Method	Description
ProcessBuilder directory(File *dir*)	Sets the current working directory of the invoking object. Returns a reference to the invoking object.
Map<String, String> environment( )	Returns the environmental variables associated with the invoking object as key/value pairs.
ProcessBuilder inheritIO( )	Causes the invoked process to use the same source and target for the standard I/O streams as the invoking process.
ProcessBuilder.Redirect redirectError( )	Returns the target for standard error as a **ProcessBuilder.Redirect** object.
ProcessBuilder redirectError(File *f*)	Sets the target for standard error to the specified file. Returns a reference to the invoking object.
ProcessBuilder redirectError(ProcessBuilder.Redirect *target*)	Sets the target for standard error as specified by *target*. Returns a reference to the invoking object.
boolean redirectErrorStream( )	Returns **true** if the standard error stream has been redirected to the standard output stream. Returns **false** if the streams are separate.
ProcessBuilder redirectErrorStream(boolean *merge*)	If *merge* is **true**, then the standard error stream is redirected to standard output. If *merge* is **false**, the streams are separated, which is the default state. Returns a reference to the invoking object.
ProcessBuilder.Redirect redirectInput( )	Returns the source for standard input as a **ProcessBuilder.Redirect** object.
ProcessBuilder redirectInput(File *f*)	Sets the source for standard input to the specified file. Returns a reference to the invoking object.
ProcessBuilder redirectInput(ProcessBuilder.Redirect *source*)	Sets the source for standard input as specified by *source*. Returns a reference to the invoking object.
ProcessBuilder.Redirect redirectOutput( )	Returns the target for standard output as a **ProcessBuilder.Redirect** object.
ProcessBuilder redirectOutput(File *f*)	Sets the target for standard output to the specified file. Returns a reference to the invoking object.
ProcessBuilder redirectOutput(ProcessBuilder.Redirect *target*)	Sets the target for standard output as specified by *target*. Returns a reference to the invoking object.
Process start( ) throws IOException	Begins the process specified by the invoking object. In other words, it runs the specified program.

Table 17-12 The Methods Defined by **ProcessBuilder** *(continued)*

To create a process using **ProcessBuilder**, simply create an instance of **ProcessBuilder**, specifying the name of the program and any needed arguments. To begin execution of the program, call **start( )** on that instance. Here is an example that executes the Windows text editor **notepad**. Notice that it specifies the name of the file to edit as an argument.

```
class PBDemo {
  public static void main(String args[]) {

    try {
      ProcessBuilder proc =
        new ProcessBuilder("notepad.exe", "testfile");
      proc.start();
    } catch (Exception e) {
      System.out.println("Error executing notepad.");
    }
  }
}
```


System

The **System** class holds a collection of static methods and variables. The standard input, output, and error output of the Java run time are stored in the **in**, **out**, and **err** variables. The methods defined by **System** are shown in Table 17-13. Many of the methods throw a **SecurityException** if the operation is not permitted by the security manager.

Let's look at some common uses of **System**.

Method	Description
static void arraycopy(Object *source*, int *sourceStart*, Object *target*, int *targetStart*, int *size*)	Copies an array. The array to be copied is passed in *source*, and the index at which point the copy will begin within *source* is passed in *sourceStart*. The array that will receive the copy is passed in *target*, and the index at which point the copy will begin within *target* is passed in *targetStart*. *size* is the number of elements that are copied.
static String clearProperty(String *which*)	Deletes the environmental variable specified by *which*. The previous value associated with *which* is returned.
static Console console( )	Returns the console associated with the JVM. **null** is returned if the JVM currently has no console.
static long currentTimeMillis( )	Returns the current time in terms of milliseconds since midnight, January 1, 1970.
static void exit(int *exitCode*)	Halts execution and returns the value of *exitCode* to the parent process (usually the operating system). By convention, 0 indicates normal termination. All other values indicate some form of error.
static void gc( )	Initiates garbage collection.

Table 17-13 The Methods Defined by **System**

Method	Description
static Map<String, String> getenv( )	Returns a **Map** that contains the current environmental variables and their values.
static String getenv(String *which*)	Returns the value associated with the environmental variable passed in *which*.
static Properties getProperties( )	Returns the properties associated with the Java run-time system. (The **Properties** class is described in Chapter 18.)
static String getProperty(String *which*)	Returns the property associated with *which*. A **null** object is returned if the desired property is not found.
static String getProperty(String *which*, String *default*)	Returns the property associated with *which*. If the desired property is not found, *default* is returned.
static SecurityManager getSecurityManager( )	Returns the current security manager or a **null** object if no security manager is installed.
static int identityHashCode(Object *obj*)	Returns the identity hash code for *obj*.
static Channel inheritedChannel( ) throws IOException	Returns the channel inherited by the Java Virtual Machine. Returns **null** if no channel is inherited.
static String lineSeparator( )	Returns a string that contains the line-separator characters.
static void load(String *libraryFileName*)	Loads the dynamic library whose file is specified by *libraryFileName*, which must specify its complete path.
static void loadLibrary(String *libraryName*)	Loads the dynamic library whose name is associated with *libraryName*.
static String mapLibraryName(String *lib*)	Returns a platform-specific name for the library named *lib*.
static long nanoTime( )	Obtains the most precise timer in the system and returns its value in terms of nanoseconds since some arbitrary starting point. The accuracy of the timer is unknowable.
static void runFinalization( )	Initiates calls to the **finalize( )** methods of unused but not yet recycled objects.
static void setErr(PrintStream *eStream*)	Sets the standard **err** stream to *eStream*.
static void setIn(InputStream *iStream*)	Sets the standard **in** stream to *iStream*.
static void setOut(PrintStream *oStream*)	Sets the standard **out** stream to *oStream*.
static void setProperties(Properties *sysProperties*)	Sets the current system properties as specified by *sysProperties*.
static String setProperty(String *which*, String *v*)	Assigns the value *v* to the property named *which*.
static void setSecurityManager(SecurityManager *secMan*)	Sets the security manager to that specified by *secMan*.

Table 17-13 The Methods Defined by **System** *(continued)*

Using currentTimeMillis( ) to Time Program Execution

One use of the **System** class that you might find particularly interesting is to use the **currentTimeMillis( )** method to time how long various parts of your program take to execute. The **currentTimeMillis( )** method returns the current time in terms of milliseconds since midnight, January 1, 1970. To time a section of your program, store this value just before beginning the section in question. Immediately upon completion, call **currentTimeMillis( )** again. The elapsed time will be the ending time minus the starting time. The following program demonstrates this:

```
// Timing program execution.

class Elapsed {
  public static void main(String args[]) {
    long start, end;

    System.out.println("Timing a for loop from 0 to 100,000,000");

    // time a for loop from 0 to 100,000,000

    start = System.currentTimeMillis(); // get starting time
    for(long i=0; i < 100000000L; i++) ;
    end = System.currentTimeMillis(); // get ending time

    System.out.println("Elapsed time: " + (end-start));
  }
}
```

Here is a sample run (remember that your results probably will differ):

```
Timing a for loop from 0 to 100,000,000
Elapsed time: 10
```

If your system has a timer that offers nanosecond precision, then you could rewrite the preceding program to use **nanoTime( )** rather than **currentTimeMillis( )**. For example, here is the key portion of the program rewritten to use **nanoTime( )**:

```
start = System.nanoTime(); // get starting time
for(long i=0; i < 100000000L; i++) ;
end = System.nanoTime(); // get ending time
```

Using arraycopy( )

The **arraycopy( )** method can be used to copy quickly an array of any type from one place to another. This is much faster than the equivalent loop written out longhand in Java. Here is an example of two arrays being copied by the **arraycopy( )** method. First, **a** is copied to **b**. Next, all of **a**'s elements are shifted *down* by one. Then, **b** is shifted *up* by one.

```
// Using arraycopy().

class ACDemo {
  static byte a[] = { 65, 66, 67, 68, 69, 70, 71, 72, 73, 74 };
  static byte b[] = { 77, 77, 77, 77, 77, 77, 77, 77, 77, 77 };
```

```
  public static void main(String args[]) {
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, b, 0, a.length);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
    System.arraycopy(a, 0, a, 1, a.length - 1);
    System.arraycopy(b, 1, b, 0, b.length - 1);
    System.out.println("a = " + new String(a));
    System.out.println("b = " + new String(b));
  }
}
```

As you can see from the following output, you can copy using the same source and destination in either direction:

```
a = ABCDEFGHIJ
b = MMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Environment Properties

The following properties are available in all cases:

file.separator	java.specification.version	java.vm.version
java.class.path	java.vendor	line.separator
java.class.version	java.vendor.url	os.arch
java.compiler	java.version	os.name
java.ext.dirs	java.vm.name	os.version
java.home	java.vm.specification.name	path.separator
java.io.tmpdir	java.vm.specification.vendor	user.dir
java.library.path	java.vm.specification.version	user.home
java.specification.name	java.vm.vendor	user.name
java.specification.vendor		

You can obtain the values of various environment variables by calling the **System.getProperty( )** method. For example, the following program displays the path to the current user directory:

