PART

# II

# The Java Library

CHAPTER

# 16 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 17.

## 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( )

When Java converts data into its string representation during concatenation, it does so 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. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange.

### toCharArray( )

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

```
char[ ] toCharArray( )
```

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

## String Comparison

The **String** class includes 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( )

The **trim( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed. It has this general form:

String trim( )

Here is an example:

```
String s = "   Hello World    ".trim();
```

This puts the string "Hello World" into **s**.

The **trim( )** method is quite useful when you process user commands. For example, the following program prompts the user for the name of a state and then displays that state's capital. It uses **trim( )** to remove any leading or trailing whitespace that may have inadvertently been entered by the user.

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

# 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( )** is called when a string representation of some other type of data is needed—for example, during concatenation operations. 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 adds 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 17 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 18, among others. See Chapter 17 for information on **Iterable**.

## Additional String Methods

In addition to those methods discussed earlier, **String** has many other methods, including those 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. |
| 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*. |

| Method | Description |
|---|---|
| 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*)

The string representation of each parameter is obtained, often by calling **String.valueOf( )**. 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 several others, including those 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

Introduced by JDK 5, **StringBuilder** is a relatively recent addition to Java's string handling capabilities. **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**.

This page has been intentionally left blank

CHAPTER

# 17 Exploring java.lang

This chapter discusses those 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.

**java.lang** includes the following classes:

| | | | |
|---|---|---|---|
| Boolean | Enum | Process | String |
| Byte | Float | ProcessBuilder | StringBuffer |
| Character | InheritableThreadLocal | ProcessBuilder.Redirect | StringBuilder |
| Character.Subset | Integer | Runtime | System |
| Character.UnicodeBlock | Long | RuntimePermission | Thread |
| Class | Math | SecurityManager | ThreadGroup |
| ClassLoader | Number | Short | ThreadLocal |
| ClassValue | Object | StackTraceElement | Throwable |
| Compiler | Package | StrictMath | Void |
| Double | | | |

**java.lang** defines the following interfaces:

| | | |
|---|---|---|
| Appendable | Cloneable | Readable |
| AutoCloseable | Comparable | Runnable |
| CharSequence | Iterable | Thread.UncaughtExceptionHandler |

Several of the classes contained in **java.lang** contain deprecated methods, most dating back to Java 1.0. These deprecated methods are still provided by Java to support an ever-shrinking pool of legacy code and are not recommended for new code. Because of this, 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 18 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.

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.

The methods defined by **Float** include those shown in Table 17-1. The methods defined by **Double** include those shown in Table 17-2. Both **Float** and **Double** define the following constants:

| | |
|---|---|
| BYTES | The width of a **float** or **double** in bytes (Added by JDK 8.) |
| 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** |

| 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*. (Added by JDK 8.) |
| static float intBitsToFloat(int *num*) | Returns **float** equivalent of the IEEE-compatible, single-precision bit pattern specified by *num*. |

**Table 17-1** The Methods Defined by **Float**

| Method | Description |
|---|---|
| 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. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| static float min(float *val*, float *val2*) | Returns the minimum of *val* and *val2*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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 17-1** The Methods Defined by **Float** *(continued)*

| 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*. |

**Table 17-2** The Methods Defined by **Double**

| Method | Description |
|---|---|
| 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*. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| static double min(double *val*, double *val2*) | Returns the minimum of *val* and *val2*. (Added by JDK 8.) |
| static double parseDouble(String *str*) throws NumberFormatException | Returns the **double** equivalent of the number contained in the string specified by *str* using radix 10. |

**Table 17-2** The Methods Defined by **Double** *(continued)*

| Method | Description |
|---|---|
| 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*. (Added by JDK 8.) |
| 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 17-2** The Methods Defined by **Double** *(continued)*

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 = new Double(3.14159);
    Double d2 = new Double("314159E-5");

    System.out.println(d1 + " = " + d2 + " -> " + d1.equals(d2));
  }
}
```

As you can see from the following output, both constructors 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 = new Double(1/0.);
     Double d2 = new Double(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.

The methods defined by these classes are shown in Tables 17-3 through 17-6. 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.

The following constants are defined:

| | |
|---|---|
| BYTES | The width of the integer type in bytes (Added by JDK 8.) |
| 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** |

| 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 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*. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| static long toUnsignedLong(byte *val*) | Returns the value of *val* as an unsigned long integer. (Added by JDK 8.) |
| 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 17-3** 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 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*. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| static long toUnsignedLong(short *val*) | Returns the value of *val* as an unsigned long integer. (Added by JDK 8.) |
| 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 17-4** 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*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| static int min(int *val*, int *val2*) | Returns the minimum of *val* and *val2*. (Added by JDK 8.) |
| 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. |

**Table 17-5** The Methods Defined by **Integer**

| Method | Description |
|---|---|
| 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(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(String *str*) throws NumberFormatException | Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix 10. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| static int remainderUnsigned(int *dividend*, int *divisor*) | Returns the remainder, as an unsigned value, of the unsigned division of *dividend* by *divisor*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| static String toUnsignedString(int *val*) | Returns a string that contains the decimal value of *val* as an unsigned integer. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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 17-5** The Methods Defined by **Integer** *(continued)*

| 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*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| static long min(long *val*, long *val2*) | Returns the minimum of *val* and *val2*. (Added by JDK 8.) |
| 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. |

**Table 17-6** The Methods Defined by **Long**

| Method | Description |
|---|---|
| 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(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(String *str*) throws NumberFormatException | Returns the unsigned integer equivalent of the number contained in the string specified by *str* using the radix 10. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| static long remainderUnsigned( long *dividend*, long *divisor*) | Returns the remainder, as an unsigned value, of the unsigned division of *dividend* by *divisor*. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| 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*. (Added by JDK 8.) |
| 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 17-6** The Methods Defined by **Long** *(continued)*

## 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.

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 (Added by JDK 8.) |
| 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 17-7. The following example demonstrates several of these 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.

| 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 17-7** Various **Character** Methods

**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 the **equals( )** and **hashCode( )** 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

Relatively recently, 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*. New 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 17-7 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 17-8.

## 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)
```

| 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 17-8** A Sampling of Methods That Provide Support for 32-Bit Unicode Code Points

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**.

