic** − Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

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.

```
class IncDec {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c;
int d;
c = ++b;c=3,b=3
d = a++;d=1,a=2
c++;c=4
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
System.out.println("d = " + d);
}
}
```

a=2

b=3

c=4

d=1

Inheritance

it allows the creation of hierarchical classifications. 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.*

Inheritance Basics

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

```
// 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
```

The general form of a **class** declaration that inherits a superclass is shown here:

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

Member Access and Inheritance

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**. For example, consider the following simple class hierarchy:

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**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private members remain
private to their class.
This program contains an error and will not
compile.
*/
// Create a superclass.
class A {
int i; // public by default
private int j; // private to A
```

```
void setij(int x, int y) {
i = x;
j = y;
}
}
// A's j is not accessible here.
class B extends A {
int total;
void sum() {
total = i + j; // ERROR, j is not accessible here
}
}
class Access {
public static void main(String args[]) {
B subOb = new B();
subOb.setij(10, 12);
subOb.sum();
System.out.println("Total is " + subOb.total);
}
    }
```

REMEMBER *A class member that has been declared as private will remain private to its class. It is*
not accessible by any code outside its class, including subclasses.

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.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded. For example, consider this modified version of the preceding example:

```
// Methods with differing type signatures are overloaded – not

// overridden.
class A {
int i, j;
A(int a, int b) {
i = a;
j = b;
}
// display i and j
void show() {
System.out.println("i and j: " + i + " " + j);
}
}
```

```
// Create a subclass by extending class A.
class B extends A {
int k;
B(int a, int b, int c) {
super(a, b);
k = c;
}
// overload show()
void show(String msg) {
System.out.println(msg + k);
}
}
class Override {
public static void main(String args[]) {
B subOb = new B(1, 2, 3);
subOb.show("This is k: "); // this calls show() in B
subOb.show(); // this calls show() in A
}
}
```

The output produced by this program is shown here:

```
This is k: 3
    i and j: 1 2
```

The version of **show( )** in **B** takes a string parameter. This makes its type signature different from the one in **A**, which takes no parameters. Therefore, no overriding (or name hiding) takes place. Instead, the version of **show( )** in **B** simply overloads the version of **show( )** in **A**.

Dynamic Method Dispatch

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.

When differenttypes 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.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
class A {
```

```
void callme() {
System.out.println("Inside A's callme method");
}
}           A<-B,A<-C

class B extends A {
// override callme()
void callme() {
System.out.println("Inside B's callme method");
}
}
class C extends A {
// override callme()
void callme() {
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]) {
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
A r; // obtain a reference of type A
r = a; // r refers to an A object
r.callme(); // calls A's version of callme
r = b; // r refers to a B object
r.callme(); // calls B's version of callme
r = c; // r refers to a C object
r.callme(); // calls C's version of callme
}
}
```

The output from the program is shown here:

```
Inside A’s callme method
Inside B’s callme method
Inside C’s callme method
```

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.

use this general form:
abstract *type name(parameter-list)*;

```
// A Simple demonstration of abstract.
abstract class A {
abstract void callme();
// concrete methods are still allowed in abstract classes
void callmetoo() {
System.out.println("This is a concrete method.");
}
}
class B extends A {
void callme() {
System.out.println("B's implementation of callme.");
}
}
class AbstractDemo {
public static void main(String args[]) {
B b = new B();
b.callme();
b.callmetoo();
}
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

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**:

```
class A {
final void meth() {
System.out.println("This is a final method.");
}
}
class B extends A {
void meth() { // ERROR! Can't override.
System.out.println("Illegal!");
}
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do

so, a compile-time error will result.

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.

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

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

A Short Package Example

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

```
// A simple package
package MyPack;
class Balance {
String name;
double bal;
Balance(String n, double b) {
name = n;
bal = b;
}
void show() {
if(bal<0)
System.out.print("--> ");
System.out.println(name + ": $" + bal);
}
}
class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

Call this file **AccountBalance.java** and put it in a directory called **MyPack**.
Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then, try executing the **AccountBalance** class, using the following command line:
java MyPack.AccountBalance

Importing Packages

This is the general form of the **import** statement:
import *pkg1*[.*pkg2*].(*classname*|*);

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:
import java.lang.*;

Interfaces
Using the keyword **interface**, you can fully abstract a class' interface from its implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

NOTE *Interfaces add most of the functionality that is required for many applications that would normally resort to using multiple inheritance in a language such as C++.*

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;
}

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. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. 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 defined by the interface. The general form of a class that includes the **implements** clause looks like this:

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**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements the **Callback** interface shown earlier.

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

Notice that **callback( )** is declared using the **public** access specifier.

REMEMBER *When you implement an interface method, it must be declared as* ***public.***

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

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

The output of this program is shown here:

```
callback called with 42
```

the second implementation of **Callback**,
shown here:

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

Now, try the following class:

```
class TestIface2 {
public static void main(String args[]) {
Callback c = new Client();
```

```
AnotherClient ob = new AnotherClient();
c.callback(42);
c = ob; // c now refers to AnotherClient object
c.callback(42);
}
}
```

The output from this program is shown here:

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

Stack that implemented a simple fixed-size stack. For example, the stack can be of a fixed size or it can be "growable." the methods **push( )** and **pop( )** define the interface to the stack independently of the details of the implementation.

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

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

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

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

Exception-Handling Fundamentals

AJava 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.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

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

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

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

}
Here, *ExceptionType* is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

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

Here is the exception generated when this example is executed:

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

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.
```

finally creates a block of code that will be executed after a **try**/**catch** block has

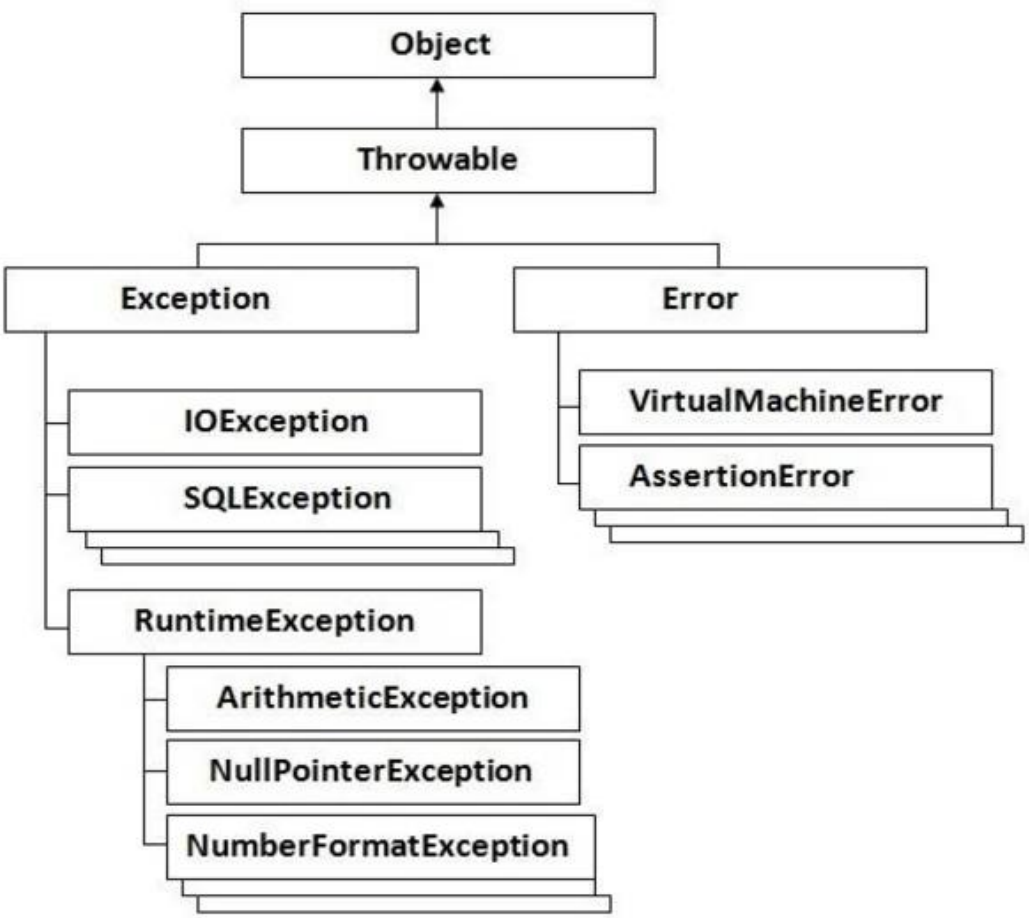
completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller 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.

```
// 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.

Here is the output generated by the preceding program:
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally

Java's Built-in Exceptions



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

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

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

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

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.
there are two distinct types of multitasking: process based and thread-based. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere.
Threads exist in several states. A thread can be *running.* It can be *ready to run* as soon as it gets CPU time. Arunning thread can be *suspended,* which temporarily suspends its activity. Asuspended 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.

Synchronization
The monitor is a control mechanism first defined by C.A.R. Hoare. 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. a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

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.
To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface.
The **Thread** class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

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

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.

the main thread is created automatically when your program is started, it can be controlled through a **Thread** object.