```
class ShowUserDir {
  public static void main(String args[]) {
    System.out.println(System.getProperty("user.dir"));
  }
}
```

Object

As mentioned in Part I, **Object** is a superclass of all other classes. **Object** defines the methods shown in Table 17-14, which are available to every object.

Using clone( ) and the Cloneable Interface

Most of the methods defined by **Object** are discussed elsewhere in this book. However, one deserves special attention: **clone( )**. The **clone( )** method generates a duplicate copy of the object on which it is called. Only classes that implement the **Cloneable** interface can be cloned.

The **Cloneable** interface defines no members. It is used to indicate that a class allows a bitwise copy of an object (that is, a *clone*) to be made. If you try to call **clone( )** on a class that does not implement **Cloneable**, a **CloneNotSupportedException** is thrown. When a clone is made, the constructor for the object being cloned is *not* called. As implemented by **Object**, a clone is simply an exact copy of the original.

Cloning is a potentially dangerous action, because it can cause unintended side effects. For example, if the object being cloned contains a reference variable called *obRef*, then when the clone is made, *obRef* in the clone will refer to the same object as does *obRef* in the

Method	Description
Object clone( ) throws CloneNotSupportedException	Creates a new object that is the same as the invoking object.
boolean equals(Object *object*)	Returns **true** if the invoking object is equivalent to *object*.
void finalize( ) throws Throwable	Default **finalize( )** method. It is called before an unused object is recycled.
final Class<?> getClass( )	Obtains a **Class** object that describes the invoking object.
int hashCode( )	Returns the hash code associated with the invoking object.
final void notify( )	Resumes execution of a thread waiting on the invoking object.
final void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
final void wait( ) throws InterruptedException	Waits on another thread of execution.
final void wait(long *milliseconds*) throws InterruptedException	Waits up to the specified number of *milliseconds* on another thread of execution.
final void wait(long *milliseconds*, int *nanoseconds*) throws InterruptedException	Waits up to the specified number of *milliseconds* plus *nanoseconds* on another thread of execution.

Table 17-14 The Methods Defined by **Object**

original. If the clone makes a change to the contents of the object referred to by *obRef*, then it will be changed for the original object, too. Here is another example: If an object opens an I/O stream and is then cloned, two objects will be capable of operating on the same stream. Further, if one of these objects closes the stream, the other object might still attempt to write to it, causing an error. In some cases, you will need to override the **clone( )** method defined by **Object** to handle these types of problems.

Because cloning can cause problems, **clone( )** is declared as **protected** inside **Object**. This means that it must either be called from within a method defined by the class that implements **Cloneable**, or it must be explicitly overridden by that class so that it is public. Let's look at an example of each approach.

The following program implements **Cloneable** and defines the method **cloneTest( )**, which calls **clone( )** in **Object**:

```
// Demonstrate the clone() method

class TestClone implements Cloneable {
  int a;
  double b;

  // This method calls Object's clone().
  TestClone cloneTest() {
    try {
      // call clone in Object.
      return (TestClone) super.clone();
    } catch(CloneNotSupportedException e) {
      System.out.println("Cloning not allowed.");
      return this;
    }
  }
}

class CloneDemo {
  public static void main(String args[]) {
    TestClone x1 = new TestClone();
    TestClone x2;

    x1.a = 10;
    x1.b = 20.98;

    x2 = x1.cloneTest(); // clone x1

    System.out.println("x1: " + x1.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

Here, the method **cloneTest( )** calls **clone( )** in **Object** and returns the result. Notice that the object returned by **clone( )** must be cast into its appropriate type (**TestClone**).

The following example overrides **clone( )** so that it can be called from code outside of its class. To do this, its access specifier must be **public**, as shown here:

```
// Override the clone() method.

class TestClone implements Cloneable {
  int a;
  double b;

  // clone() is now overridden and is public.
  public Object clone() {
    try {
      // call clone in Object.
      return super.clone();
    } catch(CloneNotSupportedException e) {
      System.out.println("Cloning not allowed.");
      return this;
    }
  }
}

class CloneDemo2 {
  public static void main(String args[]) {
    TestClone x1 = new TestClone();
    TestClone x2;

    x1.a = 10;
    x1.b = 20.98;

    // here, clone() is called directly.
    x2 = (TestClone) x1.clone();

    System.out.println("x1: " + x1.a + " " + x1.b);
    System.out.println("x2: " + x2.a + " " + x2.b);
  }
}
```

The side effects caused by cloning are sometimes difficult to see at first. It is easy to think that a class is safe for cloning when it actually is not. In general, you should not implement **Cloneable** for any class without good reason.

Class

Class encapsulates the run-time state of a class or interface. Objects of type **Class** are created automatically, when classes are loaded. You cannot explicitly declare a **Class** object. Generally, you obtain a **Class** object by calling the **getClass( )** method defined by **Object**. **Class** is a generic type that is declared as shown here:

class Class<T>

Here, **T** is the type of the class or interface represented. A sampling of methods defined by **Class** is shown in Table 17-15.

Method	Description
static Class<?> forName(String *name*) throws ClassNotFoundException	Returns a **Class** object given its complete name.
static Class<?> forName(String *name*, boolean *how*, ClassLoader *ldr*) throws ClassNotFoundException	Returns a **Class** object given its complete name. The object is loaded using the loader specified by *ldr*. If *how* is **true**, the object is initialized; otherwise, it is not.
<A extends Annotation> A getAnnotation(Class<A> *annoType*)	Returns an **Annotation** object that contains the annotation associated with *annoType* for the invoking object.
Annotation[ ] getAnnotations( )	Obtains all annotations associated with the invoking object and stores them in an array of **Annotation** objects. Returns a reference to this array.
<A extends Annotation> A[ ] getAnnotationsByType(Class<A> *annoType*)	Returns an array of the annotations (including repeated annotations) of *annoType* associated with the invoking object. (Added by JDK 8.)
Class<?>[ ] getClasses( )	Returns a **Class** object for each public class and interface that is a member of the class represented by the invoking object.
ClassLoader getClassLoader( )	Returns the **ClassLoader** object that loaded the class or interface.
Constructor<T> getConstructor(Class<?> ... *paramTypes*) throws NoSuchMethodException, SecurityException	Returns a **Constructor** object that represents the constructor for the class represented by the invoking object that has the parameter types specified by *paramTypes*.
Constructor<?>[ ] getConstructors( ) throws SecurityException	Obtains a **Constructor** object for each public constructor of the class represented by the invoking object and stores them in an array. Returns a reference to this array.
Annotation[ ] getDeclaredAnnotations( )	Obtains an **Annotation** object for all the annotations that are declared by the invoking object and stores them in an array. Returns a reference to this array. (Inherited annotations are ignored.)
<A extends Annotation> A[ ] getDeclaredAnnotationsByType(Class<A> *annoType*)	Returns an array of the non-inherited annotations (including repeated annotations) of *annoType* associated with the invoking object. (Added by JDK 8.)
Constructor<?>[ ] getDeclaredConstructors( ) throws SecurityException	Obtains a **Constructor** object for each constructor declared by the class represented by the invoking object and stores them in an array. Returns a reference to this array. (Superclass constructors are ignored.)
Field[ ] getDeclaredFields( ) throws SecurityException	Obtains a **Field** object for each field declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited fields are ignored.)

Table 17-15 A Sampling of Methods Defined by **Class**

Method	Description
Method[ ] getDeclaredMethods( ) throws SecurityException	Obtains a **Method** object for each method declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited methods are ignored.)
Field getField(String *fieldName*) throws NoSuchMethodException, SecurityException	Returns a **Field** object that represents the public field specified by *fieldName* for the class or interface represented by the invoking object.
Field[ ] getFields( ) throws SecurityException	Obtains a **Field** object for each public field of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.
Class<?>[ ] getInterfaces( )	When invoked on an object that represents a class, this method returns an array of the interfaces implemented by that class. When invoked on an object that represents an interface, this method returns an array of interfaces extended by that interface.
Method getMethod(String *methName*, Class<?> ... *paramTypes*) throws NoSuchMethodException, SecurityException	Returns a **Method** object that represents the public method specified by *methName* and having the parameter types specified by *paramTypes* in the class or interface represented by the invoking object.
Method[ ] getMethods( ) throws SecurityException	Obtains a **Method** object for each public method of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array.
String getName( )	Returns the complete name of the class or interface of the type represented by the invoking object.
ProtectionDomain getProtectionDomain( )	Returns the protection domain associated with the invoking object.
Class<? super T> getSuperclass( )	Returns the superclass of the type represented by the invoking object. The return value is **null** if the represented type is **Object** or not a class.
boolean isInterface( )	Returns **true** if the type represented by the invoking object is an interface. Otherwise, it returns **false**.
T newInstance( ) throws IllegalAccessException, InstantiationException	Creates a new instance (i.e., a new object) that is of the same type as that represented by invoking object. This is equivalent to using **new** with the class' default constructor. The new object is returned. This method will fail if the represented type is abstract, not a class, or does not have a default constructor.
String toString( )	Returns the string representation of the type represented by the invoking object or interface.

Table 17-15 A Sampling of Methods Defined by **Class** *(continued)*

The methods defined by **Class** are often useful in situations where run-time type information about an object is required. As Table 17-15 shows, methods are provided that allow you to determine additional information about a particular class, such as its public constructors, fields, and methods. Among other things, this is important for the Java Beans functionality, which is discussed later in this book.