**Boolean** defines the methods shown in Table 17-9.

| 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*. (Added by JDK 8.) |
| static boolean logicalAnd(boolean *op1*, boolean *op2*) | Performs a logical AND of *op1* and *op2* and returns the result. (Added by JDK 8.) |
| static boolean logicalOr(boolean *op1*, boolean *op2*) | Performs a logical OR of *op1* and *op2* and returns the result. (Added by JDK 8.) |
| static boolean logicalXor(boolean *op1*, boolean *op2*) | Performs a logical XOR of *op1* and *op2* and returns the result. (Added by JDK 8.) |
| 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 17-9** 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 17-10.

| Method | Description |
|---|---|
| void destroy( ) | Terminates the process. |
| Process destroyForcibly( ) | Forces termination of the invoking process. Returns a reference to the process. (Added by JDK 8.) |
| 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. |
| boolean isAlive( ) | Returns **true** if the invoking process is still active. Otherwise, returns **false**. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |

**Table 17-10** 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. Applets and other untrusted code typically cannot call any of the **Runtime** methods without raising a **SecurityException**. Several commonly used methods defined by **Runtime** are shown in Table 17-11.

| 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. |

**Table 17-11** A Sampling of Methods Defined by **Runtime**

| Method | Description |
|---|---|
| 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. |
| void traceInstructions(boolean *traceOn*) | Turns on or off instruction tracing, depending upon the value of *traceOn*. If *traceOn* is **true**, the trace is displayed. If it is **false**, tracing is turned off. |
| void traceMethodCalls(boolean *traceOn*) | Turns on or off method call tracing, depending upon the value of *traceOn*. If *traceOn* is **true**, the trace is displayed. If it is **false**, tracing is turned off. |

**Table 17-11** A Sampling of Methods Defined by **Runtime** *(continued)*

Let's look at two of the most common 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] = new Integer(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 the most common. 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 20.)

## 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 17-12 describes the methods defined by **ProcessBuilder**.

In Table 17-12, 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** and **PIPE**.

| 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. |

**Table 17-12** The Methods Defined by **ProcessBuilder**

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

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

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

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

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



## System

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

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

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

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

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

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

## Using currentTimeMillis( ) to Time Program Execution

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



```
// Timing program execution.

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

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

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

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

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

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

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

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

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

## Using arraycopy( )

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

```
// Using arraycopy().

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

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

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

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

## Environment Properties

The following properties are available in all cases:

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

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

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

# Object

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

# Using clone( ) and the Cloneable Interface

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

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

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



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

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

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

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

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

```
// Demonstrate the clone() method

class TestClone implements Cloneable {
  int a;
  double b;

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

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

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

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

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

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

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

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

class TestClone implements Cloneable {
  int a;
  double b;

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

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

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

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

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

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

## Class

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

class Class<T>

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

| Method | Description |
|---|---|
| static Class<?> forName(String *name*)<br>throws ClassNotFoundException | Returns a **Class** object given its complete name. |
| static Class<?> forName(String *name*,<br>boolean *how*,<br>ClassLoader *ldr*)<br>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<br>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[ ]<br>getAnnotationsByType(<br>Class<A> *annoType*) | Returns an array of the annotations (including repeated annotations) of *annoType* associated with the invoking object. (Added by JDK 8.) |
| Class<?>[ ] getClasses( ) | Returns a **Class** object for each public class and interface that is a member of the class represented by the invoking object. |
| ClassLoader getClassLoader( ) | Returns the **ClassLoader** object that loaded the class or interface. |
| Constructor<T><br>getConstructor(Class<?> ... *paramTypes*)<br>throws NoSuchMethodException,<br>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( )<br>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[ ]<br>getDeclaredAnnotationsByType(<br>Class<A> *annoType*) | Returns an array of the non-inherited annotations (including repeated annotations) of *annoType* associated with the invoking object. (Added by JDK 8.) |
| Constructor<?>[ ] getDeclaredConstructors( )<br>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( )<br>throws SecurityException | Obtains a **Field** object for each field declared by the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. (Inherited fields are ignored.) |

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

| Method | Description |
|---|---|
| Method[ ] getDeclaredMethods( )<br>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*)<br>throws NoSuchMethodException,<br>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( )<br>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*,<br>Class<?> ... *paramTypes*)<br>throws NoSuchMethodException,<br>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( )<br>throws SecurityException | Obtains a **Method** object for each public method of the class or interface represented by the invoking object and stores them in an array. Returns a reference to this array. |
| String getName( ) | Returns the complete name of the class or interface of the type represented by the invoking object. |
| ProtectionDomain getProtectionDomain( ) | Returns the protection domain associated with the invoking object. |
| Class<? super T> getSuperclass( ) | Returns the superclass of the type represented by the invoking object. The return value is **null** if the represented type is **Object** or not a class. |
| boolean isInterface( ) | Returns **true** if the type represented by the invoking object is an interface. Otherwise, it returns **false**. |
| T newInstance( )<br>throws IllegalAccessException,<br>InstantiationException | Creates a new instance (i.e., a new object) that is of the same type as that represented by invoking object. This is equivalent to using **new** with the class' default constructor. The new object is returned. This method will fail if the represented type is abstract, not a class, or does not have a default constructor. |
| String toString( ) | Returns the string representation of the type represented by the invoking object or interface. |

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

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

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

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

class X {
  int a;
  float b;
}

class Y extends X {
  double c;
}

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

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

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

The output from this program is shown here:

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

# ClassLoader

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

# Math

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



## Trigonometric Functions

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

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

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

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

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

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

## Exponential Functions

**Math** defines the following exponential methods:

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

## Rounding Functions

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

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

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

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

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

## Miscellaneous Math Methods

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

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

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

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

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

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

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

The output is shown here:

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



## StrictMath

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

## Compiler

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

## Thread, ThreadGroup, and Runnable

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

---

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

### The Runnable Interface

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