```
// 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. The **sleep( )** method in **Thread** might throw an **InterruptedException**.

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

Let's look more closely at the methods defined by **Thread** that are used in the program.
The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:
static void sleep(long *milliseconds*) throws InterruptedException
The number of milliseconds to suspend is specified in *milliseconds.* This method may throw an **InterruptedException**.
The **sleep( )** method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds:
static void sleep(long *milliseconds*, int *nanoseconds*) throws InterruptedException
This second form is useful only in environments that allow timing periods as short as nanoseconds.
As the preceding program shows, you can set the name of a thread by using **setName( )**.
You can obtain the name of a thread by calling **getName( )** (but note that this is not shown in the program). These methods are members of the **Thread** class and are declared like this:
final void setName(String *threadName*)
final String getName( )
Here, *threadName* specifies the name of the thread.

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.

Implementing Runnable
The easiest way to create a thread is to create a class that implements the **Runnable** interface.
Runnable abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:
public void run( )
Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.
a class that implements **Runnable**, you will instantiate an object of type
Thread from within that class. **Thread** defines several constructors. The one that we will use is shown here:
Thread(Runnable *threadOb*, String *threadName*)
In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.
After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:
void start( )

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

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

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

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

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

```
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.
```

Extending Thread

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

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

```
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

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

public Thread(String *threadName*)

Here, *threadName* specifies the name of the thread.

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");
```

```
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.

String Handling

when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically.

The String Constructors

The **String** class supports several constructors. To create an empty **String**, you call the default constructor. For example,
String s = new String();
will create an instance of **String** with no characters in it.
you will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:
String(char *chars*[ ])
Here is an example:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
```

This constructor initializes **s** with the string “abc”.
You can specify a subrange of a character array as an initializer using the following constructor:
String(char *chars*[ ], int *startIndex*, int *numChars*)
Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Here is an example:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars, 2, 3);
```

This initializes **s** with the characters **cde**.
You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:
String(String *strObj*)
Here, *strObj* is a **String** object. Consider this example:

```
// Construct one String from another.
class MakeString {
public static void main(String args[]) {
char c[] = {'J', 'a', 'v', 'a'};
String s1 = new String(c);
String s2 = new String(s1);
System.out.println(s1);
System.out.println(s2);
}
}
```

The output from this program is as follows:
Java
Java

You can construct a **String** from a **StringBuffer** by using the constructor shown here:
String(StringBuffer *strBufObj*)

The second new constructor supports the new **StringBuilder** class. It is shown here:
String(StringBuilder *strBuildObj*)
This constructs a **String** from the **StringBuilder** passed in *strBuildObj*.

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:
int length( )
The following fragment prints "3", since there are three characters in the string **s**:
```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

String Literals

The earlier examples showed how to explicitly create a **String** instance from an array of characters by using the **new** operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object. For example, the following code fragment creates two equivalent strings:
```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);
String s2 = "abc"; // use string literal
```
Because a **String** object is created for every string literal, you can use a string literal any place you can use a **String** object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the **length( )** method on the string "abc". As expected, it prints "3".
```
System.out.println("abc".length());
```

String Concatenation
In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the to this rule is the **+** operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of **+** operations. For example, the following fragment concatenates three strings:
```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

To implement **toString( )**, simply return a **String** object that contains the human-readable string that appropriately describes an object of your class.
By overriding **toString( )** for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in **print( )** and **println( )** statements and in concatenation expressions. The following program demonstrates this by overriding **toString( )** for the **Box** class:

```
// Override toString() for Box class.
class Box {
double width;
double height;
double depth;
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
public String toString() {
return "Dimensions are " + width + " by " +
depth + " by " + height + ".";
}
}
class toStringDemo {
public static void main(String args[]) {
Box b = new Box(10, 12, 14);
String s = "Box b: " + b; // concatenate Box object
System.out.println(b); // convert Box to string
System.out.println(s);
}
}
```

The output of this program is shown here:

```
Dimensions are 10.0 by 14.0 by 12.0
Box b: Dimensions are 10.0 by 14.0 by 12.0
```

As you can see, **Box**'s **toString( )** method is automatically invoked when a **Box** object is used in a concatenation expression or in a call to **println( )**.

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Each is examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

```
char charAt(int where)
```

Here, *where* is the index of the character that you want to obtain. The value of *where* must be

nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,
char ch;
ch = "abc".charAt(1);
assigns the value "**b**" to **ch**.

getChars( )
If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:
void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)
Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.
The following program demonstrates **getChars( )**:

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end - start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
}
}
```

Here is the output of this program:
demo

getBytes( )
There is an alternative to **getChars( )** that stores the characters in an array of bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:
byte[ ] getBytes( )
Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.
toCharArray( )
If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:
char[ ] toCharArray( )
This function is provided as a convenience, since it is possible to use **getChars( )** to achieve

the same result.

