The output from the program is shown here:

```
Hello 47
```

In the program, notice how **Gen** inherits **NonGen** in the following declaration:

```
class Gen<T> extends NonGen {
```

Because **NonGen** is not generic, no type argument is specified. Thus, even though **Gen** declares the type parameter **T**, it is not needed by (nor can it be used by) **NonGen**. Thus, **NonGen** is inherited by **Gen** in the normal way. No special conditions apply.

Run-Time Type Comparisons Within a Generic Hierarchy

Recall the run-time type information operator **instanceof** that was described in Chapter 13. As explained, **instanceof** determines if an object is an instance of a class. It returns true if an object is of the specified type or can be cast to the specified type. The **instanceof** operator can be applied to objects of generic classes. The following class demonstrates some of the type compatibility implications of a generic hierarchy:

```
// Use the instanceof operator with a generic class hierarchy.
class Gen<T> {
  T ob;

  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2<T> extends Gen<T> {
  Gen2(T o) {
    super(o);
  }
}

// Demonstrate run-time type ID implications of generic
// class hierarchy.
class HierDemo3 {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);
```

```
    // Create a Gen2 object for Strings.
    Gen2<String> strOb2 = new Gen2<String>("Generics Test");

    // See if iOb2 is some form of Gen2.
    if(iOb2 instanceof Gen2<?>)
      System.out.println("iOb2 is instance of Gen2");

    // See if iOb2 is some form of Gen.
    if(iOb2 instanceof Gen<?>)
      System.out.println("iOb2 is instance of Gen");

    System.out.println();

    // See if strOb2 is a Gen2.
    if(strOb2 instanceof Gen2<?>)
      System.out.println("strOb2 is instance of Gen2");

    // See if strOb2 is a Gen.
    if(strOb2 instanceof Gen<?>)
      System.out.println("strOb2 is instance of Gen");

    System.out.println();

    // See if iOb is an instance of Gen2, which it is not.
    if(iOb instanceof Gen2<?>)
      System.out.println("iOb is instance of Gen2");

    // See if iOb is an instance of Gen, which it is.
    if(iOb instanceof Gen<?>)
      System.out.println("iOb is instance of Gen");

    // The following can't be compiled because
    // generic type info does not exist at run time.
//    if(iOb2 instanceof Gen2<Integer>)
//      System.out.println("iOb2 is instance of Gen2<Integer>");
  }
}
```

The output from the program is shown here:

```
iOb2 is instance of Gen2
iOb2 is instance of Gen

strOb2 is instance of Gen2
strOb2 is instance of Gen

iOb is instance of Gen
```

In this program, **Gen2** is a subclass of **Gen**, which is generic on type parameter **T**. In **main( )**, three objects are created. The first is **iOb**, which is an object of type **Gen<Integer>**. The second is **iOb2**, which is an instance of **Gen2<Integer>**. Finally, **strOb2** is an object of type **Gen2<String>**.

Then, the program performs these **instanceof** tests on the type of **iOb2**:

```
// See if iOb2 is some form of Gen2.
if(iOb2 instanceof Gen2<?>)
  System.out.println("iOb2 is instance of Gen2");

// See if iOb2 is some form of Gen.
if(iOb2 instanceof Gen<?>)
  System.out.println("iOb2 is instance of Gen");
```

As the output shows, both succeed. In the first test, **iOb2** is checked against **Gen2<?>**. This test succeeds because it simply confirms that **iOb2** is an object of some type of **Gen2** object. The use of the wildcard enables **instanceof** to determine if **iOb2** is an object of any type of **Gen2**. Next, **iOb2** is tested against **Gen<?>**, the superclass type. This is also true because **iOb2** is some form of **Gen**, the superclass. The next few lines in **main( )** show the same sequence (and same results) for **strOb2**.

Next, **iOb**, which is an instance of **Gen<Integer>** (the superclass), is tested by these lines:

```
// See if iOb is an instance of Gen2, which it is not.
if(iOb instanceof Gen2<?>)
  System.out.println("iOb is instance of Gen2");

// See if iOb is an instance of Gen, which it is.
if(iOb instanceof Gen<?>)
  System.out.println("iOb is instance of Gen");
```

The first **if** fails because **iOb** is not some type of **Gen2** object. The second test

succeeds because **iOb** is some type of **Gen** object.

Now, look closely at these commented-out lines:

```
    // The following can't be compiled because
    // generic type info does not exist at run time.
//    if(iOb2 instanceof Gen2<Integer>)
//      System.out.println("iOb2 is instance of Gen2<Integer>");
```

As the comments indicate, these lines can't be compiled because they attempt to compare **iOb2** with a specific type of **Gen2**, in this case, **Gen2<Integer>**. Remember, there is no generic type information available at run time. Therefore, there is no way for **instanceof** to know if **iOb2** is an instance of **Gen2<Integer>** or not.

Casting

You can cast one instance of a generic class into another only if the two are otherwise compatible and their type arguments are the same. For example, assuming the foregoing program, this cast is legal:

```
(Gen<Integer>) iOb2 // legal
```

because **iOb2** includes an instance of **Gen<Integer>**. But, this cast:

```
(Gen<Long>) iOb2 // illegal
```

is not legal because **iOb2** is not an instance of **Gen<Long>**.

Overriding Methods in a Generic Class

A method in a generic class can be overridden just like any other method. For example, consider this program in which the method **getob( )** is overridden:

```
// Overriding a generic method in a generic class.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    System.out.print("Gen's getob(): " );
    return ob;
  }
}

// A subclass of Gen that overrides getob().
class Gen2<T> extends Gen<T> {

  Gen2(T o) {
    super(o);
  }

  // Override getob().
  T getob() {
    System.out.print("Gen2's getob(): ");
    return ob;
  }
}

// Demonstrate generic method override.
class OverrideDemo {
  public static void main(String args[]) {

    // Create a Gen object for Integers.
    Gen<Integer> iOb = new Gen<Integer>(88);

    // Create a Gen2 object for Integers.
    Gen2<Integer> iOb2 = new Gen2<Integer>(99);

    // Create a Gen2 object for Strings.
    Gen2<String> strOb2 = new Gen2<String> ("Generics Test");

    System.out.println(iOb.getob());
    System.out.println(iOb2.getob());
    System.out.println(strOb2.getob());
  }
}
```

The output is shown here:

```
Gen's getob(): 88
Gen2's getob(): 99
Gen2's getob(): Generics Test
```

As the output confirms, the overridden version of **getob( )** is called for objects of type **Gen2**, but the superclass version is called for objects of type **Gen**.

Type Inference with Generics

Beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic type. To begin, consider the following generic class:

```
class MyClass<T, V> {
  T ob1;
  V ob2;

  MyClass(T o1, V o2) {
    ob1 = o1;
    ob2 = o2;
  }
  // ...
}
```

Prior to JDK 7, to create an instance of **MyClass**, you would have needed to use a statement similar to the following:

```
MyClass<Integer, String> mcOb =
  new MyClass<Integer, String>(98, "A String");
```

Here, the type arguments (which are **Integer** and **String**) are specified twice: first, when **mcOb** is declared, and second, when a **MyClass** instance is created via **new**. Since generics were introduced by JDK 5, this is the form required by all versions of Java prior to JDK 7. Although there is nothing wrong, per se, with this form, it is a bit more verbose than it needs to be. In the **new** clause, the type of the type arguments can be readily inferred from the type of **mcOb**; therefore, there is really no reason that they need to be specified a second time. To address this situation, JDK 7 added a syntactic element that lets you avoid the second

specification.

Today the preceding declaration can be rewritten as shown here:

```
MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
```

Notice that the instance creation portion simply uses <>, which is an empty type argument list. This is referred to as the *diamond* operator. It tells the compiler to infer the type arguments needed by the constructor in the **new** expression. The principal advantage of this type-inference syntax is that it shortens what are sometimes quite long declaration statements.

The preceding can be generalized. When type inference is used, the declaration syntax for a generic reference and instance creation has this general form:

class-name<*type-arg-list* > *var-name* = new *class-name* <>(*cons-arg-list*);

Here, the type argument list of the constructor in the **new** clause is empty.

Type inference can also be applied to parameter passing. For example, if the following method is added to **MyClass**,

```
boolean isSame(MyClass<T, V> o) {
  if(ob1 == o.ob1 && ob2 == o.ob2) return true;
  else return false;
}
```

then the following call is legal:

```
if(mcOb.isSame(new MyClass<>(1, "test")))
System.out.println("Same");
```

In this case, the type arguments for the argument passed to **isSame( )** can be inferred from the parameter's type.

Most of the examples in this book will continue to use the full syntax when declaring instances of generic classes. This way, the examples will work with any Java compiler that supports generics. Using the full-length syntax also makes it very clear precisely what is being created, which is important in example code shown in a book. However, in your own code, the use of the type-inference syntax will streamline your declarations.

Local Variable Type Inference and Generics

As just explained, type inference is already supported for generics through the use of the diamond operator. However, you can also use the new local variable type inference feature added by JDK 10 with a generic class. For example, assuming **MyClass** used in the preceding section, this declaration:

```
MyClass<Integer, String> mcOb =
  new MyClass<Integer, String>(98, "A String");
```

can be rewritten like this using local variable type inference:

```
var mcOb = new MyClass<Integer, String>(98, "A String");
```

In this case, the type of **mcOb** is inferred to be **MyClass<Integer, String>** because that is the type of its initializer. Also notice that the use of **var** results in a shorter declaration than would be the case otherwise. In general, generic type names can often be quite long and (in some cases) complicated. The use of **var** is another way to substantially shorten such declarations. For the same reasons as just explained for the diamond operator, the remaining examples in this book will continue to use the full generic syntax, but in your own code the use of local variable type inference can be quite helpful.

Erasure

Usually, it is not necessary to know the details about how the Java compiler transforms your source code into object code. However, in the case of generics, some general understanding of the process is important because it explains why the generic features work as they do—and why their behavior is sometimes a bit surprising. For this reason, a brief discussion of how generics are implemented in Java is in order.

An important constraint that governed the way that generics were added to Java was the need for compatibility with previous versions of Java. Simply put, generic code had to be compatible with preexisting, non-generic code. Thus, any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of *erasure*.

In general, here is how erasure works. When your Java code is compiled, all generic type information is removed (erased). This means replacing type

parameters with their bound type, which is **Object** if no explicit bound is specified, and then applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments. The compiler also enforces this type compatibility. This approach to generics means that no type parameters exist at run time. They are simply a source-code mechanism.

Bridge Methods

Occasionally, the compiler will need to add a *bridge method* to a class to handle situations in which the type erasure of an overriding method in a subclass does not produce the same erasure as the method in the superclass. In this case, a method is generated that uses the type erasure of the superclass, and this method calls the method that has the type erasure specified by the subclass. Of course, bridge methods only occur at the bytecode level, are not seen by you, and are not available for your use.

Although bridge methods are not something that you will normally need to be concerned with, it is still instructive to see a situation in which one is generated. Consider the following program:

```
// A situation that creates a bridge method.
class Gen<T> {
  T ob; // declare an object of type T

  // Pass the constructor a reference to
  // an object of type T.
  Gen(T o) {
    ob = o;
  }

  // Return ob.
  T getob() {
    return ob;
  }
}

// A subclass of Gen.
class Gen2 extends Gen<String> {
```

```
  Gen2(String o) {
    super(o);
  }

  // A String-specific override of getob().
  String getob() {
    System.out.print("You called String getob(): ");
    return ob;
  }
}

// Demonstrate a situation that requires a bridge method.
class BridgeDemo {
  public static void main(String args[]) {

    // Create a Gen2 object for Strings.
    Gen2 strOb2 = new Gen2("Generics Test");

    System.out.println(strOb2.getob());
  }
}
```

In the program, the subclass **Gen2** extends **Gen**, but does so using a **String**-specific version of **Gen**, as its declaration shows:

```
class Gen2 extends Gen<String> {
```

Furthermore, inside **Gen2**, **getob( )** is overridden with **String** specified as the return type:

```
// A String-specific override of getob().
String getob() {
  System.out.print("You called String getob(): ");
  return ob;
}
```

All of this is perfectly acceptable. The only trouble is that because of type erasure, the expected form of **getob( )** will be

```
Object getob() { // ...
```

To handle this problem, the compiler generates a bridge method with the preceding signature that calls the **String** version. Thus, if you examine the class file for **Gen2** by using **javap**, you will see the following methods:

```
class Gen2 extends Gen<java.lang.String> {
  Gen2(java.lang.String);
  java.lang.String getob();
  java.lang.Object getob(); // bridge method
}
```

As you can see, the bridge method has been included. (The comment was added by the author and not by **javap**, and the precise output you see may vary based on the version of Java that you are using.)

There is one last point to make about this example. Notice that the only difference between the two **getob( )** methods is their return type. Normally, this would cause an error, but because this does not occur in your source code, it does not cause a problem and is handled correctly by the JVM.

Ambiguity Errors

The inclusion of generics gives rise to another type of error that you must guard against: *ambiguity*. Ambiguity errors occur when erasure causes two seemingly distinct generic declarations to resolve to the same erased type, causing a conflict. Here is an example that involves method overloading:

```
// Ambiguity caused by erasure on
// overloaded methods.
class MyGenClass<T, V> {
  T ob1;
  V ob2;

  // ...

  // These two overloaded methods are ambiguous
  // and will not compile.
  void set(T o) {
    ob1 = o;
  }

  void set(V o) {
    ob2 = o;
  }
}
```

Notice that **MyGenClass** declares two generic types: **T** and **V**. Inside **MyGenClass**, an attempt is made to overload **set( )** based on parameters of type **T** and **V**. This looks reasonable because **T** and **V** appear to be different types. However, there are two ambiguity problems here.

First, as **MyGenClass** is written, there is no requirement that **T** and **V** actually be different types. For example, it is perfectly correct (in principle) to construct a **MyGenClass** object as shown here:

```
MyGenClass<String, String> obj = new MyGenClass<String, String>()
```

In this case, both **T** and **V** will be replaced by **String**. This makes both versions of **set( )** identical, which is, of course, an error.

The second and more fundamental problem is that the type erasure of **set( )** reduces both versions to the following:

```
void set(Object o) { // ...
```

Thus, the overloading of **set( )** as attempted in **MyGenClass** is inherently ambiguous.

Ambiguity errors can be tricky to fix. For example, if you know that **V** will always be some type of **Number**, you might try to fix **MyGenClass** by rewriting

its declaration as shown here:

```
class MyGenClass<T, V extends Number> { // almost OK!
```

This change causes **MyGenClass** to compile, and you can even instantiate objects like the one shown here:

```
MyGenClass<String, Number> x = new MyGenClass<String, Number>();
```

This works because Java can accurately determine which method to call. However, ambiguity returns when you try this line:

```
MyGenClass<Number, Number> x = new MyGenClass<Number, Number>();
```

In this case, since both **T** and **V** are **Number**, which version of **set( )** is to be called? The call to **set( )** is now ambiguous.

Frankly, in the preceding example, it would be much better to use two separate method names, rather than trying to overload **set( )**. Often, the solution to ambiguity involves the restructuring of the code, because ambiguity frequently means that you have a conceptual error in your design.

Some Generic Restrictions

There are a few restrictions that you need to keep in mind when using generics. They involve creating objects of a type parameter, static members, exceptions, and arrays. Each is examined here.

Type Parameters Can’t Be Instantiated

It is not possible to create an instance of a type parameter. For example, consider this class:

```
// Can't create an instance of T.
class Gen<T> {
  T ob;

  Gen() {
    ob = new T(); // Illegal!!!
  }
}
```

Here, it is illegal to attempt to create an instance of **T**. The reason should be easy to understand: the compiler does not know what type of object to create. **T** is simply a placeholder.

Restrictions on Static Members

No **static** member can use a type parameter declared by the enclosing class. For example, both of the **static** members of this class are illegal:

```
class Wrong<T> {
  // Wrong, no static variables of type T.
  static T ob;

  // Wrong, no static method can use T.
  static T getob() {
    return ob;
  }
}
```

Although you can't declare **static** members that use a type parameter declared by the enclosing class, you can declare **static** generic methods, which define their own type parameters, as was done earlier in this chapter.

Generic Array Restrictions

There are two important generics restrictions that apply to arrays. First, you cannot instantiate an array whose element type is a type parameter. Second, you cannot create an array of type-specific generic references. The following short program shows both situations:

```
// Generics and arrays.
class Gen<T extends Number> {
  T ob;

  T vals[]; // OK

  Gen(T o, T[] nums) {
    ob = o;

    // This statement is illegal.
    // vals = new T[10]; // can't create an array of T

    // But, this statement is OK.
    vals = nums; // OK to assign reference to existent array
  }
}

class GenArrays {
  public static void main(String args[]) {
    Integer n[] = { 1, 2, 3, 4, 5 };

    Gen<Integer> iOb = new Gen<Integer>(50, n);

    // Can't create an array of type-specific generic references.
    // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!

    // This is OK.
    Gen<?> gens[] = new Gen<?>[10]; // OK
  }
}
```

As the program shows, it's valid to declare a reference to an array of type **T**, as this line does:

```
T vals[]; // OK
```

But, you cannot instantiate an array of **T**, as this commented-out line attempts:

```
// vals = new T[10]; // can't create an array of T
```

The reason you can’t create an array of **T** is that there is no way for the compiler to know what type of array to actually create.

However, you can pass a reference to a type-compatible array to **Gen( )** when an object is created and assign that reference to **vals**, as the program does in this line:

```
vals = nums; // OK to assign reference to existent array
```

This works because the array passed to **Gen** has a known type, which will be the same type as **T** at the time of object creation.

Inside **main( )**, notice that you can’t declare an array of references to a specific generic type. That is, this line

```
// Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
```

won’t compile.

You *can* create an array of references to a generic type if you use a wildcard, however, as shown here:

```
Gen<?> gens[] = new Gen<?>[10]; // OK
```

This approach is better than using an array of raw types, because at least some type checking will still be enforced.

Generic Exception Restriction

A generic class cannot extend **Throwable**. This means that you cannot create generic exception classes.

CHAPTER

15

Lambda Expressions

During Java's ongoing development and evolution, many features have been added since its original 1.0 release. However, two stand out because they have profoundly reshaped the language, fundamentally changing the way that code is written. The first was the addition of generics, added by JDK 5. (See Chapter 14.) The second is the *lambda expression*, which is the subject of this chapter.

Added by JDK 8, lambda expressions (and their related features) significantly enhanced Java because of two primary reasons. First, they added new syntax elements that increased the expressive power of the language. In the process, they streamlined the way that certain common constructs are implemented. Second, the addition of lambda expressions resulted in new capabilities being incorporated into the API library. Among these new capabilities are the ability to more easily take advantage of the parallel processing capabilities of multicore environments, especially as it relates to the handling of for-each style operations, and the new stream API, which supports pipeline operations on data. The addition of lambda expressions also provided the catalyst for other new Java features, including the default method (described in Chapter 9), which lets you define default behavior for an interface method, and the method reference (described here), which lets you refer to a method without executing it.

In the final analysis, in much the same way that generics reshaped Java several years ago, lambda expressions continue to reshape Java today. Simply put, lambda expressions will impact virtually all Java programmers. They truly are that important.

Introducing Lambda Expressions

Key to understanding Java's implementation of lambda expressions are two constructs. The first is the lambda expression, itself. The second is the functional interface. Let's begin with a simple definition of each.

A *lambda expression* is, essentially, an anonymous (that is, unnamed) method. However, this method is not executed on its own. Instead, it is used to implement a method defined by a functional interface. Thus, a lambda expression results in a form of anonymous class. Lambda expressions are also commonly referred to

as *closures*.

A *functional interface* is an interface that contains one and only one abstract method. Normally, this method specifies the intended purpose of the interface. Thus, a functional interface typically represents a single action. For example, the standard interface **Runnable** is a functional interface because it defines only one method: **run( )**. Therefore, **run( )** defines the action of **Runnable**. Furthermore, a functional interface defines the *target type* of a lambda expression. Here is a key point: a lambda expression can be used only in a context in which its target type is specified. One other thing: a functional interface is sometimes referred to as a *SAM type*, where SAM stands for Single Abstract Method.

NOTE A functional interface may specify any public method defined by **Object**, such as **equals( )**, without affecting its "functional interface" status. The public **Object** methods are considered implicit members of a functional interface because they are automatically implemented by an instance of a functional interface.

Let's now look more closely at both lambda expressions and functional interfaces.

Lambda Expression Fundamentals

The lambda expression introduced a new syntax element and operator into the Java language. The new operator, sometimes referred to as the *lambda operator* or the *arrow operator*, is `->`. It divides a lambda expression into two parts. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.) On the right side is the *lambda body*, which specifies the actions of the lambda expression. The `->` can be verbalized as "becomes" or "goes to."

Java defines two types of lambda bodies. One consists of a single expression, and the other type consists of a block of code. We will begin with lambdas that define a single expression. Lambdas with block bodies are discussed later in this chapter.

At this point, it will be helpful to look at a few examples of lambda expressions before continuing. Let's begin with what is probably the simplest type of lambda expression you can write. It evaluates to a constant value and is shown here:

```
() -> 123.45
```

This lambda expression takes no parameters, thus the parameter list is empty. It

returns the constant value 123.45. Therefore, it is similar to the following method:

```
double myMeth() { return 123.45; }
```

Of course, the method defined by a lambda expression does not have a name.

A slightly more interesting lambda expression is shown here:

```
() -> Math.random() * 100
```

This lambda expression obtains a pseudo-random value from **Math.random( )**, multiplies it by 100, and returns the result. It, too, does not require a parameter.

When a lambda expression requires a parameter, it is specified in the parameter list on the left side of the lambda operator. Here is a simple example:

```
(n) -> (n % 2)==0
```

This lambda expression returns **true** if the value of parameter **n** is even. Although it is possible to explicitly specify the type of a parameter, such as **n** in this case, often you won't need to do so because in many cases its type can be inferred. Like a named method, a lambda expression can specify as many parameters as needed.

Functional Interfaces

As stated, a functional interface is an interface that specifies only one abstract method. If you have been programming in Java for some time, you might at first think that all interface methods are implicitly abstract. Although this was true prior to JDK 8, the situation has changed. As explained in Chapter 9, beginning with JDK 8, it is possible to specify a default implementation for a method declared in an interface. Private and static interface methods also supply an implementation. As a result, today, an interface method is abstract only if it does not specify an implementation. Because non-default non-static, non-private interface methods are implicitly abstract, there is no need to use the **abstract** modifier (although you can specify it, if you like).

Here is an example of a functional interface:

```
interface MyNumber {
  double getValue();
}
```

In this case, the method **getValue( )** is implicitly abstract, and it is the only

method defined by **MyNumber**. Thus, **MyNumber** is a functional interface, and its function is defined by **getValue( )**.

As mentioned earlier, a lambda expression is not executed on its own. Rather, it forms the implementation of the abstract method defined by the functional interface that specifies its target type. As a result, a lambda expression can be specified only in a context in which a target type is defined. One of these contexts is created when a lambda expression is assigned to a functional interface reference. Other target type contexts include variable initialization, **return** statements, and method arguments, to name a few.

Let's work through an example that shows how a lambda expression can be used in an assignment context. First, a reference to the functional interface **MyNumber** is declared:

```
// Create a reference to a MyNumber instance.
MyNumber myNum;
```

Next, a lambda expression is assigned to that interface reference:

```
// Use a lambda in an assignment context.
myNum = () -> 123.45;
```

When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface. When that method is called through the target, the lambda expression is executed. Thus, a lambda expression gives us a way to transform a code segment into an object.

In the preceding example, the lambda expression becomes the implementation for the **getValue( )** method. As a result, the following displays the value 123.45:

```
// Call getValue(), which is implemented by the previously assigned
// lambda expression.
System.out.println(myNum.getValue());
```

Because the lambda expression assigned to **myNum** returns the value 123.45, that is the value obtained when **getValue( )** is called.

In order for a lambda expression to be used in a target type context, the type of the abstract method and the type of the lambda expression must be compatible. For example, if the abstract method specifies two **int** parameters, then the lambda must specify two parameters whose type either is explicitly **int**

or can be implicitly inferred as **int** by the context. In general, the type and number of the lambda expression's parameters must be compatible with the method's parameters; the return types must be compatible; and any exceptions thrown by the lambda expression must be acceptable to the method.

Some Lambda Expression Examples

With the preceding discussion in mind, let's look at some simple examples that illustrate the basic lambda expression concepts. The first example puts together the pieces shown in the foregoing section.

```
// Demonstrate a simple lambda expression.

// A functional interface.
interface MyNumber {
  double getValue();
}

class LambdaDemo {
  public static void main(String args[])
  {
    MyNumber myNum;  // declare an interface reference

    // Here, the lambda expression is simply a constant expression.
    // When it is assigned to myNum, a class instance is
    // constructed in which the lambda expression implements
    // the getValue() method in MyNumber.
    myNum = () -> 123.45;

    // Call getValue(), which is provided by the previously assigned
    // lambda expression.
    System.out.println("A fixed value: " + myNum.getValue());

    // Here, a more complex expression is used.
    myNum = () -> Math.random() * 100;


    // These call the lambda expression in the previous line.
    System.out.println("A random value: " + myNum.getValue());
    System.out.println("Another random value: " + myNum.getValue());

    // A lambda expression must be compatible with the method
    // defined by the functional interface. Therefore, this won't work:
//  myNum = () -> "123.03"; // Error!
  }
}
```

Sample output from the program is shown here:

```
A fixed value: 123.45
A random value: 88.90663650412304
Another random value: 53.00582701784129
```

As mentioned, the lambda expression must be compatible with the abstract method that it is intended to implement. For this reason, the commented-out line at the end of the preceding program is illegal because a value of type **String** is not compatible with **double**, which is the return type required by **getValue( )**.

The next example shows the use of a parameter with a lambda expression:

```
// Demonstrate a lambda expression that takes a parameter.

// Another functional interface.
interface NumericTest {
 boolean test(int n);
}

class LambdaDemo2 {
  public static void main(String args[])
  {
    // A lambda expression that tests if a number is even.
    NumericTest isEven = (n) -> (n % 2)==0;

    if(isEven.test(10)) System.out.println("10 is even");
    if(!isEven.test(9)) System.out.println("9 is not even");

    // Now, use a lambda expression that tests if a number
    // is non-negative.
    NumericTest isNonNeg = (n) -> n >= 0;

    if(isNonNeg.test(1)) System.out.println("1 is non-negative");
    if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
  }
}
```

The output from this program is shown here:

```
10 is even
9 is not even
1 is non-negative
-1 is negative
```

This program demonstrates a key fact about lambda expressions that warrants close examination. Pay special attention to the lambda expression that performs the test for evenness. It is shown again here:

```
(n) -> (n % 2)==0
```

Notice that the type of **n** is not specified. Rather, its type is inferred from the context. In this case, its type is inferred from the parameter type of **test( )** as defined by the **NumericTest** interface, which is **int**. It is also possible to explicitly specify the type of a parameter in a lambda expression. For example, this is also a valid way to write the preceding:

```
(int n) -> (n % 2)==0
```

Here, **n** is explicitly specified as **int**. Usually it is not necessary to explicitly specify the type, but you can in those situations that require it. Beginning with JDK 11, you can also use **var** to explicitly indicate local variable type inference for a lambda expression parameter.

This program demonstrates another important point about lambda expressions: A functional interface reference can be used to execute any lambda expression that is compatible with it. Notice that the program defines two different lambda expressions that are compatible with the **test( )** method of the

functional interface **NumericTest**. The first, called **isEven**, determines if a value is even. The second, called **isNonNeg**, checks if a value is non-negative. In each case, the value of the parameter **n** is tested. Because each lambda expression is compatible with **test( )**, each can be executed through a **NumericTest** reference.

One other point before moving on. When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side of the lambda operator. For example, this is also a valid way to write the lambda expression used in the program:

```
n -> (n % 2)==0
```

For consistency, this book will surround all lambda expression parameter lists with parentheses, even those containing only one parameter. Of course, you are free to adopt a different style.

The next program demonstrates a lambda expression that takes two parameters. In this case, the lambda expression tests if one number is a factor of another.

```
// Demonstrate a lambda expression that takes two parameters.

interface NumericTest2 {
  boolean test(int n, int d);
}

class LambdaDemo3 {
  public static void main(String args[])
  {
    // This lambda expression determines if one number is
    // a factor of another.
    NumericTest2 isFactor = (n, d) -> (n % d) == 0;

    if(isFactor.test(10, 2))
      System.out.println("2 is a factor of 10");

    if(!isFactor.test(10, 3))
      System.out.println("3 is not a factor of 10");
  }
}
```

The output is shown here:

```
2 is a factor of 10
3 is not a factor of 10
```

In this program, the functional interface **NumericTest2** defines the **test( )** method:

```
boolean test(int n, int d);
```

In this version, **test( )** specifies two parameters. Thus, for a lambda expression to be compatible with **test( )**, the lambda expression must also specify two

parameters. Notice how they are specified:

```
(n, d) -> (n % d) == 0
```

The two parameters, **n** and **d**, are specified in the parameter list, separated by commas. This example can be generalized. Whenever more than one parameter is required, the parameters are specified, separated by commas, in a parenthesized list on the left side of the lambda operator.

Here is an important point about multiple parameters in a lambda expression: If you need to explicitly declare the type of a parameter, then all of the parameters must have declared types. For example, this is legal:

```
(int n, int d) -> (n % d) == 0
```

But this is not:

```
(int n, d) -> (n % d) == 0
```

Block Lambda Expressions

The body of the lambdas shown in the preceding examples consist of a single expression. These types of lambda bodies are referred to as *expression bodies*, and lambdas that have expression bodies are sometimes called *expression lambdas.* In an expression body, the code on the right side of the lambda operator must consist of a single expression. While expression lambdas are quite useful, sometimes the situation will require more than a single expression. To handle such cases, Java supports a second type of lambda expression in which the code on the right side of the lambda operator consists of a block of code that can contain more than one statement. This type of lambda body is called a *block body.* Lambdas that have block bodies are sometimes referred to as *block lambdas*.

A block lambda expands the types of operations that can be handled within a lambda expression because it allows the body of the lambda to contain multiple statements. For example, in a block lambda you can declare variables, use loops, specify **if** and **switch** statements, create nested blocks, and so on. A block lambda is easy to create. Simply enclose the body within braces as you would any other block of statements.

Aside from allowing multiple statements, block lambdas are used much like the expression lambdas just discussed. One key difference, however, is that you

must explicitly use a **return** statement to return a value. This is necessary because a block lambda body does not represent a single expression.

Here is an example that uses a block lambda to compute and return the factorial of an **int** value:

```
// A block lambda that computes the factorial of an int value.

interface NumericFunc {
  int func(int n);
}

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

    // This block lambda computes the factorial of an int value.
    NumericFunc factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

The output is shown here:

```
The factorial of 3 is 6
The factorial of 5 is 120
```

In the program, notice that the block lambda declares a variable called **result**, uses a **for** loop, and has a **return** statement. These are legal inside a block lambda body. In essence, the block body of a lambda is similar to a method body. One other point. When a **return** statement occurs within a lambda expression, it simply causes a return from the lambda. It does not cause an enclosing method to return.

Another example of a block lambda is shown in the following program. It reverses the characters in a string.

```
// A block lambda that reverses the characters in a string.

interface StringFunc {
  String func(String n);
}

class BlockLambdaDemo2 {
  public static void main(String args[])
  {

    // This block lambda reverses the characters in a string.
    StringFunc reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " +
                       reverse.func("Lambda"));
    System.out.println("Expression reversed is " +
                       reverse.func("Expression"));
  }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
```

In this example, the functional interface **StringFunc** declares the **func( )** method. This method takes a parameter of type **String** and has a return type of **String**. Thus, in the **reverse** lambda expression, the type of **str** is inferred to be **String**. Notice that the **charAt( )** method is called on **str**. This is legal because

of the inference that **str** is of type **String**.

Generic Functional Interfaces

A lambda expression, itself, cannot specify type parameters. Thus, a lambda expression cannot be generic. (Of course, because of type inference, all lambda expressions exhibit some “generic-like” qualities.) However, the functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is determined, in part, by the type argument or arguments specified when a functional interface reference is declared.

To understand the value of generic functional interfaces, consider this. The two examples in the previous section used two different functional interfaces, one called **NumericFunc** and the other called **StringFunc**. However, both defined a method called **func( )** that took one parameter and returned a result. In the first case, the type of the parameter and return type was **int**. In the second case, the parameter and return type was **String**. Thus, the only difference between the two methods was the type of data they required. Instead of having two functional interfaces whose methods differ only in their data types, it is possible to declare one generic interface that can be used to handle both circumstances. The following program shows this approach:

```
// Use a generic functional interface with lambda expressions.