```
void run( )
```

Threads that you create must implement this method.

## Thread

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

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

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

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

MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY

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

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

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

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

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

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

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

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

## ThreadGroup

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

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

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

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

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

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

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

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

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

  NewThread(String threadname, ThreadGroup tgOb) {
    super(tgOb, threadname);
    System.out.println("New thread: " + this);
    suspendFlag = false;
    start(); // Start the thread
  }

  // This is the entry point for thread.
  public void run() {
    try {
      for(int i = 5; i > 0; i--) {
        System.out.println(getName() + ": " + i);
        Thread.sleep(1000);
        synchronized(this) {
          while(suspendFlag) {
            wait();
          }
        }
      }
    } catch (Exception e) {
      System.out.println("Exception in " + getName());
    }
      System.out.println(getName() + " exiting.");
    }

    synchronized void mysuspend() {
      suspendFlag = true;
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

## ThreadLocal and InheritableThreadLocal

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

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

# Package

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

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

    pkgs = Package.getPackages();

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

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

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

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

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

## RuntimePermission

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

## Throwable

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

## SecurityManager

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

## StackTraceElement

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

**StackTraceElement** has one constructor:

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

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

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



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

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

# Enum

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

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

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

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

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

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

## ClassValue

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

Class ClassValue<T>

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

## The CharSequence Interface

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



## The Comparable Interface

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

interface Comparable<T>

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

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

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

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

int compareTo(T *obj*)

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

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

## The Appendable Interface

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

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

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

## The Iterable Interface

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

interface Iterable<T>

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

Iterator<T> iterator( )

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

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

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

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

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

default Spliterator<T> spliterator( )

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

---

**NOTE** Iterators are described in detail in Chapter 18.

## The Readable Interface

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

int read(CharBuffer *buf*) throws IOException

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



## The AutoCloseable Interface

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

void close( ) throws Exception

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

## The Thread.UncaughtExceptionHandler Interface

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

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

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

## The java.lang Subpackages

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

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

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

Each is briefly described here.

## java.lang.annotation

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

## java.lang.instrument

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

## java.lang.invoke

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

## java.lang.management

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

## java.lang.ref

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

## java.lang.reflect

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

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

CHAPTER

# 18 java.util Part 1: The Collections Framework

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

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

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

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

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

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

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

## Collections Overview

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

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

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

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

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

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

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

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

---

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

# JDK 5 Changed the Collections Framework

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

## Generics Fundamentally Changed the Collections Framework

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

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

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

## Autoboxing Facilitates the Use of Primitive Types

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

## The For-Each Style for Loop

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

# The Collection Interfaces

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

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



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

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

The following sections examine the collection interfaces.

## The Collection Interface

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

```
interface Collection<E>
```

Here, **E** specifies the type of objects that the collection will hold. **Collection** extends the **Iterable** interface. This means that all collections can be cycled through by use of the for-each style **for** loop. (Recall that only classes that implement **Iterable** can be cycled through by the **for**.)

**Collection** declares the core methods that all collections will have. These methods are summarized in Table 18-1. Because all collections implement **Collection**, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an **UnsupportedOperationException**. As explained, this occurs if a collection cannot be modified. A **ClassCastException** is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object to a collection. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the collection. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

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

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

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

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

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

You can remove an object by using **remove( )**. To remove a group of objects, call **removeAll( )**. You can remove all elements except those of a specified group by calling **retainAll( )**. Beginning with JDK 8, to remove an element only if it statisfies some condition, you can use **removeIf( )**. (**Predicate** is a functional interface added by JDK 8. See Chapter 19.) To empty a collection, call **clear( )**.

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

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

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

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

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

## The List Interface

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