String Comparison

The **String** class includes several methods that compare strings or substrings within strings. Each is examined here.

equals( ) and equalsIgnoreCase( )

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case-sensitive.

To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

Here is an example that demonstrates **equals( )** and **equalsIgnoreCase( )**:

```
// Demonstrate equals() and equalsIgnoreCase().
class equalsDemo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = "Hello";
String s3 = "Good-bye";
String s4 = "HELLO";
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
s1==s2;
System.out.println(s1 + " equals " + s3 + " -> " +
s1.equals(s3));
System.out.println(s1 + " equals " + s4 + " -> " +
s1.equals(s4));
System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
s1.equalsIgnoreCase(s4));
}
}
```

The output from the program is shown here:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

equals( ) Versus ==

It is important to understand that the **equals( )** method and the == operator perform two different operations. As just explained, the **equals( )** method compares the characters inside a **String** object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can

contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " -> " +
s1.equals(s2));
System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
}
}
```

The variable **s1** refers to the **String** instance created by "**Hello**". The object referred to by **s2** is created with **s1** as an initializer. Thus, the contents of the two **String** objects are identical, but they are distinct objects. This means that **s1** and **s2** do not refer to the same objects and are, therefore, not **==**, as is shown here by the output of the preceding example:

```
Hello equals Hello -> true
Hello == Hello -> false
```

compareTo( )

Often, it is not enough to simply know whether two strings are identical. For sorting applications, you need to know which is *less than, equal to,* or *greater than* the next. A string is less than another if it comes before the other in dictionary order. A string is greater than another if it comes after the other in dictionary order. The **String** method **compareTo( )** serves this purpose. It has this general form:

```
int compareTo(String str)
```

Here, *str* is the **String** being compared with the invoking **String**. The result of the comparison is returned and is interpreted, as shown here:

```
class SortString {
static String arr[] = {
"Now", "is", "the", "time", "for", "all", "good", "men",
"to", "come", "to", "the", "aid", "of", "their", "country"
};
public static void main(String args[]) {
for(int j = 0; j < arr.length; j++) {
for(int i = j + 1; i < arr.length; i++) {
if(arr[i].compareTo(arr[j]) < 0) {
String t = arr[j];
arr[j] = arr[i];
arr[i] = t;
}
}
System.out.println(arr[j]);
}
}
```

}
The output of this program is the list of words:
Now
aid
all
come
country
for
good
is
men
of
the
the
their
time
to
to
compareTo( ) takes into account uppercase
and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

Searching Strings

The **String** class provides two methods that allow you to search a string for a specified character or substring:

• **indexOf( )** Searches for the first occurrence of a character or substring.
• **lastIndexOf( )** Searches for the last occurrence of a character or substring.

These two methods are overloaded in several different ways. In all cases, the methods return the index at which the character or substring was found, or –1 on failure.

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use one of the following **String** methods, which will construct a new copy of the string with your modifications complete.

substring( )

You can extract a substring using **substring( )**. It has two forms. The first is
String substring(int *startIndex*)
Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.
The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:
String substring(int *startIndex*, int *endIndex*)
Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.
The following program uses **substring( )** to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
public static void main(String args[]) {
String org = "This is a test. This is, too.";
String search = "is";
String sub = "was";
String result = "";
int i;
do { // replace all matching substrings
System.out.println(org);
i = org.indexOf(search);
if(i != -1) {
result = org.substring(0, i);
result = result + sub;
result = result + org.substring(i + search.length());
org = result;
}
} while(i != -1);
}
}
```

The output from this program is shown here:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

concat( )

You can concatenate two strings using **concat( )**, shown here:

String concat(String *str*)

This method creates a new object that contains the invoking string with the contents of *str* appended to the end. **concat( )** performs the same function as +. For example,

```
String s1 = "one";
String s2 = s1.concat("two");
```

puts the string “onetwo” into **s2**. It generates the same result as the following sequence:

```
String s1 = "one";
String s2 = s1 + "two";
```

replace( )

The **replace( )** method has two forms. The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char *original*, char *replacement*)

Here, *original* specifies the character to be replaced by the character specified by *replacement*. The resulting string is returned. For example,

```
String s = "Hello".replace('l', 'w');
```

puts the string “Hewwo” into **s**.