// A generic functional interface.
interface SomeFunc<T> {
  T func(T t);
}

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

    // Use a String-based version of SomeFunc.
    SomeFunc<String> reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    System.out.println("Lambda reversed is " +
                       reverse.func("Lambda"));
    System.out.println("Expression reversed is " +
                       reverse.func("Expression"));

    // Now, use an Integer-based version of SomeFunc.
    SomeFunc<Integer> factorial = (n) ->  {
      int result = 1;

      for(int i=1; i <= n; i++)
        result = i * result;

      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.func(3));
    System.out.println("The factoral of 5 is " + factorial.func(5));
  }
}
```

The output is shown here:

```
Lambda reversed is adbmaL
Expression reversed is noisserpxE
The factoral of 3 is 6
The factoral of 5 is 120
```

In the program, the generic functional interface **SomeFunc** is declared as shown here:

```
interface SomeFunc<T> {
  T func(T t);
}
```

Here, **T** specifies both the return type and the parameter type of **func( )**. This means that it is compatible with any lambda expression that takes one parameter and returns a value of the same type.

The **SomeFunc** interface is used to provide a reference to two different types of lambdas. The first uses type **String**. The second uses type **Integer**. Thus, the same functional interface can be used to refer to the **reverse** lambda and the **factorial** lambda. Only the type argument passed to **SomeFunc** differs.

Passing Lambda Expressions as Arguments

As explained earlier, a lambda expression can be used in any context that provides a target type. One of these is when a lambda expression is passed as an argument. In fact, passing a lambda expression as an argument is a common use of lambdas. Moreover, it is a very powerful use because it gives you a way to pass executable code as an argument to a method. This greatly enhances the expressive power of Java.

To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda. Although using a lambda expression as an argument is straightforward, it is still helpful to see it in action. The following program demonstrates the process:

```
// Use lambda expressions as an argument to a method.

interface StringFunc {
  String func(String n);
}

class LambdasAsArgumentsDemo {

  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed a reference to
  // any instance of that interface, including the instance created
  // by a lambda expression.
  // The second parameter specifies the string to operate on.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }
```

```
  public static void main(String args[])
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    System.out.println("Here is input string: " + inStr);

    // Here, a simple expression lambda that uppercases a string
    // is passed to stringOp( ).
    outStr = stringOp((str) -> str.toUpperCase(), inStr);
    System.out.println("The string in uppercase: " + outStr);

    // This passes a block lambda that removes spaces.
    outStr = stringOp((str) ->  {
                        String result = "";
                        int i;

                        for(i = 0; i < str.length(); i++)
                        if(str.charAt(i) != ' ')
                          result += str.charAt(i);

                        return result;
                      }, inStr);

    System.out.println("The string with spaces removed: " + outStr);


    // Of course, it is also possible to pass a StringFunc instance
    // created by an earlier lambda expression. For example,
    // after this declaration executes, reverse refers to an
    // instance of StringFunc.
    StringFunc reverse = (str) ->  {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
    };

    // Now, reverse can be passed as the first parameter to stringOp()
    // since it refers to a StringFunc object.
    System.out.println("The string reversed: " +
                        stringOp(reverse, inStr));
  }
}
```

The output is shown here:

```
Here is input string: Lambdas add power to Java
The string in uppercase: LAMBDAS ADD POWER TO JAVA
The string with spaces removed: LambdasaddpowertoJava
The string reversed: avaJ ot rewop dda sadbmaL
```

In the program, first notice the **stringOp( )** method. It has two parameters. The first is of type **StringFunc**, which is a functional interface. Thus, this parameter can receive a reference to any instance of **StringFunc**, including one created by a lambda expression. The second argument of **stringOp( )** is of type **String**, and this is the string operated on.

Next, notice the first call to **stringOp( )**, shown again here:

```
outStr = stringOp((str) -> str.toUpperCase(), inStr);
```

Here, a simple expression lambda is passed as an argument. When this occurs, an instance of the functional interface **StringFunc** is created and a reference to that object is passed to the first parameter of **stringOp( )**. Thus, the lambda code, embedded in a class instance, is passed to the method. The target type context is determined by the type of parameter. Because the lambda expression is compatible with that type, the call is valid. Embedding simple lambdas, such as the one just shown, inside a method call is often a convenient technique—especially when the lambda expression is intended for a single use.

Next, the program passes a block lambda to **stringOp( )**. This lambda removes spaces from a string. It is shown again here:

```
outStr = stringOp((str) ->  {
                    String result = "";
                    int i;

                    for(i = 0; i < str.length(); i++)
                    if(str.charAt(i) != ' ')
                      result += str.charAt(i);

                    return result;
                  }, inStr);
```

Although this uses a block lambda, the process of passing the lambda expression is the same as just described for the simple expression lambda. In this case,

however, some programmers will find the syntax a bit awkward.

When a block lambda seems overly long to embed in a method call, it is an easy matter to assign that lambda to a functional interface variable, as the previous examples have done. Then, you can simply pass that reference to the method. This technique is shown at the end of the program. There, a block lambda is defined that reverses a string. This lambda is assigned to **reverse**, which is a reference to a **StringFunc** instance. Thus, **reverse** can be used as an argument to the first parameter of **stringOp( )**. The program then calls **stringOp( )**, passing in **reverse** and the string on which to operate. Because the instance obtained by the evaluation of each lambda expression is an implementation of **StringFunc**, each can be used as the first parameter to **stringOp( )**.

One last point: In addition to variable initialization, assignment, and argument passing, the following also constitute target type contexts: casts, the **?** operator, array initializers, **return** statements, and lambda expressions, themselves.

Lambda Expressions and Exceptions

A lambda expression can throw an exception. However, it if throws a checked exception, then that exception must be compatible with the exception(s) listed in the **throws** clause of the abstract method in the functional interface. Here is an example that illustrates this fact. It computes the average of an array of **double** values. If a zero-length array is passed, however, it throws the custom exception **EmptyArrayException**. As the example shows, this exception is listed in the **throws** clause of **func( )** declared inside the **DoubleNumericArrayFunc** functional interface.

```
// Throw an exception from a lambda expression.

interface DoubleNumericArrayFunc {
  double func(double[] n) throws EmptyArrayException;
}

class EmptyArrayException extends Exception {
  EmptyArrayException() {
    super("Array Empty");
  }
}

class LambdaExceptionDemo {

  public static void main(String args[]) throws EmptyArrayException
  {
    double[] values  = { 1.0, 2.0, 3.0, 4.0 };

    // This block lambda computes the average of an array of doubles.
    DoubleNumericArrayFunc average = (n) ->  {
      double sum = 0;

      if(n.length == 0)
        throw new EmptyArrayException();

      for(int i=0; i < n.length; i++)
        sum += n[i];

      return sum / n.length;
    };

    System.out.println("The average is " + average.func(values));

    // This causes an exception to be thrown.
    System.out.println("The average is " + average.func(new double[0]));
  }
}
```

The first call to **average.func( )** returns the value 2.5. The second call, which passes a zero-length array, causes an **EmptyArrayException** to be thrown. Remember, the inclusion of the **throws** clause in **func( )** is necessary. Without it, the program will not compile because the lambda expression will no longer be compatible with **func( )**.

This example demonstrates another important point about lambda expressions. Notice that the parameter specified by **func( )** in the functional interface **DoubleNumericArrayFunc** is an array. However, the parameter to the lambda expression is simply **n**, rather than **n[ ]**. Remember, the type of a lambda expression parameter will be inferred from the target context. In this case, the target context is **double[ ]**, thus the type of **n** will be **double[ ]**. It is not necessary (or legal) to specify it as **n[ ]**. It would be legal to explicitly declare it as **double[ ] n**, but doing so gains nothing in this case.

Lambda Expressions and Variable Capture

Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression. For example, a lambda expression can use an instance or **static** variable defined by its enclosing class. A lambda expression also has access to **this** (both explicitly and implicitly), which refers to the invoking instance of the lambda expression's enclosing class. Thus, a lambda expression can obtain or set the value of an instance or **static** variable and call a method defined by its enclosing class.

However, when a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a *variable capture*. In this case, a lambda expression may only use local variables that are *effectively final*. An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as **final**, although doing so would not be an error. (The **this** parameter of an enclosing scope is automatically effectively final, and lambda expressions do not have a **this** of their own.)

It is important to understand that a local variable of the enclosing scope cannot be modified by the lambda expression. Doing so would remove its effectively final status, thus rendering it illegal for capture.

The following program illustrates the difference between effectively final and mutable local variables:

```
// An example of capturing a local variable from the enclosing scope.

interface MyFunc {
  int func(int n);
}

class VarCapture {
  public static void main(String args[])
  {
    // A local variable that can be captured.
    int num = 10;

    MyFunc myLambda = (n) -> {
      // This use of num is OK. It does not modify num.
      int v = num + n;

      // However, the following is illegal because it attempts
      // to modify the value of num.
//    num++;

      return v;
    };

    // The following line would also cause an error, because
    // it would remove the effectively final status from num.
//  num = 9;
  }
}
```

As the comments indicate, **num** is effectively final and can, therefore, be used inside **myLambda**. However, if **num** were to be modified, either inside the lambda or outside of it, **num** would lose its effectively final status. This would cause an error, and the program would not compile.

It is important to emphasize that a lambda expression can use and modify an instance variable from its invoking class. It just can't use a local variable of its enclosing scope unless that variable is effectively final.

Method References

There is an important feature related to lambda expressions called the *method*

reference. A method reference provides a way to refer to a method without executing it. It relates to lambda expressions because it, too, requires a target type context that consists of a compatible functional interface. When evaluated, a method reference also creates an instance of the functional interface.

There are different types of method references. We will begin with method references to **static** methods.

Method References to static Methods

To create a **static** method reference, use this general syntax:

ClassName::*methodName*

Notice that the class name is separated from the method name by a double colon. The **::** is a separator that was added to Java by JDK 8 expressly for this purpose. This method reference can be used anywhere in which it is compatible with its target type.

The following program demonstrates a **static** method reference:

```
// Demonstrate a method reference for a static method.

// A functional interface for string operations.
interface StringFunc {
  String func(String n);
}

// This class defines a static method called strReverse().
class MyStringOps {
  // A static method that reverses a string.
  static String strReverse(String str) {
```

```
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
  }
}

class MethodRefDemo {

  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed any instance
  // of that interface, including a method reference.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[])
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Here, a method reference to strReverse is passed to stringOp().
    outStr = stringOp(MyStringOps::strReverse, inStr);

    System.out.println("Original string: " + inStr);
    System.out.println("String reversed: " + outStr);
  }
}
```

The output is shown here:

```
Original string: Lambdas add power to Java
String reversed: avaJ ot rewop dda sadbmaL
```

In the program, pay special attention to this line:

```
outStr = stringOp(MyStringOps::strReverse, inStr);
```

Here, a reference to the **static** method **strReverse( )**, declared inside **MyStringOps**, is passed as the first argument to **stringOp( )**. This works

because **strReverse** is compatible with the **StringFunc** functional interface. Thus, the expression **MyStringOps::strReverse** evaluates to a reference to an object in which **strReverse** provides the implementation of **func( )** in **StringFunc**.

Method References to Instance Methods

To pass a reference to an instance method on a specific object, use this basic syntax:

objRef::*methodName*

As you can see, the syntax is similar to that used for a **static** method, except that an object reference is used instead of a class name. Here is the previous program rewritten to use an instance method reference:

```
// Demonstrate a method reference to an instance method

// A functional interface for string operations.
interface StringFunc {
  String func(String n);
}

// Now, this class defines an instance method called strReverse().
class MyStringOps {
  String strReverse(String str) {
      String result = "";
      int i;

      for(i = str.length()-1; i >= 0; i--)
        result += str.charAt(i);

      return result;
  }
}

class MethodRefDemo2 {

  // This method has a functional interface as the type of
  // its first parameter. Thus, it can be passed any instance
  // of that interface, including method references.
  static String stringOp(StringFunc sf, String s) {
    return sf.func(s);
  }

  public static void main(String args[])
  {
    String inStr = "Lambdas add power to Java";
    String outStr;

    // Create a MyStringOps object.
    MyStringOps strOps = new MyStringOps( );

    // Now, a method reference to the instance method strReverse
    // is passed to stringOp().
    outStr = stringOp(strOps::strReverse, inStr);

    System.out.println("Original string: " + inStr);
    System.out.println("String reversed: " + outStr);
  }
}
```

This program produces the same output as the previous version.

In the program, notice that **strReverse( )** is now an instance method of **MyStringOps**. Inside **main( )**, an instance of **MyStringOps** called **strOps** is created. This instance is used to create the method reference to **strReverse** in the call to **stringOp**, as shown again, here:

```
outStr = stringOp(strOps::strReverse, inStr);
```

In this example, **strReverse( )** is called on the **strOps** object.

It is also possible to handle a situation in which you want to specify an instance method that can be used with any object of a given class—not just a specified object. In this case, you will create a method reference as shown here:

ClassName::*instanceMethodName*

Here, the name of the class is used instead of a specific object, even though an instance method is specified. With this form, the first parameter of the functional interface matches the invoking object and the second parameter matches the parameter specified by the method. Here is an example. It defines a method called **counter( )** that counts the number of objects in an array that satisfy the condition defined by the **func( )** method of the **MyFunc** functional interface. In this case, it counts instances of the **HighTemp** class.

```
// Use an instance method reference with different objects.

// A functional interface that takes two reference arguments
// and returns a boolean result.
interface MyFunc<T> {
  boolean func(T v1, T v2);
}

// A class that stores the temperature high for a day.
class HighTemp {
  private int hTemp;

  HighTemp(int ht) { hTemp = ht; }

  // Return true if the invoking HighTemp object has the same
  // temperature as ht2.
  boolean sameTemp(HighTemp ht2) {
    return hTemp == ht2.hTemp;
  }

  // Return true if the invoking HighTemp object has a temperature
  // that is less than ht2.
  boolean lessThanTemp(HighTemp ht2) {
    return hTemp < ht2.hTemp;
  }
}

class InstanceMethWithObjectRefDemo {

  // A method that returns the number of occurrences
  // of an object for which some criteria, as specified by
  // the MyFunc parameter, is true.
  static <T> int counter(T[] vals, MyFunc<T> f, T v) {
```

```
    int count = 0;

    for(int i=0; i < vals.length; i++)
      if(f.func(vals[i], v)) count++;

    return count;
  }

  public static void main(String args[])
  {
    int count;

    // Create an array of HighTemp objects.
    HighTemp[] weekDayHighs = { new HighTemp(89), new HighTemp(82),
                                new HighTemp(90), new HighTemp(89),
                                new HighTemp(89), new HighTemp(91),
                                new HighTemp(84), new HighTemp(83) };

    // Use counter() with arrays of the class HighTemp.
    // Notice that a reference to the instance method
    // sameTemp() is passed as the second argument.
    count = counter(weekDayHighs, HighTemp::sameTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high of 89");

    // Now, create and use another array of HighTemp objects.
    HighTemp[] weekDayHighs2 = { new HighTemp(32), new HighTemp(12),
                                 new HighTemp(24), new HighTemp(19),
                                 new HighTemp(18), new HighTemp(12),
                                 new HighTemp(-1), new HighTemp(13) };

    count = counter(weekDayHighs2, HighTemp::sameTemp,
                    new HighTemp(12));
    System.out.println(count + " days had a high of 12");


    // Now, use lessThanTemp() to find days when temperature was less
    // than a specified value.
    count = counter(weekDayHighs, HighTemp::lessThanTemp,
                    new HighTemp(89));
    System.out.println(count + " days had a high less than 89");

    count = counter(weekDayHighs2, HighTemp::lessThanTemp,
                    new HighTemp(19));
    System.out.println(count + " days had a high of less than 19");
  }
}
```

The output is shown here:

```
3 days had a high of 89
2 days had a high of 12
3 days had a high less than 89
5 days had a high of less than 19
```

In the program, notice that **HighTemp** has two instance methods: **sameTemp( )** and **lessThanTemp( )**. The first returns **true** if two **HighTemp** objects contain the same temperature. The second returns **true** if the temperature of the invoking object is less than that of the passed object. Each method has a parameter of type **HighTemp** and each method returns a **boolean** result. Thus, each is compatible with the **MyFunc** functional interface because the invoking object type can be mapped to the first parameter of **func( )** and the argument mapped to **func( )**'s second parameter. Thus, when the expression

```
HighTemp::sameTemp
```

is passed to the **counter( )** method, an instance of the functional interface **MyFunc** is created in which the parameter type of the first parameter is that of the invoking object of the instance method, which is **HighTemp**. The type of the second parameter is also **HighTemp** because that is the type of the parameter to **sameTemp( )**. The same is true for the **lessThanTemp( )** method.

One other point: you can refer to the superclass version of a method by use of **super**, as shown here:

super::*name*

The name of the method is specified by *name*. Another form is

typeName.**super**::*name*

where *typeName* refers to an enclosing class or super interface.

Method References with Generics

You can use method references with generic classes and/or generic methods. For example, consider the following program:

```
// Demonstrate a method reference to a generic method
// declared inside a non-generic class.

// A functional interface that operates on an array
// and a value, and returns an int result.
interface MyFunc<T> {
  int func(T[] vals, T v);
}

// This class defines a method called countMatching() that
// returns the number of items in an array that are equal
// to a specified value. Notice that countMatching()
// is generic, but MyArrayOps is not.
class MyArrayOps {
  static <T> int countMatching(T[] vals, T v) {
    int count = 0;

    for(int i=0; i < vals.length; i++)
      if(vals[i] == v) count++;

      return count;
  }
}
```

```
class GenericMethodRefDemo {

  // This method has the MyFunc functional interface as the
  // type of its first parameter. The other two parameters
  // receive an array and a value, both of type T.
  static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
    return f.func(vals, v);
  }

  public static void main(String args[])
  {
    Integer[] vals = { 1, 2, 3, 4, 2, 3, 4, 4, 5 };
    String[] strs = { "One", "Two", "Three", "Two" };
    int count;

    count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
    System.out.println("vals contains " + count + " 4s");

    count = myOp(MyArrayOps::<String>countMatching, strs, "Two");
    System.out.println("strs contains " + count + " Twos");
  }
}
```

The output is shown here:

```
vals contains 3 4s
strs contains 2 Twos
```

In the program, **MyArrayOps** is a non-generic class that contains a generic method called **countMatching( )**. The method returns a count of the elements in an array that match a specified value. Notice how the generic type argument is specified. For example, its first call in **main( )**, shown here:

```
count = myOp(MyArrayOps::<Integer>countMatching, vals, 4);
```

passes the type argument **Integer**. Notice that it occurs after the **::**. This syntax can be generalized: When a generic method is specified as a method reference, its type argument comes after the **::** and before the method name. It is important to point out, however, that explicitly specifying the type argument is not required in this situation (and many others) because the type argument would have been

automatically inferred. In cases in which a generic class is specified, the type argument follows the class name and precedes the **::**.

Although the preceding examples show the mechanics of using method references, they don't show their real benefits. One place method references can be quite useful is in conjunction with the Collections Framework, which is described later in Chapter 19. However, for completeness, a short, but effective, example that uses a method reference to help determine the largest element in a collection is included here. (If you are unfamiliar with the Collections Framework, return to this example after you have worked through Chapter 19.)

One way to find the largest element in a collection is to use the **max( )** method defined by the **Collections** class. For the version of **max( )** used here, you must pass a reference to the collection and an instance of an object that implements the **Comparator<T>** interface. This interface specifies how two objects are compared. It defines only one abstract method, called **compare( )**, that takes two arguments, each the type of the objects being compared. It must return greater than zero if the first argument is greater than the second, zero if the two arguments are equal, and less than zero if the first object is less than the second.

In the past, to use **max( )** with user-defined objects, an instance of **Comparator<T>** had to be obtained by first explicitly implementing it by a class, and then creating an instance of that class. This instance was then passed as the comparator to **max( )**. Beginning with JDK 8, it is now possible to simply pass a reference to a comparison method to **max( )** because doing so automatically implements the comparator. The following simple example shows the process by creating an **ArrayList** of **MyClass** objects and then finding the one in the list that has the highest value (as defined by the comparison method).

```
// Use a method reference to help find the maximum value in a collection.
import java.util.*;

class MyClass {
  private int val;

  MyClass(int v) { val = v; }

  int getVal() { return val; }
}

class UseMethodRef {
  // A compare() method compatible with the one defined by Comparator<T>.
  static int compareMC(MyClass a, MyClass b) {
    return a.getVal() - b.getVal();
  }

  public static void main(String args[])
  {
    ArrayList<MyClass> al = new ArrayList<MyClass>();

    al.add(new MyClass(1));
    al.add(new MyClass(4));
    al.add(new MyClass(2));
    al.add(new MyClass(9));
    al.add(new MyClass(3));
    al.add(new MyClass(7));

    // Find the maximum value in al using the compareMC() method.
    MyClass maxValObj = Collections.max(al, UseMethodRef::compareMC);

    System.out.println("Maximum value is: " + maxValObj.getVal());
  }
}
```

The output is shown here:

```
Maximum value is: 9
```

In the program, notice that **MyClass** neither defines any comparison method of its own, nor does it implement **Comparator**. However, the maximum value of a list of **MyClass** items can still be obtained by calling **max( )** because **UseMethodRef** defines the static method **compareMC( )**, which is compatible

with the **compare( )** method defined by **Comparator**. Therefore, there is no need to explicitly implement and create an instance of **Comparator**.

Constructor References

Similar to the way that you can create references to methods, you can create references to constructors. Here is the general form of the syntax that you will use:

classname::new

This reference can be assigned to any functional interface reference that defines a method compatible with the constructor. Here is a simple example:

```
// Demonstrate a Constructor reference.

// MyFunc is a functional interface whose method returns
// a MyClass reference.
interface MyFunc {
   MyClass func(int n);
}

class MyClass {
  private int val;

  // This constructor takes an argument.
  MyClass(int v) { val = v; }

  // This is the default constructor.
  MyClass() { val = 0; }

  // ...

  int getVal() { return val; };
}

class ConstructorRefDemo {
  public static void main(String args[])
  {
    // Create a reference to the MyClass constructor.
    // Because func() in MyFunc takes an argument, new
    // refers to the parameterized constructor in MyClass,
    // not the default constructor.
    MyFunc myClassCons = MyClass::new;

    // Create an instance of MyClass via that constructor reference.
    MyClass mc = myClassCons.func(100);

    // Use the instance of MyClass just created.
    System.out.println("val in mc is " + mc.getVal( ));
  }
}
```

The output is shown here:

```
val in mc is 100
```

In the program, notice that the **func( )** method of **MyFunc** returns a reference of type **MyClass** and has an **int** parameter. Next, notice that **MyClass** defines two constructors. The first specifies a parameter of type **int**. The second is the default, parameterless constructor. Now, examine the following line:

```
MyFunc myClassCons = MyClass::new;
```

Here, the expression **MyClass::new** creates a constructor reference to a **MyClass** constructor. In this case, because **MyFunc**'s **func( )** method takes an **int** parameter, the constructor being referred to is **MyClass(int v)** because it is the one that matches. Also notice that the reference to this constructor is assigned to a **MyFunc** reference called **myClassCons**. After this statement executes, **myClassCons** can be used to create an instance of **MyClass**, as this line shows:

```
MyClass mc = myClassCons.func(100);
```

In essence, **myClassCons** has become another way to call **MyClass(int v)**.

Constructor references to generic classes are created in the same fashion. The only difference is that the type argument can be specified. This works the same as it does for using a generic class to create a method reference: simply specify the type argument after the class name. The following illustrates this by modifying the previous example so that **MyFunc** and **MyClass** are generic.

```
// Demonstrate a constructor reference with a generic class.

// MyFunc is now a generic functional interface.
interface MyFunc<T> {
   MyClass<T> func(T n);
}

class MyClass<T> {
  private T val;

  // A constructor that takes an argument.
  MyClass(T v) { val = v; }

  // This is the default constructor.
  MyClass( ) { val = null;  }

  // ...

  T getVal() { return val; };
}

class ConstructorRefDemo2 {

  public static void main(String args[])
  {
    // Create a reference to the MyClass<T> constructor.
    MyFunc<Integer> myClassCons = MyClass<Integer>::new;

    // Create an instance of MyClass<T> via that constructor reference.
    MyClass<Integer> mc = myClassCons.func(100);

    // Use the instance of MyClass<T> just created.
    System.out.println("val in mc is " + mc.getVal( ));
  }
}
```

This program produces the same output as the previous version. The difference is that now both **MyFunc** and **MyClass** are generic. Thus, the sequence that creates a constructor reference can include a type argument (although one is not always needed), as shown here:

```
MyFunc<Integer> myClassCons = MyClass<Integer>::new;
```

Because the type argument **Integer** has already been specified when **myClassCons** is created, it can be used to create a **MyClass<Integer>** object, as the next line shows:

```
MyClass<Integer> mc = myClassCons.func(100);
```

Although the preceding examples demonstrate the mechanics of using a constructor reference, no one would use a constructor reference as just shown because nothing is gained. Furthermore, having what amounts to two names for the same constructor creates a confusing situation (to say the least). However, to give you the flavor of a more practical usage, the following program uses a **static** method, called **myClassFactory( )**, that is a factory for objects of any type of **MyFunc** objects. It can be used to create any type of object that has a constructor compatible with its first parameter.

```
// Implement a simple class factory using a constructor reference.

interface MyFunc<R, T> {
   R func(T n);
}

// A simple generic class.
class MyClass<T> {
  private T val;

  // A constructor that takes an argument.
  MyClass(T v) { val = v; }
```

```
  // The default constructor. This constructor
  // is NOT used by this program.
  MyClass() { val = null; }
  // ...

  T getVal() { return val; };
}

// A simple, non-generic class.
class MyClass2 {
  String  str;

  // A constructor that takes an argument.
  MyClass2(String s) { str = s; }

  // The default constructor. This
  // constructor is NOT used by this program.
  MyClass2() { str = ""; }

  // ...

  String getVal() { return str; };
}

class ConstructorRefDemo3 {

  // A factory method for class objects. The class must
  // have a constructor that takes one parameter of type T.
  // R specifies the type of object being created.
  static <R,T> R myClassFactory(MyFunc<R, T> cons, T v) {
    return cons.func(v);
  }

  public static void main(String args[])
  {
    // Create a reference to a MyClass constructor.
    // In this case, new refers to the constructor that
    // takes an argument.
    MyFunc<MyClass<Double>, Double> myClassCons = MyClass<Double>::new;

    // Create an instance of MyClass by use of the factory method.
    MyClass<Double> mc = myClassFactory(myClassCons, 100.1);

    // Use the instance of MyClass just created.
    System.out.println("val in mc is " + mc.getVal( ));

    // Now, create a different class by use of myClassFactory().
    MyFunc<MyClass2, String> myClassCons2 = MyClass2::new;

    // Create an instance of MyClass2 by use of the factory method.
    MyClass2 mc2 = myClassFactory(myClassCons2, "Lambda");

    // Use the instance of MyClass just created.
    System.out.println("str in mc2 is " + mc2.getVal( ));
  }
}
```

The output is shown here:

```
val in mc is 100.1
str in mc2 is Lambda
```

As you can see, **myClassFactory( )** is used to create objects of type **MyClass<Double>** and **MyClass2**. Although both classes differ, for example **MyClass** is generic and **MyClass2** is not, both can be created by **myClassFactory( )** because they both have constructors that are compatible with **func( )** in **MyFunc**. This works because **myClassFactory( )** is passed the constructor for the object that it builds. You might want to experiment with this program a bit, trying different classes that you create. Also try creating instances of different types of **MyClass** objects. As you will see, **myClassFactory( )** can create any type of object whose class has a constructor that is compatible with **func( )** in **MyFunc**. Although this example is quite simple, it hints at the power that constructor references bring to Java.

Before moving on, it is important to mention a second form of the constructor reference syntax that is used for arrays. To create a constructor reference for an array, use this construct:

type[]::new

Here, *type* specifies the type of object being created. For example, assuming the form of **MyClass** as shown in the first constructor reference example (**ConstructorRefDemo**) and given the **MyArrayCreator** interface shown here:

```
interface MyArrayCreator<T> {
   T func(int n);
}
```

the following creates a two-element array of **MyClass** objects and gives each element an initial value:

```
MyArrayCreator<MyClass[]> mcArrayCons = MyClass[]::new;
MyClass[] a = mcArrayCons.func(2);
a[0] = new MyClass(1);
a[1] = new MyClass(2);
```

Here, the call to **func(2)** causes a two-element array to be created. In general, a functional interface must contain a method that takes a single **int** parameter if it

is to be used to refer to an array constructor.

Predefined Functional Interfaces

Up to this point, the examples in this chapter have defined their own functional interfaces so that the fundamental concepts behind lambda expressions and functional interfaces could be clearly illustrated. However, in many cases, you won't need to define your own functional interface because the package called **java.util.function** provides several predefined ones. Although we will look at them more closely in Part II, here is a sampling:

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type **T** and return the result, which is also of type **T**. Its method is called **apply**( ).
BinaryOperator<T>	Apply an operation to two objects of type **T** and return the result, which is also of type **T**. Its method is called **apply**( ).
Consumer<T>	Apply an operation on an object of type **T**. Its method is called **accept**( ).
Supplier<T>	Return an object of type **T**. Its method is called **get**( ).
Function<T, R>	Apply an operation to an object of type **T** and return the result as an object of type **R**. Its method is called **apply**( ).
Predicate<T>	Determine if an object of type **T** fulfills some constraint. Return a **boolean** value that indicates the outcome. Its method is called **test**( ).

The following program shows the **Function** interface in action by using it to rework the earlier example called **BlockLambdaDemo** that demonstrated block lambdas by implementing a factorial example. That example created its own functional interface called **NumericFunc**, but the built-in **Function** interface could have been used, as this version of the program illustrates:

```
// Use the Function built-in functional interface.