```
interface List<E>
```

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

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

To the versions of **add( )** and **addAll( )** defined by **Collection**, **List** adds the methods **add(int, E)** and **addAll(int, Collection)**. These methods insert elements at the specified index. Also, the semantics of **add(E)** and **addAll(Collection)** defined by **Collection** are changed by **List** so that they add elements to the end of the list. You can modify each element in the collection by using **replaceAll( )**. (**UnaryOperator** is a functional interface added by JDK 8. See Chapter 19.)

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

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

## The Set Interface

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

| Method | Description |
|---|---|
| void add(int *index*, E *obj*) | Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. |
| boolean addAll(int *index*, Collection<? extends E> *c*) | Inserts all elements of *c* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise. |
| E get(int *index*) | Returns the object stored at the specified index within the invoking collection. |
| int indexOf(Object *obj*) | Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned. |
| int lastIndexOf(Object *obj*) | Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned. |
| ListIterator<E> listIterator( ) | Returns an iterator to the start of the invoking list. |
| ListIterator<E> listIterator(int *index*) | Returns an iterator to the invoking list that begins at the specified *index*. |
| E remove(int *index*) | Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one. |
| default void replaceAll(UnaryOperator<E> *opToApply*) | Updates each element in the list with the value obtained from the *opToApply* function. (Added by JDK 8.) |
| E set(int *index*, E *obj*) | Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value. |
| default void sort(Comparator<? super E> *comp*) | Sorts the list using the comparator specified by *comp*. (Added by JDK 8.) |
| List<E> subList(int *start*, int *end*) | Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object. |

**Table 18-2** The Methods Declared by **List**

**false** if an attempt is made to add duplicate elements to a set. It does not specify any additional methods of its own. **Set** is a generic interface that has this declaration:

```
interface Set<E>
```

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

## The SortedSet Interface

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

```
interface SortedSet<E>
```

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

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

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

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

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

## The NavigableSet Interface

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

```
interface NavigableSet<E>
```

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



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

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

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

## The Queue Interface

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

```
interface Queue<E>
```

Here, **E** specifies the type of objects that the queue will hold. The methods declared by **Queue** are shown in Table 18-5.

Several methods throw a **ClassCastException** when an object is incompatible with the elements in the queue. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the queue. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length queue that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty queue.

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

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

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

## The Deque Interface

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

```
interface Deque<E>
```

Here, **E** specifies the type of objects that the deque will hold. In addition to the methods that it inherits from **Queue**, **Deque** adds those methods summarized in Table 18-6. Several



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

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

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

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

methods throw a **ClassCastException** when an object is incompatible with the elements in the deque. A **NullPointerException** is thrown if an attempt is made to store a **null** object and **null** elements are not allowed in the deque. An **IllegalArgumentException** is thrown if an invalid argument is used. An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length deque that is full. A **NoSuchElementException** is thrown if an attempt is made to remove an element from an empty deque.

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

## The Collection Classes

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

The core collection classes are summarized in the following table:

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



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

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

## The ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:

```
class ArrayList<E>
```

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

**ArrayList** supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**. In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

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

**ArrayList** has the constructors shown here:

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

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

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

```
// Demonstrate ArrayList.
import java.util.*;

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

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

     // Add elements to the array list.
     al.add("C");
     al.add("A");
     al.add("E");
     al.add("B");
     al.add("D");
     al.add("F");
     al.add(1, "A2");

     System.out.println("Size of al after additions: " +
                        al.size());

     // Display the array list.
     System.out.println("Contents of al: " + al);

     // Remove elements from the array list.
     al.remove("F");
     al.remove(2);

     System.out.println("Size of al after deletions: " +
                        al.size());

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

The output from this program is shown here:

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

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

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

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

void ensureCapacity(int *cap*)

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

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

void trimToSize( )

## Obtaining an Array from an ArrayList

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

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

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

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

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

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

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

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

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

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

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

    int sum = 0;

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

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

The output from the program is shown here:

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

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

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

## The LinkedList Class

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

```
class LinkedList<E>
```

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

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

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

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

The following program illustrates **LinkedList**:

```
// Demonstrate LinkedList.
import java.util.*;

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

    // Add elements to the linked list.
    ll.add("F");
    ll.add("B");
    ll.add("D");
    ll.add("E");
    ll.add("C");
    ll.addLast("Z");
    ll.addFirst("A");

    ll.add(1, "A2");

    System.out.println("Original contents of ll: " + ll);

    // Remove elements from the linked list.
    ll.remove("F");
    ll.remove(2);

    System.out.println("Contents of ll after deletion: "
                       + ll);
```

```
    // 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 capacity*) 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 six constructors shown here:

```
PriorityQueue( )
PriorityQueue(int capacity)
PriorityQueue(Comparator<? super E> comp) (Added by JDK 8.)
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 18-7 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.

# 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 18-8. The methods declared by **ListIterator** (along with those inherited from **Iterator**) are shown in Table 18-9. 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 |
|---|---|
| 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 18-7** The Methods Declared by **EnumSet**

| Method | Description |
|---|---|
| default void forEachRemaining( Consumer<? super E> *action*) | The action specified by *action* is executed on each unprocessed element in the collection. (Added by JDK 8.) |
| 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 18-8** 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. (Added by JDK 8.) |
| 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 18-9** 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 adds a new 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 18-10.

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 19 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.

| 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 18-10** The Methods Declared by **Spliterator**



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 18-10, 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 new 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 18-11. 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.

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.

| Method | Description |
|---|---|
| void clear( ) | Removes all key/value pairs from the invoking map. |
| default V compute(K *k*,<br>BiFunction<? super K, ? super V,<br>? 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. (Added by JDK 8.) |
| default V computeIfAbsent(K *k*,<br>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. (Added by JDK 8.) |
| default V computeIfPresent(K *k*,<br>BiFunction<? super K, ? super V,<br>? 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. (Added by JDK 8.) |
| 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**. |
| 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<<br>? super K,<br>? 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. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| 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. |

**Table 18-11** The Methods Declared by **Map**

| Method | Description |
|---|---|
| default V merge(K *k*, V *v*,<br>BiFunction<? super V, ? super V,<br>? 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. (Added by JDK 8.) |
| 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,<br>? 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**. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| 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. (Added by JDK 8.) |
| default void replaceAll(BiFunction<<br>? super K,<br>? super V,<br>? 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. (Added by JDK 8.) |
| 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 18-11** The Methods Declared by **Map** *(continued)*