The second form of **replace( )** replaces one character sequence with another. It has this

general form:
String replace(CharSequence *original*, CharSequence *replacement*)
This form was added by J2SE 5.

trim( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:
String trim( )
Here is an example:
String s = " Hello World ".trim();
This puts the string "Hello World" into **s**.
The **trim( )** method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses **trim( )** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

Changing the Case of Characters Within a String

The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase. The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the general forms of these methods:
String toLowerCase( )
String toUpperCase( )
Both methods return a **String** object that contains the uppercase or lowercase equivalent of the invoking **String**.

StringBuffer

StringBuffer is a peer class of **String** that provides much of the functionality of strings. As you know, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writeable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth. Java uses both classes heavily, but many programmers deal only with **String** and let Java manipulate **StringBuffer**s behind the scenes by using the overloaded **+** operator.

StringBuffer Constructors

StringBuffer defines these four constructors:
StringBuffer( )
StringBuffer(int *size*)
StringBuffer(String *str*)
StringBuffer(CharSequence *chars*)
The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation. **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations

can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars.*

length( ) and capacity( )

The current length of a **StringBuffer** can be found via the **length( )** method, while the total allocated capacity can be found through the **capacity( )** method. They have the following general forms:

```
int length( )
int capacity( )
```

Here is an example:

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer = " + sb);
System.out.println("length = " + sb.length());
System.out.println("capacity = " + sb.capacity());
}
}
```

Here is the output of this program, which shows how **StringBuffer** reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

The Java I/O Classes and Interfaces

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

BufferedInputStream	FileWriter	PipedOutputStream
BufferedOutputStream	FilterInputStream	PipedReader
BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackInputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile

Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

The following interfaces are defined by **java.io**:

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

File

A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies. a list of filenames that can be examined by the **list( )** method.

The following constructors can be used to create **File** objects:
File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)
Here, *directoryPath* is the path name of the file, *filename* is the name of the file or subdirectory, *dirObj* is a **File** object that specifies a directory, and *uriObj* is a **URI** object that describes a file. The following example creates three files: **f1**, **f2**, and **f3**. The first **File** object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to **f1** and a filename; **f3** refers to the same file as **f2**.

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

NOTE *Java does the right thing with path separators between UNIX and Windows conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly. Remember, if you are using the Windows convention of a backslash character (\), you will need to use its escape sequence (\\) within a string.*

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

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

File also includes two useful utility methods. The first is **renameTo( )**, shown here:
boolean renameTo(File *newName*)
Here, the filename specified by *newName* becomes the new name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed (if you either attempt to rename a file so that it moves from one directory to another or use an existing filename, for example).
The second utility method is **delete( )**, which deletes the disk file represented by the path of the invoking **File** object. It is shown here:
boolean delete( )
You can also use **delete( )** to delete a directory if the directory is empty. **delete( )** returns **true** if it deletes the file and **false** if the file cannot be removed.

Directories
Adirectory is a **File** that contains a list of other files and directories. When you create a **File** object and it is a directory, the **isDirectory( )** method will return **true**. In this case, you can call **list( )** on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

String[ ] list( )
The list of files is returned in an array of **String** objects.
The program shown here illustrates how to use **list( )** to examine the contents of a directory:

```
// Using directories.
import java.io.File;

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

Using FilenameFilter
You will often want to limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*. To do this, you must use a second form of **list( )**, shown here:
String[ ] list(FilenameFilter *FFObj*)
In this form, *FFObj* is an object of a class that implements the **FilenameFilter** interface. **FilenameFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:
boolean accept(File *directory*, String *filename*)
The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument), and returns **false** for those files that should be excluded.
The **OnlyExt** class, shown next, implements **FilenameFilter**. It will be used to modify the preceding program so that it restricts the visibility of the filenames returned by **list( )** to files with names that end in the file extension specified when the object is constructed.

```
import java.io.*;
public class OnlyExt implements FilenameFilter {
```

```
String ext;
public OnlyExt(String ext) {
this.ext = "." + ext;
}
public boolean accept(File dir, String name) {
return name.endsWith(ext);
}
}
```

The modified directory listing program is shown here. Now it will only display files that use the **.html** extension.

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

The listFiles( ) Alternative

There is a variation to the **list( )** method, called **listFiles( )**, which you might find useful. The signatures for **listFiles( )** are shown here:

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

These methods return the file list as an array of **File** objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified **FilenameFilter**. Aside from returning an array of **File** objects, these two versions of **listFiles( )** work like their equivalent **list( )** methods.

FileFilter on the other hand deals with "File" objects i.e. filtering based on a given File attributes like; is the file hidden, is it read only; something which a file name can't give you.

The Stream Classes

InputStream and **OutputStream** are designed for byte streams. **Reader** and **Writer** are designed for character streams. The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings, and use the byte stream classes when working with bytes or other binary objects.