The following program demonstrates **getClass( )** (inherited from **Object**) and **getSuperclass( )** (from **Class**):

```
// Demonstrate Run-Time Type Information.

class X {
  int a;
  float b;
}

class Y extends X {
  double c;
}

class RTTI {
  public static void main(String args[]) {
    X x = new X();
    Y y = new Y();
    Class<?> clObj;

    clObj = x.getClass(); // get Class reference
    System.out.println("x is object of type: " +
                       clObj.getName());

    clObj = y.getClass(); // get Class reference
    System.out.println("y is object of type: " +
                       clObj.getName());
    clObj = clObj.getSuperclass();
    System.out.println("y's superclass is " +
                       clObj.getName());
  }
}
```

The output from this program is shown here:

```
x is object of type: X
y is object of type: Y
y's superclass is X
```

ClassLoader

The abstract class **ClassLoader** defines how classes are loaded. Your application can create subclasses that extend **ClassLoader**, implementing its methods. Doing so allows you to load classes in some way other than the way they are normally loaded by the Java run-time system. However, this is not something that you will normally need to do.

Math

The **Math** class contains all the floating-point functions that are used for geometry and trigonometry, as well as several general-purpose methods. **Math** defines two **double** constants: **E** (approximately 2.72) and **PI** (approximately 3.14).

Trigonometric Functions

The following methods accept a **double** parameter for an angle in radians and return the result of their respective trigonometric function:

Method	Description
static double sin(double *arg*)	Returns the sine of the angle specified by *arg* in radians.
static double cos(double *arg*)	Returns the cosine of the angle specified by *arg* in radians.
static double tan(double *arg*)	Returns the tangent of the angle specified by *arg* in radians.

The next methods take as a parameter the result of a trigonometric function and return, in radians, the angle that would produce that result. They are the inverse of their non-arc companions.

Method	Description
static double asin(double *arg*)	Returns the angle whose sine is specified by *arg*.
static double acos(double *arg*)	Returns the angle whose cosine is specified by *arg*.
static double atan(double *arg*)	Returns the angle whose tangent is specified by *arg*.
static double atan2(double *x*, double *y*)	Returns the angle whose tangent is *x*/*y*.

The next methods compute the hyperbolic sine, cosine, and tangent of an angle:

Method	Description
static double sinh(double *arg*)	Returns the hyperbolic sine of the angle specified by *arg*.
static double cosh(double *arg*)	Returns the hyperbolic cosine of the angle specified by *arg*.
static double tanh(double *arg*)	Returns the hyperbolic tangent of the angle specified by *arg*.

Exponential Functions

Math defines the following exponential methods:

Method	Description
static double cbrt(double *arg*)	Returns the cube root of *arg*.
static double exp(double *arg*)	Returns e to the *arg*.
static double expm1(double *arg*)	Returns e to the *arg*–1.
static double log(double *arg*)	Returns the natural logarithm of *arg*.
static double log10(double *arg*)	Returns the base 10 logarithm for *arg*.
static double log1p(double *arg*)	Returns the natural logarithm for *arg* + 1.
static double pow(double *y*, double *x*)	Returns *y* raised to the *x*; for example, pow(2.0, 3.0) returns 8.0.
static double scalb(double *arg*, int *factor*)	Returns $arg \times 2^{factor}$.
static float scalb(float *arg*, int *factor*)	Returns $arg \times 2^{factor}$.
static double sqrt(double *arg*)	Returns the square root of *arg*.

Rounding Functions

The **Math** class defines several methods that provide various types of rounding operations. They are shown in Table 17-16. Notice the two **ulp( )** methods at the end of the table. In this context, *ulp* stands for *units in the last place*. It indicates the distance between a value and the next higher value. It can be used to help assess the accuracy of a result.

Method	Description
static int abs(int *arg*)	Returns the absolute value of *arg*.
static long abs(long *arg*)	Returns the absolute value of *arg*.
static float abs(float *arg*)	Returns the absolute value of *arg*.
static double abs(double *arg*)	Returns the absolute value of *arg*.
static double ceil(double *arg*)	Returns the smallest whole number greater than or equal to *arg*.
static double floor(double *arg*)	Returns the largest whole number less than or equal to *arg*.
static int floorDiv(int *dividend*, int *divisor*)	Returns the floor of the result of *dividend*/*divisor*. (Added by JDK 8.)
static long floorDiv(long *dividend*, long *divisor*)	Returns the floor of the result of *dividend*/ *divisor*. (Added by JDK 8.)
static int floorMod(int *dividend*, int *divisor*)	Returns the floor of the remainder of *dividend*/ *divisor*. (Added by JDK 8.)
static long floorMod(long *dividend*, long *divisor*)	Returns the floor of the remainder of *dividend*/ *divisor*. (Added by JDK 8.)

Table 17-16 The Rounding Methods Defined by **Math**

Method	Description
static int max(int *x*, int *y*)	Returns the maximum of *x* and *y*.
static long max(long *x*, long *y*)	Returns the maximum of *x* and *y*.
static float max(float *x*, float *y*)	Returns the maximum of *x* and *y*.
static double max(double *x*, double *y*)	Returns the maximum of *x* and *y*.
static int min(int *x*, int *y*)	Returns the minimum of *x* and *y*.
static long min(long *x*, long *y*)	Returns the minimum of *x* and *y*.
static float min(float *x*, float *y*)	Returns the minimum of *x* and *y*.
static double min(double *x*, double *y*)	Returns the minimum of *x* and *y*.
static double nextAfter(double *arg*, double *toward*)	Beginning with the value of *arg*, returns the next value in the direction of *toward*. If *arg* == *toward*, then *toward* is returned.
static float nextAfter(float *arg*, double *toward*)	Beginning with the value of *arg*, returns the next value in the direction of *toward*. If *arg* == *toward*, then *toward* is returned.
static double nextDown(double *val*)	Returns the next value lower than *val*. (Added by JDK 8.)
static float nextDown(float *val*)	Returns the next value lower than *val*. (Added by JDK 8.)
static double nextUp(double *arg*)	Returns the next value in the positive direction from *arg*.
static float nextUp(float *arg*)	Returns the next value in the positive direction from *arg*.
static double rint(double *arg*)	Returns the integer nearest in value to *arg*.
static int round(float *arg*)	Returns *arg* rounded up to the nearest **int**.
static long round(double *arg*)	Returns *arg* rounded up to the nearest **long**.
static float ulp(float *arg*)	Returns the ulp for *arg*.
static double ulp(double *arg*)	Returns the ulp for *arg*.

Table 17-16 The Rounding Methods Defined by **Math** *(continued)*

Miscellaneous Math Methods

In addition to the methods just shown, **Math** defines several other methods, which are shown in Table 17-17. Notice that several of the methods use the suffix **Exact**. These were added by JDK 8. They throw an **ArithmeticException** if overflow occurs. Thus, these methods give you an easy way to watch various operations for overflow.

Method	Description
static int addExact(int *arg1*, int *arg2*)	Returns *arg1* + *arg2*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static long addExact(long *arg1*, long *arg2*	Returns *arg1* + *arg2*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static double copySign(double *arg*, double *signarg*)	Returns *arg* with same sign as that of *signarg*.
static float copySign(float *arg*, float *signarg*)	Returns *arg* with same sign as that of *signarg*.
static int decrementExact(int *arg*)	Returns *arg* – 1. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static long decrementExact(long *arg*)	Returns *arg* – 1. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static int getExponent(double *arg*)	Returns the base-2 exponent used by the binary representation of *arg*.
static int getExponent(float *arg*)	Returns the base-2 exponent used by the binary representation of *arg*.
static hypot(double *side1*, double *side2*)	Returns the length of the hypotenuse of a right triangle given the length of the two opposing sides.
static double IEEEremainder(double *dividend*, double *divisor*)	Returns the remainder of *dividend* / *divisor*.
static int incrementExact(int *arg*)	Returns *arg* + 1. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static long incrementExact(long *arg*)	Returns *arg* + 1. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static int multiplyExact(int *arg1*, int *arg2*)	Returns *arg1* * *arg2*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static long multiplyExact(long *arg1*, long *arg2*)	Returns *arg1* * *arg2*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static int negateExact(int *arg*)	Returns –*arg*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static long negateExact(long *arg*)	Returns –*arg*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static double random( )	Returns a pseudorandom number between 0 and 1.
static float signum(double *arg*)	Determines the sign of a value. It returns 0 if *arg* is 0, 1 if *arg* is greater than 0, and –1 if *arg* is less than 0.
static float signum(float *arg*)	Determines the sign of a value. It returns 0 if *arg* is 0, 1 if *arg* is greater than 0, and –1 if *arg* is less than 0.
static int subtractExact(int *arg1*, int *arg2*)	Returns *arg1* – *arg2*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static long subtractExact(long *arg1*, long *arg2*)	Returns *arg1* – *arg2*. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static double toDegrees(double *angle*)	Converts radians to degrees. The angle passed to *angle* must be specified in radians. The result in degrees is returned.
static int toIntExact(long *arg*)	Returns *arg* as an int. Throws an **ArithmeticException** if overflow occurs. (Added by JDK 8.)
static double toRadians(double *angle*)	Converts degrees to radians. The *angle* passed to angle must be specified in degrees. The result in radians is returned.