// Import the Function interface.
import java.util.function.Function;

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

    // This block lambda computes the factorial of an int value.
    // This time, Function is the functional interface.
    Function<Integer, Integer> factorial = (n) ->  {
      int result = 1;
      for(int i=1; i <= n; i++)
        result = i * result;
      return result;
    };

    System.out.println("The factoral of 3 is " + factorial.apply(3));
    System.out.println("The factoral of 5 is " + factorial.apply(5));
  }
}
```

It produces the same output as previous versions of the program.

CHAPTER

16

Modules

JDK 9 introduced a new and important feature called *modules*. Modules give you a way to describe the relationships and dependencies of the code that comprises an application. Modules also let you control which parts of a module are accessible to other modules and which are not. Through the use of modules you can create more reliable, scalable programs.

As a general rule, modules are most helpful to large applications because they help reduce the management complexity often associated with a large software system. However, small programs also benefit from modules because the Java API library has now been organized into modules. Thus, it is now possible to specify which parts of the API are required by your program and which are not. This makes it possible to deploy programs with a smaller run-time footprint, which is especially important when creating code for small devices, such as those intended to be part of the Internet of Things (IoT).

Support for modules is provided both by language elements, including several keywords, and by enhancements to **javac**, **java**, and other JDK tools. Furthermore, new tools and file formats were introduced. As a result, the JDK and the run-time system were substantially upgraded to support modules. In short, modules constitute a major addition to, and evolution of, the Java language.

Module Basics

In its most fundamental sense, a *module* is a grouping of packages and resources that can be collectively referred to by the module's name. A *module declaration* specifies the name of a module and defines the relationship a module and its packages have to other modules. Module declarations are program statements in a Java source file and are supported by several module-related keywords. They are shown here:

exports	module	open	opens
provides	requires	to	transitive
uses	with		

It is important to understand that these keywords are recognized *as keywords* only in the context of a module declaration. Otherwise, they are interpreted as identifiers in other situations. Thus, the keyword **module** could, for example, also be used as a parameter name, although such a use is certainly not recommended. However, making the module-related keywords context-sensitive prevents problems with pre-existing code that may use one or more of them as identifiers. Because they are context-sensitive, the module-related keywords are formally called *restricted keywords*.

A module declaration is contained in a file called **module-info.java**. Thus, a module is defined in a Java source file. This file is then compiled by **javac** into a class file and is known as its *module descriptor*. The **module-info.java** file must contain only a module definition. It cannot contain other types of declarations.

A module declaration begins with the keyword **module**. Here is its general form:

```
module moduleName {
   // module definition
}
```

The name of the module is specified by *moduleName*, which must be a valid Java identifier or a sequence of identifiers separated by periods. The module definition is specified within the braces. Although a module definition may be empty (which results in a declaration that simply names the module), typically it specifies one or more clauses that define the characteristics of the module.

A Simple Module Example

At the foundation of a module's capabilities are two key features. The first is a module's ability to specify that it requires another module. In other words, one module can specify that it *depends* on another. A dependence relationship is specified by use of a **requires** statement. By default, the presence of the required module is checked at both compile time and at run time. The second key feature is a module's ability to control which, if any, of its packages are accessible by another module. This is accomplished by use of the **exports** keyword. The

public and protected types within a package are accessible to other modules only if they are explicitly exported. Here we will develop an example that introduces both of these features.

The following example creates a modular application that demonstrates some simple mathematical functions. Although this application is purposely very small, it illustrates the core concepts and procedures required to create, compile, and run module-based code. Furthermore, the general approach shown here also applies to larger, real-world applications. It is strongly recommended that you work through the example on your computer, carefully following each step.

NOTE This chapter shows the process of creating, compiling, and running module-based code by use of the command-line tools. This approach has two advantages. First, it works for all Java programmers because no IDE is required. Second, it very clearly shows the fundamentals of the module system, including how it utilizes directories. To follow along, you will need to manually create a number of directories and ensure that each file is placed in its proper directory. As you might expect, when creating real-world, module-based applications you will likely find a module-aware IDE easier to use because, typically, it will automate much of the process. However, learning the fundamentals of modules using the command-line tools ensures that you have a solid understanding of the topic.

The application defines two modules. The first module is called **appstart**. It contains a package called **appstart.mymodappdemo** that defines the application's entry point in a class called **MyModAppDemo**. Thus, **MyModAppDemo** contains the application's **main( )** method. The second module is called **appfuncs**. It contains a package called **appfuncs.simplefuncs** that includes the class **SimpleMathFuncs**. This class defines three static methods that implement some simple mathematical functions. The entire application will be contained in a directory tree that begins at **mymodapp**.

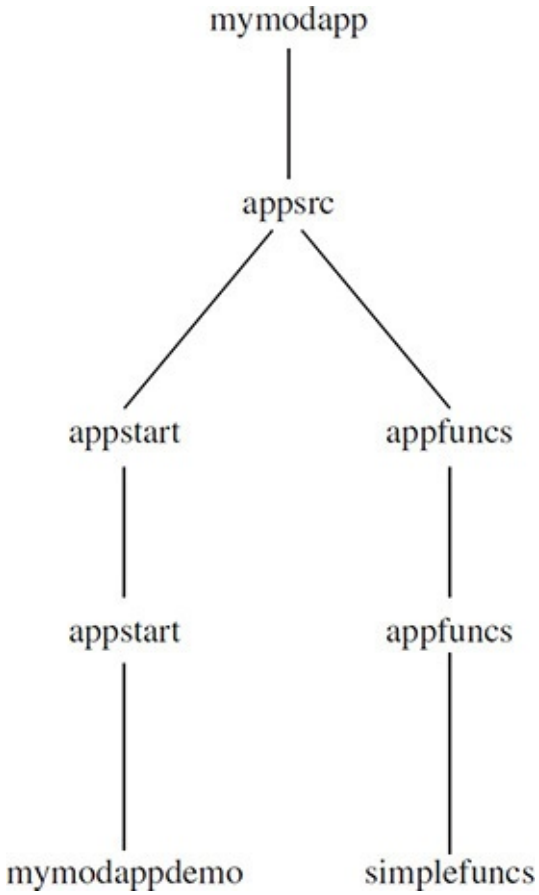
Before continuing, a few words about module names are appropriate. First, in the examples that follow, the name of a module (such as **appfuncs**) is the prefix of the name of a package (such as **appfuncs.simplefuncs**) that it contains. This is *not* required, but is used here as a way of clearly indicating to what module a package belongs. In general, when learning about and experimenting with modules, short, simple names, such as those used in this chapter, are helpful, and you can use any sort of convenient names that you like. However, when creating modules suitable for distribution, you must be careful with the names you choose because you will want those names to be unique. At the time of this writing, the suggested way to achieve this is to use the reverse domain name method. In this method, the reverse domain name of the domain that "owns" the project is used as a prefix for the module. For example, a project associated with **herbschildt.com** would use **com.herbschildt** as the module prefix. (The same goes for package names.) Because modules are a recent addition to Java, naming

conventions may evolve over time. You will want to check the Java documentation for current recommendations.

Let's now begin. Start by creating the necessary source code directories by following these steps:

1. Create a directory called **mymodapp**. This is the top-level directory for the entire application.
2. Under **mymodapp**, create a subdirectory called **appsrc**. This is the top-level directory for the application's source code.
3. Under **appsrc**, create the subdirectory **appstart**. Under this directory, create a subdirectory also called **appstart**. Under this directory, create the directory **mymodappdemo**. Thus, beginning with **appsrc**, you will have created this tree:
   ```
   appsrc\appstart\appstart\mymodappdemo
   ```
4. Also under **appsrc**, create the subdirectory **appfuncs**. Under this directory, create a subdirectory also called **appfuncs**. Under this directory, create the directory called **simplefuncs**. Thus, beginning with **appsrc**, you will have created this tree:
   ```
   appsrc\appfuncs\appfuncs\simplefuncs
   ```

Your directory tree should look like that shown here.



After you have set up these directories, you can create the application's source files.

This example will use four source files. Two are the source files that define the application. The first is **SimpleMathFuncs.java**, shown here. Notice that **SimpleMathFuncs** is packaged in **appfuncs.simplefuncs**.

```
// Some simple math functions.

package appfuncs.simplefuncs;

public class SimpleMathFuncs {

  // Determine if a is a factor of b.
  public static boolean isFactor(int a, int b) {
    if((b%a) == 0) return true;
    return false;
  }

  // Return the smallest positive factor that a and b have in common.
  public static int lcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = 2; i <= min/2; i++) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }
```

```
    return 1;
  }

  // Return the largest positive factor that a and b have in common.
  public static int gcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = min/2; i >= 2; i--) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }

    return 1;
  }
}
```

SimpleMathFuncs defines three simple **static** math functions. The first, **isFactor( )**, returns true if **a** is a factor of **b**. The **lcf( )** method returns the smallest factor common to both **a** and **b**. In other words, it returns the least common factor of **a** and **b**. The **gcf( )** method returns the greatest common factor of **a** and **b**. In both cases, 1 is returned if no common factors are found. This file must be put in the following directory:

```
appsrc\appfuncs\appfuncs\simplefuncs
```

This is the **appfuncs.simplefuncs** package directory.

The second source file is **MyModAppDemo.java**, shown next. It uses the methods in **SimpleMathFuncs**. Notice that it is packaged in **appstart.mymodappdemo**. Also note that it imports the **SimpleMathFuncs** class because it depends on **SimpleMathFuncs** for its operation.

```
// Demonstrate a simple module-based application.
package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;

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

    if(SimpleMathFuncs.isFactor(2, 10))
      System.out.println("2 is a factor of 10");

    System.out.println("Smallest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.lcf(35, 105));

    System.out.println("Largest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.gcf(35, 105));

  }
}
```

This file must be put in the following directory:

```
appsrc\appstart\appstart\mymodappdemo
```

This is the directory for the **appstart.mymodappdemo** package.

Next, you will need to add **module-info.java** files for each module. These files contain the module definitions. First, add this one, which defines the **appfuncs** module:

```
// Module definition for the functions module.
module appfuncs {
  // Exports the package appfuncs.simplefuncs.
  exports appfuncs.simplefuncs;
}
```

Notice that **appfuncs** exports the package **appfuncs.simplefuncs**, which makes it accessible to other modules. This file must be put into this directory:

```
appsrc\appfuncs
```

Thus, it goes in the **appfuncs** module directory, which is above the package directories.

Finally, the **module-info.java** file for the **appstart** module is shown next. Notice that **appstart** requires the module **appfuncs**.

```
// Module definition for the main application module.
module appstart {
  // Requires the module appfuncs.
  requires appfuncs;
}
```

This file must be put into its module directory:

```
appsrc\appstart
```

Before examining the **requires**, **exports**, and **module** statements more closely, let's first compile and run this example. Be sure that you have correctly created the directories and entered each file into its proper directory, as just explained.

Compile and Run the First Module Example

Beginning with JDK 9, **javac** has been updated to support modules. Thus, like all other Java programs, module-based programs are compiled using **javac**. The process is easy, with the primary difference being that you will usually explicitly specify a *module path*. A module path tells the compiler where the compiled files will be located. When following along with this example, be sure that you execute the **javac** commands from the **mymodapp** directory in order for the paths to be correct. Recall that **mymodapp** is the top-level directory for the entire module application.

To begin, compile **SimpleMathFuncs.java** using this command:

```
javac -d appmodules\appfuncs
  appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java
```

Remember, this command *must be* executed from the **mymodapp** directory. Notice the use of the **-d** option. This tells **javac** where to put the output **.class** file. For the examples in this chapter, the top of the directory tree for compiled code is **appmodules**. This command will create the output package directories for **appfuncs.simplefuncs** under **appmodules\appfuncs** as needed.

Next, here is the **javac** command that compiles the **module-info.java** file for the **appfuncs** module:

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java
```

This puts the **module-info.class** file into the **appmodules\appfuncs** directory.

Although the preceding two-step process works, it was shown primarily for

the sake of discussion. It is usually easier to compile a module's **module-info.java** file and its source files in one command line. Here, the preceding two **javac** commands are combined into one:

```
javac -d appmodules\appfuncs appsrc\appfuncs\module-info.java
  appsrc\appfuncs\appfuncs\simplefuncs\SimpleMathFuncs.java
```

In this case, each compiled file is put in its proper module or package directory.

Now, compile **module-info.java** and **MyModAppDemo.java** for the **appstart** module, using this command:

```
javac --module-path appmodules -d appmodules\appstart
  appsrc\appstart\module-info.java
  appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

Notice the **--module-path** option. It specifies the module path, which is the path on which the compiler will look for the user-defined modules required by the **module-info.java** file. In this case, it will look for the **appfuncs** module because it is needed by the **appstart** module. Also, notice that it specifies the output directory as **appmodules\appstart**. This means that the **module-info.class** file will be in the **appmodules\appstart** module directory and **MyModAppDemo.class** will be in the **appmodules\appstart\appstart\mymodappdemo** package directory.

Once you have completed the compilation, you can run the application with this **java**command:

```
java --module-path appmodules -m
appstart/appstart.mymodappdemo.MyModAppDemo
```

Here, the --**module-path** option specifies the path to the application's modules. As mentioned, **appmodules** is the directory at the top of the compiled modules tree. The **-m** option specifies the class that contains the entry point of the application and, in this case, the name of the class that contains the **main( )** method. When you run the program, you will see the following output:

```
2 is a factor of 10
Smallest factor common to both 35 and 105 is 5
Largest factor common to both 35 and 105 is 7
```

A Closer Look at requires and exports

The preceding module-based example relies on the two foundational features of the module system: the ability to specify a dependence and the ability to satisfy

that dependence. These capabilities are specified through the use of the **requires** and **exports** statements within a **module** declaration. Each merits a closer examination at this time.

Here is the form of the **requires** statement used in the example:

requires *moduleName*;

Here, *moduleName* specifies the name of a module that is required by the module in which the **requires** statement occurs. This means that the required module must be present in order for the current module to compile. In the language of modules, the current module is said to *read* the module specified in the **requires** statement. When more than one module is required, it must be specified in its own **requires** statement. Thus, a module declaration may include several different **requires** statements. In general, the **requires** statement gives you a way to ensure that your program has access to the modules that it needs.

Here is the general form of the **exports** statement used in the example:

exports *packageName*;

Here, *packageName* specifies the name of the package that is exported by the module inwhich this statement occurs. A module can export as many packages as needed, with eachone specified in a separate **exports** statement. Thus, a module may have several **exports** statements.

When a module exports a package, it makes all of the public and protected types in the package accessible to other modules. Furthermore, the public and protected members of those types are also accessible. However, if a package within a module is not exported, then it is private to that module, including all of its public types. For example, even though a class is declared as **public** within a package, if that package is not explicitly exported by an **exports** statement, then that class is not accessible to other modules. It is important to understand that the public and protected types of a package, whether exported or not, are always accessible within that package's module. The **exports** statement simply makes them accessible to outside modules. Thus, any nonexported package is only for the internal use of its module.

The key to understanding **requires** and **exports** is that they work together. If one module depends on another, then it must specify that dependence with **requires**. The module on which another depends must explicitly export (i.e., make accessible) the packages that the dependent module needs. If either side of this dependence relationship is missing, the dependent module will not compile. As it relates to the foregoing example, **MyModAppDemo** uses the functions in

SimpleMathFuncs. As a result, the **appstart** module declaration contains a **requires** statement that names the **appfuncs** module. The **appfuncs** module declaration exports the **appfuncs.simplefuncs** package, thus making the public types in the **SimpleMathFuncs** class available. Since both sides of the dependence relationship have been fulfilled, the application can compile and run. If either is missing, the compilation will fail.

It is important to emphasize that **requires** and **exports** statements must occur only within a **module** statement. Furthermore, a **module** statement must occur by itself in a file called **module-info.java**.

java.base and the Platform Modules

As mentioned at the start of this chapter, beginning with JDK 9 the Java API packages have been incorporated into modules. In fact, the modularization of the API is one of the primary benefits realized by the addition of the modules. Because of their special role, the API modules are referred to as *platform modules*, and their names all begin with the prefix **java**. Here are some examples: **java.base**, **java.desktop**, and **java.xml**. By modularizing the API, it becomes possible to deploy an application with only the packages that it requires, rather than the entire Java Runtime Environment (JRE). Because of the size of the full JRE, this is a very important improvement.

The fact that all of the Java API library packages are now in modules gives rise to the following question: How can the **main( )** method in **MyModAppDemo** in the preceding example use **System.out.println( )** without specifying a **requires** statement for the module that contains the **System** class? Obviously, the program will not compile and run unless **System** is present. The same question also applies to the use of the **Math** class in **SimpleMathFuncs**. The answer to this question is found in **java.base**.

Of the platform modules, the most important is **java.base**. It includes and exports those packages fundamental to Java, such as **java.lang**, **java.io**, and **java.util**, among many others. Because of its importance, **java.base** is *automatically accessible* to all modules. Furthermore, all other modules automatically require **java.base**. There is no need to include a **requires java.base** statement in a module declaration. (As a point of interest, it is not wrong to explicitly specify **java.base**, it’s just not necessary.) Thus, in much the same way that **java.lang** is automatically available to all programs without the use of an **import** statement, the **java.base** module is automatically accessible to all module-based programs without explicitly requesting it.

Because **java.base** contains the **java.lang** package, and **java.lang** contains the **System** class, **MyModAppDemo** in the preceding example can automatically use **System.out.println( )** without an explicit **requires** statement. The same applies to the use of the **Math** class in **SimpleMathFuncs**, because the **Math** class is also in **java.lang**. As you will see when you begin to create your own module-based applications, many of the API classes you will commonly need are in the packages included in **java.base**. Thus, the automatic inclusion of **java.base** simplifies the creation of module-based code because Java's core packages are automatically accessible.

One last point: Beginning with JDK 9, the documentation for the Java API now tells you the name of the module in which a package is contained. If the module is **java.base**, then you can use the contents of that package directly. Otherwise, your module declaration must include a **requires** clause for the desired module.

Legacy Code and the Unnamed Module

Another question may have occurred to you when working through the first example module program. Because Java now supports modules, and the API packages are also contained in modules, why do all of the other programs in the preceding chapters compile and run without error even though they do not use modules? More generally, since there is now over 20 years of Java code in existence and (at the time of this writing) the vast majority of that code does not use modules, how is it possible to compile, run, and maintain that legacy code with a JDK 9 or later compiler? Given Java's original philosophy of "write once, run everywhere," this is a very important question because backward capability must be maintained. As you will see, Java answers this question by providing an elegant, nearly transparent means of ensuring backward compatibility with pre-existing code.

Support for legacy code is provided by two key features. The first is the *unnamed module*. When you use code that is not part of a named module, it automatically becomes part of theunnamed module. The unnamed module has two important attributes. First, all of the packages in the unnamed module are automatically exported. Second, the unnamed module can access any and all other modules. Thus, when a program does not use modules, all API modules in the Java platform are automatically accessible through the unnamed module.

The second key feature that supports legacy code is the automatic use of the class path, rather than the module path. When you compile a program that does

not use modules, the class path mechanism is employed, just as it has been since Java's original release. As a result, the program is compiled and run in the same way it was prior to the advent of modules.

Because of the unnamed module and the automatic use of the class path, there was no need to declare any modules for the sample programs shown elsewhere in this book. They run properly whether you compile them with a modern compiler or an earlier one, such as JDK8. Thus, even though modules are a feature that has significant impact on Java, compatibility with legacy code is maintained. This approach also provides a smooth, nonintrusive, nondisruptive transition path to modules. Thus, it enables you to move a legacy application to modules at your own pace. Furthermore, it allows you to avoid the use of modules when they are not needed.

Before moving on, an important point needs to be made. For the types of example programs used elsewhere in this book, and for example programs in general, there is no benefit in using modules. Modularizing them would simply add clutter and complicate them for no reason or benefit. Furthermore, for many simple programs, there is no need to contain them in modules. For the reasons stated at the start of this chapter, modules are often of the greatest benefit when creating commercial programs. Therefore, no examples outside this chapter will use modules. This also allows the examples to be compiled and run in a pre–JDK 9 environment, which is important to readers using an older version of Java. Thus, except for the examples in this chapter, the examples in this book work for both pre-module and post-module JDKs.

Exporting to a Specific Module

The basic form of the **exports** statement makes a package accessible to any and all other modules. This is often exactly what you want. However, in some specialized development situations, it can be desirable to make a package accessible to only a *specific set* of modules, not *all* other modules. For example, a library developer might want to export a support package to certain other modules within the library, but not make it available for general use. Adding a **to** clause to the **exports** statement provides a means by which this can be accomplished.

In an **exports** statement, the **to** clause specifies a list of one or more modules that have access to the exported package. Furthermore, only those modules named in the **to** clause will have access. In the language of modules, the **to** clause creates what is known as a *qualified export*.

The form of **exports** that includes **to** is shown here:

exports *packageName* to *moduleNames*;

Here, *moduleNames* is a comma-separated list of modules to which the exporting module grants access.

You can try the **to** clause by changing the **module-info.java** file for the **appfuncs** module, as shown here:

```
// Module definition that uses a to clause.
module appfuncs {
  // Exports the package appfuncs.simplefuncs to appstart.
  exports appfuncs.simplefuncs to appstart;
}
```

Now, **simplefuncs** is exported only to **appstart** and to no other modules. After making this change, you can recompile the application by using this **javac** command:

```
javac -d appmodules --module-source-path appsrc
  appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

After compiling, you can run the application as shown earlier.

This example also uses another module-related feature. Look closely at the preceding **javac** command. First, notice that it specifies the **--module-source-path** option. The module source path specifies the top of the module source tree. The --**module-source-path** option automatically compiles the files in the tree under the specified directory, which is **appsrc** in this example. The --**module-source-path** option must be used with the **-d** option to ensure that the compiled modules are stored in their proper directories under **appmodules**. This form of **javac** is called *multi-module mode* because it enables more than one module to be compiled at a time. The multi-module compilation mode is especially helpful here because the **to** clause refers to a specific module, and the requiring module must have access to the exported package. Thus, in this case, both **appstart** and **appfuncs** are needed to avoid warnings and/or errors during compilation. Multi-module mode avoids this problem because both modules are being compiled at the same time.

The multi-module mode of **javac** has another advantage. It automatically finds and compiles all source files for the application, creating the necessary output directories. Because of the advantages that multi-module compilation mode offers, it will be used for the subsequent examples.

NOTE As a general rule, qualified export is a special case feature. Most often, your modules will either provide unqualified export of a package or not export the package at all, keeping it inaccessible. As such, qualified export is discussed here primarily for the sake of completeness. Also, qualified export by itself does not prevent the exported package from being misused by malicious code in a module that masquerades as the targeted module. The security techniques required to prevent this from happening are beyond the scope of this book. Consult the Oracle documentation for details on security in this regard, and Java security details in general.

Using requires transitive

Consider a situation in which there are three modules, A, B, and C, that have the following dependences:

- A requires B.
- B requires C.

Given this situation, it is clear that since A depends on B and B depends on C, A has an indirect dependence on C. As long as A does not directly use any of the contents of C, then you can simply have A require B in its module-info file, and have B export the packages required by A in its module-info file, as shown here:

```
// A's module-info file:
module A {
  requires B;
}

// B's module-info file.
module B {
  exports somepack;
  requires C;
}
```

Here, *somepack* is a placeholder for the package exported by B and used by A. Although this works as long as A does not need to use anything defined in C, a problem occurs if A *does* want to access a type in C. In this case, there are two solutions.

The first solution is to simply add a **requires C** statement to A's file, as shown here:

```
// A's module-info file updated to explicitly require C:
module A {
  requires B;
  requires C; // also require C
}
```

This solution certainly works, but if B will be used by many modules, you must add **requiresC** to all module definitions that require B. This is not only tedious, it is also error prone. Fortunately, there is a better solution. You can create an *implied dependence* on C. Implied dependence is also referred to as *implied readability*.

To create an implied dependence, add the **transitive** keyword after **requires** in the clause that requires the module upon which an implied readability is needed. In the case of this example, you would change B's module-info file as shown here:

```
// B's module-info file.
module B {
  exports somepack;
  requires transitive C;
}
```

Here, C is now required as transitive. After making this change, any module that depends on B will also, automatically, depend on C. Thus, A would automatically have access to C.

You can experiment with **requires transitive** by reworking the preceding modular application example so that the **isFactor( )** method is removed from the **SimpleMathFuncs** class in the **appfuncs.simplefuncs** package and put into a new class, module, and package. The new class will be called **SupportFuncs**, the module will be called **appsupport**, and the package will be called **appsupport.supportfuncs**. The **appfuncs** module will then add a dependence on the **appsupport** module by use of **requires transitive**. This will enable both the **appfuncs** and **appstart** modules to access it without **appstart** having to provide its own **requires** statement. This works because **appstart** receives access to it through an **appfuncs requires transitive** statement. The following describes the process in detail.

To begin, create the source directories that support the new **appsupport** module. First, create **appsupport** under the **appsrc** directory. This is the module directory for the support functions. Under **appsupport**, create the package directory by adding the **appsupport** subdirectory followed by the **supportfuncs**

subdirectory. Thus, the directory tree for **appsupport** should now look like this:

```
appsrc\appsupport\appsupport\supportfuncs
```

Once the directories have been established, create the **SupportFuncs** class. Notice that **SupportFuncs** is part of the **appsupport.supportfuncs** package. Therefore, you must put it in the **appsupport.supportfuncs** package directory.

```
// Support functions.

package appsupport.supportfuncs;

public class SupportFuncs {

  // Determine if a is a factor of b.
  public static boolean isFactor(int a, int b) {
    if((b%a) == 0) return true;
    return false;
  }
}
```

Notice that **isFactor( )** is now part of **SupportFuncs**, rather than **SimpleMathFuncs**.

Next, create the **module-info.java** file for the **appsupport** module and put it in **appsrc\appsupport** directory.

```
// Module definition for appsupport.
module appsupport {
  exports appsupport.supportfuncs;
}
```

As you can see, it exports the **appsupport.supportfuncs** package.

Because **isFactor( )** is now part of **Supportfuncs**, remove it from **SimpleMathFuncs**. Thus, **SimpleMathFuncs.java** will now look like this:

```
// Some simple math functions, with isFactor() removed.

package appfuncs.simplefuncs;
import appsupport.supportfuncs.SupportFuncs;
```

```
public class SimpleMathFuncs {

  // Return the smallest positive factor that a and b have in common.
  public static int lcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = 2; i <= min/2; i++) {
      if(SupportFuncs.isFactor(i, a) && SupportFuncs.isFactor(i, b))
        return i;
    }

    return 1;
  }

  // Return the largest positive factor that a and b have in common.
  public static int gcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = min/2; i >= 2; i--) {
      if(SupportFuncs.isFactor(i, a) && SupportFuncs.isFactor(i, b))
        return i;
    }

    return 1;
  }
}
```

Notice that now the **SupportFuncs** class is imported and calls to **isFactor( )** are referred to through the class name **SupportFuncs**.

Next, change the **module-info.java** file for **appfuncs** so that in its **requires** statement, **appsupport** is specified as **transitive**, as shown here:

```
// Module definition for appfuncs.
module appfuncs {
  // Exports the package appfuncs.simplefuncs.
  exports appfuncs.simplefuncs;

  // Requires appsupport and makes it transitive.
  requires transitive appsupport;
}
```

Because **appfuncs** requires **appsupport** as **transitive**, there is no need for the **module-info.java** file for **appstart** to also require it. Its dependence on **appsupport** is implied. Thus, no changes to the **module-info.java** file for **appstart** are needed.

Finally, update **MyModAppDemo.java** to reflect these changes. Specifically, it must now import the **SupportFuncs** class and specify it when invoking **isFactor( )**, as shown here:

```
// Updated to use SupportFuncs.
package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;
import appsupport.supportfuncs.SupportFuncs;

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

    // Now, isFactor() is referred to via SupportFuncs,
    // not SimpleMathFuncs.
    if(SupportFuncs.isFactor(2, 10))
      System.out.println("2 is a factor of 10");

    System.out.println("Smallest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.lcf(35, 105));

    System.out.println("Largest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.gcf(35, 105));

  }
}
```

Once you have completed all of the preceding steps, you can recompile the entire program using this multi-module compilation command:

```
javac -d appmodules --module-source-path appsrc
   appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java
```

As explained earlier, the multi-module compilation will automatically create the parallel module subdirectories, under the **appmodules** directory.

You can run the program using this command:

```
java --module-path appmodules -m
appstart/appstart.mymodappdemo.MyModAppDemo
```

It will produce the same output as the previous version. However, this time three different modules are required.

To prove that the **transitive** modifier is actually required by the application, remove it from the **module-info.java** file for **appfuncs**. Then, try to compile the program. As you will see, an error will result because **appsupport** is no longer accessible by **appstart**.

Here is another experiment. In the module-info file for **appsupport**, try exporting the**appsupport.supportfuncs** package to only **appfuncs** by use of a qualified export, as shown here:

```
exports appsupport.supportfuncs to appfuncs;
```

Next, attempt to compile the program. As you see, the program will not compile because now the support function **isFactor( )** is not available to **MyModAppDemo**, which is in the **appstart** module. As explained previously, a qualified export restricts access to a package to only those modules specified by the **to** clause.

One final point, because of a special exception in the Java language syntax, in a **requires** statement, if **transitive** is immediately followed by a separator (such as a semicolon), it is interpreted as an identifier (for example, as a module name) rather than a keyword.

Use Services

In programming, it is often useful to separate *what* must be done from *how* it is done. As you learned in Chapter 9, one way this is accomplished in Java is through the use of interfaces. The interface specifies the *what*, and the implementing class specifies the *how*. This concept can be expanded so that the implementing class is provided by code that is outside your program, through the use of a *plug-in*. Using such an approach, the capabilities of an application can be enhanced, upgraded, or altered by simply changing the plug-in. The core of

the application itself remains unchanged. One way that Java supports a pluggable application architecture is through the use of *services* and *service providers*. Because of their importance, especially in large, commercial applications, Java's module system provides support for them.

Before we begin, it is necessary to state that applications that use services and service providers are typically fairly sophisticated. Therefore, you may find that you do not often need the service-based module features. However, because support for services constitutes a rather significant part of the module system, it is important that you have a general understanding of how these features work. Also, a simple example is presented that illustrates the core techniques needed to use them.

Service and Service Provider Basics

In Java, a *service* is a program unit whose functionality is defined by an interface or abstract class. Thus, a service specifies in a general way some form of program activity. A concrete implementation of a service is supplied by a *service provider*. In other words, a service defines the form of some action, and the service provider supplies that action.

As mentioned, services are often used to support a pluggable architecture. For example, a service might be used to support the translation of one language into another. In this case, the service supports translation in general. The service provider supplies a specific translation, such as German to English or French to Chinese. Because all service providers implement the same interface, different translators can be used to translate different languages without having to change the core of the application. You can simply change the service provider.

Service providers are supported by the **ServiceLoader** class. **ServiceLoader** is a generic class packaged in **java.util**. It is declared like this:

```
class ServiceLoader<S>
```

Here, **S** specifies the service type. Service providers are loaded by the **load( )** method. It has several forms; the one we will use is shown here:

```
public static <S> ServiceLoader<S> load(Class <S> serviceType)
```

Here, *serviceType* specifies the **Class** object for the desired service type. Recall that **Class** is a class that encapsulates information about a class. There are a variety of ways to obtain a **Class** instance. The way we will use here involves a class literal. Recall that a class literal has this general form:

className.class

Here, *className* specifies the name of the class.

When **load( )** is called, it returns a **ServiceLoader** instance for the application. This object supports iteration and can be cycled through by use of a for-each **for** loop. Therefore, to find a specific provider, simply search for it using a loop.

The Service-Based Keywords

Modules support services through the use of the keywords **provides**, **uses**, and **with**. Essentially, a module specifies that it provides a service with a **provides** statement. A module indicates that it requires a service with a **uses** statement. The specific type of service provider is declared by **with**. When used together, they enable you to specify a module that provides a service, a module that needs that service, and the specific implementation of that service. Furthermore, the module system ensures that the service and service providers are available and will be found.

Here is the general form of **provides**:

provides *serviceType* with *implementationTypes*;

Here, *serviceType* specifies the type of the service, which is often an interface, although abstract classes are also used. A comma-separated list of the implementation types is specified by *implementationTypes*. Therefore, to provide a service, the module indicates both the name of the service and its implementation.

Here is the general form of the **uses** statement:

uses *serviceType*;

Here, *serviceType* specifies the type of the service required.

A Module-Based Service Example

To demonstrate the use of services we will add a service to the modular application example that we have been evolving. For simplicity, we will begin with the first version of the application shown at the start of this chapter. To it we will add two new modules. The first is called **userfuncs**. It will define interfaces that support functions that perform binary operations in which each argument is

an **int** and the result is an **int**. The second module is called **userfuncsimp**, and it contains concrete implementations of the interfaces.

Begin by creating the necessary source directories.

1. Under the **appsrc** directory add directories called **userfuncs** and **userfuncsimp**.
2. Under **userfuncs**, add the subdirectory also called **userfuncs**. Under that directory, add the subdirectory **binaryfuncs**. Thus, beginning with **appsrc**, you will have created this tree:
   `appsrc\userfuncs\userfuncs\binaryfuncs`
3. Under **userfuncsimp**, add the subdirectory also called **userfuncsimp**. Under that directory, add the subdirectory **binaryfuncsimp**. Thus, beginning with **appsrc**, you will have created this tree:

   `appsrc\userfuncsimp\userfuncsimp\binaryfuncsimp`

This example expands the original version of the application by providing support for functions beyond those built into the application. Recall that the **SimpleMathFuncs** class supplies three built-in functions: **isFactor( )**, **lcf( )**, and **gcf( )**. Although it would be possible to add more functions to this class, doing so requires modifying and recompiling the application. By implementing services, it becomes possible to "plug in" new functions at run time, without modifying the application, and that is what this example will do. In this case, the service supplies functions that take two **int** arguments and return an **int** result. Of course, other types of functions can be supported if additional interfaces are provided, but support for binary integer functions is sufficient for our purposes and keeps the source code size of the example manageable.