The Byte Streams

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. It implements the **Closeable** interface.

OutputStream

OutputStream is an abstract class that defines streaming byte output. It implements the **Closeable** and **Flushable** interfaces. Most of the methods in this class return **void** and throw an **IOException** in the case of errors.

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.

TABLE 19-1 The Methods Defined by **InputStream**

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*].

TABLE 19-2 The Methods Defined by **OutputStream**

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 **FileInputStream**s that use the same disk file and each of the two constructors:

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

```
// Demonstrate FileInputStream.
import java.io.*;
class FileInputStreamDemo {
public static void main(String args[]) throws IOException {
int size;
FileInputStream 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**. 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.

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

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

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

Next, **file2.txt**:

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

Finally, **file3.txt**:

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

java.util Part 1:
The Collections Framework

java.util contains a wide array of functionality, it is quite large. Here is a list of its classes:

AbstractCollection	EventObject	Random
AbstractList	FormattableFlags	ResourceBundle
AbstractMap	Formatter	Scanner
AbstractQueue	GregorianCalendar	ServiceLoader (Added by Java SE 6.)
AbstractSequentialList	HashMap	SimpleTimeZone
AbstractSet	HashSet	Stack
ArrayDeque (Added by Java SE 6.)	Hashtable	StringTokenizer
ArrayList	IdentityHashMap	Timer
Arrays	LinkedHashMap	TimerTask
BitSet	LinkedHashSet	TimeZone
Calendar	LinkedList	TreeMap
Collections	ListResourceBundle	TreeSet
Currency	Locale	UUID
Date	Observable	Vector
Dictionary	PriorityQueue	WeakHashMap
EnumMap	Properties	
EnumSet	PropertyPermission	
EventListenerProxy	PropertyResourceBundle	

The interfaces defined by **java.util** are shown next:

Collection	List	Queue
Comparator	ListIterator	RandomAccess
Deque (Added by Java SE 6.)	Map	Set
Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap (Added by Java SE 6.)	SortedSet
Formattable	NavigableSet (Added by Java SE 6.)	
Iterator	Observer	

The Collections Framework was designed to meet several goals. First, the framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.

The Collection Interfaces

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue. (Added by Java SE 6.)
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches. (Added by Java SE 6.)
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional.

The Collection Interface

The **Collection** interface is the foundation upon which the Collections Framework is built because it must be implemented by any class that defines a collection. **Collection** is a generic interface that has this declaration:

interface Collection<E>

Here, **E** specifies the type of objects that the collection will hold.

A**NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the collection. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

Objects are added to a collection by calling **add( )**. Notice that **add( )** takes an argument of type **E**, which means that objects added to a collection must be compatible with the type of data expected by the collection. You can add the entire contents of one collection to another by calling **addAll( ).**

You can remove an object by using **remove( )**. To remove a group of objects, call **removeAll( )**. You can remove all elements except those of a specified group by calling **retainAll( )**. To empty a collection, call **clear( )**.

The List Interface

The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements. Elements can be inserted or accessed by their position in the list, using a zero-based index. A list may contain duplicate elements. **List** is a generic interface that has this declaration:

interface List<E>

Here, **E** specifies the type of objects that the list will hold.

Method	Description
void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int *index*, Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified index.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

TABLE 17-2 The Methods Defined by **List**

To the versions of **add( )** and **addAll( )** defined by **Collection**, **List** adds the methods **add(int, E)** and **addAll(int, Collection)**. These methods insert elements at the specified index. Also, the semantics of **add(E)** and **addAll(Collection)** defined by **Collection** are changed by **List** so that they add elements to the end of the list.

The Set Interface

The **Set** interface defines a set. It extends **Collection** and declares the behavior of a collection that does not allow duplicate elements. Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own. **Set** is a generic interface that has this declaration:

interface Set<E>

Here, **E** specifies the type of objects that the set will hold.

The SortedSet Interface

The **SortedSet** interface extends **Set** and declares the behavior of a set sorted in ascending order. **SortedSet** is a generic interface that has this declaration:

interface SortedSet<E>

Here, **E** specifies the type of objects that the set will hold.

The Queue Interface

The **Queue** interface extends **Collection** and declares the behavior of a queue, which is often a first-in, first-out list. However, there are types of queues in which the ordering is based upon other criteria. **Queue** is a generic interface that has this declaration:

interface Queue<E>

Here, **E** specifies the type of objects that the queue will hold.

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

TABLE 17-5 The Methods Defined by **Queue**

The Deque Interface

The **Deque** interface was added by Java SE 6. It extends **Queue** and declares the behavior of a double-ended queue. Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks. **Deque** is a generic interface that has this declaration:

interface Deque<E>

Here, **E** specifies the type of objects that the deque will hold. In addition to the methods that it inherits from **Queue**.

Deque includes the methods **push( )** and **pop( )**. These methods enable a **Deque** to function as a stack. Also, notice the **descendingIterator( )** method. It returns an iterator that returns elements in reverse order. In other words, it returns an iterator that moves from the end of the collection to the start. A**Deque** implementation can be *capacity-restricted*, which means that only a limited number of elements can be added to the deque. When this is the case, an attempt to add an element to the deque can fail. **Deque** allows you to handle such a failure in two ways. First, methods such as **addFirst( )** and **addLast( )** throw an **IllegalStateException** if a capacity-restricted deque is full. Second, methods such as **offerFirst( )** and **offerLast( )** return false if the element can not be added.

The Collection Classes

Class	Description
AbstractCollection	Implements most of the **Collection** interface.
AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface.
AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
LinkedList	Implements a linked list by extending **AbstractSequentialList**.
ArrayList	Implements a dynamic array by extending **AbstractList**.
ArrayDeque	Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface. (Added by Java SE 6.)
AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
EnumSet	Extends **AbstractSet** for use with **enum** elements.
HashSet	Extends **AbstractSet** for use with a hash table.
LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
TreeSet	Implements a set stored in a tree. Extends **AbstractSet**.

The ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:
class ArrayList<E>
Here, **E** specifies the type of objects that the list will hold.
ArrayList supports dynamic arrays that can grow as needed.
an **ArrayList** can dynamically increase or decrease in size. Array
lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.
ArrayList has the constructors shown here:
ArrayList( )
ArrayList(Collection<? extends E> *c*)
ArrayList(int *capacity*)