## 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 18-12. 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.

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 18-13. 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 |
|---|---|
| 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 18-12** The Methods Declared by **SortedMap**

| 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. |

**Table 18-13** The Methods Declared by **NavigableMap**

| Method | Description |
|---|---|
| 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><br>subMap(K *lowerBound*,<br>boolean *lowIncl*,<br>K *upperBound*<br>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><br>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 18-13** The Methods Declared by **NavigableMap** *(continued)*

## The Map.Entry Interface

The **Map.Entry** interface enables you to work with a map entry. 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 18-14 summarizes the non-static methods declared by **Map.Entry**. JDK 8 adds 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 18-14** 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", new Double(3434.34));
    hm.put("Tom Smith", new Double(123.22));
    hm.put("Jane Baker", new Double(1378.00));
    hm.put("Tod Hall", new Double(99.22));
    hm.put("Ralph Smith", new Double(-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", new Double(3434.34));
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Tod Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-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 has dramatically changed. JDK 8 adds 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 **nullsLast( )**, shown here:

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> comp)
static <T> Comparator<T> nullsLast(Comparator<? super T> comp)
```

The **nullsFirst( )** method returns a comparator that views **null** values as less than other values. The **nullsLast( )** method returns a comparator that views **null** values as greater than other values. In both cases, if the two values being compared are non-**null**, *comp* performs the comparison. If *comp* is passed **null**, then all non-**null** values are viewed as equivalent.

Another default method added by JDK 8 is **thenComparing( )**. It returns a comparator that performs a second comparison when the outcome of the first comparison indicates that the objects being compared are equal. Thus, it can be used to create a "compare by X then compare by Y" sequence. For example, when comparing cities, the first comparison might compare names, with the second comparison comparing states. (Therefore, Springfield, Illinois, would come before Springfield, Missouri, assuming normal, alphabetical order.) The **thenComparing( )** method has three forms. The first, shown here, lets you specify the second comparator by passing an instance of **Comparator**:

```
default Comparator<T> thenComparing(Comparator<? super T> thenByComp)
```

Here, *thenByComp* specifies the comparator that is called if the first comparison returns equal.

The next versions of **thenComparing( )** let you specify the standard functional interface **Function** (defined by **java.util.function**). They are shown here:

```
default <U extends Comparable<? super U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> getKey)

default <U> Comparator<T>
    thenComparing(Function<? super T, ? extends U> getKey,
                  Comparator<? super U> keyComp)
```

In both, *getKey* refers to function that obtains the next comparison key, which is used if the first comparison returns equal. In the second version, *keyComp* specifies the comparator used to compare keys. (Here, and in subsequent uses, **U** specifies the type of the key.)

**Comparator** also adds the following specialized versions of "then comparing" methods for the primitive types:

```
default Comparator<T>
    thenComparingDouble(ToDoubleFunction<? super T> getKey)

default Comparator<T>
    thenComparingInt(ToIntFunction<? super T> getKey)

default Comparator<T>
    thenComparingLong(ToLongFunction<? super T> getKey)
```

In all methods, *getKey* refers to a function that obtains the next comparison key.

Finally, JDK 8 adds to **Comparator** a method called **comparing( )**. It returns a comparator that obtains its comparison key from a function passed to the method. There are two versions of **comparing( )**, shown here:

```
static <T, U extends Comparable<? super U>> Comparator<T>
    comparing(Function<? super T, ? extends U> getKey)

static <T, U> Comparator<T>
    comparing(Function<? super T, ? extends U> getKey,
              Comparator<? super U> keyComp)
```

In both, *getKey* refers to a function that obtains the next comparison key. In the second version, *keyComp* specifies the comparator used to compare keys. **Comparator** also adds the following specialized versions of these methods for the primitive types:

```
static <T> Comparator<T>
    ComparingDouble(ToDoubleFunction<? super T> getKey)

static <T> Comparator<T>
    ComparingInt(ToIntFunction<? super T> getKey)

static <T> Comparator<T>
    ComparingLong(ToLongFunction<? super T> getKey)
```

In all methods, *getKey* refers to a function that obtains the next comparison key.

## Using a Comparator

The following is an example that demonstrates the power of a custom comparator. It implements the **compare( )** method for strings that operates in reverse of normal. Thus, it causes a tree set to be sorted in reverse order.

```
// Use a custom comparator.
import java.util.*;

// A reverse comparator for strings.
class MyComp implements Comparator<String> {
```

```
  public int compare(String aStr, String bStr) {

    // Reverse the comparison.
    return bStr.compareTo(aStr);
  }

  // No need to override equals or the default methods.
}

class CompDemo {
  public static void main(String args[]) {
    // Create a tree set.
    TreeSet<String> ts = new TreeSet<String>(new MyComp());

    // Add elements to the tree set.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");

    // Display the elements.
    for(String element : ts)
      System.out.print(element + " ");

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



As the following output shows, the tree is now sorted in reverse order:

```
F E D C B A
```

Look closely at the **MyComp** class, which implements **Comparator** by implementing **compare( )**. (As explained earlier, overriding **equals( )** is neither necessary nor common. It is also not necessary to override the default methods added by JDK 8.) Inside **compare( )**, the **String** method **compareTo( )** compares the two strings. However, **bStr**—not **aStr**—invokes **compareTo( )**. This causes the outcome of the comparison to be reversed.