The Service Interfaces

Two service-related interfaces are needed. One specifies the form of an action, and the other specifies the form of the provider of that action. Both go in the **binaryfuncs** directory, and both are in the **userfuncs.binaryfuncs** package. The first, called **BinaryFunc**, declares the form of a binary function. It is shown here:

```
// This interface defines a function that takes two int
// arguments and returns an int result. Thus, it can
// describe any binary operation on two ints that
// returns an int.

package userfuncs.binaryfuncs;

public interface BinaryFunc {
  // Obtain the name of the function.
  public String getName();

  // This is the function to perform. It will be
  // provided by specific implementations.
  public int func(int a, int b);
}
```

BinaryFunc declares the form of an object that can implement a binary integer function. This is specified by the **func( )** method. The name of the function is obtainable from **getName( )**. The name will be used to determine what type of function is implemented. This interface is implemented by a class that supplies a binary function.

The second interface declares the form of the service provider. It is called **BinFuncProvider** and is shown here:

```
// This interface defines the form of a service provider that
// obtains BinaryFunc instances.
package userfuncs.binaryfuncs;
import userfuncs.binaryfuncs.BinaryFunc;

public interface BinFuncProvider {

  // Obtain a BinaryFunc.
  public BinaryFunc get();
}
```

BinFuncProvider declares only one method, **get( )**, which is used to obtain an instance of **BinaryFunc**. This interface must be implemented by a class that wants to provide instances of **BinaryFunc**.

The Implementation Classes

In this example, two concrete implementations of **BinaryFunc** are supported. The first is **AbsPlus**, which returns the sum of the absolute values of its arguments. The second is **AbsMinus**, which returns the result of subtracting the absolute value of the second argument from the absolute value of the first argument. These are provided by the classes **AbsPlusProvider** and **AbsMinusProvider**. The source code for these classes must be stored in the **binaryfuncsimp** directory, and they are all part of the **userfuncsimp.binaryfuncsimp** package.

The code for **AbsPlus** is shown here:

```
// AbsPlus provides a concrete implementation of
// BinaryFunc. It returns the result of abs(a) + abs(b).
package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsPlus implements BinaryFunc {

  // Return name of this function.
  public String getName() {
    return "absPlus";
  }

  // Implement the AbsPlus function.
  public int func(int a, int b) { return Math.abs(a) + Math.abs(b); }
}
```

AbsPlus implements **func( )** such that it returns the result of adding the absolute values of **a** and **b**. Notice that **getName( )** returns the "absPlus" string. It identifies this function.

The **AbsMinus** class is shown next:

```
// AbsMinus provides a concrete implementation of
// BinaryFunc. It returns the result of abs(a) - abs(b).

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.BinaryFunc;

public class AbsMinus implements BinaryFunc {
```

```
  // Return name of this function.
  public String getName() {
    return "absMinus";
  }

  // Implement the AbsMinus function.
  public int func(int a, int b) { return Math.abs(a) - Math.abs(b); }
}
```

Here, **func( )** is implemented to return the difference between the absolute values of **a** and **b**, and the string "absMinus" is returned by **getName( )**.

To obtain an instance of **AbsPlus**, the **AbsPlusProvider** is used. It implements **BinFuncProvider** and is shown here:

```
// This is a provider for the AbsPlus function.

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.*;

public class AbsPlusProvider implements BinFuncProvider {

  // Provide an AbsPlus object.
  public BinaryFunc get() { return new AbsPlus(); }
}
```

The **get( )** method simply returns a new **AbsPlus( )** object. Although this provider is very simple, it is important to point out that some service providers will be much more complex.

The provider for **AbsMinus** is called **AbsMinusProvider** and is shown next:

```
// This is a provider for the AbsMinus function.

package userfuncsimp.binaryfuncsimp;

import userfuncs.binaryfuncs.*;

public class AbsMinusProvider implements BinFuncProvider {

  // Provide an AbsMinus object.
  public BinaryFunc get() { return new AbsMinus(); }
}
```

Its **get( )** method returns an object of **AbsMinus**.

The Module Definition Files

Next, two module definition files are needed. The first is for the **userfuncs** module. It is shown here:

```
module userfuncs {
  exports userfuncs.binaryfuncs;
}
```

This code must be contained in a **module-info.java** file that is in the **userfuncs** module directory. Notice that it exports the **userfuncs.binaryfuncs** package. This is the package that defines the **BinaryFunc** and **BinFuncProvider** interfaces.

The second **module-info.java** file is shown next. It defines the module that contains the implementations. It goes in the **userfuncsimp** module directory.

```
module userfuncsimp {
  requires userfuncs;

  provides userfuncs.binaryfuncs.BinFuncProvider with
    userfuncsimp.binaryfuncsimp.AbsPlusProvider,
    userfuncsimp.binaryfuncsimp.AbsMinusProvider;
}
```

This module requires **userfuncs** because that is where **BinaryFunc** and **BinFuncProvider** are contained, and those interfaces are needed by the implementations. The module provides **BinFuncProvider** implementations with the classes **AbsPlusProvider** and **AbsMinusProvider**.

Demonstrate the Service Providers in MyModAppDemo

To demonstrate the use of the services, the **main( )** method of **MyModAppDemo** is expanded to use **AbsPlus** and **AbsMinus**. It does so by loading them at run time by use of **ServiceLoader.load( )**. Here is the updated code:

```
// A module-based application that demonstrates services
// and service providers.

package appstart.mymodappdemo;

import java.util.ServiceLoader;

import appfuncs.simplefuncs.SimpleMathFuncs;
import userfuncs.binaryfuncs.*;

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

    // First, use built-in functions as before.
    if(SimpleMathFuncs.isFactor(2, 10))
      System.out.println("2 is a factor of 10");

    System.out.println("Smallest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.lcf(35, 105));

    System.out.println("Largest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.gcf(35, 105));
```

```
    // Now, use service-based, user-defined operations.

    // Get a service loader for binary functions.
    ServiceLoader<BinFuncProvider> ldr =
      ServiceLoader.load(BinFuncProvider.class);

    BinaryFunc binOp = null;

    // Find the provider for absPlus and obtain the function.
    for(BinFuncProvider bfp : ldr) {
      if(bfp.get().getName().equals("absPlus")) {
        binOp = bfp.get();
        break;
      }
    }

    if(binOp != null)
      System.out.println("Result of absPlus function: " +
                         binOp.func(12, -4));
    else
      System.out.println("absPlus function not found");

    binOp = null;

    // Now, find the provider for absMinus and obtain the function.
    for(BinFuncProvider bfp : ldr) {
      if(bfp.get().getName().equals("absMinus")) {
        binOp = bfp.get();
        break;
      }
    }

    if(binOp != null)
      System.out.println("Result of absMinus function: " +
                         binOp.func(12, -4));
    else
      System.out.println("absMinus function not found");

  }
}
```

Let’s take a close look at how a service is loaded and executed by the preceding code. First, a service loader for services of type **BinFuncProvider** is created with this statement:

```
ServiceLoader<BinFuncProvider> ldr =
  ServiceLoader.load(BinFuncProvider.class);
```

Notice that the type parameter to **ServiceLoader** is **BinFuncProvider**. This is also the type used in the call to **load( )**. This means that providers that implement this interface will be found. Thus, after this statement executes, **BinFuncProvider** classes in the module will be available through **ldr**. In this case, both **AbsPlusProvider** and **AbsMinusProvider** will be available.

Next, a reference of type **BinaryFunc** called **binOp** is declared and initialized to **null**. It will be used to refer to an implementation that supplies a specific type of binary function. Next, the following loop searches **ldr** for one that has the "absPlus" name.

```
// Find the provider for absPlus and obtain the function.
for(BinFuncProvider bfp : ldr) {
  if(bfp.get().getName().equals("absPlus")) {
    binOp = bfp.get();
    break;
  }
}
```

Here, a for-each loop iterates through **ldr**. Inside the loop, the name of the function supplied by the provider is checked. If it matches "absPlus", that function is assigned to **binOp** by calling the provider’s **get( )** method.

Finally, if the function is found, as it will be in this example, it is executed by this statement:

```
if(binOp != null)
  System.out.println("Result of absPlus function: " +
                     binOp.func(12, -4));
```

In this case, because **binOp** refers to an instance of **AbsPlus**, the call to **func( )** performs an absolute value addition. A similar sequence is used to find and execute **AbsMinus**.

Because **MyModAppDemo** now uses **BinFuncProvider**, its module definition file must include a **uses** statement that specifies this fact. Recall that **MyModAppDemo** is in the **appstart** module. Therefore, you must change the **module-info.java** file for **appstart** as shown here:

```
// Module definition for the main application module.
// It now uses BinFuncProvider.
module appstart {
  // Requires the modules appfuncs and userfuncs.
  requires appfuncs;
  requires userfuncs;

  // appstart now uses BinFuncProvider.
  uses userfuncs.binaryfuncs.BinFuncProvider;
}
```

Compile and Run the Module-Based Service Example

Once you have performed all of the preceding steps, you can compile and run the example by executing the following commands:

```
javac  -d appmodules --module-source-path appsrc
  appsrc\userfuncsimp\module-info.java
  appsrc\appstart\appstart\mymodappdemo\MyModAppDemo.java

java --module-path appmodules -m appstart/appstart.mymodappdemo.MyModAppDemo
```

Here is the output:

```
2 is a factor of 10
Smallest factor common to both 35 and 105 is 5
Largest factor common to both 35 and 105 is 7
Result of absPlus function: 16
Result of absMinus function: 8
```

As the output shows, the binary functions were located and executed. It is important to emphasize that if either the **provides** statement in the **userfuncsimp** module or the **uses** statement in the **appstart** module were missing, the application would fail.

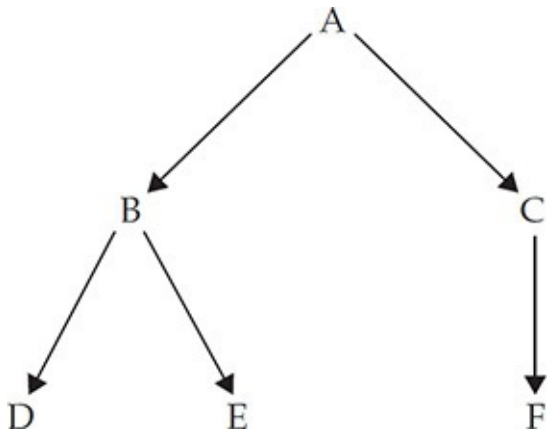
One last point: The preceding example was kept very simple in order to clearly illustrate module support for services, but much more sophisticated uses are possible. For example, you might use a service to provide a **sort( )** method that sorts a file. Various sorting algorithms could be supported and made available through the service. The specific sort could then be chosen based on the desired run-time characteristics, the nature and/or size of the data, and whether random access to the data is supported. You might want to try implementing such a service as a way to further experiment with services in

modules.

Module Graphs

A term you are likely to encounter when working with modules is *module graph*. During compilation, the compiler resolves the dependence relationships between modules by creating a module graph that represents the dependences. The process ensures that *all dependences* are resolved, including those that occur indirectly. For example, if module A requires module B, and B requires module C, then the module graph will contain module C even if A does not use it directly.

Module graphs can be depicted visually in a drawing to illustrate the relationship between modules. Here is a simple example. Assume six modules called **A**, **B**, **C**, **D**, **E**, and **F**. Further assume that **A** requires **B** and **C**, **B** requires **D** and **E**, and **C** requires **F**. The following visually depicts this relationship. (Because **java.base** is automatically included, it is not shown in the diagram.)



In Java, the arrows point from the dependent module to the required module. Thus, a drawing of a module graph depicts what modules have access to what other modules. Frankly, only the smallest applications can have their module graphs visually represented because of the complexity typically involved in many commercial applications.

Three Specialized Module Features

The preceding discussions have described the key features of modules supported by the Java language, and they are the features on which you will typically rely when creating your own modules. However, there are three additional module-

related features that can be quite useful in certain circumstances. These are the **open** module, the **opens** statement, and the use of **requires static**. Each of these features is designed to handle a specialized situation, and each constitutes a fairly advanced aspect of the module system. That said, it is important for all Java programmers to have a general understanding of their purpose.

Open Modules

As you learned earlier in this chapter, by default, the types in a module's packages are accessible only if they are explicitly exported via an **exports** statement. While this is usually what you will want, there can be circumstances in which it is useful to enable run-time access to all packages in the module, whether a package is exported or not. To allow this, you can create an *open module*. An open module is declared by preceding the **module** keyword with the **open** modifier, as shown here:

```
open module moduleName {
  // module definition
}
```

In an open module, types in all its packages are accessible at run time. Understand, however, that only those packages that are explicitly exported are available at compile time. Thus, the **open** modifier affects only run-time accessibility. The primary reason for an open module is to enable the packages in the module to be accessed through reflection. As explained in Chapter 12, reflection is the feature that lets a program analyze code at run time.

The opens Statement

It is possible for a module to open a specific package for run-time access by other modules and for reflective access rather than opening an entire module. To do so, use the **opens** statement, shown here:

```
opens packageName;
```

Here, *packageName* specifies the package to open. It is also possible to include a **to** clause, which names those modules for which the package is opened.

It is important to understand **opens** does not grant compile-time access. It is used only to open a package for run-time and reflective access. However, you can both export and open a module. One other point: an **opens** statement cannot

be used in an open module. Remember, all packages in an open module are already open.

requires static

As you know, **requires** specifies a dependence that, by default, is enforced both during compilation and at run time. However, it is possible to relax this requirement in such a way that a module is not required at run time. This is accomplished by use of the **static** modifier in a **requires** statement. For example, this specifies that **mymod** is required for compilation, but not at run time:

```
requires static mymod;
```

In this case, the addition of **static** makes **mymod** optional at run time. This can be helpful in a situation in which a program can utilize functionality if it is present, but not require it.

Introducing jlink and Module JAR Files

As the preceding discussions have shown, modules represent a substantial enhancement to the Java language. The module system also supports enhancements at run time. One of the most important is the ability to create a run-time image that is specifically tailored to your application. To accomplish this, JDK 9 added a new tool called **jlink**. It combines a group of modules into an optimized run-time image. You can use **jlink** to link modular JAR files, the new JMOD files, or even modules in their unarchived, "exploded directory" form.

Linking Files in an Exploded Directory

Let's look first at using **jlink** to create a run-time image from unarchived modules. That is, the files are contained in their raw form in a fully expanded (i.e., exploded) directory. Assuming a Windows environment, the following command links the modules for the first example in this chapter. It must be executed from a directory *directly above* **mymodapp**.

```
jlink --launcher MyModApp=appstart/appstart.mymodappdemo.MyModAppDemo
      --module-path "%JAVA_HOME%"\jmods;mymodapp\appmodules
      --add-modules appstart --output mylinkedmodapp
```

Let’s look closely at this command. First, the option **--launcher** tells **jlink** to create a command that starts the application. It specifies the name of the application and the path to the main class. In this case, the main class is **MyModAppDemo**. The **--module-path** option specifies the path to the required modules. The first is the path to the platform modules; the second is the path to the application modules. Notice the use of the environmental variable **JAVA_HOME**. It represents the path to the standard JDK directory. For example, in a standard Windows installation, the path will typically be something similar to **"C:\program files"\java\jdk-11\jmods**, but the use of **JAVA_HOME** is both shorter and able to work no matter in what directory the JDK was installed. The **--add-modules** option specifies the module or modules to add. Notice that only **appstart** is specified. This is because **jlink** automatically resolves all dependencies and includes all required modules. Finally, **--output** specifies the output directory.

After you run the preceding command, a directory called **mylinkedmodapp** will have been created that contains the run-time image. In its **bin** directory, you will find a launcher file called **MyModApp** that you can use to run the application. For example, in Windows, this will be a batch file that executes the program.

Linking Modular JAR Files

Although linking modules from their exploded directory is convenient, when working on real-world code you will often be using JAR files. (Recall that JAR stands for *J*ava *AR*chive. It is a file format typically used for application deployment.) In the case of modular code, you will be using *modular JAR files*. A modular JAR file is one that contains a **module-info.class** file. Beginning with JDK 9, the **jar** tool has the ability to create modular JAR files. For example, it can now recognize a module path. Once you have created modular JAR files, you can use **jlink** to link them into a run-time image. To understand the process, let’s work through an example. Again assuming the first example in this chapter, here are the **jar** commands that create modular JAR files for the **MyModAppDemo** program. Each must be executed from a directory directly above **mymodapp**. Also, you will need to create a directory called **applib** under **mymodapp**.

```
jar --create --file=mymodapp\applib\appfuncs.jar
     -C mymodapp\appmodules\appfuncs .

jar --create --file=mymodapp\applib\appstart.jar
    --main-class=appstart.mymodappdemo.MyModAppDemo
    -C mymodapp\appmodules\appstart .
```

Here, **--create** tells **jar** to create a new JAR file. The **--file** option specifies the name of the JAR file. The files to include are specified by the **-C** option. The class that contains **main( )** is specified by the **--main-class** option. After running these commands, the JAR files for the application will be in the **applib** directory under **mymodapp**.

Given the modular JAR files just created, here is the command that links them:

```
jlink --launcher MyModApp=appstart
      --module-path "%JAVA_HOME%"\jmods;mymodapp\applib
      --add-modules appstart --output mylinkedmodapp
```

Here, the module path to the JAR files is specified, not the path to the exploded directories. Otherwise, the **jlink** command is the same as before.

As a point of interest, you can use the following command to run the application directly from the JAR files. It must be executed from a directory directly above **mymodapp**.

```
java -p mymodapp\applib -m appstart
```

Here, **-p** specifies the module path and **-m** specifies the module that contains the program's entry point.

JMOD Files

The **jlink** tool can also link files that use the new JMOD format introduced by JDK 9. JMOD files can include things that are not applicable to a JAR file. They are created by the new **jmod** tool. Although most applications will still use module JAR files, JMOD files will be of value in specialized situations. As a point of interest, beginning with JDK 9 the platform modules are contained in JMOD files.

A Brief Word About Layers and Automatic

Modules

When learning about modules you are likely to encounter reference to two additional module-related features. These are *layers* and *automatic modules*. Both are designed for specialized, advanced work with modules or when migrating preexisting applications. Although it is likely that most programmers will not need to make use of these features, a brief description of each is given here in the interest of completeness.

A module layer associates the modules in a module graph with a class loader. Thus, different layers can use different class loaders. Layers enable certain specialized types of applications to be more easily constructed.

An automatic module is created by specifying a nonmodular JAR file on the module path, with its name being automatically derived. (It is also possible to explicitly specify a name for an automatic module in the manifest file.) Automatic modules enable normal modules to have a dependence on code in the automatic module. Automatic modules are provided as an aid in migration from pre-modular code to fully modular code. Thus, they are primarily a transitional feature.

Final Thoughts on Modules

The preceding discussions have introduced and demonstrated the core elements of Java's module system. These are the features about which every Java programmer should have at least a basic understanding. As you might guess, the module system provides additional features that give you fine-grained control over the creation and use of modules. For example, both **javac** and **java** have many more options related to modules than described in this chapter. Because modules are both a recent and significant addition to Java, it is likely that the module system will evolve over time. You will want to watch for enhancements to this innovative aspect of Java.

In conclusion, modules are expected to play an important role in Java programming. Although their use is not required at this time, they offer important benefits for commercial applications that no Java programmer can afford to ignore. It is likely that module-based development will be in every Java programmer's future.

PART

II

The Java Library

The Stream API

CHAPTER 30
Regular Expressions and Other Packages

CHAPTER

17

String Handling

A brief overview of Java's string handling was presented in Chapter 7. In this chapter, it is described in detail. As is the case in most other programming languages, in Java a string is a sequence of characters. But, unlike some other languages that implement strings as character arrays, Java implements strings as objects of type **String**.

Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. Also, **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.

Somewhat unexpectedly, when you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically. All are declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface.

One last point: To say that the strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

The String Constructors

The **String** class supports several constructors. To create an empty **String**, call the default constructor. For example,

```
String s = new String();
```

will create an instance of **String** with no characters in it.

Frequently, you will want to create strings that have initial values. The **String** class provides a variety of constructors to handle this. To create a **String** initialized by an array of characters, use the constructor shown here:

String(char *chars*[ ])

Here is an example:

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

This constructor initializes **s** with the string "abc".

You can specify a subrange of a character array as an initializer using the following constructor:

String(char *chars*[ ], int *startIndex*, int *numChars*)

Here, *startIndex* specifies the index at which the subrange begins, and *numChars* specifies the number of characters to use. Here is an example:

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

This initializes **s** with the characters **cde**.

You can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

String(String *strObj*)

Here, *strObj* is a **String** object. Consider this example:

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

    System.out.println(s1);
    System.out.println(s2);
  }
}
```

The output from this program is as follows:

```
Java
Java
```

As you can see, **s1** and **s2** contain the same string.

Even though Java's **char** type uses 16 bits to represent the basic Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array. Two forms are shown here:

String(byte *chrs*[ ])
String(byte *chrs*[ ], int *startIndex*, int *numChars*)

Here, *chrs* specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

```
// Construct string from subset of char array.
class SubStringCons {
  public static void main(String args[]) {
    byte ascii[] = {65, 66, 67, 68, 69, 70 };

    String s1 = new String(ascii);
    System.out.println(s1);

    String s2 = new String(ascii, 2, 3);
    System.out.println(s2);
  }
}
```

This program generates the following output:

```
ABCDEF
CDE
```

Extended versions of the byte-to-string constructors are also defined in which you can specify the character encoding that determines how bytes are converted to characters. However, you will often want to use the default encoding provided by the platform.

NOTE The contents of the array are copied whenever you create a **String** object from an array. If you modify the contents of the array after you have created the string, the **String** will be unchanged.

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

String(StringBuffer *strBufObj*)

You can construct a **String** from a **StringBuilder** by using this constructor:

String(StringBuilder *strBuildObj*)

The following constructor supports the extended Unicode character set:

String(int *codePoints*[ ], int *startIndex*, int *numChars*)

Here, *codePoints* is an array that contains Unicode code points. The resulting string is constructed from the range that begins at *startIndex* and runs for *numChars*.

There are also constructors that let you specify a **Charset**.

NOTE A discussion of Unicode code points and how they are handled by Java is found in Chapter 18.

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

```
int length( )
```

The following fragment prints "3", since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Special String Operations

Because strings are a common and important part of programming, Java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new **String** instances from string literals, concatenation of multiple **String** objects by use of the + operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but Java does them automatically as a convenience for the programmer and to add clarity.

String Literals

The earlier examples showed how to explicitly create a **String** instance from an array of characters by using the **new** operator. However, there is an easier way to do this using a string literal. For each string literal in your program, Java automatically constructs a **String** object. Thus, you can use a string literal to initialize a **String** object. For example, the following code fragment creates two equivalent strings:

```
char chars[] = { 'a', 'b', 'c' };
String s1 = new String(chars);

String s2 = "abc"; // use string literal
```

Because a **String** object is created for every string literal, you can use a string

literal any place you can use a **String** object. For example, you can call methods directly on a quoted string as if it were an object reference, as the following statement shows. It calls the **length( )** method on the string "abc". As expected, it prints "3".

```
System.out.println("abc".length());
```

String Concatenation

In general, Java does not allow operators to be applied to **String** objects. The one exception to this rule is the + operator, which concatenates two strings, producing a **String** object as the result. This allows you to chain together a series of + operations. For example, the following fragment concatenates three strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

This displays the string "He is 9 years old."

One practical use of string concatenation is found when you are creating very long strings. Instead of letting long strings wrap around within your source code, you can break them into smaller pieces, using the + to concatenate them. Here is an example:

```
// Using concatenation to prevent long lines.
class ConCat {
  public static void main(String args[]) {
    String longStr = "This could have been " +
      "a very long line that would have " +
      "wrapped around.  But string concatenation " +
      "prevents this.";

    System.out.println(longStr);
  }
}
```

String Concatenation with Other Data Types

You can concatenate strings with other types of data. For example, consider this slightly different version of the earlier example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

In this case, **age** is an **int** rather than another **String**, but the output produced is the same as before. This is because the **int** value in **age** is automatically converted into its string representation within a **String** object. This string is then concatenated as before. The compiler will convert an operand to its string equivalent whenever the other operand of the **+** is an instance of **String**.

Be careful when you mix other types of operations with string concatenation expressions, however. You might get surprising results. Consider the following:

```
String s = "four: " + 2 + 2;
System.out.println(s);
```

This fragment displays

```
four: 22
```

rather than the

```
four: 4
```

that you probably expected. Here's why. Operator precedence causes the concatenation of "four" with the string equivalent of 2 to take place first. This result is then concatenated with the string equivalent of 2 a second time. To complete the integer addition first, you must use parentheses, like this:

```
String s = "four: " + (2 + 2);
```

Now **s** contains the string "four: 4".

String Conversion and toString( )

One way to convert data into its string representation is by calling one of the overloaded versions of the string conversion method **valueOf( )** defined by **String**. **valueOf( )** is overloaded for all the primitive types and for type **Object**. For the primitive types, **valueOf( )** returns a string that contains the human-readable equivalent of the value with which it is called. For objects, **valueOf( )** calls the **toString( )** method on the object. We will look more closely at **valueOf( )** later in this chapter. Here, let's examine the **toString( )** method, because it is the means by which you can determine the string representation for objects of classes that you create.

Every class implements **toString( )** because it is defined by **Object**. However, the default implementation of **toString( )** is seldom sufficient. For most important classes that you create, you will want to override **toString( )** and provide your own string representations. Fortunately, this is easy to do. The **toString( )** method has this general form:

String toString( )

To implement **toString( )**, simply return a **String** object that contains the human-readable string that appropriately describes an object of your class.

By overriding **toString( )** for classes that you create, you allow them to be fully integrated into Java's programming environment. For example, they can be used in **print( )** and **println( )** statements and in concatenation expressions. The following program demonstrates this by overriding **toString( )** for the **Box** class:

```
// Override toString() for Box class.
class Box {
  double width;
  double height;
  double depth;

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

  public String toString() {
    return "Dimensions are " + width + " by " +
```

```
            depth + " by " + height + ".";
  }
}

class toStringDemo {
  public static void main(String args[]) {
    Box b = new Box(10, 12, 14);
    String s = "Box b: " + b; // concatenate Box object

    System.out.println(b); // convert Box to string
    System.out.println(s);
  }
}
```

The output of this program is shown here:

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

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

Character Extraction

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

charAt( )

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

char charAt(int *where*)

Here, *where* is the index of the character that you want to obtain. The value of *where* must be nonnegative and specify a location within the string. **charAt( )**

returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value **b** to **ch**.

getChars( )

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*. The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

The following program demonstrates **getChars( )**:

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

    s.getChars(start, end, buf, 0);
    System.out.println(buf);
  }
}
```

Here is the output of this program:

```
demo
```

getBytes( )

There is an alternative to **getChars( )** that stores the characters in an array of

bytes. This method is called **getBytes( )**, and it uses the default character-to-byte conversions provided by the platform. Here is its simplest form:

byte[ ] getBytes( )

Other forms of **getBytes( )** are also available. **getBytes( )** is most useful when you are exporting a **String** value into an environment that does not support 16-bit Unicode characters.

toCharArray( )

If you want to convert all the characters in a **String** object into a character array, the easiest way is to call **toCharArray( )**. It returns an array of characters for the entire string. It has this general form:

char[ ] toCharArray( )

This function is provided as a convenience, since it is possible to use **getChars( )** to achieve the same result.

String Comparison

The **String** class includes a number of methods that compare strings or substrings within strings. Several are examined here.

equals( ) and equalsIgnoreCase( )

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

boolean equals(Object *str*)

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

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

boolean equalsIgnoreCase(String *str*)

Here, *str* is the **String** object being compared with the invoking **String** object. It,

too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

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

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

The output from the program is shown here:

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

regionMatches( )

The **regionMatches( )** method compares a specific region inside a string with another specific region in another string. There is an overloaded form that allows you to ignore case in such comparisons. Here are the general forms for these two methods:

boolean regionMatches(int *startIndex*, String *str2*,
int *str2StartIndex*, int *numChars*)

boolean regionMatches(boolean *ignoreCase*,
int *startIndex*, String *str2*,
int *str2StartIndex*, int *numChars*)

For both versions, *startIndex* specifies the index at which the region begins within the invoking **String** object. The **String** being compared is specified by *str2*. The index at which the comparison will start within *str2* is specified by *str2StartIndex*. The length of the substring being compared is passed in *numChars*. In the second version, if *ignoreCase* is **true**, the case of the characters is ignored. Otherwise, case is significant.

startsWith( ) and endsWith( )

String defines two methods that are, more or less, specialized forms of **regionMatches( )**. The **startsWith( )** method determines whether a given **String** begins with a specified string. Conversely, **endsWith( )** determines whether the **String** in question ends with a specified string. They have the following general forms:

boolean startsWith(String *str*)
boolean endsWith(String *str*)

Here, *str* is the **String** being tested. If the string matches, **true** is returned. Otherwise, **false** is returned. For example,

```
"Foobar".endsWith("bar")
```

and

```
"Foobar".startsWith("Foo")
```

are both **true**.

A second form of **startsWith( )**, shown here, lets you specify a starting point:

boolean startsWith(String *str*, int *startIndex*)

Here, *startIndex* specifies the index into the invoking string at which point the search will begin. For example,

```
"Foobar".startsWith("bar", 3)
```

returns **true**.

equals( ) Versus ==

It is important to understand that the **equals( )** method and the == operator perform two different operations. As just explained, the **equals( )** method compares the characters inside a **String** object. The == operator compares two object references to see whether they refer to the same instance. The following program shows how two different **String** objects can contain the same characters, but references to these objects will not compare as equal:

```
// equals() vs ==
class EqualsNotEqualTo {
  public static void main(String args[]) {
    String s1 = "Hello";
    String s2 = new String(s1);

    System.out.println(s1 + " equals " + s2 + " -> " +
                       s1.equals(s2));
    System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
  }
}
```

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

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

compareTo( )

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

int compareTo(String *str*)

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

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

Here is a sample program that sorts an array of strings. The program uses **compareTo( )** to determine sort ordering for a bubble sort:

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

The output of this program is the list of words:

```
Now
aid
all
come
country
for
good
is
```

```
men
of
the
the
their
time
to
to
```

As you can see from the output of this example, **compareTo( )** takes into account uppercase and lowercase letters. The word "Now" came out before all the others because it begins with an uppercase letter, which means it has a lower value in the ASCII character set.

If you want to ignore case differences when comparing two strings, use **compareToIgnoreCase( )**, as shown here:

int compareToIgnoreCase(String *str*)

This method returns the same results as **compareTo( )**, except that case differences are ignored. You might want to try substituting it into the previous program. After doing so, "Now" will no longer be first.

Searching Strings

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

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

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

To search for the first occurrence of a character, use

int indexOf(int *ch*)

To search for the last occurrence of a character, use

int lastIndexOf(int *ch*)

Here, *ch* is the character being sought.

To search for the first or last occurrence of a substring, use

int indexOf(String *str*)
int lastIndexOf(String *str*)

Here, *str* specifies the substring.

You can specify a starting point for the search using these forms:

int indexOf(int *ch*, int *startIndex*)
int lastIndexOf(int *ch*, int *startIndex*)

int indexOf(String *str*, int *startIndex*)
int lastIndexOf(String *str*, int *startIndex*)

Here, *startIndex* specifies the index at which point the search begins. For **indexOf( )**, the search runs from *startIndex* to the end of the string. For **lastIndexOf( )**, the search runs from *startIndex* to zero.

The following example shows how to use the various index methods to search inside of a **String**:

```
// Demonstrate indexOf() and lastIndexOf().
class indexOfDemo {
  public static void main(String args[]) {
    String s = "Now is the time for all good men " +
               "to come to the aid of their country.";

    System.out.println(s);
    System.out.println("indexOf(t) = " +
                       s.indexOf('t'));
    System.out.println("lastIndexOf(t) = " +
                       s.lastIndexOf('t'));
    System.out.println("indexOf(the) = " +
                       s.indexOf("the"));
    System.out.println("lastIndexOf(the) = " +
                       s.lastIndexOf("the"));
    System.out.println("indexOf(t, 10) = " +
                       s.indexOf('t', 10));
    System.out.println("lastIndexOf(t, 60) = " +
                       s.lastIndexOf('t', 60));
    System.out.println("indexOf(the, 10) = " +
                       s.indexOf("the", 10));
    System.out.println("lastIndexOf(the, 60) = " +
                       s.lastIndexOf("the", 60));
  }
}
```

Here is the output of this program:

```
Now is the time for all good men to come to the aid of their country.
indexOf(t) = 7
lastIndexOf(t) = 65
indexOf(the) = 7
lastIndexOf(the) = 55
indexOf(t, 10) = 11
lastIndexOf(t, 60) = 55
indexOf(the, 10) = 44
lastIndexOf(the, 60) = 55
```

Modifying a String

Because **String** objects are immutable, whenever you want to modify a **String**, you must either copy it into a **StringBuffer** or **StringBuilder**, or use a **String** method that constructs a new copy of the string with your modifications complete. A sampling of these methods are described here.

substring( )

You can extract a substring using **substring( )**. It has two forms. The first is

String substring(int *startIndex*)

Here, *startIndex* specifies the index at which the substring will begin. This form returns a copy of the substring that begins at *startIndex* and runs to the end of the invoking string.