```
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +
al.size());
// Add elements to the array list.
al.add("C");
al.add("A");
```

```
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +
al.size());
// Display the array list.
System.out.println("Contents of al: " + al);
// Remove elements from the array list.
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());
System.out.println("Contents of al: " + al);
}
}
```

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

The LinkedList Class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure. **LinkedList** is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold. **LinkedList** has the two constructors shown here:

```
LinkedList( )
LinkedList(Collection<? extends E> c)
```

Because **LinkedList** implements the **Deque** interface, you have access to the methods defined by **Deque**. For example, to add elements to the start of a list you can use **addFirst( )** or **offerFirst( )**. To add elements to the end of the list, use **addLast( )** or **offerLast( )**. To obtain the first element, you can use **getFirst( )** or **peekFirst( )**. To obtain the last element, use **getLast( )** or **peekLast( )**. To remove the first element, use **removeFirst( )** or **pollFirst( )**. To remove the last element, use **removeLast( )** or **pollLast( )**.

The following program illustrates **LinkedList**:

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo {
public static void main(String args[]) {
// Create a linked list.
LinkedList<String> ll = new LinkedList<String>();
```

```
// Add elements to the linked list.
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
// Remove elements from the linked list.
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "
+ ll);
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "
+ ll);
// Get and set a value.
String val = ll.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

The output from this program is shown here:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

Because **LinkedList** implements the **List** interface, calls to **add(E)** append items to the end of the list, as do calls to **addLast( )**. To insert items at a specific location, use the **add(int, E)** form of **add( )**, as illustrated by the call to **add(1, "A2")** in the example.
Notice how the third element in **ll** is changed by employing calls to **get( )** and **set( )**. To obtain the current value of an element, pass **get( )** the index at which the element is stored. To assign a new value to that index, pass **set( )** the index and its new value.

Using an Iterator

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an **iterator( )** method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time. In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
2. Set up a loop that makes a call to **hasNext( )**. Have the loop iterate as long as **hasNext( )** returns **true**.
3. Within the loop, obtain each element by calling **next( )**.

Method	Description
void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise, returns **false**.
E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
E previous( )	Returns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

TABLE 17-9 The Methods Defined by **ListIterator**

For collections that implement **List**, you can also obtain an iterator by calling **listIterator( )**. As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, **ListIterator** is used just like **Iterator**.
The following example implements these steps, demonstrating both the **Iterator** and **ListIterator** interfaces. It uses an **ArrayList** object, but the general principles apply to any type of collection. Of course, **ListIterator** is available only to those collections that implement the **List** interface.