Table 17-17 Other Methods Defined by **Math**

The following program demonstrates **toRadians( )** and **toDegrees( )**:

```
// Demonstrate toDegrees() and toRadians().
class Angles {
  public static void main(String args[]) {
    double theta = 120.0;

    System.out.println(theta + " degrees is " +
                       Math.toRadians(theta) + " radians.");

    theta = 1.312;
    System.out.println(theta + " radians is " +
                       Math.toDegrees(theta) + " degrees.");
  }
}
```

The output is shown here:

```
120.0 degrees is 2.0943951023931953 radians.
1.312 radians is 75.17206272116401 degrees.
```


StrictMath

The **StrictMath** class defines a complete set of mathematical methods that parallel those in **Math**. The difference is that the **StrictMath** version is guaranteed to generate precisely identical results across all Java implementations, whereas the methods in **Math** are given more latitude in order to improve performance.

Compiler

The **Compiler** class supports the creation of Java environments in which Java bytecode is compiled into executable code rather than interpreted. It is not for normal programming use.

Thread, ThreadGroup, and Runnable

The **Runnable** interface and the **Thread** and **ThreadGroup** classes support multithreaded programming. Each is examined next.

NOTE An overview of the techniques used to manage threads, implement the **Runnable** interface, and create multithreaded programs is presented in Chapter 11.

The Runnable Interface

The **Runnable** interface must be implemented by any class that will initiate a separate thread of execution. **Runnable** only defines one abstract method, called **run( )**, which is the entry point to the thread. It is defined like this:

```
void run( )
```

Threads that you create must implement this method.

Thread

Thread creates a new thread of execution. It implements **Runnable** and defines the following commonly used constructors:

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

threadOb is an instance of a class that implements the **Runnable** interface and defines where execution of the thread will begin. The name of the thread is specified by *threadName*. When a name is not specified, one is created by the Java Virtual Machine. *groupOb* specifies the thread group to which the new thread will belong. When no thread group is specified, the new thread belongs to the same group as the parent thread.

The following constants are defined by **Thread**:

MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY

As expected, these constants specify the maximum, minimum, and default thread priorities.

The methods defined by **Thread** are shown in Table 17-18. In early versions of Java, **Thread** also included the methods **stop( )**, **suspend( )**, and **resume( )**. However, as explained in Chapter 11, these were deprecated because they were inherently unstable. Also deprecated are **countStackFrames( )**, because it calls **suspend( )**, and **destroy( )**, because it can cause deadlock.

Method	Description
static int activeCount( )	Returns the approximate number of active threads in the group to which the thread belongs.
final void checkAccess( )	Causes the security manager to verify that the current thread can access and/or change the thread on which **checkAccess( )** is called.
static Thread currentThread( )	Returns a **Thread** object that encapsulates the thread that calls this method.
static void dumpStack( )	Displays the call stack for the thread.
static int enumerate(Thread *threads*[ ])	Puts copies of all **Thread** objects in the current thread's group into *threads*. The number of threads is returned.
static Map<Thread, StackTraceElement[ ]> getAllStackTraces( )	Returns a **Map** that contains the stack traces for all active threads. In the map, each entry consists of a key, which is the **Thread** object, and its value, which is an array of **StackTraceElement**.

Table 17-18 The Methods Defined by **Thread**

Method	Description
ClassLoader getContextClassLoader( )	Returns the context class loader that is used to load classes and resources for this thread.
static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler( )	Returns the default uncaught exception handler.
long getID( )	Returns the ID of the invoking thread.
final String getName( )	Returns the thread's name.
final int getPriority( )	Returns the thread's priority setting.
StackTraceElement[ ] getStackTrace( )	Returns an array containing the stack trace for the invoking thread.
Thread.State getState( )	Returns the invoking thread's state.
final ThreadGroup getThreadGroup( )	Returns the **ThreadGroup** object of which the invoking thread is a member.
Thread.UncaughtExceptionHandler getUncaughtExceptionHandler( )	Returns the invoking thread's uncaught exception handler.
static boolean holdsLock(Object *ob*)	Returns **true** if the invoking thread owns the lock on *ob*. Returns **false** otherwise.
void interrupt( )	Interrupts the thread.
static boolean interrupted( )	Returns **true** if the currently executing thread has been interrupted. Otherwise, it returns **false**.
final boolean isAlive( )	Returns **true** if the thread is still active. Otherwise, it returns **false**.
final boolean isDaemon( )	Returns **true** if the thread is a daemon thread. Otherwise, it returns **false**.
boolean isInterrupted( )	Returns **true** if the invoking thread has been interrupted. Otherwise, it returns **false**.
final void join( ) throws InterruptedException	Waits until the thread terminates.
final void join(long *milliseconds*) throws InterruptedException	Waits up to the specified number of milliseconds for the thread on which it is called to terminate.
final void join(long *milliseconds*, int *nanoseconds*) throws InterruptedException	Waits up to the specified number of milliseconds plus nanoseconds for the thread on which it is called to terminate.
void run( )	Begins execution of a thread.
void setContextClassLoader(ClassLoader *cl*)	Sets the context class loader that will be used by the invoking thread to *cl*.
final void setDaemon(boolean *state*)	Flags the thread as a daemon thread.
static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler *e*)	Sets the default uncaught exception handler to *e*.

Table 17-18 The Methods Defined by **Thread** *(continued)*

Method	Description
final void setName(String *threadName*)	Sets the name of the thread to that specified by *threadName*.
final void setPriority(int *priority*)	Sets the priority of the thread to that specified by *priority*.
void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler *e*)	Sets the invoking thread's default uncaught exception handler to *e*.
static void sleep(long *milliseconds*) throws InterruptedException	Suspends execution of the thread for the specified number of milliseconds.
static void sleep(long *milliseconds*, int *nanoseconds*) throws InterruptedException	Suspends execution of the thread for the specified number of milliseconds plus nanoseconds.
void start( )	Starts execution of the thread.
String toString( )	Returns the string equivalent of a thread.
static void yield( )	The calling thread offers to yield the CPU to another thread.

Table 17-18 The Methods Defined by **Thread** *(continued)*

ThreadGroup

ThreadGroup creates a group of threads. It defines these two constructors:

ThreadGroup(String *groupName*)
ThreadGroup(ThreadGroup *parentOb*, String *groupName*)

For both forms, *groupName* specifies the name of the thread group. The first version creates a new group that has the current thread as its parent. In the second form, the parent is specified by *parentOb*. The non-deprecated methods defined by **ThreadGroup** are shown in Table 17-19.

Method	Description
int activeCount( )	Returns the approximate number of active threads in the invoking group (including those in subgroups).
int activeGroupCount( )	Returns the approximate number of active groups (including subgroups) for which the invoking thread is a parent.
final void checkAccess( )	Causes the security manager to verify that the invoking thread may access and/or change the group on which **checkAccess( )** is called.
final void destroy( )	Destroys the thread group (and any child groups) on which it is called.

Table 17-19 The Methods Defined by **ThreadGroup**

Method	Description
int enumerate(Thread *group*[ ])	Puts the active threads that comprise the invoking thread group (including those in subgroups) into the *group* array.
int enumerate(Thread *group*[ ], boolean *all*)	Puts the active threads that comprise the invoking thread group into the *group* array. If *all* is **true**, then threads in all subgroups of the thread are also put into *group*.
int enumerate(ThreadGroup *group*[ ])	Puts the active subgroups (including subgroups of subgroups and so on) of the invoking thread group into the *group* array.
int enumerate(ThreadGroup *group*[ ], boolean *all*)	Puts the active subgroups of the invoking thread group into the *group* array. If *all* is **true**, then all active subgroups of the subgroups (and so on) are also put into *group*.
final int getMaxPriority( )	Returns the maximum priority setting for the group.
final String getName( )	Returns the name of the group.
final ThreadGroup getParent( )	Returns **null** if the invoking **ThreadGroup** object has no parent. Otherwise, it returns the parent of the invoking object.
final void interrupt( )	Invokes the **interrupt( )** method of all threads in the group and any subgroups.
final boolean isDaemon( )	Returns **true** if the group is a daemon group. Otherwise, it returns **false**.
boolean isDestroyed( )	Returns **true** if the group has been destroyed. Otherwise, it returns **false**.
void list( )	Displays information about the group.
final boolean parentOf(ThreadGroup *group*)	Returns **true** if the invoking thread is the parent of *group* (or *group*, itself). Otherwise, it returns **false**.
final void setDaemon(boolean *isDaemon*)	If *isDaemon* is **true**, then the invoking group is flagged as a daemon group.
final void setMaxPriority(int *priority*)	Sets the maximum priority of the invoking group to *priority*.
String toString( )	Returns the string equivalent of the group.
void uncaughtException(Thread *thread*, Throwable *e*)	This method is called when an exception goes uncaught.