The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:

String substring(int *startIndex*, int *endIndex*)

Here, *startIndex* specifies the beginning index, and *endIndex* specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

The following program uses **substring( )** to replace all instances of one substring with another within a string:

```
// Substring replacement.
class StringReplace {
  public static void main(String args[]) {
    String org = "This is a test. This is, too.";
    String search = "is";
    String sub = "was";
    String result = "";
    int i;

    do { // replace all matching substrings
      System.out.println(org);
      i = org.indexOf(search);
```

```
      if(i != -1) {
        result = org.substring(0, i);
        result = result + sub;
        result = result + org.substring(i + search.length());
        org = result;
      }
    } while(i != -1);
  }
}
```

The output from this program is shown here:

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

concat( )

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

String concat(String *str*)

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

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

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

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

replace( )

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

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

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

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

puts the string "Hewwo" into **s**.

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

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

trim( ) and strip( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing spaces have been removed. As it relates to this method, spaces consist of those characters with a value of 32 or less. The **trim( )** method has this general form:

String trim( )

Here is an example:

```
String s = "   Hello World    ".trim();
```

This puts the string "Hello World" into **s**.

The **trim( )** method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state’s capital. It uses **trim( )** to remove any leading or trailing spaces that may have inadvertently been entered by the user.

```
// Using trim() to process commands.
import java.io.*;

class UseTrim {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String str;

    System.out.println("Enter 'stop' to quit.");
    System.out.println("Enter State: ");
    do {
      str = br.readLine();
      str = str.trim(); // remove whitespace

      if(str.equals("Illinois"))
        System.out.println("Capital is Springfield.");
      else if(str.equals("Missouri"))
        System.out.println("Capital is Jefferson City.");
      else if(str.equals("California"))
        System.out.println("Capital is Sacramento.");
      else if(str.equals("Washington"))
        System.out.println("Capital is Olympia.");
      // ...
    } while(!str.equals("stop"));
  }
}
```

Beginning with JDK 11, Java provides the method **strip( )**,which removes all whitespace characters (as defined by Java) from the beginning and end of the invoking string and returns the result. Such whitespace characters include, among others, spaces, tabs, carriage returns, and line feeds. It has this general form:

String strip( )

Also provided by JDK 11 are the methods **stripLeading( )** and **stripTrailing( )**,

which delete whitespace characters from the start or end, respectively, of the invoking string and return the result.

Data Conversion Using valueOf( )

The **valueOf( )** method converts data from its internal format into a human-readable form. It is a static method that is overloaded within **String** for all of Java's built-in types so that each type can be converted properly into a string. **valueOf( )** is also overloaded for type **Object**, so an object of any class type you create can also be used as an argument. (Recall that **Object** is a superclass for all classes.) Here are a few of its forms:

static String valueOf(double *num*)
static String valueOf(long *num*)
static String valueOf(Object *ob*)
static String valueOf(char *chars*[ ])

As discussed earlier, **valueOf( )** can be called when a string representation of some other type of data is needed. You can call this method directly with any data type and get a reasonable **String** representation. All of the simple types are converted to their common **String** representation. Any object that you pass to **valueOf( )** will return the result of a call to the object's **toString( )** method. In fact, you could just call **toString( )** directly and get the same result.

For most arrays, **valueOf( )** returns a rather cryptic string, which indicates that it is an array of some type. For arrays of **char**, however, a **String** object is created that contains the characters in the **char** array. There is a special version of **valueOf( )** that allows you to specify a subset of a **char** array. It has this general form:

static String valueOf(char *chars*[ ], int *startIndex*, int *numChars*)

Here, *chars* is the array that holds the characters, *startIndex* is the index into the array of characters at which the desired substring begins, and *numChars* specifies the length of the substring.

Changing the Case of Characters Within a String

The method **toLowerCase( )** converts all the characters in a string from uppercase to lowercase. The **toUpperCase( )** method converts all the characters in a string from lowercase to uppercase. Nonalphabetical characters, such as digits, are unaffected. Here are the simplest forms of these methods:

String toLowerCase( )
String toUpperCase( )

Both methods return a **String** object that contains the uppercase or lowercase equivalent of the invoking **String**. The default locale governs the conversion in both cases.

Here is an example that uses **toLowerCase( )** and **toUpperCase( )**:

```
// Demonstrate toUpperCase() and toLowerCase().
class ChangeCase {
  public static void main(String args[])
  {
    String s = "This is a test.";

    System.out.println("Original: " + s);

    String upper = s.toUpperCase();
    String lower = s.toLowerCase();

    System.out.println("Uppercase: " + upper);
    System.out.println("Lowercase: " + lower);
  }
}
```

The output produced by the program is shown here:

```
Original: This is a test.
Uppercase: THIS IS A TEST.
Lowercase: this is a test.
```

One other point: Overloaded versions of **toLowerCase( )** and **toUpperCase( )** that let you specify a **Locale** object to govern the conversion are also supplied. Specifying the locale can be quite important in some cases and can help internationalize your application.

Joining Strings

JDK 8 added a new method to **String** called **join()**. It is used to concatenate two or more strings, separating each string with a delimiter, such as a space or a comma. It has two forms. Its first is shown here:

static String join(CharSequence *delim*, CharSequence . . . *strs*)

Here, *delim* specifies the delimiter used to separate the character sequences specified by *strs*. Because **String** implements the **CharSequence** interface, *strs* can be a list of strings. (See Chapter 18 for information on **CharSequence**.) The following program demonstrates this version of **join()**:

```
// Demonstrate the join() method defined by String.
class StringJoinDemo {
  public static void main(String args[]) {

    String result = String.join(" ", "Alpha", "Beta", "Gamma");
    System.out.println(result);

    result = String.join(", ", "John", "ID#: 569",
                         "E-mail: John@HerbSchildt.com");
    System.out.println(result);
  }
}
```

The output is shown here:

```
Alpha Beta Gamma
John, ID#: 569, E-mail: John@HerbSchildt.com
```

In the first call to **join()**, a space is inserted between each string. In the second call, the delimiter is a comma followed by a space. This illustrates that the delimiter need not be just a single character.

The second form of **join()** lets you join a list of strings obtained from an object that implements the **Iterable** interface. **Iterable** is implemented by the Collections Framework classes described in Chapter 19, among others. See Chapter 18 for information on **Iterable**.

Additional String Methods

In addition to those methods discussed earlier, **String** has many other methods. Several are summarized in the following table:

Method	Description
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
boolean contains(CharSequence *str*)	Returns **true** if the invoking object contains the string specified by *str*. Returns **false** otherwise.
boolean contentEquals(CharSequence *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
boolean contentEquals(StringBuffer *str*)	Returns **true** if the invoking string contains the same string as *str*. Otherwise, returns **false**.
static String format(String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. (See Chapter 19 for details on formatting.)
static String format(Locale *loc*, String *fmtstr*, Object ... *args*)	Returns a string formatted as specified by *fmtstr*. Formatting is governed by the locale specified by *loc*. (See Chapter 19 for details on formatting.)
boolean isEmpty( )	Returns **true** if the invoking string contains no characters and has a length of zero.
Stream<String> lines( )	Decomposes a string into individual lines based on carriage return and line feed characters, and returns a **Stream** containing the lines. (Added by JDK 11.)
boolean matches(string *regExp*)	Returns **true** if the invoking string matches the regular expression passed in *regExp*. Otherwise, returns **false**.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.

String replaceFirst(String *regExp*, String *newStr*)	Returns a string in which the first substring that matches the regular expression specified by *regExp* is replaced by *newStr*.
String replaceAll(String *regExp*, String *newStr*)	Returns a string in which all substrings that match the regular expression specified by *regExp* are replaced by *newStr*.
String[ ] split(String *regExp*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*.
String[ ] split(String *regExp*, int *max*)	Decomposes the invoking string into parts and returns an array that contains the result. Each part is delimited by the regular expression passed in *regExp*. The number of pieces is specified by *max*. If *max* is negative, then the invoking string is fully decomposed. Otherwise, if *max* contains a nonzero value, the last entry in the returned array contains the remainder of the invoking string. If *max* is zero, the invoking string is fully decomposed, but no trailing empty strings will be included.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **String**.

Notice that several of these methods work with regular expressions. Regular expressions are described in Chapter 30.

StringBuffer

StringBuffer supports a modifiable string. As you know, **String** represents fixed-length, immutable character sequences. In contrast, **StringBuffer** represents growable and writable character sequences. **StringBuffer** may have characters and substrings inserted in the middle or appended to the end. **StringBuffer** will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.

StringBuffer Constructors

StringBuffer defines these four constructors:

```
StringBuffer( )
```

StringBuffer(int *size*)
StringBuffer(String *str*)
StringBuffer(CharSequence *chars*)

The default constructor (the one with no parameters) reserves room for 16 characters without reallocation. The second version accepts an integer argument that explicitly sets the size of the buffer. The third version accepts a **String** argument that sets the initial contents of the **StringBuffer** object and reserves room for 16 more characters without reallocation. **StringBuffer** allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly process in terms of time. Also, frequent reallocations can fragment memory. By allocating room for a few extra characters, **StringBuffer** reduces the number of reallocations that take place. The fourth constructor creates an object that contains the character sequence contained in *chars* and reserves room for 16 more characters.

length( ) and capacity( )

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

int length( )
int capacity( )

Here is an example:

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

    System.out.println("buffer = " + sb);
    System.out.println("length = " + sb.length());
    System.out.println("capacity = " + sb.capacity());
  }
}
```

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

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

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

ensureCapacity( )

If you want to preallocate room for a certain number of characters after a **StringBuffer** has been constructed, you can use **ensureCapacity( )** to set the size of the buffer. This is useful if you know in advance that you will be appending a large number of small strings to a **StringBuffer**. **ensureCapacity( )** has this general form:

void ensureCapacity(int *minCapacity*)

Here, *minCapacity* specifies the minimum size of the buffer. (A buffer larger than *minCapacity* may be allocated for reasons of efficiency.)

setLength( )

To set the length of the string within a **StringBuffer** object, use **setLength( )**. Its general form is shown here:

void setLength(int *len*)

Here, *len* specifies the length of the string. This value must be nonnegative.

When you increase the size of the string, null characters are added to the end. If you call **setLength( )** with a value less than the current value returned by **length( )**, then the characters stored beyond the new length will be lost. The **setCharAtDemo** sample program in the following section uses **setLength( )** to shorten a **StringBuffer**.

charAt( ) and setCharAt( )

The value of a single character can be obtained from a **StringBuffer** via the **charAt( )** method. You can set the value of a character within a **StringBuffer** using **setCharAt( )**. Their general forms are shown here:

char charAt(int *where*)
void setCharAt(int *where*, char *ch*)

For **charAt( )**, *where* specifies the index of the character being obtained. For **setCharAt( )**, *where* specifies the index of the character being set, and *ch* specifies the new value of that character. For both methods, *where* must be nonnegative and must not specify a location beyond the end of the string.

The following example demonstrates **charAt( )** and **setCharAt( )**:

```
// Demonstrate charAt() and setCharAt().
class setCharAtDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("Hello");
    System.out.println("buffer before = " + sb);
    System.out.println("charAt(1) before = " + sb.charAt(1));

    sb.setCharAt(1, 'i');
    sb.setLength(2);
    System.out.println("buffer after = " + sb);
    System.out.println("charAt(1) after = " + sb.charAt(1));
  }
}
```

Here is the output generated by this program:

```
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
```

getChars( )

To copy a substring of a **StringBuffer** into an array, use the **getChars( )** method. It has this general form:

void getChars(int *sourceStart*, int *sourceEnd*, char *target*[ ], int *targetStart*)

Here, *sourceStart* specifies the index of the beginning of the substring, and *sourceEnd* specifies an index that is one past the end of the desired substring. This means that the substring contains the characters from *sourceStart* through *sourceEnd*–1. The array that will receive the characters is specified by *target*.

The index within *target* at which the substring will be copied is passed in *targetStart*. Care must be taken to assure that the *target* array is large enough to hold the number of characters in the specified substring.

append( )

The **append( )** method concatenates the string representation of any other type of data to the end of the invoking **StringBuffer** object. It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String *str*)
StringBuffer append(int *num*)
StringBuffer append(Object *obj*)

First, the string representation of each parameter is obtained. Then, the result is appended to the current **StringBuffer** object. The buffer itself is returned by each version of **append( )**. This allows subsequent calls to be chained together, as shown in the following example:

```
// Demonstrate append().
class appendDemo {
  public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);

    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
  }
}
```

The output of this example is shown here:

```
a = 42!
```

insert( )

The **insert( )** method inserts one string into another. It is overloaded to accept values of all the primitive types, plus **String**s, **Object**s, and **CharSequence**s. Like **append( )**, it obtains the string representation of the value it is called with.

This string is then inserted into the invoking **StringBuffer** object. These are a few of its forms:

StringBuffer insert(int *index*, String *str*)
StringBuffer insert(int *index*, char *ch*)
StringBuffer insert(int *index*, Object *obj*)

Here, *index* specifies the index at which point the string will be inserted into the invoking **StringBuffer** object.

The following sample program inserts "like" between "I" and "Java":

```
// Demonstrate insert().
class insertDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("I Java!");

    sb.insert(2, "like ");
    System.out.println(sb);
  }
}
```

The output of this example is shown here:

```
I like Java!
```

reverse( )

You can reverse the characters within a **StringBuffer** object using **reverse( )**, shown here:

StringBuffer reverse( )

This method returns the reverse of the object on which it was called. The following program demonstrates **reverse( )**:

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
  public static void main(String args[]) {
    StringBuffer s = new StringBuffer("abcdef");

    System.out.println(s);
    s.reverse();
    System.out.println(s);
  }
}
```

Here is the output produced by the program:

```
abcdef
fedcba
```

delete( ) and deleteCharAt( )

You can delete characters within a **StringBuffer** by using the methods **delete( )** and **deleteCharAt( )**. These methods are shown here:

StringBuffer delete(int *startIndex*, int *endIndex*)
StringBuffer deleteCharAt(int *loc*)

The **delete( )** method deletes a sequence of characters from the invoking object. Here, *startIndex* specifies the index of the first character to remove, and *endIndex* specifies an index one past the last character to remove. Thus, the substring deleted runs from *startIndex* to *endIndex*–1. The resulting **StringBuffer** object is returned.

The **deleteCharAt( )** method deletes the character at the index specified by *loc*. It returns the resulting **StringBuffer** object.

Here is a program that demonstrates the **delete( )** and **deleteCharAt( )** methods:

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.delete(4, 7);
    System.out.println("After delete: " + sb);

    sb.deleteCharAt(0);
    System.out.println("After deleteCharAt: " + sb);
  }
}
```

The following output is produced:

```
After delete: This a test.
After deleteCharAt: his a test.
```

replace( )

You can replace one set of characters with another set inside a **StringBuffer** object by calling **replace( )**. Its signature is shown here:

StringBuffer replace(int *startIndex*, int *endIndex*, String *str*)

The substring being replaced is specified by the indexes *startIndex* and *endIndex*. Thus, the substring at *startIndex* through *endIndex*–1 is replaced. The replacement string is passed in *str*. The resulting **StringBuffer** object is returned.

The following program demonstrates **replace( )**:

```
// Demonstrate replace()
class replaceDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("This is a test.");

    sb.replace(5, 7, "was");
    System.out.println("After replace: " + sb);
  }
}
```

Here is the output:

```
After replace: This was a test.
```

substring( )

You can obtain a portion of a **StringBuffer** by calling **substring( )**. It has the following two forms:

String substring(int *startIndex*)
String substring(int *startIndex*, int *endIndex*)

The first form returns the substring that starts at *startIndex* and runs to the end of the invoking **StringBuffer** object. The second form returns the substring that starts at *startIndex* and runs through *endIndex*–1. These methods work just like those defined for **String** that were described earlier.

Additional StringBuffer Methods

In addition to those methods just described, **StringBuffer** supplies others. Several are summarized in the following table:

Method	Description
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int offsetByCodePoints(int *start*, int *num*)	Returns the index within the invoking string that is *num* code points beyond the starting index specified by *start*.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is implemented by **StringBuffer**.
void trimToSize( )	Requests that the size of the character buffer for the invoking object be reduced to better fit the current contents.

The following program demonstrates **indexOf( )** and **lastIndexOf( )**:

```
class IndexOfDemo {
  public static void main(String args[]) {
    StringBuffer sb = new StringBuffer("one two one");
    int i;

    i = sb.indexOf("one");
    System.out.println("First index: " + i);

    i = sb.lastIndexOf("one");
    System.out.println("Last index: " + i);
  }
}
```

The output is shown here:

```
First index: 0
Last index: 8
```

StringBuilder

StringBuilder is similar to **StringBuffer** except for one important difference: it is not synchronized, which means that it is not thread-safe. The advantage of **StringBuilder** is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use **StringBuffer** rather than **StringBuilder**.

CHAPTER

18

Exploring java.lang

This chapter discusses classes and interfaces defined by **java.lang**. As you know, **java.lang** is automatically imported into all programs. It contains classes and interfaces that are fundamental to virtually all of Java programming. It is Java's most widely used package. Beginning with JDK 9, all of **java.lang** is part of the **java.base** module.

java.lang includes the following classes:

Boolean	Float	Process	StrictMath
Byte	InheritableThreadLocal	ProcessBuilder	String
Character	Integer	ProcessBuilder.Redirect	StringBuffer
Character.Subset	Long	Runtime	StringBuilder
Character.UnicodeBlock	Math	RuntimePermission	System
Class	Module	Runtime.Version	System.LoggerFinder
ClassLoader	ModuleLayer	SecurityManager	Thread
ClassValue	ModuleLayer.Controller	Short	ThreadGroup
Compiler	Number	StackFramePermission	ThreadLocal
Double	Object	StackTraceElement	Throwable
Enum	Package	StackWalker	Void

java.lang defines the following interfaces:

Appendable	Iterable	StackWalker.StackFrame
AutoCloseable	ProcessHandle	System.Logger
CharSequence	ProcessHandle.Info	Thread.UncaughtExceptionHandler
Cloneable	Readable	
Comparable	Runnable	

Several of the classes contained in **java.lang** contain deprecated methods, many dating back to Java 1.0. Deprecated methods are still provided by Java to support legacy code but are not recommended for new code. Because of this, in

most cases the deprecated methods are not discussed here.

Primitive Type Wrappers

As mentioned in Part I of this book, Java uses primitive types, such as **int** and **char**, for performance reasons. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. Also, there is no way for two methods to refer to the *same instance* of an **int**. At times, you will need to create an object representation for one of these primitive types. For example, there are collection classes discussed in Chapter 19 that deal only with objects; to store a primitive type in one of these classes, you need to wrap the primitive type in a class. To address this need, Java provides classes that correspond to each of the primitive types. In essence, these classes encapsulate, or *wrap*, the primitive types within a class. Thus, they are commonly referred to as *type wrappers*. The type wrappers were introduced in Chapter 12. They are examined in detail here.

Number

The abstract class **Number** defines a superclass that is implemented by the classes that wrap the numeric types **byte**, **short**, **int**, **long**, **float**, and **double**. **Number** has abstract methods that return the value of the object in each of the different number formats. For example, **doubleValue( )** returns the value as a **double**, **floatValue( )** returns the value as a **float**, and so on. These methods are shown here:

```
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )
```

The values returned by these methods might be rounded, truncated, or result in a "garbage" value due to the effects of a narrowing conversion.

Number has concrete subclasses that hold explicit values of each primitive numeric type: **Double**, **Float**, **Byte**, **Short**, **Integer**, and **Long**.

Double and Float

Double and **Float** are wrappers for floating-point values of type **double** and **float**, respectively. The constructors for **Float** are shown here:

Float(double *num*)
Float(float *num*)
Float(String *str*) throws NumberFormatException

As you can see, **Float** objects can be constructed with values of type **float** or **double**. They can also be constructed from the string representation of a floating-point number. Beginning with JDK 9, these constructors have been deprecated. The **valueOf( )** method is the recommended alternative.

The constructors for **Double** are shown here:

Double(double *num*)
Double(String *str*) throws NumberFormatException

Double objects can be constructed with a **double** value or a string containing a floating-point value. Beginning with JDK 9, these constructors have been deprecated. The **valueOf( )** method is the recommended alternative.

The methods defined by **Float** include those shown in Table 18-1. The methods defined by **Double** include those shown in Table 18-2. Both **Float** and **Double** define the following constants:

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(float *num1*, float *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Float *f*)	Compares the numerical value of the invoking object with that of *f*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *FloatObj*)	Returns **true** if the invoking **Float** object is equivalent to *FloatObj*. Otherwise, it returns **false**.
static int floatToIntBits(float *num*)	Returns the IEEE-compatible, single-precision bit pattern that corresponds to *num*.
static int floatToRawIntBits(float *num*)	Returns the IEEE-compatible single-precision bit pattern that corresponds to *num*. A NaN value is preserved.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(float *num*)	Returns the hash code for *num*.
static float intBitsToFloat(int *num*)	Returns **float** equivalent of the IEEE-compatible, single-precision bit pattern specified by *num*.
int intValue( )	Returns the value of the invoking object as an **int**.
static boolean isFinite(float *num*)	Returns **true** if *num* is not **NaN** and is not infinite.
boolean isInfinite( )	Returns **true** if the invoking object contains an infinite value. Otherwise, it returns **false**.

static boolean isInfinite(float *num*)	Returns **true** if *num* specifies an infinite value. Otherwise, it returns **false**.
boolean isNaN( )	Returns **true** if the invoking object contains a value that is not a number. Otherwise, it returns **false**.
static boolean isNaN(float *num*)	Returns **true** if *num* specifies a value that is not a number. Otherwise, it returns **false**.
long longValue( )	Returns the value of the invoking object as a **long**.
static float max(float *val*, float *val2*)	Returns the maximum of *val* and *val2*.
static float min(float *val*, float *val2*)	Returns the minimum of *val* and *val2*.
static float parseFloat(String *str*) throws NumberFormatException	Returns the **float** equivalent of the number contained in the string specified by *str* using radix 10.
short shortValue( )	Returns the value of the invoking object as a **short**.
static float sum(float *val*, float *val2*)	Returns the result of *val* + *val2*.
static String toHexString(float *num*)	Returns a string containing the value of *num* in hexadecimal format.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(float *num*)	Returns the string equivalent of the value specified by *num*.
static Float valueOf(float *num*)	Returns a **Float** object containing the value passed in *num*.
static Float valueOf(String *str*) throws NumberFormatException	Returns the **Float** object that contains the value specified by the string in *str*.

Table 18-1 The Methods Defined by **Float**

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(double *num1*, double *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Double *d*)	Compares the numerical value of the invoking object with that of *d*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static long doubleToLongBits(double *num*)	Returns the IEEE-compatible, double-precision bit pattern that corresponds to *num*.
static long doubleToRawLongBits(double *num*)	Returns the IEEE-compatible double-precision bit pattern that corresponds to *num*. A NaN value is preserved.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *DoubleObj*)	Returns **true** if the invoking **Double** object is equivalent to *DoubleObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashcode( )	Returns the hash code for the invoking object.
static int hashCode(double *num*)	Returns the hash code for *num*.
int intValue( )	Returns the value of the invoking object as an **int**.
static boolean isFinite(double *num*)	Returns **true** if *num* is not **NaN** and is not infinite.
boolean isInfinite( )	Returns **true** if the invoking object contains an infinite value. Otherwise, it returns **false**.

static boolean isInfinite(double *num*)	Returns **true** if *num* specifies an infinite value. Otherwise, it returns **false**.
boolean isNaN( )	Returns **true** if the invoking object contains a value that is not a number. Otherwise, it returns **false**.
static boolean isNaN(double *num*)	Returns **true** if *num* specifies a value that is not a number. Otherwise, it returns **false**.
static double longBitsToDouble(long *num*)	Returns **double** equivalent of the IEEE-compatible, double-precision bit pattern specified by *num*.
long longValue( )	Returns the value of the invoking object as a **long**.
static double max(double *val*, double *val2*)	Returns the maximum of *val* and *val2*.
static double min(double *val*, double *val2*)	Returns the minimum of *val* and *val2*.
static double parseDouble(String *str*) throws NumberFormatException	Returns the **double** equivalent of the number contained in the string specified by *str* using radix 10.
short shortValue( )	Returns the value of the invoking object as a **short**.
static double sum(double *val*, double *val2*)	Returns the result of *val* + *val2*.
static String toHexString(double *num*)	Returns a string containing the value of *num* in hexadecimal format.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(double *num*)	Returns the string equivalent of the value specified by *num*.
static Double valueOf(double *num*)	Returns a **Double** object containing the value passed in *num*.
static Double valueOf(String *str*) throws NumberFormatException	Returns a **Double** object that contains the value specified by the string in *str*.

Table 18-2 The Methods Defined by **Double**

BYTES	The width of a **float** or **double** in bytes
MAX_EXPONENT	Maximum exponent
MAX_VALUE	Maximum positive value
MIN_EXPONENT	Minimum exponent
MIN_NORMAL	Minimum positive normal value
MIN_VALUE	Minimum positive value
NaN	Not a number
POSITIVE_INFINITY	Positive infinity
NEGATIVE_INFINITY	Negative infinity
SIZE	The bit width of the wrapped value
TYPE	The **Class** object for **float** or **double**

The following example creates two **Double** objects—one by using a **double** value and the other by passing a string that can be parsed as a **double**:

```
class DoubleDemo {
  public static void main(String args[]) {
    Double d1 = Double.valueOf(3.14159);
    Double d2 = Double.valueOf("314159E-5");

    System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
  }
}
```

As you can see from the following output, both versions of **valueOf( )** created identical **Double** instances, as shown by the **equals( )** method returning **true**:

```
3.14159 = 3.14159 -> true
```

Understanding isInfinite( ) and isNaN( )

Float and **Double** provide the methods **isInfinite( )** and **isNaN( )**, which help when manipulating two special **double** and **float** values. These methods test for two unique values defined by the IEEE floating-point specification: infinity and NaN (not a number). **isInfinite( )** returns **true** if the value being tested is infinitely large or small in magnitude. **isNaN( )** returns **true** if the value being tested is not a number.

The following example creates two **Double** objects; one is infinite, and the

other is not a number:

```
// Demonstrate isInfinite() and isNaN()
class InfNaN {
  public static void main(String args[]) {
     Double d1 = Double.valueOf(1/0.);
     Double d2 = Double.valueOf(0/0.);

     System.out.println(d1 + ": " + d1.isInfinite() + ", " + d1.isNaN());
     System.out.println(d2 + ": " + d2.isInfinite() + ", " + d2.isNaN());
  }
}
```

This program generates the following output:

```
Infinity: true, false
NaN: false, true
```

Byte, Short, Integer, and Long

The **Byte**, **Short**, **Integer**, and **Long** classes are wrappers for **byte**, **short**, **int**, and **long** integer types, respectively. Their constructors are shown here:

Byte(byte *num*)
Byte(String *str*) throws NumberFormatException

Short(short *num*)
Short(String *str*) throws NumberFormatException

Integer(int *num*)
Integer(String *str*) throws NumberFormatException

Long(long *num*)
Long(String *str*) throws NumberFormatException

As you can see, these objects can be constructed from numeric values or from strings that contain valid whole number values. Beginning with JDK 9, these constructors have been deprecated. The **valueOf( )** method is the recommended alternative.

The methods defined by these classes are shown in Tables 18-3 through 18-6. As you can see, they define methods for parsing integers from strings and converting strings back into integers. Variants of these methods allow you to specify the *radix*, or numeric base, for conversion. Common radixes are 2 for

binary, 8 for octal, 10 for decimal, and 16 for hexadecimal.

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(byte *num1*, byte *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Byte *b*)	Compares the numerical value of the invoking object with that of *b*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(byte *num1*, byte *num2*)	Performs an unsigned comparison of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
static Byte decode(String *str*) throws NumberFormatException	Returns a **Byte** object that contains the value specified by the string in *str*.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *ByteObj*)	Returns **true** if the invoking **Byte** object is equivalent to *ByteObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(byte *num*)	Returns the hash code for *num*.
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.

static byte parseByte(String *str*) throws NumberFormatException	Returns the **byte** equivalent of the number contained in the string specified by *str* using radix 10.
static byte parseByte(String *str*, int *radix*) throws NumberFormatException	Returns the **byte** equivalent of the number contained in the string specified by *str* using the specified radix.
short shortValue( )	Returns the value of the invoking object as a **short**.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(byte *num*)	Returns a string that contains the decimal equivalent of *num*.
static int toUnsignedInt(byte *val*)	Returns the value of *val* as an unsigned integer.
static long toUnsignedLong(byte *val*)	Returns the value of *val* as an unsigned long integer.
static Byte valueOf(byte *num*)	Returns a **Byte** object containing the value passed in *num*.
static Byte valueOf(String *str*) throws NumberFormatException	Returns a **Byte** object that contains the value specified by the string in *str*.
static Byte valueOf(String *str*, int *radix*) throws NumberFormatException	Returns a **Byte** object that contains the value specified by the string in *str* using the specified *radix*.

Table 18-3 The Methods Defined by **Byte**

Method	Description
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(short *num1*, short *num2*	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Short *s*)	Compares the numerical value of the invoking object with that of *s*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(short *num1*, short *num2*)	Performs an unsigned comparison of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
static Short decode(String *str*) throws NumberFormatException	Returns a **Short** object that contains the value specified by the string in *str*.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *ShortObj*)	Returns **true** if the invoking **Short** object is equivalent to *ShortObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(short *num*)	Returns the hash code for *num*.
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.

static short parseShort(String *str*) throws NumberFormatException	Returns the **short** equivalent of the number contained in the string specified by *str* using radix 10.
static short parseShort(String *str*, int *radix*) throws NumberFormatException	Returns the **short** equivalent of the number contained in the string specified by *str* using the specified *radix*.
static short reverseBytes(short *num*)	Exchanges the high- and low-order bytes of *num* and returns the result.
short shortValue( )	Returns the value of the invoking object as a **short**.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(short *num*)	Returns a string that contains the decimal equivalent of *num*.
static int toUnsignedInt(short *val*)	Returns the value of *val* as an unsigned integer.
static long toUnsignedLong(short *val*)	Returns the value of *val* as an unsigned long integer.
static Short valueOf(short *num*)	Returns a **Short** object containing the value passed in *num*.
static Short valueOf(String *str*) throws NumberFormatException	Returns a **Short** object that contains the value specified by the string in *str* using radix 10.
static Short valueOf(String *str*, int *radix*) throws NumberFormatException	Returns a **Short** object that contains the value specified by the string in *str* using the specified *radix*.

Table 18-4 The Methods Defined by **Short**

Method	Description
static int bitCount(int *num*)	Returns the number of set bits in *num*.
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(int *num1*, int *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Integer *i*)	Compares the numerical value of the invoking object with that of *i*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(int *num1*, int *num2*)	Performs an unsigned comparison of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
static Integer decode(String *str*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str*.
static int divideUnsigned(int *dividend*, int *divisor*)	Returns the result, as an unsigned value, of the unsigned division of *dividend* by *divisor*.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *IntegerObj*)	Returns **true** if the invoking **Integer** object is equivalent to *IntegerObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
static Integer getInteger(String *propertyName*)	Returns the value associated with the environmental property specified by *propertyName*. A **null** is returned on failure.
static Integer getInteger(String *propertyName*, int *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.
static Integer getInteger(String *propertyName*, Integer *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.

int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(int *num*)	Returns the hash code for *num*.
static int highestOneBit(int *num*)	Determines the position of the highest order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.
static int lowestOneBit(int *num*)	Determines the position of the lowest order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
static int max(int *val*, int *val2*)	Returns the maximum of *val* and *val2*.
static int min(int *val*, int *val2*)	Returns the minimum of *val* and *val2*.
static int numberOfLeadingZeros(int *num*)	Returns the number of high-order zero bits that precede the first high-order set bit in *num*. If *num* is zero, 32 is returned.
static int numberOfTrailingZeros(int *num*)	Returns the number of low-order zero bits that precede the first low-order set bit in *num*. If *num* is zero, 32 is returned.
static int parseInt(CharSequence *chars*, int *startIdx*, int *stopIdx*, int *radix*) throws NumberFormatException	Returns the integer equivalent of the number contained in the sequence specified by *chars*, between the indices *startIdx* and *stopIdx*-1, using the specified *radix*.
static int parseInt(String *str*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using radix 10.
static int parseInt(String *str*, int *radix*) throws NumberFormatException	Returns the integer equivalent of the number contained in the string specified by *str* using the specified *radix*.
static int parseUnsignedInt(CharSequence *chars*, int *startIdx*, int *stopIdx*, int *radix*) throws NumberFormatException	Returns the integer equivalent of the unsigned number contained in the sequence specified by *chars*, between the indices *startIdx* and *stopIdx*-1, using the specified *radix*.

static int parseUnsignedInt(String *str*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix 10.
static int parseUnsignedInt(String *str*, int *radix*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix specified by *radix*.
static int remainderUnsigned(int *dividend*, int *divisor*)	Returns the remainder, as an unsigned value, of the unsigned division of *dividend* by *divisor*.
static int reverse(int *num*)	Reverses the order of the bits in *num* and returns the result.
static int reverseBytes(int *num*)	Reverses the order of the bytes in *num* and returns the result.
static int rotateLeft(int *num*, int *n*)	Returns the result of rotating *num* left *n* positions.
static int rotateRight(int *num*, int *n*)	Returns the result of rotating *num* right *n* positions.
short shortValue( )	Returns the value of the invoking object as a **short**.
static int signum(int *num*)	Returns –1 if *num* is negative, 0 if it is zero, and 1 if it is positive.
static int sum(int *val*, int *val2*)	Returns the result of *val* + *val2*.
static String toBinaryString(int *num*)	Returns a string that contains the binary equivalent of *num*.
static String toHexString(int *num*)	Returns a string that contains the hexadecimal equivalent of *num*.
static String toOctalString(int *num*)	Returns a string that contains the octal equivalent of *num*.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(int *num*)	Returns a string that contains the decimal equivalent of *num*.
static String toString(int *num*, int *radix*)	Returns a string that contains the decimal equivalent of *num* using the specified *radix*.
static long toUnsignedLong(int *val*)	Returns the value of *val* as an unsigned long integer.
static String toUnsignedString(int *val*)	Returns a string that contains the decimal value of *val* as an unsigned integer.
static String toUnsignedString(int *val*, int *radix*)	Returns a string that contains the value of *val* as an unsigned integer in the radix specified by *radix*.
static Integer valueOf(int *num*)	Returns an **Integer** object containing the value passed in *num*.
static Integer valueOf(String *str*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str*.
static Integer valueOf(String *str*, int *radix*) throws NumberFormatException	Returns an **Integer** object that contains the value specified by the string in *str* using the specified *radix*.

Table 18-5 The Methods Defined by **Integer**

Method	Description
static int bitCount(long *num*)	Returns the number of set bits in *num*.
byte byteValue( )	Returns the value of the invoking object as a **byte**.
static int compare(long *num1*, long *num2*)	Compares the values of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
int compareTo(Long *l*)	Compares the numerical value of the invoking object with that of *l*. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compareUnsigned(long *num1*, long *num2*)	Performs an unsigned comparison of *num1* and *num2*. Returns 0 if the values are equal. Returns a negative value if *num1* is less than *num2*. Returns a positive value if *num1* is greater than *num2*.
static Long decode(String *str*) throws NumberFormatException	Returns a **Long** object that contains the value specified by the string in *str*.
static long divideUnsigned(long *dividend*, long *divisor*)	Returns the result, as an unsigned value, of the unsigned division of *dividend* by *divisor*.
double doubleValue( )	Returns the value of the invoking object as a **double**.
boolean equals(Object *LongObj*)	Returns **true** if the invoking **Long** object is equivalent to *LongObj*. Otherwise, it returns **false**.
float floatValue( )	Returns the value of the invoking object as a **float**.
static Long getLong(String *propertyName*)	Returns the value associated with the environmental property specified by *propertyName*. A **null** is returned on failure.
static Long getLong(String *propertyName*, long *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.
static Long getLong(String *propertyName*, Long *default*)	Returns the value associated with the environmental property specified by *propertyName*. The value of *default* is returned on failure.

int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(long *num*)	Returns the hash code for *num*.
static long highestOneBit(long *num*)	Determines the position of the highest-order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
int intValue( )	Returns the value of the invoking object as an **int**.
long longValue( )	Returns the value of the invoking object as a **long**.
static long lowestOneBit(long *num*)	Determines the position of the lowest-order set bit in *num*. It returns a value in which only this bit is set. If no bit is set to one, then zero is returned.
static long max(long *val*, long *val2*)	Returns the maximum of *val* and *val2*.
static long min(long *val*, long *val2*)	Returns the minimum of *val* and *val2*.
static int numberOfLeadingZeros(long *num*)	Returns the number of high-order zero bits that precede the first high-order set bit in *num*. If *num* is zero, 64 is returned.
static int numberOfTrailingZeros(long *num*)	Returns the number of low-order zero bits that precede the first low-order set bit in *num*. If *num* is zero, 64 is returned.
static long parseLong(CharSequence *chars*, int *startIdx*, int *stopIdx*, int *radix*) throws NumberFormatException	Returns the **long** equivalent of the number contained in the sequence specified by *chars*, between the indices *startIdx* and *stopIdx*-1, using the specified *radix*.
static long parseLong(String *str*) throws NumberFormatException	Returns the **long** equivalent of the number contained in the string specified by *str* using radix 10.
static long parseLong(String *str*, int *radix*) throws NumberFormatException	Returns the **long** equivalent of the number contained in the string specified by *str* using the specified *radix*.
static long parseUnsignedLong (CharSequence *chars*, int *startIdx*, int *stopIdx*, int *radix*) throws NumberFormatException	Returns the **long** equivalent of the unsigned number contained in the sequence specified by *chars*, between the indices *startIdx* and *stopIdx*-1, using the specified *radix*.

static long parseUnsignedLong(String *str*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix 10.
static long parseUnsignedLong(String *str*, int *radix*) throws NumberFormatException	Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix specified by *radix*.
static long remainderUnsigned (long *dividend*, long *divisor*)	Returns the remainder, as an unsigned value, of the unsigned division of *dividend* by *divisor*.
static long reverse(long *num*)	Reverses the order of the bits in *num* and returns the result.
static long reverseBytes(long *num*)	Reverses the order of the bytes in *num* and returns the result.
static long rotateLeft(long *num*, int *n*)	Returns the result of rotating *num* left *n* positions.
static long rotateRight(long *num*, int *n*)	Returns the result of rotating *num* right *n* positions.
short shortValue( )	Returns the value of the invoking object as a **short**.
static int signum(long *num*)	Returns –1 if *num* is negative, 0 if it is zero, and 1 if it is positive.
static long sum(long *val*, long *val2*)	Returns the result of *val* + *val2*.
static String toBinaryString(long *num*)	Returns a string that contains the binary equivalent of *num*.
static String toHexString(long *num*)	Returns a string that contains the hexadecimal equivalent of *num*.
static String toOctalString(long *num*)	Returns a string that contains the octal equivalent of *num*.
String toString( )	Returns a string that contains the decimal equivalent of the invoking object.
static String toString(long *num*)	Returns a string that contains the decimal equivalent of *num*.
static String toString(long *num*, int *radix*)	Returns a string that contains the decimal equivalent of *num* using the specified *radix*.
static String toUnsignedString(long *val*)	Returns a string that contains the decimal value of *val* as an unsigned integer.
static String toUnsignedString(long *val*, int *radix*)	Returns a string that contains the value of *val* as an unsigned integer in the radix specified by *radix*.
static Long valueOf(long *num*)	Returns a **Long** object containing the value passed in *num*.
static Long valueOf(String *str*) throws NumberFormatException	Returns a **Long** object that contains the value specified by the string in *str*.
static Long valueOf(String *str*, int *radix*) throws NumberFormatException	Returns a **Long** object that contains the value specified by the string in *str* using the specified *radix*.

Table 18-6 The Methods Defined by **Long**

The following constants are defined:

BYTES	The width of the integer type in bytes
MIN_VALUE	Minimum value
MAX_VALUE	Maximum value
SIZE	The bit width of the wrapped value
TYPE	The **Class** object for **byte**, **short**, **int**, or **long**

Converting Numbers to and from Strings

One of the most common programming chores is converting the string representation of a number into its internal, binary format. Fortunately, Java provides an easy way to accomplish this. The **Byte**, **Short**, **Integer**, and **Long** classes provide the **parseByte( )**, **parseShort( )**, **parseInt( )**, and **parseLong( )** methods, respectively. These methods return the **byte**, **short**, **int**, or **long** equivalent of the numeric string with which they are called. (Similar methods also exist for the **Float** and **Double** classes.)

The following program demonstrates **parseInt( )**. It sums a list of integers entered by the user. It reads the integers using **readLine( )** and uses **parseInt( )** to convert these strings into their **int** equivalents.

```
/* This program sums a list of numbers entered
   by the user.  It converts the string representation
   of each number into an int using parseInt().
*/
import java.io.*;

class ParseDemo {
  public static void main(String args[])
    throws IOException
  {
    // create a BufferedReader using System.in
    BufferedReader br = new
      BufferedReader(new InputStreamReader(System.in));
    String str;
    int i;
    int sum=0;

    System.out.println("Enter numbers, 0 to quit.");
    do {
      str = br.readLine();
      try {
        i = Integer.parseInt(str);
      } catch(NumberFormatException e) {
        System.out.println("Invalid format");
        i = 0;
      }
      sum += i;
      System.out.println("Current sum is: " + sum);
    } while(i != 0);
  }
}
```

To convert a whole number into a decimal string, use the versions of **toString( )** defined in the **Byte**, **Short**, **Integer**, or **Long** classes. The **Integer** and **Long** classes also provide the methods **toBinaryString( )**, **toHexString( )**, and **toOctalString( )**, which convert a value into a binary, hexadecimal, or octal string, respectively.

The following program demonstrates binary, hexadecimal, and octal conversion:

```
/* Convert an integer into binary, hexadecimal,
   and octal.
*/

class StringConversions {
  public static void main(String args[]) {
    int num = 19648;
    System.out.println(num + " in binary: " +
                       Integer.toBinaryString(num));

    System.out.println(num + " in octal: " +
                       Integer.toOctalString(num));

    System.out.println(num + " in hexadecimal: " +
                       Integer.toHexString(num));
  }
}
```

The output of this program is shown here:

```
19648 in binary: 100110011000000
19648 in octal: 46300
19648 in hexadecimal: 4cc0
```

Character

Character is a simple wrapper around a **char**. The constructor for **Character** is

Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created. Beginning with JDK 9, this constructor has been deprecated. The **valueOf( )** method is the recommended alternative.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

char charValue( )

It returns the character.

The **Character** class defines several constants, including the following:

BYTES	The width of a **char** in bytes
MAX_RADIX	The largest radix
MIN_RADIX	The smallest radix
MAX_VALUE	The largest character value
MIN_VALUE	The smallest character value
TYPE	The **Class** object for **char**

Character includes several static methods that categorize characters and alter their case. A sampling is shown in Table 18-7. The following example demonstrates several of these methods:

Method	Description
static boolean isDefined(char *ch*)	Returns **true** if *ch* is defined by Unicode. Otherwise, it returns **false**.
static boolean isDigit(char *ch*)	Returns **true** if *ch* is a digit. Otherwise, it returns **false**.
static boolean isIdentifierIgnorable(char *ch*)	Returns **true** if *ch* should be ignored in an identifier. Otherwise, it returns **false**.
static boolean islSOControl(char *ch*)	Returns **true** if *ch* is an ISO control character. Otherwise, it returns **false**.
static boolean isJavaIdentifierPart(char *ch*)	Returns **true** if *ch* is allowed as part of a Java identifier (other than the first character). Otherwise, it returns **false**.
static boolean isJavaIdentifierStart(char *ch*)	Returns **true** if *ch* is allowed as the first character of a Java identifier. Otherwise, it returns **false**.
static boolean isLetter(char *ch*)	Returns **true** if *ch* is a letter. Otherwise, it returns false.
static boolean isLetterOrDigit(char *ch*)	Returns **true** if *ch* is a letter or a digit. Otherwise, it returns **false**.
static boolean isLowerCase(char *ch*)	Returns **true** if *ch* is a lowercase letter. Otherwise, it returns **false**.
static boolean isMirrored(char *ch*)	Returns **true** if *ch* is a mirrored Unicode character. A mirrored character is one that is reversed for text that is displayed right-to-left.
static boolean isSpaceChar(char *ch*)	Returns **true** if *ch* is a Unicode space character. Otherwise, it returns **false**.
static boolean isTitleCase(char *ch*)	Returns **true** if *ch* is a Unicode titlecase character. Otherwise, it returns **false**.
static boolean isUnicodeIdentifierPart(char *ch*)	Returns **true** if *ch* is allowed as part of a Unicode identifier (other than the first character). Otherwise, it returns **false**.
static Boolean isUnicodeIdentifierStart(char *ch*)	Returns **true** if *ch* is allowed as the first character of a Unicode identifier. Otherwise, it returns **false**.
static boolean isUpperCase(char *ch*)	Returns **true** if *ch* is an uppercase letter. Otherwise, it returns **false**.
static boolean isWhitespace(char *ch*)	Returns **true** if *ch* is whitespace. Otherwise, it returns **false**.
static char toLowerCase(char *ch*)	Returns lowercase equivalent of *ch*.
static char toTitleCase(char *ch*)	Returns titlecase equivalent of *ch*.
static char toUpperCase(char *ch*)	Returns uppercase equivalent of *ch*.

Table 18-7 Various Character **Methods**

```
// Demonstrate several Is... methods.

class IsDemo {
  public static void main(String args[]) {
    char a[] = {'a', 'b', '5', '?', 'A', ' '};

    for(int i=0; i<a.length; i++) {
      if(Character.isDigit(a[i]))
        System.out.println(a[i] + " is a digit.");
      if(Character.isLetter(a[i]))
        System.out.println(a[i] + " is a letter.");
      if(Character.isWhitespace(a[i]))
        System.out.println(a[i] + " is whitespace.");
      if(Character.isUpperCase(a[i]))
        System.out.println(a[i] + " is uppercase.");
      if(Character.isLowerCase(a[i]))
        System.out.println(a[i] + " is lowercase.");
    }
  }
}
```

The output from this program is shown here:

```
a is a letter.
a is lowercase.
b is a letter.
b is lowercase.
5 is a digit.
A is a letter.
A is uppercase.
  is whitespace.
```

Character defines two methods, **forDigit( )** and **digit( )**, that enable you to convert between integer values and the digits they represent. They are shown here:

static char forDigit(int *num*, int *radix*)
static int digit(char *digit*, int *radix*)

forDigit( ) returns the digit character associated with the value of *num*. The radix of the conversion is specified by *radix*. **digit( )** returns the integer value associated with the specified character (which is presumably a digit) according to the specified radix. (There is a second form of **digit( )** that takes a code point.

See the following section for a discussion of code points.)

Another method defined by **Character** is **compareTo( )**, which has the following form:

int compareTo(Character *c*)

It returns zero if the invoking object and *c* have the same value. It returns a negative value if the invoking object has a lower value. Otherwise, it returns a positive value.

Character includes a method called **getDirectionality( )** which can be used to determine the direction of a character. Several constants are defined that describe directionality. Most programs will not need to use character directionality.

Character also overrides **equals( )** and **hashCode( )**, and provides a number of other methods.

Two other character-related classes are **Character.Subset**, used to describe a subset of Unicode, and **Character.UnicodeBlock**, which contains Unicode character blocks.

Additions to Character for Unicode Code Point Support

A number of years ago, major additions were made to **Character**. Beginning with JDK 5, the **Character** class has included support for 32-bit Unicode characters. In the past, all Unicode characters could be held by 16 bits, which is the size of a **char** (and the size of the value encapsulated within a **Character**), because those values ranged from 0 to FFFF. However, the Unicode character set has been expanded, and more than 16 bits are required. Characters can now range from 0 to 10FFFF.

Here are three important terms. A *code point* is a character in the range 0 to 10FFFF. Characters that have values greater than FFFF are called *supplemental characters*. The *basic multilingual plane (BMP)* are those characters between 0 and FFFF.

The expansion of the Unicode character set caused a fundamental problem for Java. Because a supplemental character has a value greater than a **char** can hold, some means of handling the supplemental characters was needed. Java addressed this problem in two ways. First, Java uses two **chars** to represent a supplemental character. The first **char** is called the *high surrogate*, and the second is called the *low surrogate*. Methods, such as **codePointAt( )**, were provided to translate

between code points and supplemental characters.

Secondly, Java overloaded several preexisting methods in the **Character** class. The overloaded forms use **int** rather than **char** data. Because an **int** is large enough to hold any character as a single value, it can be used to store any character. For example, all of the methods in Table 18-7 have overloaded forms that operate on **int**. Here is a sampling:

static boolean isDigit(int *cp*)
static boolean isLetter(int *cp*)
static int toLowerCase(int *cp*)

In addition to the methods overloaded to accept code points, **Character** adds methods that provide additional support for code points. A sampling is shown in Table 18-8.

Method	Description
static int charCount(int *cp*)	Returns 1 if *cp* can be represented by a single **char**. It returns 2 if two **char**s are needed.
static int codePointAt(CharSequence *chars*, int *loc*)	Returns the code point at the location specified by *loc*.
static int codePointAt(char *chars*[ ], int *loc*)	Returns the code point at the location specified by *loc*.
static int codePointBefore(CharSequence *chars*, int *loc*)	Returns the code point at the location that precedes that specified by *loc*.
static int codePointBefore(char *chars*[ ], int *loc*)	Returns the code point at the location that precedes that specified by *loc*.
static boolean isBmpCodePoint(int *cp*)	Returns **true** if *cp* is part of the basic multilingual plane and **false** otherwise.
static boolean isHighSurrogate(char *ch*)	Returns **true** if *ch* contains a valid high surrogate character.
static boolean isLowSurrogate(char *ch*)	Returns **true** if *ch* contains a valid low surrogate character.
static boolean isSupplementaryCodePoint(int *cp*)	Returns **true** if *cp* contains a supplemental character.
static boolean isSurrogatePair(char *highCh*, char *lowCh*)	Returns **true** if *highCh* and *lowCh* form a valid surrogate pair.
static boolean isValidCodePoint(int *cp*)	Returns **true** if *cp* contains a valid code point.
static char[ ] toChars(int *cp*)	Converts the code point in *cp* into its **char** equivalent, which might require two **char**s. An array holding the result is returned.
static int toChars(int *cp*, char *target*[ ], int *loc*)	Converts the code point in *cp* into its **char** equivalent, storing the result in *target*, beginning at *loc*. Returns 1 if *cp* can be represented by a single **char**. It returns 2 otherwise.
static int toCodePoint(char *highCh*, char *lowCh*)	Converts *highCh* and *lowCh* into their equivalent code point.

Table 18-8 A Sampling of Methods That Provide Support for 32-Bit Unicode Code Points

Boolean

Boolean is a very thin wrapper around **boolean** values, which is useful mostly

when you want to pass a **boolean** variable by reference. It contains the constants **TRUE** and **FALSE**, which define true and false **Boolean** objects. **Boolean** also defines the **TYPE** field, which is the **Class** object for **boolean**. **Boolean** defines these constructors:

Boolean(boolean *boolValue*)
Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**. In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be **true**. Otherwise, it will be **false**. Beginning with JDK 9, these constructors have been deprecated. The **valueOf( )** method is the recommended alternative.

Boolean defines the methods shown in Table 18-9.

Method	Description
boolean booleanValue( )	Returns **boolean** equivalent.
static int compare(boolean *b1*, boolean *b2*)	Returns zero if *b1* and *b2* contain the same value. Returns a positive value if *b1* is **true** and *b2* is **false**. Otherwise, returns a negative value.
int compareTo(Boolean *b*)	Returns zero if the invoking object and *b* contain the same value. Returns a positive value if the invoking object is **true** and *b* is **false**. Otherwise, returns a negative value.
boolean equals(Object *boolObj*)	Returns **true** if the invoking object is equivalent to *boolObj*. Otherwise, it returns **false**.
static Boolean getBoolean(String *propertyName*)	Returns **true** if the system property specified by *propertyName* is **true**. Otherwise, it returns **false**.
int hashCode( )	Returns the hash code for the invoking object.
static int hashCode(boolean *boolVal*)	Returns the hash code for *boolVal*.
static boolean logicalAnd(boolean *op1*, boolean *op2*)	Performs a logical AND of *op1* and *op2* and returns the result.
static boolean logicalOr(boolean *op1*, boolean *op2*)	Performs a logical OR of *op1* and *op2* and returns the result.
static boolean logicalXor(boolean *op1*, boolean *op2*)	Performs a logical XOR of *op1* and *op2* and returns the result.
static boolean parseBoolean(String *str*)	Returns **true** if *str* contains the string "true". Case is not significant. Otherwise, returns **false**.
String toString( )	Returns the string equivalent of the invoking object.
static String toString(boolean *boolVal*)	Returns the string equivalent of *boolVal*.
static Boolean valueOf(boolean *boolVal*)	Returns the **Boolean** equivalent of *boolVal*.
static Boolean valueOf(String *boolString*)	Returns **true** if *boolString* contains the string "true" (in uppercase or lowercase). Otherwise, it returns **false**.

Table 18-9 The Methods Defined by **Boolean**

Void

The **Void** class has one field, **TYPE**, which holds a reference to the **Class** object for type **void**. You do not create instances of this class.

Process

The abstract **Process** class encapsulates a *process*—that is, an executing program. It is used primarily as a superclass for the type of objects created by **exec( )** in the **Runtime** class, or by **start( )** in the **ProcessBuilder** class. **Process** contains the methods shown in Table 18-10. Beginning with JDK 9, you can obtain a handle to the process in the form of a **ProcessHandle** instance, and that you can obtain information about the process encapsulated in a **ProcessHandle.Info** instance. These offer additional control and information about a process. One particularly interesting piece of information is the amount of CPU time that a process receives. This is obtained by calling **totalCpuDuration( )** defined by **ProcessHandle.Info**. Another especially helpful piece of information is obtained by calling **isAlive( )** on a **ProcessHandle**. It will return **true** if the process is still executing.

Method	Description
Stream<ProcessHandle> children( )	Returns a stream that contains **ProcessHandle** objects that represent the immediate children of the invoking process.
Stream<ProcessHandle> descendants( )	Returns a stream that contains **ProcessHandle** objects that represent both the immediate children of the invoking process, plus all of their descendants.
void destroy( )	Terminates the process.
Process destroyForcibly( )	Forces termination of the invoking process. Returns a reference to the process.
int exitValue( )	Returns an exit code obtained from a subprocess.
InputStream getErrorStream( )	Returns an input stream that reads input from the process' **err** output stream.
InputStream getInputStream( )	Returns an input stream that reads input from the process' **out** output stream.
OutputStream getOutputStream( )	Returns an output stream that writes output to the process' **in** input stream.
ProcessHandle.Info info( )	Returns information about the process in the form of a **ProcessHandle.Info** object.
boolean isAlive( )	Returns **true** if the invoking process is still active. Otherwise, returns **false**.
CompletableFuture<Process> onExit( )	Returns a **CompletableFuture** for the invoking process, which can be used to perform tasks at termination.
long pid( )	Returns the process ID associated with the invoking process.
boolean supportsNormalTermination( )	Determines if a call to **destroy( )** will result in normal or forced termination. Returns **true** if termination is normal, and **false** otherwise.
ProcessHandle toHandle( )	Returns a handle to the invoking process in the form of a **ProcessHandle** object.
int waitFor( ) throws InterruptedException	Returns the exit code returned by the process. This method does not return until the process on which it is called terminates.
boolean waitFor(long *waitTime*, TimeUnit *timeUnit*) throws InterruptedException	Waits for the invoking process to end. The amount of time to wait is specified by *waitTime* in the units specified by *timeUnit*. Returns **true** if the process has ended and **false** if the wait time runs out.

Table 18-10 The Methods Defined by **Process**

Runtime

The **Runtime** class encapsulates the run-time environment. You cannot instantiate a **Runtime** object. However, you can get a reference to the current

Runtime object by calling the static method **Runtime.getRuntime( )**. Once you obtain a reference to the current **Runtime** object, you can call several methods that control the state and behavior of the Java Virtual Machine. Untrusted code typically cannot call any of the **Runtime** methods without raising a **SecurityException**. A sampling of methods defined by **Runtime** are shown in Table 18-11.

Method	Description
void addShutdownHook(Thread *thrd*)	Registers *thrd* as a thread to be run when the Java Virtual Machine terminates.
Process exec(String *progName*) throws IOException	Executes the program specified by *progName* as a separate process. An object of type **Process** is returned that describes the new process.
Process exec(String *progName*, String *environment*[ ]) throws IOException	Executes the program specified by *progName* as a separate process with the environment specified by *environment*. An object of type **Process** is returned that describes the new process.
Process exec(String *comLineArray*[ ]) throws IOException	Executes the command line specified by the strings in *comLineArray* as a separate process. An object of type **Process** is returned that describes the new process.
Process exec(String *comLineArray*[ ], String *environment*[ ]) throws IOException	Executes the command line specified by the strings in *comLineArray* as a separate process with the environment specified by *environment*. An object of type **Process** is returned that describes the new process.
void exit(int *exitCode*)	Halts execution and returns the value of *exitCode* to the parent process. By convention, 0 indicates normal termination. All other values indicate some form of error.
long freeMemory( )	Returns the approximate number of bytes of free memory available to the Java run-time system.
void gc( )	Initiates garbage collection.
static Runtime getRuntime( )	Returns the current **Runtime** object.
void halt(int *code*)	Immediately terminates the Java Virtual Machine. No termination threads or finalizers are run. The value of *code* is returned to the invoking process.
void load(String *libraryFileName*)	Loads the dynamic library whose file is specified by *libraryFileName*, which must specify its complete path.
void loadLibrary(String *libraryName*)	Loads the dynamic library whose name is associated with *libraryName*.
Boolean removeShutdownHook(Thread *thrd*)	Removes *thrd* from the list of threads to run when the Java Virtual Machine terminates. It returns **true** if successful—that is, if the thread was removed.
void runFinalization( )	Initiates calls to the **finalize( )** methods of unused but not yet recycled objects.
long totalMemory( )	Returns the total number of bytes of memory available to the program.
static Runtime.Version version( )	Returns the Java version being used. See **Runtime .Version** for details.

Table 18-11 A Sampling of Methods Defined by **Runtime**

Let’s look at two of the more interesting uses of the **Runtime** class: memory management and executing additional processes.

Memory Management

Although Java provides automatic garbage collection, sometimes you will want to know how large the object heap is and how much of it is left. You can use this information, for example, to check your code for efficiency or to approximate how many more objects of a certain type can be instantiated. To obtain these values, use the **totalMemory( )** and **freeMemory( )** methods.

As mentioned in Part I, Java’s garbage collector runs periodically to recycle unused objects. However, sometimes you will want to collect discarded objects prior to the collector’s next appointed rounds. You can run the garbage collector on demand by calling the **gc( )** method. A good thing to try is to call **gc( )** and then call **freeMemory( )** to get a baseline memory usage. Next, execute your code and call **freeMemory( )** again to see how much memory it is allocating. The following program illustrates this idea:

```
// Demonstrate totalMemory(), freeMemory() and gc().

class MemoryDemo {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    long mem1, mem2;
    Integer someints[] = new Integer[1000];

    System.out.println("Total memory is: " +
                       r.totalMemory());
    mem1 = r.freeMemory();
    System.out.println("Initial free memory: " + mem1);
    r.gc();
    mem1 = r.freeMemory();
    System.out.println("Free memory after garbage collection: "
                       + mem1);

    for(int i=0; i<1000; i++)
      someints[i] = Integer.valueOf(i); // allocate integers

    mem2 = r.freeMemory();
    System.out.println("Free memory after allocation: "
                       + mem2);
    System.out.println("Memory used by allocation: "
                       + (mem1-mem2));

    // discard Integers
    for(int i=0; i<1000; i++) someints[i] = null;

    r.gc(); // request garbage collection

    mem2 = r.freeMemory();
    System.out.println("Free memory after collecting" +
                       " discarded Integers: " + mem2);

  }
}
```

Sample output from this program is shown here (of course, your actual results may vary):

```
Total memory is: 1048568
Initial free memory: 751392
```

```
Free memory after garbage collection: 841424
Free memory after allocation: 824000
Memory used by allocation: 17424
Free memory after collecting discarded Integers: 842640
```

Executing Other Programs

In safe environments, you can use Java to execute other heavyweight processes (that is, programs) on your multitasking operating system. Several forms of the **exec( )** method allow you to name the program you want to run as well as its input parameters. The **exec( )** method returns a **Process** object, which can then be used to control how your Java program interacts with this new running process. Because Java can run on a variety of platforms and under a variety of operating systems, **exec( )** is inherently environment-dependent.

The following example uses **exec( )** to launch **notepad**, Windows' simple text editor. Obviously, this example must be run under the Windows operating system.