Although the way in which the reverse order comparator is implemented by the preceding program is perfectly adequate, beginning with JDK 8, there is another way to approach a solution. It is now possible to simply call **reversed( )** on a natural-order comparator. It will return an equivalent comparator, except that it runs in reverse. For example, assuming the preceding program, you can rewrite **MyComp** as a natural-order comparator, as shown here:

```
class MyComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    return aStr.compareTo(bStr);
  }
}
```

Next, you can use the following sequence to create a **TreeSet** that orders its string elements in reverse:

```
MyComp mc = new MyComp(); // Create a comparator

// Pass a reverse order version of MyComp to TreeSet.
TreeSet<String> ts = new TreeSet<String>(mc.reversed());
```

If you plug this new code into the preceding program, it will produce the same results as before. In this case, there is no advantage gained by using **reversed( )**. However, in cases in which you need to create both a natural-order comparator and a reversed comparator, then using **reversed( )** gives you an easy way to obtain the reverse-order comparator without having to code it explicitly.

Beginning with JDK 8, it is not actually necessary to create the **MyComp** class in the preceding examples because a lambda expression can be easily used instead. For example, you can remove the **MyComp** class entirely and create the string comparator by using this statement:

```
// Use a lambda expression to implement Comparator<String>.
Comparator<String> mc = (aStr, bStr) -> aStr.compareTo(bStr);
```

One other point: in this simple example, it would also be possible to specify a reverse comparator via a lambda expression directly in the call to the **TreeSet( )** constructor, as shown here:

```
// Pass a reversed comparator to TreeSet() via a
// lambda expression.
TreeSet<String> ts = new TreeSet<String>(
                          (aStr, bStr) -> bStr.compareTo(aStr));
```

By making these changes, the program is substantially shortened, as its final version shown here illustrates:

```
// Use a lambda expression to create a reverse comparator.
import java.util.*;

class CompDemo2 {
  public static void main(String args[]) {

    // Pass a reverse comparator to TreeSet() via a
    // lambda expression.
    TreeSet<String> ts = new TreeSet<String>(
                              (aStr, bStr) -> bStr.compareTo(aStr));

    // Add elements to the tree set.
    ts.add("C");
    ts.add("A");
    ts.add("B");
    ts.add("E");
    ts.add("F");
    ts.add("D");
```

```
    // Display the elements.
    for(String element : ts)
      System.out.print(element + " ");

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

For a more practical example that uses a custom comparator, the following program is an updated version of the **TreeMap** program shown earlier that stores account balances. In the previous version, the accounts were sorted by name, but the sorting began with the first name. The following program sorts the accounts by last name. To do so, it uses a comparator that compares the last name of each account. This results in the map being sorted by last name.



```
// Use a comparator to sort accounts by last name.
import java.util.*;

// Compare last whole words in two strings.
class TComp implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j, k;

    // Find index of beginning of last name.
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');

    k = aStr.substring(i).compareToIgnoreCase (bStr.substring(j));
    if(k==0) // last names match, check entire name
      return aStr.compareToIgnoreCase (bStr);
    else
      return k;
  }

  // No need to override equals.
}

class TreeMapDemo2 {
  public static void main(String args[]) {
    // Create a tree map.
    TreeMap<String, Double> tm = new TreeMap<String, Double>(new TComp());

    // Put elements to the map.
    tm.put("John Doe", new Double(3434.34));
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Tod Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-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"));
  }
}
```

Here is the output; notice that the accounts are now sorted by last name:

```
Jane Baker: 1378.0
John Doe: 3434.34
Todd Hall: 99.22
Ralph Smith: -19.08
Tom Smith: 123.22

John Doe's new balance: 4434.34
```

The comparator class **TComp** compares two strings that hold first and last names. It does so by first comparing last names. To do this, it finds the index of the last space in each string and then compares the substrings of each element that begin at that point. In cases where last names are equivalent, the first names are then compared. This yields a tree map that is sorted by last name, and within last name by first name. You can see this because Ralph Smith comes before Tom Smith in the output.

If you are using JDK 8 or later, then there is another way that you could code the preceding program so the map is sorted by last name and then by first name. This approach uses the **thenComparing( )** method. Recall that **thenComparing( )** lets you specify a second comparator that will be used if the invoking comparator returns equal. This approach is put into action by the following program, which reworks the preceding example to use **thenComparing( )**:

```
// Use thenComparing() to sort by last, then first name.
import java.util.*;

// A comparator that compares last names.
class CompLastNames implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j;

    // Find index of beginning of last name.
    i = aStr.lastIndexOf(' ');
    j = bStr.lastIndexOf(' ');
```

```
    return aStr.substring(i).compareToIgnoreCase(bStr.substring(j));
  }
}

// Sort by entire name when last names are equal.
class CompThenByFirstName implements Comparator<String> {
  public int compare(String aStr, String bStr) {
    int i, j;

    return aStr.compareToIgnoreCase(bStr);
  }
}

class TreeMapDemo2A {
  public static void main(String args[]) {
    // Use thenComparing() to create a comparator that compares
    // last names, then compares entire name when last names match.
    CompLastNames compLN = new CompLastNames();
    Comparator<String> compLastThenFirst =
                         compLN.thenComparing(new CompThenByFirstName());

    // Create a tree map.
    TreeMap<String, Double> tm =
                         new TreeMap<String, Double>(compLastThenFirst);