Table 17-19 The Methods Defined by **ThreadGroup** *(continued)*

Thread groups offer a convenient way to manage groups of threads as a unit. This is particularly valuable in situations in which you want to suspend and resume a number of related threads. For example, imagine a program in which one set of threads is used for printing a document, another set is used to display the document on the screen, and another set saves the document to a disk file. If printing is aborted, you will want an easy way to stop all threads related to printing. Thread groups offer this convenience. The following program, which creates two thread groups of two threads each, illustrates this usage:

```
// Demonstrate thread groups.
class NewThread extends Thread {
  boolean suspendFlag;

  NewThread(String threadname, ThreadGroup tgOb) {
    super(tgOb, threadname);
    System.out.println("New thread: " + this);
    suspendFlag = false;
    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(getName() + ": " + i);
        Thread.sleep(1000);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (Exception e) {
      System.out.println("Exception in " + getName());
    }
      System.out.println(getName() + " exiting.");
    }

    synchronized void mysuspend() {
      suspendFlag = true;
    }

    synchronized void myresume() {
      suspendFlag = false;
      notify();
    }
  }

class ThreadGroupDemo {
  public static void main(String args[]) {
    ThreadGroup groupA = new ThreadGroup("Group A");
    ThreadGroup groupB = new ThreadGroup("Group B");
```

```
    NewThread ob1 = new NewThread("One", groupA);
    NewThread ob2 = new NewThread("Two", groupA);
    NewThread ob3 = new NewThread("Three", groupB);
    NewThread ob4 = new NewThread("Four", groupB);

    System.out.println("\nHere is output from list():");
    groupA.list();
    groupB.list();
    System.out.println();

    System.out.println("Suspending Group A");
    Thread tga[] = new Thread[groupA.activeCount()];
    groupA.enumerate(tga); // get threads in group
    for(int i = 0; i < tga.length; i++) {
      ((NewThread)tga[i]).mysuspend(); // suspend each thread
    }

    try {
      Thread.sleep(4000);
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted.");
    }

    System.out.println("Resuming Group A");
    for(int i = 0; i < tga.length; i++) {
      ((NewThread)tga[i]).myresume(); // resume threads in group
    }

    // wait for threads to finish
    try {
      System.out.println("Waiting for threads to finish.");
      ob1.join();
      ob2.join();
      ob3.join();
      ob4.join();
    } catch (Exception e) {
      System.out.println("Exception in Main thread");
    }

    System.out.println("Main thread exiting.");
  }
}
```

Sample output from this program is shown here (the precise output you see may differ):

```
New thread: Thread[One,5,Group A]
New thread: Thread[Two,5,Group A]
New thread: Thread[Three,5,Group B]
New thread: Thread[Four,5,Group B]
Here is output from list():
java.lang.ThreadGroup[name=Group A,maxpri=10]
  Thread[One,5,Group A]
  Thread[Two,5,Group A]
```

```
java.lang.ThreadGroup[name=Group B,maxpri=10]
  Thread[Three,5,Group B]
  Thread[Four,5,Group B]
Suspending Group A
Three: 5
Four: 5
Three: 4
Four: 4
Three: 3
Four: 3
Three: 2
Four: 2
Resuming Group A
Waiting for threads to finish.
One: 5
Two: 5
Three: 1
Four: 1
One: 4
Two: 4
Three exiting.
Four exiting.
One: 3
Two: 3
One: 2
Two: 2
One: 1
Two: 1
One exiting.
Two exiting.
Main thread exiting.
```

Inside the program, notice that thread group A is suspended for four seconds. As the output confirms, this causes threads One and Two to pause, but threads Three and Four continue running. After the four seconds, threads One and Two are resumed. Notice how thread group A is suspended and resumed. First, the threads in group A are obtained by calling **enumerate( )** on group A. Then, each thread is suspended by iterating through the resulting array. To resume the threads in A, the list is again traversed and each thread is resumed. One last point: This example uses the recommended approach to suspending and resuming threads. It does not rely upon the deprecated methods **suspend( )** and **resume( )**.

ThreadLocal and InheritableThreadLocal

Java defines two additional thread-related classes in **java.lang**:

- **ThreadLocal** Used to create thread local variables. Each thread will have its own copy of a thread local variable.
- **InheritableThreadLocal** Creates thread local variables that may be inherited.

Package

Package encapsulates version data associated with a package. **Package** version information is becoming more important because of the proliferation of packages and because a Java program may need to know what version of a package is available. The methods defined by **Package** are shown in Table 17-20. The following program demonstrates **Package**, displaying the packages about which the program currently is aware:

```
// Demonstrate Package
class PkgTest {
  public static void main(String args[]) {
    Package pkgs[];

    pkgs = Package.getPackages();

    for(int i=0; i < pkgs.length; i++)
      System.out.println(
             pkgs[i].getName() + " " +
             pkgs[i].getImplementationTitle() + " " +
             pkgs[i].getImplementationVendor() + " " +
             pkgs[i].getImplementationVersion()
      );
  }
}
```

Method	Description
<A extends Annotation> A getAnnotation(Class<A> *annoType*)	Returns an **Annotation** object that contains the annotation associated with *annoType* for the invoking object.
Annotation[ ] getAnnotations( )	Returns all annotations associated with the invoking object in an array of **Annotation** objects. Returns a reference to this array.
<A extends Annotation> A[ ] getAnnotationsByType(Class<A> *annoType*)	Returns an array of the annotations (including repeated annotations) of *annoType* associated with the invoking object. (Added by JDK 8.)
<A extends Annotation> A getDeclaredAnnotation(Class<A> *annoType*)	Returns an **Annotation** object that contains the non-inherited annotation associated with *annoType*. (Added by JDK 8.)
Annotation[ ] getDeclaredAnnotations( )	Returns an **Annotation** object for all the annotations that are declared by the invoking object. (Inherited annotations are ignored.)
<A extends Annotation> A[ ] getDeclaredAnnotationsByType(Class<A> *annoType*)	Returns an array of the non-inherited annotations (including repeated annotations) of *annoType* associated with the invoking object. (Added by JDK 8.)
String getImplementationTitle( )	Returns the title of the invoking package.

Table 17-20 The Methods Defined by **Package**

Method	Description
String getImplementationVendor( )	Returns the name of the implementor of the invoking package.
String getImplementationVersion( )	Returns the version number of the invoking package.
String getName( )	Returns the name of the invoking package.
static Package getPackage(String *pkgName*)	Returns a **Package** object with the name specified by *pkgName*.
static Package[ ] getPackages( )	Returns all packages about which the invoking program is currently aware.
String getSpecificationTitle( )	Returns the title of the invoking package's specification.
String getSpecificationVendor( )	Returns the name of the owner of the specification for the invoking package.
String getSpecificationVersion( )	Returns the invoking package's specification version number.
int hashCode( )	Returns the hash code for the invoking package.
boolean isAnnotationPresent(Class<? extends Annotation> *anno*)	Returns **true** if the annotation described by *anno* is associated with the invoking object. Returns **false** otherwise.
boolean isCompatibleWith(String *verNum*) throws NumberFormatException	Returns **true** if *verNum* is less than or equal to the invoking package's version number.
boolean isSealed( )	Returns **true** if the invoking package is sealed. Returns **false** otherwise.
boolean isSealed(URL *url*)	Returns **true** if the invoking package is sealed relative to *url*. Returns **false** otherwise.
String toString( )	Returns the string equivalent of the invoking package.

Table 17-20 The Methods Defined by **Package** *(continued)*

RuntimePermission

RuntimePermission relates to Java's security mechanism and is not examined further here.

Throwable

The **Throwable** class supports Java's exception-handling system and is the class from which all exception classes are derived. It is discussed in Chapter 10.

SecurityManager

SecurityManager supports Java's security system. A reference to the current security manager can be obtained by calling **getSecurityManager( )** defined by the **System** class.

StackTraceElement

The **StackTraceElement** class describes a single *stack frame*, which is an individual element of a stack trace when an exception occurs. Each stack frame represents an *execution point*, which includes such things as the name of the class, the name of the method, the name of the file, and the source-code line number. An array of **StackTraceElement**s is returned by the **getStackTrace( )** method of the **Throwable** class.

StackTraceElement has one constructor:

StackTraceElement(String *className*, String *methName*, string *fileName*, int *line*)

Here, the name of the class is specified by *className*, the name of the method is specified in *methName*, the name of the file is specified by *fileName*, and the line number is passed in *line*. If there is no valid line number, use a negative value for *line*. Furthermore, a value of –2 for *line* indicates that this frame refers to a native method.

The methods supported by **StackTraceElement** are shown in Table 17-21. These methods give you programmatic access to a stack trace.

Method	Description
boolean equals(Object *ob*)	Returns **true** if the invoking **StackTraceElement** is the same as the one passed in *ob*. Otherwise, it returns **false**.
String getClassName( )	Returns the name of the class in which the execution point described by the invoking **StackTraceElement** occurred.
String getFileName( )	Returns the name of the file in which the source code of the execution point described by the invoking **StackTraceElement** is stored.
int getLineNumber( )	Returns the source-code line number at which the execution point described by the invoking **StackTraceElement** occurred. In some situations, the line number will not be available, in which case a negative value is returned.
String getMethodName( )	Returns the name of the method in which the execution point described by the invoking **StackTraceElement** occurred.
int hashCode( )	Returns the hash code for the invoking **StackTraceElement**.
boolean isNativeMethod( )	Returns **true** if the execution point described by the invoking **StackTraceElement** occurred in a native method. Otherwise, it returns **false**.
String toString( )	Returns the **String** equivalent of the invoking sequence.