```
// Demonstrate exec().
class ExecDemo {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
    } catch (Exception e) {
      System.out.println("Error executing notepad.");
    }
  }
}
```

There are several alternative forms of **exec( )**, but the one shown in the example is often sufficient. The **Process** object returned by **exec( )** can be manipulated by **Process**' methods after the new program starts running. You can kill the subprocess with the **destroy( )** method. The **waitFor( )** method causes your program to wait until the subprocess finishes. The **exitValue( )** method returns the value returned by the subprocess when it is finished. This is typically 0 if no problems occur. Here is the preceding **exec( )** example modified to wait for the running process to exit:

```
// Wait until notepad is terminated.
class ExecDemoFini {
  public static void main(String args[]) {
    Runtime r = Runtime.getRuntime();
    Process p = null;

    try {
      p = r.exec("notepad");
      p.waitFor();
    } catch (Exception e) {
      System.out.println("Error executing notepad.");
    }
    System.out.println("Notepad returned " + p.exitValue());
  }
}
```

While a subprocess is running, you can write to and read from its standard input and output. The **getOutputStream( )** and **getInputStream( )** methods return the handles to standard **in** and **out** of the subprocess. (I/O is examined in detail in Chapter 21.)

Runtime.Version

Runtime.Version encapsulates version information (which includes the version number) pertaining to the Java environment. You can obtain an instance of **Runtime.Version** for the current platform by calling **Runtime.version( )**. Originally added by JDK 9, **Runtime.Version** was substantially changed with the release of JDK 10 to better accommodate the faster, time-based release cadence. As discussed earlier in this book, starting with JDK 10, a feature release is anticipated to occur on a strict schedule, with the time between feature releases expected to be six months.

In the past, the JDK version number used the well-known *major.minor* approach. This mechanism did not, however, provide a good fit with the time-based release schedule. As a result, a different meaning was given to the elements of a version number. Today, the first four elements specify *counters,* which occur in the following order: feature release counter, interim release counter, update release counter, and patch release counter. Each number is separated by a period. However, trailing zeros, along with their preceding periods, are removed. Although additional elements may also be included, only the meaning of the first four are predefined.

The feature release counter specifies the number of the release. This counter is updated with each feature release. To smooth the transition from the previous version scheme, the feature release counter began at 10. Thus, the feature release counter for JDK 10 is 10, the one for JDK 11 is 11, and so on.

The interim release counter indicates the number of a release that occurs between feature releases. At the time of this writing, the value of the interim release counter will be zero because interim releases are not expected to be part of the increased release cadence. (It is defined for possible future use.) An interim release will not cause breaking changes to the JDK feature set. The update release counter indicates the number of a release that addresses security and possibly other problems. The patch release counter specifies a number of a release that addresses a serious flaw that must be fixed as soon as possible. With each new feature release, the interim, update, and patch counters are reset to zero.

It is useful to point out that the version number just described is a necessary component of the *version string*, but optional elements may also be included in the string. For example, a version string may include information for a pre-release version. Optional elements follow the version number in the version string.

Beginning with JDK 10, **Runtime.Version** was updated to include the following methods that support the new feature, interim, update, and patch counter values:

```
int feature( )
int interim( )
int update( )
int patch( )
```

Each returns an integer value that represents the indicated value. Here is a short program that demonstrates their use:

```
// Demonstrate Runtime.Version release counters.
class VerDemo {
  public static void main(String args[]) {
    Runtime.Version ver = Runtime.version();

    // Display individual counters.
    System.out.println("Feature release counter: " + ver.feature());
    System.out.println("Interim release counter: " + ver.interim());
    System.out.println("Update release counter: " + ver.update());
    System.out.println("Patch release counter: " + ver.patch());
  }
}
```

As a result of the change to time-based releases, the following methods in **Runtime.Version** have been deprecated: **major( )**, **minor( )**, and **security( )**. Previously, these returned the major version number, the minor version number, and the security update number. These values have been superseded by the feature, interim, and update numbers, as just described.

In addition to the methods just discussed, **Runtime.Version** has methods that obtain various pieces of optional data. For example, you can obtain the build number, if present, by calling **build( )**. Pre-release information, if present, is returned by **pre( )**. Other optional information may also be present and is obtained by calling **optional( )**. You can compare versions by using **compareTo( )** or **compareToIgnoreOptional( )**. You can use **equals( )** and **equalsIgnoreOptional( )** to determine version equality. The **version( )** method returns a list of the version numbers. You can convert a valid version string into a **Runtime.Version** object by calling **parse( )**.

ProcessBuilder

ProcessBuilder provides another way to start and manage processes (that is, programs). As explained earlier, all processes are represented by the **Process** class, and a process can be started by **Runtime.exec( )**. **ProcessBuilder** offers more control over the processes. For example, you can set the current working directory.

ProcessBuilder defines these constructors:

ProcessBuilder(List<String> *args*)
ProccessBuilder(String ... *args*)

Here, *args* is a list of arguments that specify the name of the program to be

executed along with any required command-line arguments. In the first constructor, the arguments are passed in a **List**. In the second, they are specified through a varargs parameter. Table 18-12 describes the methods defined by **ProcessBuilder**.

Method	Description
List<String> command( )	Returns a reference to a **List** that contains the name of the program and its arguments. Changes to this list affect the invoking object.
ProcessBuilder command(List<String> *args*)	Sets the name of the program and its arguments to those specified by *args*. Changes to this list affect the invoking object. Returns a reference to the invoking object.
ProcessBuilder command(String ... *args*)	Sets the name of the program and its arguments to those specified by *args*. Returns a reference to the invoking object.
File directory( )	Returns the current working directory of the invoking object. This value will be **null** if the directory is the same as that of the Java program that started the process.
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.
static List<Process> startPipeline(List<ProcessBuilder> *pbList*) *throws IOException*	Pipelines the processes in *pbList*.

Table 18-12 The Methods Defined by **ProcessBuilder**

In Table 18-12, notice the methods that use the **ProcessBuilder.Redirect** class. This abstract class encapsulates an I/O source or target linked to a subprocess. Among other things, these methods enable you to redirect the source or target of I/O operations. For example, you can redirect to a file by calling **to( )**, redirect from a file by calling **from( )**, and append to a file by calling **appendTo( )**. A **File** object linked to the file can be obtained by calling **file( )**. These methods are shown here:

```
static ProcessBuilder.Redirect to(File f)
static ProcessBuilder.Redirect from(File f)
static ProcessBuilder.Redirect appendTo(File f)
File file( )
```

Another method supported by **ProcessBuilder.Redirect** is **type( )**, which returns a value of the enumeration type **ProcessBuilder.Redirect.Type**. This enumeration describes the type of the redirection. It defines these values:

APPEND, **INHERIT**, **PIPE**, **READ**, or **WRITE**. **ProcessBuilder.Redirect** also defines the constants **INHERIT**, **PIPE,** and **DISCARD.**

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 18-13. Many of the methods throw a **SecurityException** if the operation is not permitted by the security manager.

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.
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 System.Logger getLogger(String *logName)*	Returns a reference to an object that can be used for program logging. The name of the logger is passed in *logName*.
static System.Logger getLogger(String *logName*, ResourceBundle *rb*)	Returns a reference to an object that can be used for program logging. The name of the logger is passed in *logName*. Localization is supported by the resource bundle passed in *rb*.
static Properties getProperties( )	Returns the properties associated with the Java run-time system. (The **Properties** class is described in Chapter 19.)

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 18-13 The Methods Defined by **System**

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

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 = MMMMMMMMMM
a = ABCDEFGHIJ
b = ABCDEFGHIJ
a = AABCDEFGHI
b = BCDEFGHIJJ
```

Environment Properties

The following properties are available in all cases:

file.separator	java.vendor	java.vm.version
java.class.path	java.vendor.url	line.separator
java.class.version	java.vendor.version	os.arch
java.compiler	java.version	os.name
java.home	java.version.date	os.version
java.io.tmpdir	java.vm.name	path.separator
java.library.path	java.vm.specification.name	user.dir
java.specification.name	java.vm.specification.vendor	user.home
java.specification.vendor	java.vm.specification.version	user.name
java.specification.version	java.vm.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"));
  }
}
```

System.Logger and System.LoggerFinder

The **System.Logger** interface and **System.LoggerFinder** class support a program log. A logger can be found by use of **System.getLogger( )**. **System.Logger** provides the interface to the logger.

Object

As mentioned in Part I, **Object** is a superclass of all other classes. **Object** defines the methods shown in Table 18-14, which are available to every object.

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. (Deprecated by JDK 9.)
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( )	Notifies a thread waiting on the invoking object.
final void notifyAll( )	Notifies 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 18-14 The Methods Defined by **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 an exact 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 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 18-15. In the table, notice the **getModule( )** method. It is part of the support for the modules feature added by JDK 9.

Method	Description
static Class<?> forName(Module *mod*, String *name*)	Returns a **Class** object corresponding to its complete name and the module in which is resides.
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.
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.
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.)
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.

Module getModule( )	Returns a **Module** object that represents the module in which the invoking class type resides.
String getName( )	Returns the complete name of the class or interface of the type represented by the invoking object.
String getPackageName( )	Returns the name of the package of which the invoking class type is a part.
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**.
String toString( )	Returns the string representation of the type represented by the invoking object or interface.

Table 18-15 A Sampling of Methods Defined by **Class**

The methods defined by **Class** are often useful in situations where run-time type information about an object is required. As Table 18-15 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
```

Before moving on, it is useful to mention a new **Class** capability that you may find interesting. Beginning with JDK 11, **Class** provides three methods that relate to a nest. A *nest* is a group of classes and/or interfaces nested within an outer class or interface. The nest concept enables the JVM to more efficiently handle certain situations involving access between nest members. It is important to state that a nest is *not* a source code mechanism, and it does *not* change the Java language or how it defines accessibility. Nests relate specifically to how the

compiler and JVM work. However, it is now possible to obtain a nest's top-level class/interface, which is called the *nest host*, by use of **getNestHost( )**. You can determine if one class/interface is a member of the same nest as another by use of **isNestMateOf( )**. Finally, you can get an array containing a list of the nest members by calling **getNestMembers( )**. You may find these methods useful when using reflection, for example.

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 18-16. 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*.
static long floorDiv(long *dividend*, int *divisor*)	Returns the floor of the result of *dividend/divisor*.
static long floorDiv(long *dividend*, long *divisor*)	Returns the floor of the result of *dividend/divisor*.
static int floorMod(int *dividend*, int *divisor*)	Returns the floor of the remainder of *dividend/divisor*.
static int floorMod(long *dividend*, int *divisor*)	Returns the floor of the remainder of *dividend/divisor*.
static long floorMod(long *dividend*, long *divisor*)	Returns the floor of the remainder of *dividend/divisor*.

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*.
static float nextDown(float *val*)	Returns the next value lower than *val*.
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 18-16 The Rounding Methods Defined by **Math**

Miscellaneous Math Methods

In addition to the methods just shown, **Math** defines several other methods, which are shown in Table 18-17. Notice that several of the methods use the suffix **Exact**. 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.
static long addExact(long *arg1*, long *arg2*	Returns *arg1* + *arg2*. Throws an **ArithmeticException** if overflow occurs.
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.
static long decrementExact(long *arg*)	Returns *arg* – 1. Throws an **ArithmeticException** if overflow occurs.
static double fma(double *arg1*, double *arg2*, double *arg3*)	Adds *arg3* to the product of *arg1* and *arg2* and returns the rounded result. The name is short for fused multiply add.
static float fma(float *arg1*, float *arg2*, float *arg3*)	Adds *arg3* to the product of *arg1* and *arg2* and returns the rounded result. The name is short for fused multiply add.
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.
static long incrementExact(long *arg*)	Returns *arg* + 1. Throws an **ArithmeticException** if overflow occurs.

static int multiplyExact(int *arg1*, int *arg2*)	Returns *arg1* * *arg2*. Throws an **ArithmeticException** if overflow occurs.
static long multiplyExact(long *arg1*, int *arg2*)	Returns *arg1* * *arg2*. Throws an **ArithmeticException** if overflow occurs.
static long multiplyExact(long *arg1*, long *arg2*)	Returns *arg1* * *arg2*. Throws an **ArithmeticException** if overflow occurs.
static long multiplyFull(int *arg1*, int *arg2*)	Returns *arg1* * *arg2* as a **long** value.
static long multiplyHigh(long *arg1*, long *arg2*)	Returns a **long** value that contains the most significant bits of *arg1* * *arg2*.
static int negateExact(int *arg*)	Returns –*arg*. Throws an **ArithmeticException** if overflow occurs.
static long negateExact(long *arg*)	Returns –*arg*. Throws an **ArithmeticException** if overflow occurs.
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.
static long subtractExact(long *arg1*, long *arg2*)	Returns *arg1* – *arg2*. Throws an **ArithmeticException** if overflow occurs.
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.
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 18-17 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 and has been deprecated by JDK 9.

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 a number of constructors. Several are shown here:

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, by default 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 non-deprecated methods defined by **Thread** are shown in Table 18-18. **Thread** also includes the deprecated 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. Furthermore, beginning with JDK 11, **destroy( )** and one version of **stop( )** have now been removed from **Thread**.

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**.
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.
static void onSpinWait( )	Called to signify that execution is currently inside a wait loop, possibly enabling a runtime optimization.
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*.
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 18-18 The Methods Defined by **Thread**

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 18-19.

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.
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 18-19 The Methods Defined by **ThreadGroup**

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;
  }

  // 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);

    ob1.start();
    ob2.start();
    ob3.start();
    ob4.start();

    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.

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 information about a package. The methods defined by **Package** are shown in Table 18-20. The following program demonstrates **Package**, displaying the packages about which the program currently is aware:

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.
<A extends Annotation> A getDeclaredAnnotation(Class<A> *annoType*)	Returns an **Annotation** object that contains the non-inherited annotation associated with *annoType*.
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.
String getImplementationTitle( )	Returns the title of the invoking package.
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[ ] 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 18-20 The Methods Defined by **Package**

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

Module

Added by JDK 9, the **Module** class encapsulates a module. Using a **Module** instance you can add various access rights to a module, determine access rights, or obtain information about a module. For example, to export a package to a specified module, call **addExports( )**; to open a package to a specified module, call **addOpens( )**; to read another module, call **addReads( )**; and to add a service requirement, call **addUses( )**. You can determine if a module can access another by calling **canRead( )**. To determine if a module uses a service, call **canUse( )**. Although these methods will be most useful in specialized situations, **Module** defines several others that may be of more general interest.

For example, you can obtain the name of a module by calling **getName( )**. If called from within a named module, the name is returned. If called from the unnamed module, **null** is returned. You can obtain a **Set** of the packages in a module by calling **getPackages( )**. A module descriptor, in the form of a **ModuleDescriptor** instance, is returned by **getDescriptor( )**. (**ModuleDescriptor** is a class declared in **java.lang.module**.) You can determine if a package is exported or opened by the invoking module by calling **isExported( )** or **isOpen( )**. Use **isNamed( )** to determine if a module is named or unnamed. Other methods include **getAnnotation( )**,

getDeclaredAnnotations( ), **getLayer( )**, **getClassLoader( )**, and **getResourceAsStream( )**. The **toString( )** method is also overridden for **Module**.

Assuming the modules defined by the examples in Chapter 16, you can easily experiment with the **Module** class. For example, try adding the following lines to the **MyModAppDemo** class:

```
Module myMod = MyModAppDemo.class.getModule();
System.out.println("Module is " + myMod.getName());

System.out.print("Packages: ");
for(String pkg : myMod.getPackages()) System.out.println(pkg + "
");
```

Here, the methods **getName( )** and **getPackages( )** are used. Notice that a **Module** instance is obtained by calling **getModule( )** on the **Class** instance for **MyModAppDemo**. When run, these lines produce the following output:

```
Module is appstart
Packages: appstart.mymodappdemo
```

ModuleLayer

ModuleLayer, added by JDK 9, encapsulates a module layer. The nested class **ModuleLayer.Controller**, also added by JDK 9, is the controller for a module layer. In general, these classes are for specialized applications.

RuntimePermission

RuntimePermission relates to Java's security mechanism.

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. Beginning with JDK 9, module information is also included. **StackTraceElement** defines two constructors, but typically you won't need to use them because an array of **StackTraceElement**s is returned by various methods, such as the **getStackTrace( )** method of the **Throwable** and **Thread** classes.

The methods supported by **StackTraceElement** are shown in Table 18-21. 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 getClassLoaderName( )	Returns the name of the class loader used to load the class in which the execution point described by the invoking **StackTraceElement** occurred. If the object does not include class loader information, **null** is returned.
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.
String getModuleName( )	Returns the name of the module in which the execution point described by the invoking **StackTraceElement** occurred. If the object does not include module information, **null** is returned.
String getModuleVersion( )	Returns the version of the module in which the execution point described by the invoking **StackTraceElement** occurred. If the object does not include module information, **null** is returned.
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 18-21 The Methods Defined by **StackTraceElement**

StackWalker and StackWalker.StackFrame

Added by JDK 9, the **StackWalker** class and the **StackWalker.StackFrame** interface support stack walking operations. A **StackWalker** instance is obtained by use of the static **getInstance( )** method defined by **StackWalker**. Stack walking is initiated by calling the **walk( )** method of **StackWalker**. Each stack frame is encapsulated as a **StackWalker.StackFrame** object.

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 18-22.

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 18-22 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 18-23. This interface is implemented by **String**, **StringBuffer**, and **StringBuilder**, among others.

Method	Description
char charAt(int *idx*)	Returns the character at the index specified by *idx*.
static int compare(CharSequence *seqA*, CharSequence *seqB*)	Compares *seqA* to *seqB*. Returns 0 if the sequences are the same. Returns a negative value if *seqA* is less than *seqB*. Returns a positive value if *seqA* is greater than *seqB*. (Added by JDK 11.)
default IntStream chars( )	Returns a stream (in the form of an **IntStream**) to the characters in the invoking object.
default IntStream codePoints( )	Returns a stream (in the form of an **IntStream**) to the code points in the invoking object.
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 18-23 The Methods Defined by **CharSequence**

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.

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, such as **Byte**, **Character**, **Double**, **Float**, **Long**, **Short**, **String**, **Integer**, and **Enum**.

The Appendable Interface

An object 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.

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 defined in **java.util.function**. See Chapter 20.)

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

```
default Spliterator<T> spliterator( )
```

It returns a **Spliterator** to the sequence being iterated. (See Chapters 19 and 29 for details on spliterators.)

NOTE Iterators are described in detail in Chapter 19.

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**. Except as otherwise noted, these packages are in the **java.base** module.

- java.lang.annotation
- java.lang.instrument
- java.lang.invoke
- java.lang.management
- java.lang.module
- 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. This package is in the **java.instrument** module.

java.lang.invoke

java.lang.invoke supports dynamic language features. 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. This package is in the **java.management** module.

java.lang.module

The **java.lang.module** package supports modules. It includes classes such as **ModuleDescriptor** and **ModuleReference**, and the interfaces **ModuleFinder** and **ModuleReader**.

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

19

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. Beginning with JDK 9, **java.util** is part of the **java.base** module.

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	SimpleTimeZone
Arrays	LinkedHashMap	Spliterators
Base64	LinkedHashSet	SplitableRandom
BitSet	LinkedList	Stack

Calendar	ListResourceBundle	StringJoiner
Collections	Locale	StringTokenizer
Currency	LongSummaryStatistics	Timer
Date	Objects	TimerTask
Dictionary	Observable (Deprecated by JDK 9.)	TimeZone
DoubleSummaryStatistics	Optional	TreeMap
EnumMap	OptionalDouble	TreeSet
EnumSet	OptionalInt	UUID
EventListenerProxy	OptionalLong	Vector
EventObject	PriorityQueue	WeakHashMap

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

Collection	Map.Entry	ServiceLoader.Provider
Comparator	NavigableMap	Set
Deque	NavigableSet	SortedMap
Enumeration	Observer (Deprecated by JDK 9.)	SortedSet
EventListener	PrimitiveIterator	Spliterator
Formattable	PrimitiveIterator.OfDouble	Spliterator.OfDouble
Iterator	PrimitiveIterator.OfInt	Spliterator.OfInt
List	PrimitiveIterator.OfLong	Spliterator.OfLong
ListIterator	Queue	Spliterator.OfPrimitive
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 20 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 were 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 added 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 added 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.

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**, **ListIterator**, and **Spliterator** interfaces, which are described in depth later in this chapter. 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 19-1. 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.
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*.
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.

default Stream<E> stream( )	Returns a stream that uses the invoking collection as its source for elements. The stream is sequential.
default <T> T[ ] toArray(IntFunction<T[ ]> *arrayGen*)	Returns an array of the elements from the invoking collection. The returned array is created by the function specified by *arrayGen*. An **ArrayStoreException** is thrown if any collection element has a type that is not compatible with the array type. (Added by JDK 11.)
Object[ ] toArray( )	Returns an array of the elements from the invoking collection.
<T> T[ ] toArray(T *array*[ ])	Returns an array of the elements from the invoking collection. 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 compatible with the array type.

Table 19-1 The Methods Declared by **Collection**

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

You can remove an object by using **remove( )**. To remove a group of objects, call **removeAll( )**. You can remove all elements except those of a specified group by calling **retainAll( )**. To remove an element only if it statisfies some condition, you can use **removeIf( )**. 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 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. Beginning with JDK 11, a third form has been added that lets you specify a function that obtains the array. 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 **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 **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 19-2. 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.

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.
static <E> List<E> copyOf(Collection<? extends E> *from*)	Returns a list that contains the same elements as that specified by *from*. The returned list is unmodifiable. Null values are not allowed.
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*.
static <E> List<E> of(*parameter-list*)	Creates an unmodifiable list containing the elements specified in *parameter-list*. Null elements are not allowed. Many overloaded versions are provided. See the discussion in the text for details.
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.
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*.
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 19-2 The Methods Declared by **List**

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

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**.

Beginning with JDK 9, **List** includes the **of( )** factory method, which has a number of overloads. Each version returns an unmodifiable, value-based collection that is comprised of the arguments that it is passed. The primary purpose of **of( )** is to provide a convenient, efficient way to create a small **List** collection. There are 12 overloads of **of( )**. One takes no arguments and creates an empty list. It is shown here:

```
static <E> List<E> of( )
```

Ten overloads take from 1 to 10 arguments and create a list that contains the specified elements. They are shown here:

```
static <E> List<E> of(E obj1)
static <E> List<E> of(E obj1, E obj2)
static <E> List<E> of(E obj, E obj2, E obj3)
...
static <E> List<E> of(E ob1, E obj2, E obj3, E obj4, E obj5,
                      E obj6, E obj7, E obj8, E obj9, E obj10)
```

The final **of( )** overload specifies a varargs parameter that takes an arbitrary number of elements or an array of elements. It is shown here:

```
static <E> List<E> of(E ... objs)
```

For all versions, **null** elements are not allowed. In all cases, the **List** implementation is unspecified.

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 **false** if an attempt is made to add duplicate elements to a set. With two exceptions, 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.

Beginning with JDK 9, **Set** includes the **of( )** factory method, which has a number of overloads. Each version returns an unmodifiable, value-based collection that is comprised of the arguments that it is passed. The primary purpose of **of( )** is to provide a convenient, efficient way to create a small **Set** collection. There are 12 overloads of **of( )**. One takes no arguments and creates an empty set. It is shown here:

static <E> Set<E> of( )

Ten overloads take from 1 to 10 arguments and create a list that contains the specified elements. They are shown here:

static <E> Set<E> of(E *obj1*)
static <E> Set<E> of(E *obj1*, E *obj2*)
static <E> Set<E> of(E *obj*, E *obj2*, E *obj3*)
...
static <E> Set<E> of(E *ob1*, E *obj2*, E *obj3*, E *obj4*, E *obj5*,
E *obj6*, E *obj7*, E *obj8*, E *obj9*, E *obj10*)

The final **of( )** overload specifies a varargs parameter that takes an arbitrary number of elements or an array of elements. It is shown here:

static <E> Set<E> of(E ... *objs*)

For all versions, **null** elements are not allowed. In all cases, the **Set** implementation is unspecified.

Beginning with JDK 10, **Set** includes the static **copyOf( )** method shown

here:

static <E> Set<E> copyOf(Collection <? extends E> *from*)

It returns a set that contains the same elements as *from*. Null values are not allowed. The returned set is unmodifiable.

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 19-3. 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.

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 19-3 The Methods Declared by **SortedSet**

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( )**.

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 19-4. A **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.

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 smallest 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 19-4 The Methods Declared by **NavigableSet**

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 19-5.

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 19-5 The Methods Declared by **Queue**

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.

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 19-6. Several 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.

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.

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 19-6 The Methods Declared by **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

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

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

```
object[ ] toArray( )
<T> T[ ] toArray(T array[ ])
default <T> T[ ] toArray(IntFunction<T[ ]> arrayGen)
```

The first returns an array of **Object**. The second and third forms return an array of elements that have the same type as **T**. Here, we will use the second form because of its convenience. The following program shows it in action.

```
// 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);
```

```
    // Remove first and last elements.
    ll.removeFirst();
    ll.removeLast();

    System.out.println("ll after deleting first and last: "
                       + ll);

    // Get and set a value.

    String val = ll.get(2);
    ll.set(2, val + " Changed");

    System.out.println("ll after change: " + ll);
  }
}
```

The output from this program is shown here:

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

Because **LinkedList** implements the **List** interface, calls to **add(E)** append items to the end of the list, as do calls to **addLast( )**. To insert items at a specific location, use the **add(int, E)** form of **add( )**, as illustrated by the call to **add(1, "A2")** in the example.

Notice how the third element in **ll** is changed by employing calls to **get( )** and **set( )**. To obtain the current value of an element, pass **get( )** the index at which the element is stored. To assign a new value to that index, pass **set( )** the index and its new value.

The HashSet Class

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. **HashSet** is a generic class that has this declaration:

class HashSet<E>

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

As most readers likely know, a hash table stores information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its *hash code*. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can't directly index the hash table. The advantage of hashing is that it allows the execution time of **add( )**, **contains( )**, **remove( )**, and **size( )** to remain constant even for large sets.

The following constructors are defined:

HashSet( )
HashSet(Collection<? extends E> c)
HashSet(int *capacity*)
HashSet(int *capacity*, float *fillRatio*)

The first form constructs a default hash set. The second form initializes the hash set by using the elements of *c*. The third form initializes the capacity of the hash set to *capacity*. (The default capacity is 16.) The fourth form initializes both the capacity and the fill ratio (also called *load factor*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used.

HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

It is important to note that **HashSet** does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as **TreeSet**, is a better choice.

Here is an example that demonstrates **HashSet**:

```
// Demonstrate HashSet.
import java.util.*;

class HashSetDemo {
  public static void main(String args[]) {
    // Create a hash set.
    HashSet<String> hs = new HashSet<String>();

    // Add elements to the hash set.
    hs.add("Beta");
    hs.add("Alpha");
    hs.add("Eta");
    hs.add("Gamma");
    hs.add("Epsilon");
    hs.add("Omega");

    System.out.println(hs);
  }
}
```

The following is the output from this program:

```
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
```

As explained, the elements are not stored in sorted order, and the precise output may vary.

The LinkedHashSet Class

The **LinkedHashSet** class extends **HashSet** and adds no members of its own. It is a generic class that has this declaration:

class LinkedHashSet<E>

Here, **E** specifies the type of objects that the set will hold. Its constructors parallel those in **HashSet**.

LinkedHashSet maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set. That is, when cycling through a **LinkedHashSet** using an iterator, the elements will be returned in the order in which they were inserted. This is also the order in which they are contained in the string returned by **toString( )** when called on a

LinkedHashSet object. To see the effect of **LinkedHashSet**, try substituting **LinkedHashSet** for **HashSet** in the preceding program. The output will be

```
[Beta, Alpha, Eta, Gamma, Epsilon, Omega]
```

which is the order in which the elements were inserted.

The TreeSet Class

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

class TreeSet<E>

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

TreeSet has the following constructors:

TreeSet( )
TreeSet(Collection<? extends E> *c*)
TreeSet(Comparator<? super E> *comp*)
TreeSet(SortedSet<E> *ss*)

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements. The second form builds a tree set that contains the elements of *c*. The third form constructs an empty tree set that will be sorted according to the comparator specified by *comp*. (Comparators are described later in this chapter.) The fourth form builds a tree set that contains the elements of *ss*.

Here is an example that demonstrates a **TreeSet**:

```
// Demonstrate TreeSet.
import java.util.*;

class TreeSetDemo {
  public static void main(String args[]) {
    // Create a tree set.
    TreeSet<String> ts = new TreeSet<String>();

    // Add elements to the tree set.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
    System.out.println(ts);
  }
}
```

The output from this program is shown here:

```
[A, B, C, D, E, F]
```

As explained, because **TreeSet** stores its elements in a tree, they are automatically arranged in sorted order, as the output confirms.

Because **TreeSet** implements the **NavigableSet** interface, you can use the methods defined by **NavigableSet** to retrieve elements of a **TreeSet**. For example, assuming the preceding program, the following statement uses **subSet( )** to obtain a subset of **ts** that contains the elements between **C** (inclusive) and **F** (exclusive). It then displays the resulting set.

```
System.out.println(ts.subSet("C", "F"));
```

The output from this statement is shown here:

```
[C, D, E]
```

You might want to experiment with the other methods defined by **NavigableSet**.

The PriorityQueue Class

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface. It

creates a queue that is prioritized based on the queue's comparator.
PriorityQueue is a generic class that has this declaration:

class PriorityQueue<E>

Here, **E** specifies the type of objects stored in the queue. **PriorityQueue**s are dynamic, growing as necessary.

PriorityQueue defines the seven constructors shown here:

PriorityQueue( )
PriorityQueue(int *capacity*)
PriorityQueue(Comparator<? super E> *comp*) PriorityQueue(int *capacity*,
Comparator<? super E> *comp*)
PriorityQueue(Collection<? extends E> *c*)
PriorityQueue(PriorityQueue<? extends E> *c*)
PriorityQueue(SortedSet<? extends E> *c*)

The first constructor builds an empty queue. Its starting capacity is 11. The second constructor builds a queue that has the specified initial capacity. The third constructor specifies a comparator, and the fourth builds a queue with the specified capacity and comparator. The last three constructors create queues that are initialized with the elements of the collection passed in *c*. In all cases, the capacity grows automatically as elements are added.

If no comparator is specified when a **PriorityQueue** is constructed, then the default comparator for the type of data stored in the queue is used. The default comparator will order the queue in ascending order. Thus, the head of the queue will be the smallest value. However, by providing a custom comparator, you can specify a different ordering scheme. For example, when storing items that include a time stamp, you could prioritize the queue such that the oldest items are first in the queue.

You can obtain a reference to the comparator used by a **PriorityQueue** by calling its **comparator( )** method, shown here:

Comparator<? super E> comparator( )

It returns the comparator. If natural ordering is used for the invoking queue, **null** is returned.

One word of caution: Although you can iterate through a **PriorityQueue** using an iterator, the order of that iteration is undefined. To properly use a **PriorityQueue**, you must call methods such as **offer( )** and **poll( )**, which are

defined by the **Queue** interface.

The ArrayDeque Class

The **ArrayDeque** class extends **AbstractCollection** and implements the **Deque** interface. It adds no methods of its own. **ArrayDeque** creates a dynamic array and has no capacity restrictions. (The **Deque** interface supports implementations that restrict capacity, but does not require such restrictions.) **ArrayDeque** is a generic class that has this declaration:

```
class ArrayDeque<E>
```

Here, **E** specifies the type of objects stored in the collection.

ArrayDeque defines the following constructors:

```
ArrayDeque( )
ArrayDeque(int size)
ArrayDeque(Collection<? extends E> c)
```

The first constructor builds an empty deque. Its starting capacity is 16. The second constructor builds a deque that has the specified initial capacity. The third constructor creates a deque that is initialized with the elements of the collection passed in *c*. In all cases, the capacity grows as needed to handle the elements added to the deque.

The following program demonstrates **ArrayDeque** by using it to create a stack:

```
// Demonstrate ArrayDeque.
import java.util.*;

class ArrayDequeDemo {
  public static void main(String args[]) {
    // Create an array deque.
    ArrayDeque<String> adq = new ArrayDeque<String>();

    // Use an ArrayDeque like a stack.
    adq.push("A");
    adq.push("B");
    adq.push("D");
    adq.push("E");
    adq.push("F");
    System.out.print("Popping the stack: ");

    while(adq.peek() != null)
      System.out.print(adq.pop() + " ");

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

The output is shown here:

```
Popping the stack: F E D B A
```

The EnumSet Class

EnumSet extends **AbstractSet** and implements **Set**. It is specifically for use with elements of an **enum** type. It is a generic class that has this declaration:

class EnumSet<E extends Enum<E>>

Here, **E** specifies the elements. Notice that **E** must extend **Enum<E>**, which enforces the requirement that the elements must be of the specified **enum** type.

EnumSet defines no constructors. Instead, it uses the factory methods shown in Table 19-7 to create objects. All methods can throw **NullPointerException**. The **copyOf( )** and **range( )** methods can also throw **IllegalArgumentException**. Notice that the **of( )** method is overloaded a number of times. This is in the interest of efficiency. Passing a known number of

arguments can be faster than using a vararg parameter when the number of arguments is small.

Method	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements in the enumeration specified by *t*.
static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*)	Creates an **EnumSet** that is comprised of those elements not stored in *e*.
static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> *c*)	Creates an **EnumSet** from the elements stored in *c*.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> *t*)	Creates an **EnumSet** that contains the elements that are not in the enumeration specified by *t*, which is an empty set by definition.
static <E extends Enum<E>> EnumSet<E> of(E *v*, E ... *varargs*)	Creates an **EnumSet** that contains *v* and zero or more additional enumeration values.
static <E extends Enum<E>> EnumSet<E> of(E *v*)	Creates an **EnumSet** that contains *v*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*)	Creates an **EnumSet** that contains *v1* and *v2*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*)	Creates an **EnumSet** that contains *v1* through *v3*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*)	Creates an **EnumSet** that contains *v1* through *v4*.
static <E extends Enum<E>> EnumSet<E> of(E *v1*, E *v2*, E *v3*, E *v4*, E *v5*)	Creates an **EnumSet** that contains *v1* through *v5*.
static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*)	Creates an **EnumSet** that contains the elements in the range specified by *start* and *end*.

Table 19-7 The Methods Declared by **EnumSet**

Accessing a Collection via an Iterator

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element. One way to do this is to employ an *iterator*, which is an object that implements either the **Iterator** or the

ListIterator interface. **Iterator** enables you to cycle through a collection, obtaining or removing elements. **ListIterator** extends **Iterator** to allow bidirectional traversal of a list, and the modification of elements. **Iterator** and **ListIterator** are generic interfaces which are declared as shown here:

```
    interface Iterator<E>
interface ListIterator<E>
```

Here, **E** specifies the type of objects being iterated. The **Iterator** interface declares the methods shown in Table 19-8. The methods declared by **ListIterator** (along with those inherited from **Iterator**) are shown in Table 19-9. In both cases, operations that modify the underlying collection are optional. For example, **remove( )** will throw **UnsupportedOperationException** when used with a read-only collection. Various other exceptions are possible.

Method	Description
default void forEachRemaining(Consumer<? super E> *action*)	The action specified by *action* is executed on each unprocessed element in the collection.
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.

Table 19-8 The Methods Declared by **Iterator**

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

Table 19-9 The Methods Provided by **ListIterator**

NOTE Beginning with JDK 8, you can also use a **Spliterator** to cycle through a collection. **Spliterator** works differently than does **Iterator**, and it is described later in this chapter.

Using an Iterator

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

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
2. Set up a loop that makes a call to **hasNext( )**. Have the loop iterate as long

as **hasNext( )** returns **true**.

3. Within the loop, obtain each element by calling **next( )**.

For collections that implement **List**, you can also obtain an iterator by calling **listIterator( )**. As explained, a list iterator gives you the ability to access the collection in either the forward or backward direction and lets you modify an element. Otherwise, **ListIterator** is used just like **Iterator**.

The following example implements these steps, demonstrating both the **Iterator** and **ListIterator** interfaces. It uses an **ArrayList** object, but the general principles apply to any type of collection. Of course, **ListIterator** is available only to those collections that implement the **List** interface.