```
// Demonstrate iterators.
import java.util.*;
class IteratorDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
// Add elements to the array list.
al.add("C");
al.add("A");
```

```
al.add("E");
al.add("B");
al.add("D");
al.add("F");
// Use iterator to display contents of al.
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Modify objects being iterated.
ListIterator<String> litr = al.listIterator();
while(litr.hasNext()) {
String element = litr.next();
litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
String element = itr.next();
System.out.print(element + " ");
}
System.out.println();
// Now, display the list backwards.
System.out.print("Modified list backwards: ");
while(litr.hasPrevious()) {
String element = litr.previous();
System.out.print(element + " ");
}
System.out.println();
}
}
```

The output is shown here:

```
Original contents of al: C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
```

Pay special attention to how the list is displayed in reverse. After the list is modified, **litr** points to the end of the list. (Remember, **litr.hasNext( )** returns **false** when the end of the list has been reached.) To traverse the list in reverse, the program continues to use **litr**, but this time it checks to see whether it has a previous element. As long as it does, that element is obtained and displayed.

Storing User-Defined Classes in Collections

collections are not limited to the storage of built-in
objects. Quite the contrary. The power of collections is that they can store any type of object, including objects of classes that you create. For example, consider the following example that uses a **LinkedList** to store mailing addresses:

```
// A simple mailing list example.
import java.util.*;
class Address {
private String name;
private String street;
private String city;
private String state;
private String code;
Address(String n, String s, String c,
String st, String cd) {
name = n;
street = s;
city = c;
state = st;
code = cd;
}
public String toString() {
return name + "\n" + street + "\n" +
city + " " + state + " " + code;
}
}

class MailList {
public static void main(String args[]) {
LinkedList<Address> ml = new LinkedList<Address>();
// Add elements to the linked list.
ml.add(new Address("J.W. West", "11 Oak Ave",
"Urbana", "IL", "61801"));
ml.add(new Address("Ralph Baker", "1142 Maple Lane",
"Mahomet", "IL", "61853"));
ml.add(new Address("Tom Carlton", "867 Elm St",
"Champaign", "IL", "61820"));
// Display the mailing list.
for(Address element : ml)
System.out.println(element + "\n");
System.out.println();
}
}
```

The output from the program is shown here:

J.W. West
11 Oak Ave
Urbana IL 61801
Ralph Baker
1142 Maple Lane
Mahomet IL 61853
Tom Carlton
867 Elm St
Champaign IL 61820

StringTokenizer

The processing of text often consists of parsing a formatted input string. *Parsing* is the division of text into a set of discrete parts, or *tokens,* which in a certain sequence can convey a semantic meaning. The **StringTokenizer** class provides the first step in this parsing process, often called the *lexer* (lexical analyzer) or *scanner.* **StringTokenizer** implements the **Enumeration** interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using **StringTokenizer**.

To use **StringTokenizer**, you specify an input string and a string that contains delimiters. *Delimiters* are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, **",;:"** sets the delimiters to a comma, semicolon, and colon.

The **StringTokenizer** constructors are shown here:
StringTokenizer(String *str*)
StringTokenizer(String *str*, String *delimiters*)
StringTokenizer(String *str*, String *delimiters*, boolean *delimAsToken*)

In all versions, *str* is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, *delimiters* is a string that specifies the delimiters. In the third version, if *delimAsToken* is **true**, then the delimiters are also returned as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

Once you have created a **StringTokenizer** object, the **nextToken( )** method is used to extract
consecutive tokens. The **hasMoreTokens( )** method returns **true** while there are more tokens to
be extracted. Since **StringTokenizer** implements **Enumeration**, the **hasMoreElements( )** and
nextElement( ) methods are also implemented, and they act the same as **hasMoreTokens( )** and
nextToken( ), respectively.

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;
class STDemo {
static String in = "title=Java: The Complete Reference;" +
"author=Schildt;" +
"publisher=Osborne/McGraw-Hill;" +
"copyright=2007";
public static void main(String args[]) {
StringTokenizer st = new StringTokenizer(in, "=;");
while(st.hasMoreTokens()) {
String key = st.nextToken();
String val = st.nextToken();
System.out.println(key + "\t" + val);
}
}
}
```

The output from this program is shown here:

```
title Java: The Complete Reference
author Schildt
publisher Osborne/McGraw-Hill
copyright 2007
```

Method	Description
int countTokens( )	Using the current set of delimiters, the method determines the number of tokens left to be parsed and returns the result.
boolean hasMoreElements( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
boolean hasMoreTokens( )	Returns **true** if one or more tokens remain in the string and returns **false** if there are none.
Object nextElement( )	Returns the next token as an **Object**.
String nextToken( )	Returns the next token as a **String**.
String nextToken(String *delimiters*)	Returns the next token as a **String** and sets the delimiters string to that specified by *delimiters*.

TABLE 18-1 The Methods Defined by **StringTokenizer**