Table 17-21 The Methods Defined by **StackTraceElement**

Enum

As described in Chapter 12, an enumeration is a list of named constants. (Recall that an enumeration is created by using the keyword **enum**.) All enumerations automatically inherit **Enum**. **Enum** is a generic class that is declared as shown here:

```
class Enum<E extends Enum<E>>
```

Here, **E** stands for the enumeration type. **Enum** has no public constructors.

Enum defines several methods that are available for use by all enumerations, which are shown in Table 17-22.

Method	Description
protected final Object clone( ) throws CloneNotSupportedException	Invoking this method causes a **CloneNotSupportedException** to be thrown. This prevents enumerations from being cloned.
final int compareTo(E *e*)	Compares the ordinal value of two constants of the same enumeration. Returns a negative value if the invoking constant has an ordinal value less than *e*'s, zero if the two ordinal values are the same, and a positive value if the invoking constant has an ordinal value greater than *e*'s.
final boolean equals(Object *obj*)	Returns **true** if *obj* and the invoking object refer to the same constant.
final Class<E> getDeclaringClass( )	Returns the type of enumeration of which the invoking constant is a member.
final int hashCode( )	Returns the hash code for the invoking object.
final String name( )	Returns the unaltered name of the invoking constant.
final int ordinal( )	Returns a value that indicates an enumeration constant's position in the list of constants.
String toString( )	Returns the name of the invoking constant. This name may differ from the one used in the enumeration's declaration.
static <T extends Enum<T>> T valueOf(Class<T> *e-type*, String *name*)	Returns the constant associated with *name* in the enumeration type specified by *e-type*.

Table 17-22 The Methods Defined by **Enum**

ClassValue

ClassValue can be used to associate a value with a type. It is a generic type defined like this:

Class ClassValue<T>

It is designed for highly specialized uses, not for normal programming.

The CharSequence Interface

The **CharSequence** interface defines methods that grant read-only access to a sequence of characters. These methods are shown in Table 17-23. This interface is implemented by **String**, **StringBuffer**, and **StringBuilder**, among others.

The Comparable Interface

Objects of classes that implement **Comparable** can be ordered. In other words, classes that implement **Comparable** contain objects that can be compared in some meaningful manner. **Comparable** is generic and is declared like this:

interface Comparable<T>

Here, **T** represents the type of objects being compared.

Method	Description
char charAt(int *idx*)	Returns the character at the index specified by *idx*.
default IntStream chars( )	Returns a stream (in the form of an **IntStream**) to the characters in the invoking object. (Added by JDK 8.)
default IntStream codePoints( )	Returns a stream (in the form of an **IntStream**) to the code points in the invoking object. (Added by JDK 8.)
int length( )	Returns the number of characters in the invoking sequence.
CharSequence subSequence(int *startIdx*, int *stopIdx*)	Returns a subset of the invoking sequence beginning at *startIdx* and ending at *stopIdx*–1.
String toString( )	Returns the **String** equivalent of the invoking sequence.

Table 17-23 The Methods Defined by **CharSequence**

The **Comparable** interface declares one method that is used to determine what Java calls the *natural ordering* of instances of a class. The signature of the method is shown here:

int compareTo(T *obj*)

This method compares the invoking object with *obj*. It returns 0 if the values are equal. A negative value is returned if the invoking object has a lower value. Otherwise, a positive value is returned.

This interface is implemented by several of the classes already reviewed in this book. Specifically, the **Byte**, **Character**, **Double**, **Float**, **Long**, **Short**, **String**, and **Integer** classes define a **compareTo( )** method. So does **Enum**.

The Appendable Interface

Objects of a class that implements **Appendable** can have a character or character sequences appended to it. **Appendable** defines these three methods:

Appendable append(char *ch*) throws IOException
Appendable append(CharSequence *chars*) throws IOException
Appendable append(CharSequence *chars*, int *begin*, int *end*) throws IOException

In the first form, the character *ch* is appended to the invoking object. In the second form, the character sequence *chars* is appended to the invoking object. The third form allows you to indicate a portion (the characters running from *begin* through *end*–1) of the sequence specified by *chars*. In all cases, a reference to the invoking object is returned.

The Iterable Interface

Iterable must be implemented by any class whose objects will be used by the for-each version of the **for** loop. In other words, in order for an object to be used within a for-each style **for** loop, its class must implement **Iterable**. **Iterable** is a generic interface that has this declaration:

interface Iterable<T>

Here, **T** is the type of the object being iterated. It defines one abstract method, **iterator( )**, which is shown here:

Iterator<T> iterator( )

It returns an iterator to the elements contained in the invoking object.

Beginning with JDK 8, **Iterable** also defines two default methods. The first is called **forEach( )**:

default void forEach(Consumer<? super T> *action*)

For each element being iterated, **forEach( )** executes the code specified by *action*. (**Consumer** is a functional interface added by JDK 8 and defined in **java.util.function**. See Chapter 19.)

The second default method is **spliterator( )**, shown next:

default Spliterator<T> spliterator( )

It returns a **Spliterator** to the sequence being iterated. (See Chapters 18 and 29 for details on spliterators.)

NOTE Iterators are described in detail in Chapter 18.

The Readable Interface

The **Readable** interface indicates that an object can be used as a source for characters. It defines one method called **read( )**, which is shown here:

int read(CharBuffer *buf*) throws IOException

This method reads characters into *buf*. It returns the number of characters read, or –1 if an EOF is encountered.

The AutoCloseable Interface

AutoCloseable provides support for the **try**-with-resources statement, which implements what is sometimes referred to as *automatic resource management* (ARM). The **try**-with-resources statement automates the process of releasing a resource (such as a stream) when it is no longer needed. (See Chapter 13 for details.) Only objects of classes that implement **AutoCloseable** can be used with **try**-with-resources. The **AutoCloseable** interface defines only the **close( )** method, which is shown here:

void close( ) throws Exception

This method closes the invoking object, releasing any resources that it may hold. It is automatically called at the end of a **try**-with-resources statement, thus eliminating the need to explicitly invoke **close( )**. **AutoCloseable** is implemented by several classes, including all of the I/O classes that open a stream that can be closed.

The Thread.UncaughtExceptionHandler Interface

The static **Thread.UncaughtExceptionHandler** interface is implemented by classes that want to handle uncaught exceptions. It is implemented by **ThreadGroup**. It declares only one method, which is shown here:

void uncaughtException(Thread *thrd*, Throwable *exc*)

Here, *thrd* is a reference to the thread that generated the exception and *exc* is a reference to the exception.

The java.lang Subpackages

Java defines several subpackages of **java.lang**:

- java.lang.annotation
- java.lang.instrument
- java.lang.invoke

- java.lang.management
- java.lang.ref
- java.lang.reflect

Each is briefly described here.

java.lang.annotation

Java's annotation facility is supported by **java.lang.annotation**. It defines the **Annotation** interface, the **ElementType** and **RetentionPolicy** enumerations, and several predefined annotations. Annotations are described in Chapter 12.

java.lang.instrument

java.lang.instrument defines features that can be used to add instrumentation to various aspects of program execution. It defines the **Instrumentation** and **ClassFileTransformer** interfaces, and the **ClassDefinition** class.

java.lang.invoke

java.lang.invoke supports dynamic languages. It includes classes such as **CallSite**, **MethodHandle**, and **MethodType**.

java.lang.management

The **java.lang.management** package provides management support for the JVM and the execution environment. Using the features in **java.lang.management**, you can observe and manage various aspects of program execution.

java.lang.ref

You learned earlier that the garbage collection facilities in Java automatically determine when no references exist to an object. The object is then assumed to be no longer needed and its memory is reclaimed. The classes in the **java.lang.ref** package provide more flexible control over the garbage collection process.

java.lang.reflect

Reflection is the ability of a program to analyze code at run time. The **java.lang.reflect** package provides the ability to obtain information about the fields, constructors, methods, and modifiers of a class. Among other reasons, you need this information to build software tools that enable you to work with Java Beans components. The tools use reflection to determine dynamically the characteristics of a component. Reflection was introduced in Chapter 12 and is also examined in Chapter 30.

java.lang.reflect defines several classes, including **Method**, **Field**, and **Constructor**. It also defines several interfaces, including **AnnotatedElement**, **Member**, and **Type**. In addition, the **java.lang.reflect** package includes the **Array** class that enables you to create and access arrays dynamically.

CHAPTER

18 java.util Part 1: The Collections Framework

This chapter begins our examination of **java.util**. This important package contains a large assortment of classes and interfaces that support a broad range of functionality. For example, **java.util** has classes that generate pseudorandom numbers, manage date and time, observe events, manipulate sets of bits, tokenize strings, and handle formatted data. The **java.util** package also contains one of Java's most powerful subsystems: the *Collections Framework*. The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects. It merits close attention by all programmers.

Because **java.util** contains a wide array of functionality, it is quite large. Here is a list of its top-level classes:

AbstractCollection	FormattableFlags	Properties
AbstractList	Formatter	PropertyPermission
AbstractMap	GregorianCalendar	PropertyResourceBundle
AbstractQueue	HashMap	Random
AbstractSequentialList	HashSet	ResourceBundle
AbstractSet	Hashtable	Scanner
ArrayDeque	IdentityHashMap	ServiceLoader
ArrayList	IntSummaryStatistics (Added by JDK 8.)	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators (Added by JDK 8.)
Base64 (Added by JDK 8.)	LinkedHashSet	SplitableRandom (Added by JDK 8.)
BitSet	LinkedList	Stack
Calendar	ListResourceBundle	StringJoiner (Added by JDK 8.)

Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics (Added by JDK 8.)	Timer
Date	Objects	TimerTask
Dictionary	Observable	TimeZone
DoubleSummaryStatistics (Added by JDK 8.)	Optional (Added by JDK 8.)	TreeMap
EnumMap	OptionalDouble (Added by JDK 8.)	TreeSet
EnumSet	OptionalInt (Added by JDK 8.)	UUID
EventListenerProxy	OptionalLong (Added by JDK 8.)	Vector
EventObject	PriorityQueue	WeakHashMap

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

Collection	Map.Entry	Set
Comparator	NavigableMap	SortedMap
Deque	NavigableSet	SortedSet
Enumeration	Observer	Spliterator (Added by JDK 8.)
EventListener	PrimitiveIterator (Added by JDK 8.)	Spliterator.OfDouble (Added by JDK 8.)
Formattable	PrimitiveIterator.OfDouble (Added by JDK 8.)	Spliterator.OfInt (Added by JDK 8.)
Iterator	PrimitiveIterator.OfInt (Added by JDK 8.)	Spliterator.OfLong (Added by JDK 8.)
List	PrimitiveIterator.OfLong (Added by JDK 8.)	Spliterator.OfPrimitive (Added by JDK 8.)
ListIterator	Queue	
Map	RandomAccess	

Because of its size, the description of **java.util** is broken into two chapters. This chapter examines those members of **java.util** that are part of the Collections Framework. Chapter 18 discusses its other classes and interfaces.

Collections Overview

The Java Collections Framework standardizes the way in which groups of objects are handled by your programs. Collections were not part of the original Java release, but were added by J2SE 1.2. Prior to the Collections Framework, Java provided ad hoc classes such as **Dictionary**, **Vector**, **Stack**, and **Properties** to store and manipulate groups of objects. Although these

classes were quite useful, they lacked a central, unifying theme. The way that you used **Vector** was different from the way that you used **Properties**, for example. Also, this early, ad hoc approach was not designed to be easily extended or adapted. Collections are an answer to these (and other) problems.

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. You seldom, if ever, need to code one of these "data engines" manually. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and/or adapting a collection had to be easy. Toward this end, the entire Collections Framework is built upon a set of standard interfaces. Several standard implementations (such as **LinkedList**, **HashSet**, and **TreeSet**) of these interfaces are provided that you may use as-is. You may also implement your own collection, if you choose. Various special-purpose implementations are created for your convenience, and some partial implementations are provided that make creating your own collection class easier. Finally, mechanisms were added that allow the integration of standard arrays into the Collections Framework.

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the **Collections** class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.

Another item closely associated with the Collections Framework is the **Iterator** interface. An *iterator* offers a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus, an iterator provides a means of *enumerating the contents of a collection*. Because each collection provides an iterator, the elements of any collection class can be accessed through the methods defined by **Iterator**. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example.

JDK 8 adds another type of iterator called a *spliterator*. In brief, spliterators are iterators that provide support for parallel iteration. The interfaces that support spliterators are **Spliterator** and several nested interfaces that support primitive types. JDK 8 also adds iterator interfaces designed for use with primitive types, such as **PrimitiveIterator** and **PrimitiveIterator.OfDouble**.

In addition to collections, the framework defines several map interfaces and classes. *Maps* store key/value pairs. Although maps are part of the Collections Framework, they are not "collections" in the strict use of the term. You can, however, obtain a *collection-view* of a map. Such a view contains the elements from the map stored in a collection. Thus, you can process the contents of a map as a collection, if you choose.

The collection mechanism was retrofitted to some of the original classes defined by **java.util** so that they too could be integrated into the new system. It is important to understand that although the addition of collections altered the architecture of many of the original utility classes, it did not cause the deprecation of any. Collections simply provide a better way of doing several things.

NOTE If you are familiar with C++, then you will find it helpful to know that the Java collections technology is similar in spirit to the Standard Template Library (STL) defined by C++. What C++ calls a container, Java calls a collection. However, there are significant differences between the Collections Framework and the STL. It is important to not jump to conclusions.

JDK 5 Changed the Collections Framework

When JDK 5 was released, some fundamental changes were made to the Collections Framework that significantly increased its power and streamlined its use. These changes include the addition of generics, autoboxing/unboxing, and the for-each style **for** loop. Although JDK 8 is three major Java releases after JDK 5, the effects of the JDK 5 features were so profound that they still warrant special attention. The main reason is that you may encounter pre-JDK 5 code. Understanding the effects and reasons for the changes is important if you will be maintaining or updating older code.

Generics Fundamentally Changed the Collections Framework

The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework was reengineered for it. All collections are now generic, and many of the methods that operate on collections take generic type parameters. Simply put, the addition of generics affected every part of the Collections Framework.

Generics added the one feature that collections had been missing: type safety. Prior to generics, all collections stored **Object** references, which meant that any collection could store any type of object. Thus, it was possible to accidentally store incompatible types in a collection. Doing so could result in run-time type mismatch errors. With generics, it is possible to explicitly state the type of data being stored, and run-time type mismatch errors can be avoided.

Although the addition of generics changed the declarations of most of its classes and interfaces, and several of their methods, overall, the Collections Framework still works the same as it did prior to generics. Of course, to gain the advantages that generics bring collections, older code will need to be rewritten. This is also important because pre-generics code will generate warning messages when compiled by a modern Java compiler. To eliminate these warnings, you will need to add type information to all your collections code.

Autoboxing Facilitates the Use of Primitive Types

Autoboxing/unboxing facilitates the storing of primitive types in collections. As you will see, a collection can store only references, not primitive values. In the past, if you wanted to store a primitive value, such as an **int**, in a collection, you had to manually box it into its type wrapper. When the value was retrieved, it needed to be manually unboxed (by using an explicit cast) into its proper primitive type. Because of autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

The For-Each Style for Loop

All collection classes in the Collections Framework were retrofitted to implement the **Iterable** interface, which means that a collection can be cycled through by use of the for-each style **for** loop. In the past, cycling through a collection required the use of an iterator (described later in this chapter), with the programmer manually constructing the loop. Although iterators are still needed for some uses, in many cases, iterator-based loops can be replaced by **for** loops.

The Collection Interfaces

The Collections Framework defines several core interfaces. This section provides an overview of each interface. Beginning with the collection interfaces is necessary because they determine the fundamental nature of the collection classes. Put differently, the concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are summarized in the following table:

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.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
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.

In addition to the collection interfaces, collections also use the **Comparator**, **RandomAccess**, **Iterator**, and **ListIterator** interfaces, which are described in depth later in this chapter. Beginning with JDK 8, **Spliterator** can also be used. Briefly, **Comparator** defines how two objects are compared; **Iterator**, **ListIterator**, and **Spliterator** enumerate the objects within a collection. By implementing **RandomAccess**, a list indicates that it supports efficient, random access to its elements.

To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional. The optional methods enable you to modify the contents of a collection. Collections that support these methods are called *modifiable*. Collections that do not allow their contents to be changed are called *unmodifiable*. If an attempt is made to use one of these methods on an unmodifiable collection, an **UnsupportedOperationException** is thrown. All the built-in collections are modifiable.

The following sections examine the collection interfaces.

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. **Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop. (Recall that only classes that implement **Iterable** can be cycled through by the **for**.)

Collection declares the core methods that all collections will have. These methods are summarized in Table 18-1. Because all collections implement **Collection**, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**. As explained, this occurs if a collection cannot be modified. A **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. 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.

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
default Stream<E> parallelStream( )	Returns a stream that uses the invoking collection as its source for elements. If possible, the stream supports parallel operations. (Added by JDK 8.)
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
default boolean removeIf(Predicate<? super E> *predicate*)	Removes from the invoking collection those elements that satisfy the condition specified by *predicate*. (Added by JDK 8.)

Table 18-1 The Methods Declared by **Collection**

Method	Description
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
default Spliterator<E> spliterator( )	Returns a spliterator to the invoking collections. (Added by JDK 8.)
default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential. (Added by JDK 8.)
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

Table 18-1 The Methods Declared by **Collection** *(continued)*

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( )**. Beginning with JDK 8, to remove an element only if it statisfies some condition, you can use **removeIf( )**. (**Predicate** is a functional interface added by JDK 8. See Chapter 19.) To empty a collection, call **clear( )**.

You can determine whether a collection contains a specific object by calling **contains( )**. To determine whether one collection contains all the members of another, call **containsAll( )**. You can determine when a collection is empty by calling **isEmpty( )**. The number of elements currently held in a collection can be determined by calling **size( )**.

The **toArray( )** methods return an array that contains the elements stored in the invoking collection. The first returns an array of **Object**. The second returns an array of elements that have the same type as the array specified as a parameter. Normally, the second form is more convenient because it returns the desired array type. These methods are more important than it might at first seem. Often, processing the contents of a

collection by using array-like syntax is advantageous. By providing a pathway between collections and arrays, you can have the best of both worlds.