```
// Demonstrate iterators.
import java.util.*;

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

    // Add elements to the array list.
    al.add("C");
    al.add("A");
    al.add("E");
    al.add("B");
    al.add("D");
    al.add("F");

    // Use iterator to display contents of al.
    System.out.print("Original contents of al: ");
    Iterator<String> itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Modify objects being iterated.
    ListIterator<String> litr = al.listIterator();
    while(litr.hasNext()) {
      String element = litr.next();
      litr.set(element + "+");
    }

    System.out.print("Modified contents of al: ");
    itr = al.iterator();
    while(itr.hasNext()) {
      String element = itr.next();
      System.out.print(element + " ");
    }
    System.out.println();

    // Now, display the list backwards.
    System.out.print("Modified list backwards: ");
    while(litr.hasPrevious()) {
      String element = litr.previous();
      System.out.print(element + " ");
    }
    System.out.println();
  }
}
```

The output is shown here:

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

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

The For-Each Alternative to Iterators

If you won't be modifying the contents of a collection or obtaining elements in reverse order, then the for-each version of the **for** loop is often a more convenient alternative to cycling through a collection than is using an iterator. Recall that the **for** can cycle through any collection of objects that implement the **Iterable** interface. Because all of the collection classes implement this interface, they can all be operated upon by the **for**.

The following example uses a **for** loop to sum the contents of a collection:

```
// Use the for-each for loop to cycle through a collection.
import java.util.*;

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

    // Add values to the array list.
    vals.add(1);
    vals.add(2);
    vals.add(3);
    vals.add(4);
    vals.add(5);

    // Use for loop to display the values.
    System.out.print("Contents of vals: ");
    for(int v : vals)
      System.out.print(v + " ");

    System.out.println();

    // Now, sum the values by using a for loop.
    int sum = 0;
    for(int v : vals)
      sum += v;

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

The output from the program is shown here:

```
Contents of vals: 1 2 3 4 5
Sum of values: 15
```

As you can see, the **for** loop is substantially shorter and simpler to use than the iterator-based approach. However, it can only be used to cycle through a collection in the forward direction, and you can’t modify the contents of the collection.

Spliterators

JDK 8 added another type of iterator called a *spliterator* that is defined by the **Spliterator** interface. A spliterator cycles through a sequence of elements, and in this regard, it is similar to the iterators just described. However, the techniques required to use it differ. Furthermore, it offers substantially more functionality than does either **Iterator** or **ListIterator**. Perhaps the most important aspect of **Spliterator** is its ability to provide support for parallel iteration of portions of the sequence. Thus, **Spliterator** supports parallel programming. (See Chapter 28 for information on concurrency and parallel programming.) However, you can use **Spliterator** even if you won't be using parallel execution. One reason you might want to do so is because it offers a streamlined approach that combines the *hasNext* and *next* operations into one method.

Spliterator is a generic interface that is declared like this:

interface Spliterator<T>

Here, **T** is the type of elements being iterated. **Spliterator** declares the methods shown in Table 19-10.

Method	Description
int characteristics( )	Returns the characteristics of the invoking spliterator, encoded into an integer.
long estimateSize( )	Estimates the number of elements left to iterate and returns the result. Returns **Long.MAX_VALUE** if the count cannot be obtained for any reason.
default void forEachRemaining(Consumer<? super T> *action*)	Applies *action* to each unprocessed element in the data source.
default Comparator<? super T> getComparator( )	Returns the comparator used by the invoking spliterator or **null** if natural ordering is used. If the sequence is unordered, **IllegalStateException** is thrown.
default long getExactSizeIfKnown( )	If the invoking spliterator is sized, returns the number of elements left to iterate. Returns –1 otherwise.
default boolean hasCharacteristics(int *val*)	Returns **true** if the invoking spliterator has the characteristics passed in *val*. Returns **false** otherwise.
boolean tryAdvance(Consumer<? super T> *action*)	Executes *action* on the next element in the iteration. Returns **true** if there is a next element. Returns **false** if no elements remain.
Spliterator<T> trySplit( )	If possible, splits the invoking spliterator, returning a reference to a new spliterator for the partition. Otherwise, returns **null**. Thus, if successful, the original spliterator iterates over one portion of the sequence and the returned spliterator iterates over the other portion.

Table 19-10 The Methods Declared by **Spliterator**

Using **Spliterator** for basic iteration tasks is quite easy: simply call **tryAdvance( )** until it returns **false**. If you will be applying the same action to each element in the sequence, **forEachRemaining( )** offers a streamlined alternative. In both cases, the action that will occur with each iteration is defined by what the **Consumer** object does with each element. **Consumer** is a functional interface that applies an action to an object. It is a generic functional interface declared in **java.util.function**. (See Chapter 20 for information on **java.util.function**.) **Consumer** specifies only one abstract method, **accept( )**, which is shown here:

void accept(T *objRef*)

In the case of **tryAdvance( )**, each iteration passes the next element in the sequence to *objRef*. Often, the easiest way to implement **Consumer** is by use of a lambda expression.

The following program provides a simple example of **Spliterator**. Notice that the program demonstrates both **tryAdvance( )** and **forEachRemaining( )**. Also notice how these methods combine the actions of **Iterator**'s **next( )** and **hasNext( )** methods into a single call.

```
// A simple Spliterator demonstration.
import java.util.*;

class SpliteratorDemo {
public static void main(String args[]) {
  // Create an array list for doubles.
  ArrayList<Double> vals = new ArrayList<>();

  // Add values to the array list.
  vals.add(1.0);
  vals.add(2.0);
  vals.add(3.0);
  vals.add(4.0);
  vals.add(5.0);

  // Use tryAdvance() to display contents of vals.
  System.out.print("Contents of vals:\n");
  Spliterator<Double> spltitr = vals.spliterator();
  while(spltitr.tryAdvance((n) -> System.out.println(n)));
  System.out.println();

  // Create new list that contains square roots.
  spltitr = vals.spliterator();
  ArrayList<Double> sqrs = new ArrayList<>();
  while(spltitr.tryAdvance((n) -> sqrs.add(Math.sqrt(n))));

    // Use forEachRemaining() to display contents of sqrs.
    System.out.print("Contents of sqrs:\n");
    spltitr = sqrs.spliterator();
    spltitr.forEachRemaining((n) -> System.out.println(n));
    System.out.println();
  }
}
```

The output is shown here:

```
Contents of vals:
1.0
2.0
3.0
4.0
5.0

Contents of sqrs:
1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
```

Although this program demonstrates the mechanics of using **Spliterator**, it does not reveal its full power. As mentioned, **Spliterator**'s maximum benefit is found in situations that involve parallel processing.

In Table 19-10, notice the methods **characteristics( )** and **hasCharacteristics( )**. Each **Spliterator** has a set of attributes, called *characteristics*, associated with it. These are defined by static **int** fields in **Spliterator**, such as **SORTED**, **DISTINCT**, **SIZED**, and **IMMUTABLE**, to name a few. You can obtain the characteristics by calling **characteristics( )**. You can determine if a characteristic is present by calling **hasCharacteristics( )**. Often, you won't need to access a **Spliterator**'s characteristics, but in some cases, they can aid in creating efficient, resilient code.

NOTE For a further discussion of **Spliterator**, see Chapter 29, where it is used in the context of the stream API. For a discussion of lambda expressions, see Chapter 15. See Chapter 28 for a discussion of parallel programming and concurrency.

There are several nested subinterfaces of **Spliterator** designed for use with the primitive types **double**, **int**, and **long**. These are called **Spliterator.OfDouble**, **Spliterator.OfInt**, and **Spliterator.OfLong**. There is also a generalized version called **Spliterator.OfPrimitive( )**, which offers additional flexibility and serves as a superinterface of the aforementioned ones.

Storing User-Defined Classes in Collections

For the sake of simplicity, the foregoing examples have stored built-in objects, such as **String** or **Integer**, in a collection. Of course, collections are not limited to the storage of built-in objects. Quite the contrary. The power of collections is that they can store any type of object, including objects of classes that you

create. For example, consider the following example that uses a **LinkedList** to store mailing addresses:

```
// A simple mailing list example.
import java.util.*;

class Address {
  private String name;
  private String street;
  private String city;
  private String state;
  private String code;

  Address(String n, String s, String c,
          String st, String cd) {

    name = n;
    street = s;
    city = c;
    state = st;
    code = cd;
  }

  public String toString() {
    return name + "\n" + street + "\n" +
           city + " " + state + " " + code;
  }
}

class MailList {
  public static void main(String args[]) {
    LinkedList<Address> ml = new LinkedList<Address>();

    // Add elements to the linked list.
    ml.add(new Address("J.W. West", "11 Oak Ave",
                       "Urbana", "IL", "61801"));
    ml.add(new Address("Ralph Baker", "1142 Maple Lane",
                       "Mahomet", "IL", "61853"));
    ml.add(new Address("Tom Carlton", "867 Elm St",
                       "Champaign", "IL", "61820"));

    // Display the mailing list.
    for(Address element : ml)
      System.out.println(element + "\n");

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

The output from the program is shown here:

```
J.W. West
11 Oak Ave
Urbana IL 61801

Ralph Baker
1142 Maple Lane
Mahomet IL 61853

Tom Carlton
867 Elm St
Champaign IL 61820
```

Aside from storing a user-defined class in a collection, another important thing to notice about the preceding program is that it is quite short. When you consider that it sets up a linked list that can store, retrieve, and process mailing addresses in about 50 lines of code, the power of the Collections Framework begins to become apparent. As most readers know, if all of this functionality had to be coded manually, the program would be several times longer. Collections offer off-the-shelf solutions to a wide variety of programming problems. You should use them whenever the situation presents itself.

The RandomAccess Interface

The **RandomAccess** interface contains no members. However, by implementing this interface, a collection signals that it supports efficient random access to its elements. Although a collection might support random access, it might not do so efficiently. By checking for the **RandomAccess** interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections. (You can use **instanceof** to determine if a class implements an interface.) **RandomAccess** is implemented by **ArrayList** and by the legacy **Vector** class, among others.

Working with Maps

A *map* is an object that stores associations between keys and values, or *key/value pairs*. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a

null key and **null** values, others cannot.

There is one key point about maps that is important to mention at the outset: they don't implement the **Iterable** interface. This means that you *cannot* cycle through a map using a for-each style **for** loop. Furthermore, you can't obtain an iterator to a map. However, as you will soon see, you can obtain a collection-view of a map, which does allow the use of either the **for** loop or an iterator.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps:

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

Each interface is examined next, in turn.

The Map Interface

The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key. **Map** is generic and is declared as shown here:

```
interface Map<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The methods declared by **Map** are summarized in Table 19-11. Several methods throw a **ClassCastException** when an object is incompatible with the elements in a map. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** is not allowed in the map. An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map. An **IllegalArgumentException** is thrown if an invalid argument is used.

Method	Description
void clear( )	Removes all key/value pairs from the invoking map.
default V compute(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	Calls *func* to construct a new value. If func returns non-**null**, the new key/value pair is added to the map, any preexisting pairing is removed, and the new value is returned. If *func* returns **null**, any preexisting pairing is removed, and **null** is returned.
default V computeIfAbsent(K *k*, Function<? super K, ? extends V> *func*)	Returns the value associated with the key *k*. Otherwise, the value is constructed through a call to *func* and the pairing is entered into the map and the constructed value is returned. If no value can be constructed, **null** is returned.
default V computeIfPresent(K *k*, BiFunction<? super K, ? super V, ? extends V> *func*)	If *k* is in the map, a new value is constructed through a call to *func* and the new value replaces the old value in the map. In this case, the new value is returned. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned.
boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.
static <K, V> Map<K, V> copyOf(Map<? extends K, ? extends V> *from*)	Returns a map that contains the same key/value pairs as that specified by *from*. The returned map is unmodifiable. Null keys or values are not allowed.
static <K, V> Map.Entry<K, V> entry(K *k*, V *v*)	Returns an unmodifiable map entry comprised of the specified key and value. A **null** key or value is not allowed.
Set<Map.Entry<K, V>> entrySet( )	Returns a **Set** that contains the entries in the map. The set contains objects of type **Map.Entry**. Thus, this method provides a set-view of the invoking map.
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map** and contains the same entries. Otherwise, returns **false**.
default void forEach(BiConsumer< ? super K, ? super V> *action*)	Executes *action* on each element in the invoking map. A **ConcurrentModificationException** will be thrown if an element is removed during the process.
V get(Object *k*)	Returns the value associated with the key *k*. Returns **null** if the key is not found.
default V getOrDefault(Object *k*, V *defVal*)	Returns the value associated with *k* if it is in the map. Otherwise, *defVal* is returned.
int hashCode( )	Returns the hash code for the invoking map.
boolean isEmpty( )	Returns **true** if the invoking map is empty. Otherwise, returns **false**.

Set<K> keySet( )	Returns a **Set** that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
default V merge(K *k*, V *v*, BiFunction<? super V, ? super V, ? extends V> *func*)	If *k* is not in the map, the pairing *k*,*v* is added to the map. In this case, *v* is returned. Otherwise, *func* returns a new value based on the old value, the key is updated to use this value, and **merge( )** returns this value. If the value returned by *func* is **null**, the existing key and value are removed from the map and **null** is returned.
static <K, V> Map<K, V> of(*parameter-list*)	Creates an unmodifiable map containing the entries specified in *parameter-list*. Null keys or values are not allowed. Many overloaded versions are provided. See the discussion in the text for details.
static <K, V> Map<K, V> ofEntries(Map.Entry<? extends K, ? extends V> ... *entries*)	Returns an unmodifiable map that contains the key/value mappings described by the entries passed in *entries*. Null keys or values are not allowed.
V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are *k* and *v*, respectively. Returns **null** if the key did not already exist. Otherwise, the previous value linked to the key is returned.
void putAll(Map<? extends K, ? extends V> *m*)	Puts all the entries from *m* into this map.
default V putIfAbsent(K *k*, V *v*)	Inserts the key/value pair into the invoking map if this pairing is not already present or if the existing value is **null**. Returns the old value. The **null** value is returned when no previous mapping exists, or the value is **null**.
V remove(Object *k*)	Removes the entry whose key equals *k*.
default boolean remove(Object *k*, Object *v*)	If the key/value pair specified by *k* and *v* is in the invoking map, it is removed and **true** is returned. Otherwise, **false** is returned.
default boolean replace(K *k*, V *oldV*, V *newV*)	If the key/value pair specified by *k* and *oldV* is in the invoking map, the value is replaced by *newV* and **true** is returned. Otherwise **false** is returned.
default V replace(K *k*, V *v*)	If the key specified by *k* is in the invoking map, its value is set to *v* and the previous value is returned. Otherwise, **null** is returned.
default void replaceAll(BiFunction< ? super K, ? super V, ? extends V> *func*)	Executes *func* on each element of the invoking map, replacing the element with the result returned by *func*. A **ConcurrentModificationException** will be thrown if an element is removed during the process.

int size( )	Returns the number of key/value pairs in the map.
Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

Table 19-11 The Methods Declared by **Map**

Maps revolve around two basic operations: **get( )** and **put( )**. To put a value into a map, use **put( )**, specifying the key and the value. To obtain a value, call **get( )**, passing the key as an argument. The value is returned.

As mentioned earlier, although part of the Collections Framework, maps are not, themselves, collections because they do not implement the **Collection** interface. However, you can obtain a collection-view of a map. To do this, you can use the **entrySet( )** method. It returns a **Set** that contains the elements in the map. To obtain a collection-view of the keys, use **keySet( )**. To get a collection-view of the values, use **values( )**. For all three collection-views, the collection is backed by the map. Changing one affects the other. Collection-views are the means by which maps are integrated into the larger Collections Framework.

Beginning with JDK 9, **Map** includes the **of( )** factory method, which has a number of overloads. Each version returns an unmodifiable, value-based map that is comprised of the arguments that it is passed. The primary purpose of **of( )** is to provide a convenient, efficient way to create a small **Map**. There are 11 overloads of **of( )**. One takes no arguments and creates an empty map. It is shown here:

```
static <K, V> Map<K, V> of( )
```

Ten overloads take from 1 to 10 arguments and create a list that contains the specified elements. They are shown here:

```
static <K, V> Map<K, V> of(K k1, V v1)
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2)
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3)
...
static <K, V> Map<K, V> of(K k1, V v1, K k2, V v2, K k3, V v3, K k4, V v4,
                           K k5, V v5, K k6, V v6, K k7, V v7, K k8, V v8,
                           K k9, V v9, K k10, V v10)
```

For all versions, **null** keys and/or values are not allowed. In all cases, the **Map** implementation is unspecified.

The SortedMap Interface

The **SortedMap** interface extends **Map**. It ensures that the entries are maintained in ascending order based on the keys. **SortedMap** is generic and is declared as shown here:

```
interface SortedMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The methods declared by **SortedMap** are summarized in Table 19-12. Several methods throw a **NoSuchElementException** when no items are in the invoking map. A **ClassCastException** is thrown when an object is incompatible with the elements in a map. A **NullPointerException** is thrown if an attempt is made to use a **null** object when **null** is not allowed in the map. An **IllegalArgumentException** is thrown if an invalid argument is used.

Method	Description
Comparator<? super K> comparator( )	Returns the invoking sorted map's comparator. If natural ordering is used for the invoking map, **null** is returned.
K firstKey( )	Returns the first key in the invoking map.
SortedMap<K, V> headMap(K *end*)	Returns a sorted map for those map entries with keys that are less than *end*.
K lastKey( )	Returns the last key in the invoking map.
SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to *start* and less than *end*.
SortedMap<K, V> tailMap(K *start*)	Returns a map containing those entries with keys that are greater than or equal to *start*.

Table 19-12 The Methods Declared by **SortedMap**

Sorted maps allow very efficient manipulations of *submaps* (in other words, subsets of a map). To obtain a submap, use **headMap( )**, **tailMap( )**, or **subMap( )**. The submap returned by these methods is backed by the invoking map. Changing one changes the other. To get the first key in the set, call **firstKey( )**. To get the last key, use **lastKey( )**.

The NavigableMap Interface

The **NavigableMap** interface extends **SortedMap** and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys. **NavigableMap** is a generic interface that has this declaration:

```
interface NavigableMap<K,V>
```

Here, **K** specifies the type of the keys, and **V** specifies the type of the values associated with the keys. In addition to the methods that it inherits from **SortedMap**, **NavigableMap** adds those summarized in Table 19-13. Several methods throw a **ClassCastException** when an object is incompatible with the keys in the map. A **NullPointerException** is thrown if an attempt is made to use a **null** object and **null** keys are not allowed in the set. An **IllegalArgumentException** is thrown if an invalid argument is used.

Method	Description
Map.Entry<K,V> ceilingEntry(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K ceilingKey(K *obj*)	Searches the map for the smallest key *k* such that *k* >= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableSet<K> descendingKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map in reverse order. Thus, it returns a reverse set-view of the keys. The resulting set is backed by the map.
NavigableMap<K,V> descendingMap( )	Returns a **NavigableMap** that is the reverse of the invoking map. The resulting map is backed by the invoking map.
Map.Entry<K,V> firstEntry( )	Returns the first entry in the map. This is the entry with the least key.
Map.Entry<K,V> floorEntry(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K floorKey(K *obj*)	Searches the map for the largest key *k* such that *k* <= *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included. The resulting map is backed by the invoking map.
Map.Entry<K,V> higherEntry(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K higherKey(K *obj*)	Searches the set for the largest key *k* such that *k* > *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.
Map.Entry<K,V> lastEntry( )	Returns the last entry in the map. This is the entry with the largest key.
Map.Entry<K,V> lowerEntry(K obj)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, its entry is returned. Otherwise, **null** is returned.
K lowerKey(K *obj*)	Searches the set for the largest key *k* such that *k* < *obj*. If such a key is found, it is returned. Otherwise, **null** is returned.

NavigableSet<K> navigableKeySet( )	Returns a **NavigableSet** that contains the keys in the invoking map. The resulting set is backed by the invoking map.
Map.Entry<K,V> pollFirstEntry( )	Returns the first entry, removing the entry in the process. Because the map is sorted, this is the entry with the least key value. **null** is returned if the map is empty.
Map.Entry<K,V> pollLastEntry( )	Returns the last entry, removing the entry in the process. Because the map is sorted, this is the entry with the greatest key value. **null** is returned if the map is empty.
NavigableMap<K,V> subMap(K *lowerBound*, boolean *lowIncl*, K *upperBound* boolean *highIncl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys 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 *highIncl* is included. The resulting map is backed by the invoking map.
NavigableMap<K,V> tailMap(K *lowerBound*, boolean *incl*)	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*. If *incl* is **true**, then an element equal to *lowerBound* is included. The resulting map is backed by the invoking map.

Table 19-13 The Methods Declared by **NavigableMap**

The Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. For example, recall that the **entrySet( )** method declared by the **Map** interface returns a **Set** containing the map entries. Each of these set elements is a **Map.Entry** object. **Map.Entry** is generic and is declared like this:

```
interface Map.Entry<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values. Table 19-14 summarizes the non-static methods declared by **Map.Entry**. It also has two static methods. The first is **comparingByKey( )**, which returns a **Comparator** that compares entries by key. The second is **comparingByValue( )**, which returns a **Comparator** that compares entries by value.

Method	Description
boolean equals(Object *obj*)	Returns **true** if *obj* is a **Map.Entry** whose key and value are equal to that of the invoking object.
K getKey( )	Returns the key for this map entry.
V getValue( )	Returns the value for this map entry.
int hashCode( )	Returns the hash code for this map entry.
V setValue(V *v*)	Sets the value for this map entry to *v*. A **ClassCastException** is thrown if *v* is not the correct type for the map. An **IllegalArgumentException** is thrown if there is a problem with *v*. A **NullPointerException** is thrown if *v* is **null** and the map does not permit **null** keys. An **UnsupportedOperationException** is thrown if the map cannot be changed.

Table 19-14 The Non-Static Methods Declared by **Map.Entry**

The Map Classes

Several classes provide implementations of the map interfaces. The classes that can be used for maps are summarized here:

Class	Description
AbstractMap	Implements most of the **Map** interface.
EnumMap	Extends **AbstractMap** for use with **enum** keys.
HashMap	Extends **AbstractMap** to use a hash table.
TreeMap	Extends **AbstractMap** to use a tree.
WeakHashMap	Extends **AbstractMap** to use a hash table with weak keys.
LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
IdentityHashMap	Extends **AbstractMap** and uses reference equality when comparing documents.

Notice that **AbstractMap** is a superclass for all concrete map implementations.

WeakHashMap implements a map that uses “weak keys,” which allows an element in a map to be garbage-collected when its key is otherwise unused. This class is not discussed further here. The other map classes are described next.

The HashMap Class

The **HashMap** class extends **AbstractMap** and implements the **Map** interface.

It uses a hash table to store the map. This allows the execution time of **get( )** and **put( )** to remain constant even for large sets. **HashMap** is a generic class that has this declaration:

```
class HashMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:

```
HashMap( )
HashMap(Map<? extends K, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)
```

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier. The default capacity is 16. The default fill ratio is 0.75.

HashMap implements **Map** and extends **AbstractMap**. It does not add any methods of its own.

You should note that a hash map does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

The following program illustrates **HashMap**. It maps names to account balances. Notice how a set-view is obtained and used.

```
import java.util.*;

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

    // Create a hash map.
    HashMap<String, Double> hm = new HashMap<String, Double>();

    // Put elements to the map
    hm.put("John Doe", 3434.34);
    hm.put("Tom Smith", 123.22);
    hm.put("Jane Baker", 1378.00);
    hm.put("Tod Hall", 99.22);
    hm.put("Ralph Smith", -19.08);

    // Get a set of the entries.
    Set<Map.Entry<String, Double>> set = hm.entrySet();

    // Display the set.
    for(Map.Entry<String, Double> me : set) {
      System.out.print(me.getKey() + ": ");
      System.out.println(me.getValue());
    }

    System.out.println();

    // Deposit 1000 into John Doe's account.
    double balance = hm.get("John Doe");
    hm.put("John Doe", balance + 1000);

    System.out.println("John Doe's new balance: " +
      hm.get("John Doe"));
  }
}
```

Output from this program is shown here (the precise order may vary):

```
Ralph Smith: -19.08
Tom Smith: 123.22
John Doe: 3434.34
Tod Hall: 99.22
Jane Baker: 1378.0

John Doe's new balance: 4434.34
```

The program begins by creating a hash map and then adds the mapping of names to balances. Next, the contents of the map are displayed by using a set-view, obtained by calling **entrySet( )**. The keys and values are displayed by calling the **getKey( )** and **getValue( )** methods that are defined by **Map.Entry**. Pay close attention to how the deposit is made into John Doe's account. The **put( )** method automatically replaces any preexisting value that is associated with the specified key with the new value. Thus, after John Doe's account is updated, the hash map will still contain just one "John Doe" account.

The TreeMap Class

The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface. It creates maps stored in a tree structure. A **TreeMap** provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval. You should note that, unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order. **TreeMap** is a generic class that has this declaration:

```
class TreeMap<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following **TreeMap** constructors are defined:

```
TreeMap( )
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)
```

The first form constructs an empty tree map that will be sorted by using the natural order of its keys. The second form constructs an empty tree-based map that will be sorted by using the **Comparator** *comp*. (Comparators are discussed later in this chapter.) The third form initializes a tree map with the entries from

m, which will be sorted by using the natural order of the keys. The fourth form initializes a tree map with the entries from *sm*, which will be sorted in the same order as *sm*.

TreeMap has no map methods beyond those specified by the **NavigableMap** interface and the **AbstractMap** class.

The following program reworks the preceding example so that it uses **TreeMap**:

```
import java.util.*;

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

    // Create a tree map.
    TreeMap<String, Double> tm = new TreeMap<String, Double>();

    // Put elements to the map.
    tm.put("John Doe", 3434.34);
    tm.put("Tom Smith", 123.22);
    tm.put("Jane Baker", 1378.00);
    tm.put("Tod Hall", 99.22);
    tm.put("Ralph Smith", -19.08);

    // Get a set of the entries.
    Set<Map.Entry<String, Double>> set = tm.entrySet();

    // Display the elements.
```

```
    for(Map.Entry<String, Double> me : set) {
       System.out.print(me.getKey() + ": ");
       System.out.println(me.getValue());
    }
    System.out.println();

    // Deposit 1000 into John Doe's account.
    double balance = tm.get("John Doe");
    tm.put("John Doe", balance + 1000);

    System.out.println("John Doe's new balance: " +
      tm.get("John Doe"));
  }
}
```

The following is the output from this program:

```
Jane Baker: 1378.0
John Doe: 3434.34
Ralph Smith: -19.08
Todd Hall: 99.22
Tom Smith: 123.22

John Doe's current balance: 4434.34
```

Notice that **TreeMap** sorts the keys. However, in this case, they are sorted by first name instead of last name. You can alter this behavior by specifying a comparator when the map is created, as described shortly.

The LinkedHashMap Class

LinkedHashMap extends **HashMap**. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted. You can also create a **LinkedHashMap** that returns its elements in the order in which they were last accessed. **LinkedHashMap** is a generic class that has this declaration:

class LinkedHashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

LinkedHashMap defines the following constructors:

LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> *m*)
LinkedHashMap(int *capacity*)
LinkedHashMap(int *capacity*, float *fillRatio*)
LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)

The first form constructs a default **LinkedHashMap**. The second form initializes the **LinkedHashMap** with the elements from *m*. The third form initializes the capacity. The fourth form initializes both capacity and fill ratio. The meaning of capacity and fill ratio are the same as for **HashMap**. The default capacity is 16. The default ratio is 0.75. The last form allows you to specify whether the elements will be stored in the linked list by insertion order, or by order of last access. If *Order* is **true**, then access order is used. If *Order* is **false**, then insertion order is used.

LinkedHashMap adds only one method to those defined by **HashMap**. This method is **removeEldestEntry( )**, and it is shown here:

protected boolean removeEldestEntry(Map.Entry<K, V> *e*)

This method is called by **put( )** and **putAll( )**. The oldest entry is passed in *e*. By default, this method returns **false** and does nothing. However, if you override this method, then you can have the **LinkedHashMap** remove the oldest entry in the map. To do this, have your override return **true**. To keep the oldest entry, return **false**.

The IdentityHashMap Class

IdentityHashMap extends **AbstractMap** and implements the **Map** interface. It is similar to **HashMap** except that it uses reference equality when comparing elements. **IdentityHashMap** is a generic class that has this declaration:

class IdentityHashMap<K, V>

Here, **K** specifies the type of key, and **V** specifies the type of value. The API documentation explicitly states that **IdentityHashMap** is not for general use.

The EnumMap Class

EnumMap extends **AbstractMap** and implements **Map**. It is specifically for use with keys of an **enum** type. It is a generic class that has this declaration:

class EnumMap<K extends Enum<K>, V>

Here, **K** specifies the type of key, and **V** specifies the type of value. Notice that **K** must extend **Enum<K>**, which enforces the requirement that the keys must be of an **enum** type.

EnumMap defines the following constructors:

EnumMap(Class<K> *kType*)
EnumMap(Map<K, ? extends V> *m*)
EnumMap(EnumMap<K, ? extends V> *em*)

The first constructor creates an empty **EnumMap** of type *kType*. The second creates an **EnumMap** map that contains the same entries as *m*. The third creates an **EnumMap** initialized with the values in *em*.

EnumMap defines no methods of its own.

Comparators

Both **TreeSet** and **TreeMap** store elements in sorted order. However, it is the comparator that defines precisely what "sorted order" means. By default, these classes store their elements by using what Java refers to as "natural ordering," which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then specify a **Comparator** when you construct the set or map. Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.

Comparator is a generic interface that has this declaration:

interface Comparator<T>

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

Prior to JDK 8, the **Comparator** interface defined only two methods: **compare( )** and **equals( )**. The **compare( )** method, shown here, compares two elements for order:

int compare(T *obj1*, T *obj2*)

obj1 and *obj2* are the objects to be compared. Normally, this method returns zero if the objects are equal. It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned. The method can throw a **ClassCastException** if the types of the objects are not compatible for comparison. By implementing **compare( )**, you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The **equals( )** method, shown here, tests whether an object equals the invoking comparator:

```
boolean equals(object obj)
```

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**. Overriding **equals( )** is not necessary, and most simple comparators will not do so.

For many years, the preceding two methods were the only methods defined by **Comparator**. With the release of JDK 8, the situation dramatically changed. JDK 8 added significant new functionality to **Comparator** through the use of default and static interface methods. Each is described here.

You can obtain a comparator that reverses the ordering of the comparator on which it is called by using **reversed( )**, shown here:

```
default Comparator<T> reversed( )
```

It returns the reverse comparator. For example, assuming a comparator that uses natural ordering for the characters A through Z, a reverse order comparator would put B before A, C before B, and so on.

A method related to **reversed( )** is **reverseOrder( )**, shown next:

```
static <T extends Comparable<? super T>> Comparator<T> reverseOrder( )
```

It returns a comparator that reverses the natural order of the elements. Conversely, you can obtain a comparator that uses natural ordering by calling the static method **naturalOrder( )**, shown next:

```
static <T extends Comparable<? super T>> Comparator<T> naturalOrder( )
```

If you want a comparator that can handle **null** values, use **nullsFirst( )** or