    // Put elements to the map.
    tm.put("John Doe", new Double(3434.34));
    tm.put("Tom Smith", new Double(123.22));
    tm.put("Jane Baker", new Double(1378.00));
    tm.put("Tod Hall", new Double(99.22));
    tm.put("Ralph Smith", new Double(-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"));
  }
}
```

This version produces the same output as before. It differs only in how it accomplishes its job. To begin, notice that a comparator called **CompLastNames** is created. This comparator

compares only the last names. A second comparator, called **CompThenByFirstName**, compares the entire name, starting with the first name. Next, the **TreeMap** is then created by the following sequence:

```
CompLastNames compLN = new CompLastNames();
Comparator<String> compLastThenFirst =
                    compLN.thenComparing(new CompThenByFirstName());
```

Here, the primary comparator is **compLN**. It is an instance of **CompLastNames**. On it is called **thenComparing( )**, passing in an instance of **CompThenByFirstName**. The result is assigned to the comparator called **compLastThenFirst**. This comparator is used to construct the **TreeMap**, as shown here:

```
TreeMap<String, Double> tm =
                    new TreeMap<String, Double>(compLastThenFirst);
```

Now, whenever the last names of the items being compared are equal, the entire name, beginning with the first name, is used to order the two. This means that names are ordered based on last name, and within last names, by first names.

One last point: in the interest of clarity, this example explicitly creates two comparator classes called **CompLastNames** and **ThenByFirstNames**, but lambda expressions could have been used instead. You might want to try this on your own. Just follow the same general approach described for the **CompDemo2** example shown earlier.

## The Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the **Collections** class. They are summarized in Table 18-15. As explained earlier, beginning with JDK 5 all of the algorithms were retrofitted for generics.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection. Other exceptions are possible, depending on the method.

One thing to pay special attention to is the set of **checked** methods, such as **checkedCollection( )**, which returns what the API documentation refers to as a "dynamically typesafe view" of a collection. This view is a reference to the collection that monitors insertions into the collection for type compatibility at run time. An attempt to insert an incompatible element will cause a **ClassCastException**. Using such a view is especially helpful during debugging because it ensures that the collection always contains valid elements. Related methods include **checkedSet( )**, **checkedList( )**, **checkedMap( )**, and so on. They obtain a type-safe view for the indicated collection.

Notice that several methods, such as **synchronizedList( )** and **synchronizedSet( )**, are used to obtain synchronized (*thread-safe*) copies of the various collections. As a general rule, the standard collections implementations are not synchronized. You must use the synchronization algorithms to provide synchronization. One other point: iterators to synchronized collections must be used within **synchronized** blocks.

| Method | Description |
|---|---|
| static <T> boolean<br>addAll(Collection <? super T> *c*,<br>T... *elements*) | Inserts the elements specified by *elements* into the collection specified by *c*. Returns **true** if the elements were added and **false** otherwise. |
| static <T> Queue<T> asLifoQueue(Deque<T> *c*) | Returns a last-in, first-out view of *c*. |
| static <T><br>int binarySearch(List<? extends T> *list*,<br>T *value*,<br>Comparator<? super T> *c*) | Searches for *value* in *list* ordered according to *c*. Returns the position of value in *list*, or a negative value if *value* is not found. |
| static <T><br>int binarySearch(List<? extends<br>Comparable<? super T>> *list*,<br>T *value*) | Searches for *value* in *list*. The list must be sorted. Returns the position of *value* in *list*, or a negative value if *value* is not found. |
| static <E> Collection<E><br>checkedCollection(Collection<E> *c*,<br>Class<E> *t*) | Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <E> List<E><br>checkedList(List<E> *c*, Class<E> *t*) | Returns a run-time type-safe view of a **List**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <K, V> Map<K, V><br>checkedMap(Map<K, V> *c*,<br>Class<K> *keyT*,<br>Class<V> *valueT*) | Returns a run-time type-safe view of a **Map**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <K, V> NavigableMap<K, V><br>checkedNavigableMap(<br>NavigableMap<K, V> *nm*,<br>Class<E> *keyT*,<br>Class<V> *valueT*) | Returns a run-time type-safe view of a **NavigableMap**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.) |
| static <E> NavigableSet<E><br>checkedNavigableSet(NavigableSet<E> *ns*,<br>Class<E> *t*) | Returns a run-time type-safe view of a **NavigableSet**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.) |
| static <E> Queue<E><br>checkedQueue(Queue<E> *q*,<br>Class<E> *t*) | Returns a run-time type-safe view of a **Queue**. An attempt to insert an incompatible element will cause a **ClassCastException**. (Added by JDK 8.) |
| static <E> List<E><br>checkedSet(Set<E> *c*, Class<E> *t*) | Returns a run-time type-safe view of a **Set**. An attempt to insert an incompatible element will cause a **ClassCastException**. |

**Table 18-15** The Algorithms Defined by **Collections**

| Method | Description |
|---|---|
| static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> *c*, Class<K> *keyT*, Class<V> *valueT*) | Returns a run-time type-safe view of a **SortedMap**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <E> SortedSet<E> checkedSortedSet(SortedSet<E> *c*, Class<E> *t*) | Returns a run-time type-safe view of a **SortedSet**. An attempt to insert an incompatible element will cause a **ClassCastException**. |