Two collections can be compared for equality by calling **equals( )**. The precise meaning of "equality" may differ from collection to collection. For example, you can implement **equals( )** so that it compares the values of elements stored in the collection. Alternatively, **equals( )** can compare references to those elements.

Another important method is **iterator( )**, which returns an iterator to a collection. The new **spliterator( )** method returns a spliterator to the collection. Iterators are frequently used when working with collections. Finally, the **stream( )** and **parallelStream( )** methods return a **Stream** that uses the collection as a source of elements. (See Chapter 29 for a detailed discussion of the new **Stream** interface.)

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.

In addition to the methods defined by **Collection**, **List** defines some of its own, which are summarized in Table 18-2. Note again that several of these methods will throw an **UnsupportedOperationException** if the list cannot be modified, and a **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a list. Also, several methods will throw an **IndexOutOfBoundsException** if an invalid index is used. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the list. An **IllegalArgumentException** is thrown if an invalid argument is used.

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. You can modify each element in the collection by using **replaceAll( )**. (**UnaryOperator** is a functional interface added by JDK 8. See Chapter 19.)

To obtain the object stored at a specific location, call **get( )** with the index of the object. To assign a value to an element in the list, call **set( )**, specifying the index of the object to be changed. To find the index of an object, use **indexOf( )** or **lastIndexOf( )**.

You can obtain a sublist of a list by calling **subList( )**, specifying the beginning and ending indexes of the sublist. As you can imagine, **subList( )** makes list processing quite convenient. One way to sort a list is with the **sort( )** method defined by **List**.

The Set Interface

The **Set** interface defines a set. It extends **Collection** and specifies the behavior of a collection that does not allow duplicate elements. Therefore, the **add( )** method returns

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.
default void replaceAll(UnaryOperator<E> *opToApply*)	Updates each element in the list with the value obtained from the *opToApply* function. (Added by JDK 8.)
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
default void sort(Comparator<? super E> *comp*)	Sorts the list using the comparator specified by *comp*. (Added by JDK 8.)
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 18-2 The Methods Declared by **List**

false if an attempt is made to add duplicate elements to a set. It does not specify 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.

In addition to those methods provided by **Set**, the **SortedSet** interface declares the methods summarized in Table 18-3. Several methods throw a **NoSuchElementException** when no items are contained in the invoking set. A **ClassCastException** is thrown when an object is incompatible with the elements in a set. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

SortedSet defines several methods that make set processing more convenient. To obtain the first object in the set, call **first( )**. To get the last element, use **last( )**. You can obtain a subset of a sorted set by calling **subSet( )**, specifying the first and last object in the set. If you need the subset that starts with the first element in the set, use **headSet( )**. If you want the subset that ends the set, use **tailSet( )**.

Method	Description
Comparator<? super E> comparator( )	Returns the invoking sorted set's comparator. If the natural ordering is used for this set, **null** is returned.
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

Table 18-3 The Methods Declared by **SortedSet**

The NavigableSet Interface

The **NavigableSet** interface extends **SortedSet** and declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values. **NavigableSet** is a generic interface that has this declaration:

```
interface NavigableSet<E>
```

Here, **E** specifies the type of objects that the set will hold. In addition to the methods that it inherits from **SortedSet**, **NavigableSet** adds those summarized in Table 18-4. A

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E pollFirst( )	Returns the first element, removing the element in the process. Because the set is sorted, this is the element with the least value. **null** is returned if the set is empty.
E pollLast( )	Returns the last element, removing the element in the process. Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
NavigableSet<E> subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
NavigableSet<E> tailSet(E *lowerBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting set is backed by the invoking set.

Table 18-4 The Methods Declared by **NavigableSet**

ClassCastException is thrown when an object is incompatible with the elements in the set. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

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. The methods declared by **Queue** are shown in Table 18-5.

Several methods throw a **ClassCastException** when an object is incompatible with the elements in the queue. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the queue. 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 queue that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty queue.

Despite its simplicity, **Queue** offers several points of interest. First, elements can only be removed from the head of the queue. Second, there are two methods that obtain and remove elements: **poll( )** and **remove( )**. The difference between them is that **poll( )** returns **null** if the queue is empty, but **remove( )** throws an exception. Third, there are two methods, **element( )** and **peek( )**, that obtain but don't remove the element at the head of the queue. They differ only in that **element( )** throws an exception if the queue is empty, but **peek( )** returns **null**. Finally, notice that **offer( )** only attempts to add an element to a queue. Because some queues have a fixed length and might be full, **offer( )** can fail.

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 18-5 The Methods Declared by **Queue**

The Deque Interface

The **Deque** interface 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** adds those methods summarized in Table 18-6. Several

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.

Table 18-6 The Methods Declared by **Deque**

Method	Description
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

Table 18-6 The Methods Declared by **Deque** *(continued)*

methods throw a **ClassCastException** when an object is incompatible with the elements in the deque. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the deque. 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 deque that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty deque.

Notice that **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 cannot be added.

The Collection Classes

Now that you are familiar with the collection interfaces, you are ready to examine the standard classes that implement them. Some of the classes provide full implementations that can be used as-is. Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections. As a general rule, the collection classes are not synchronized, but as you will see later in this chapter, it is possible to obtain synchronized versions.

The core collection classes are summarized in the following table:

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.
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 following sections examine the concrete collection classes and illustrate their use.

NOTE In addition to the collection classes, several legacy classes, such as **Vector**, **Stack**, and **Hashtable**, have been reengineered to support collections. These are examined later in this chapter.

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. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**. In essence, an **ArrayList** is a variable-length array of object references. That is, 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.

NOTE Dynamic arrays are also supported by the legacy class **Vector**, which is described later in this chapter.

ArrayList has the constructors shown here:

ArrayList( )
ArrayList(Collection<? extends E> *c*)
ArrayList(int *capacity*)

The first constructor builds an empty array list. The second constructor builds an array list that is initialized with the elements of the collection c. The third constructor builds an array list that has the specified initial *capacity*. The capacity is the size of the underlying array that is used to store the elements. The capacity grows automatically as elements are added to an array list.

The following program shows a simple use of **ArrayList**. An array list is created for objects of type **String**, and then several strings are added to it. (Recall that a quoted string is translated into a **String** object.) The list is then displayed. Some of the elements are removed and the list is displayed again.

```
// 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]
```

Notice that **a1** starts out empty and grows as elements are added to it. When elements are removed, its size is reduced.

In the preceding example, the contents of a collection are displayed using the default conversion provided by **toString( )**, which was inherited from **AbstractCollection**. Although it is sufficient for short, sample programs, you seldom use this method to display the contents of a real-world collection. Usually, you provide your own output routines. But, for the next few examples, the default output created by **toString( )** is sufficient.

Although the capacity of an **ArrayList** object increases automatically as objects are stored in it, you can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( )**. You might want to do this if you know in advance that you will be storing many more items in the collection than it can currently hold. By increasing its capacity once, at the start, you can prevent several reallocations later. Because reallocations are costly in terms of time, preventing unnecessary ones improves performance. The signature for **ensureCapacity( )** is shown here:

void ensureCapacity(int *cap*)

Here, *cap* specifies the new minimum capacity of the collection.

Conversely, if you want to reduce the size of the array that underlies an **ArrayList** object so that it is precisely as large as the number of items that it is currently holding, call **trimToSize( )**, shown here:

void trimToSize( )

Obtaining an Array from an ArrayList

When working with **ArrayList**, you will sometimes want to obtain an actual array that contains the contents of the list. You can do this by calling **toArray( )**, which is defined by **Collection**. Several reasons exist why you might want to convert a collection into an array, such as:

- To obtain faster processing times for certain operations
- To pass an array to a method that is not overloaded to accept a collection
- To integrate collection-based code with legacy code that does not understand collections

Whatever the reason, converting an **ArrayList** to an array is a trivial matter.

As explained earlier, there are two versions of **toArray( )**, which are shown again here for your convenience:

object[ ] toArray( )
<T> T[ ] toArray(T *array*[ ])

The first returns an array of **Object**. The second returns an array of elements that have the same type as **T**. Normally, the second form is more convenient because it returns the proper type of array. The following program demonstrates its use:

```
// Convert an ArrayList into an array.
import java.util.*;

class ArrayListToArray {
  public static void main(String args[]) {
    // Create an array list.
    ArrayList<Integer> al = new ArrayList<Integer>();

    // Add elements to the array list.
    al.add(1);
    al.add(2);
    al.add(3);
    al.add(4);

    System.out.println("Contents of al: " + al);

    // Get the array.
    Integer ia[] = new Integer[al.size()];
    ia = al.toArray(ia);

    int sum = 0;

    // Sum the array.
    for(int i : ia) sum += i;

    System.out.println("Sum is: " + sum);
  }
}
```

The output from the program is shown here:

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

The program begins by creating a collection of integers. Next, **toArray( )** is called and it obtains an array of **Integer**s. Then, the contents of that array are summed by use of a for-each style **for** loop.

There is something else of interest in this program. As you know, collections can store only references, not values of primitive types. However, autoboxing makes it possible to pass values of type **int** to **add( )** without having to manually wrap them within an **Integer**, as the program shows. Autoboxing causes them to be automatically wrapped. In this way, autoboxing significantly improves the ease with which collections can be used to store primitive values.

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)
```

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection *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);
```