| static <T> void copy(List<? super T> *list1*, List<? extends T> *list2*) | Copies the elements of *list2* to *list1*. |
| static boolean disjoint(Collection<?> *a*, Collection<?> *b*) | Compares the elements in *a* to elements in *b*. Returns **true** if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns **false**. |
| static <T> Enumeration<T> emptyEnumeration( ) | Returns an empty enumeration, which is an enumeration with no elements. |
| static <T> Iterator<T> emptyIterator( ) | Returns an empty iterator, which is an iterator with no elements. |
| static <T> List<T> emptyList( ) | Returns an immutable, empty **List** object of the inferred type. |
| static <T> ListIterator<T> emptyListIterator( ) | Returns an empty list iterator, which is a list iterator that has no elements. |
| static <K, V> Map<K, V> emptyMap( ) | Returns an immutable, empty **Map** object of the inferred type. |
| static <K, V> NavigableMap<K, V> emptyNavigableMap( ) | Returns an immutable, empty **NavigableMap** object of the inferred type. (Added by JDK 8.) |
| static <E> NavigableSet<E> emptyNavigableSet( ) | Returns an immutable, empty **NavigableSet** object of the inferred type. (Added by JDK 8.) |
| static <T> Set<T> emptySet( ) | Returns an immutable, empty **Set** object of the inferred type. |
| static <K, V> SortedMap<K, V> emptySortedMap( ) | Returns an immutable, empty **SortedMap** object of the inferred type. (Added by JDK 8.) |
| static <E> SortedSet<E> emptySortedSet( ) | Returns an immutable, empty **SortedSet** object of the inferred type. (Added by JDK 8.) |
| static <T> Enumeration<T> enumeration(Collection<T> *c*) | Returns an enumeration over *c*. (See "The Enumeration Interface," later in this chapter.) |
| static <T> void fill(List<? super T> *list*, T *obj*) | Assigns *obj* to each element of *list*. |
| static int frequency(Collection<?> *c*, object *obj*) | Counts the number of occurrences of *obj* in *c* and returns the result. |

**Table 18-15** The Algorithms Defined by **Collections** *(continued)*

| Method | Description |
|---|---|
| static int indexOfSubList(List<?> *list*, List<?> *subList*) | Searches *list* for the first occurrence of *subList*. Returns the index of the first match, or –1 if no match is found. |
| static int lastIndexOfSubList(List<?> *list*, List<?> *subList*) | Searches *list* for the last occurrence of *subList*. Returns the index of the last match, or –1 if no match is found. |
| static <T> ArrayList<T> list(Enumeration<T> *enum*) | Returns an **ArrayList** that contains the elements of *enum*. |
| static <T> T max(Collection<? extends T> *c*, Comparator<? super T> *comp*) | Returns the maximum element in *c* as determined by *comp*. |
| static <T extends Object & Comparable<? super T>> T max(Collection<? extends T> *c*) | Returns the maximum element in *c* as determined by natural ordering. The collection need not be sorted. |
| static <T> T min(Collection<? extends T> *c*, Comparator<? super T> *comp*) | Returns the minimum element in *c* as determined by *comp*. The collection need not be sorted. |
| static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> *c*) | Returns the minimum element in *c* as determined by natural ordering. |
| static <T> List<T> nCopies(int *num*, T *obj*) | Returns *num* copies of *obj* contained in an immutable list. *num* must be greater than or equal to zero. |
| static <E> Set<E> newSetFromMap(Map<E, Boolean> *m*) | Creates and returns a set backed by the map specified by *m*, which must be empty at the time this method is called. |
| static <T> boolean replaceAll(List<T> *list*, T *old*, T *new*) | Replaces all occurrences of *old* with *new* in *list*. Returns **true** if at least one replacement occurred. Returns **false** otherwise. |
| static void reverse(List<T> *list*) | Reverses the sequence in *list*. |
| static <T> Comparator<T> reverseOrder(Comparator<T> *comp*) | Returns a reverse comparator based on the one passed in *comp*. That is, the returned comparator reverses the outcome of a comparison that uses *comp*. |
| static <T> Comparator<T> reverseOrder( ) | Returns a reverse comparator, which is a comparator that reverses the outcome of a comparison between two elements. |
| static void rotate(List<T> *list*, int *n*) | Rotates *list* by *n* places to the right. To rotate left, use a negative value for *n*. |
| static void shuffle(List<T> *list*, Random *r*) | Shuffles (i.e., randomizes) the elements in *list* by using *r* as a source of random numbers. |
| static void shuffle(List<T> *list*) | Shuffles (i.e., randomizes) the elements in *list*. |

**Table 18-15** The Algorithms Defined by **Collections** *(continued)*

| Method | Description |
|---|---|
| static <T> Set<T> singleton(T *obj*) | Returns *obj* as an immutable set. This is an easy way to convert a single object into a set. |
| static <T> List<T> singletonList(T *obj*) | Returns *obj* as an immutable list. This is an easy way to convert a single object into a list. |
| static <K, V> Map<K, V> singletonMap(K *k*, V *v*) | Returns the key/value pair *k*/*v* as an immutable map. This is an easy way to convert a single key/value pair into a map. |
| static <T> void sort(List<T> *list*, Comparator<? super T> *comp*) | Sorts the elements of *list* as determined by *comp*. |
| static <T extends Comparable<? super T>> void sort(List<T> *list*) | Sorts the elements of *list* as determined by their natural ordering. |
| static void swap(List<?> *list*, int *idx1*, int *idx2*) | Exchanges the elements in *list* at the indices specified by *idx1* and *idx2*. |
| static <T> Collection<T> synchronizedCollection(Collection<T> *c*) | Returns a thread-safe collection backed by *c*. |
| static <T> List<T> synchronizedList(List<T> *list*) | Returns a thread-safe list backed by *list*. |
| static <K, V> Map<K, V> synchronizedMap(Map<K, V> *m*) | Returns a thread-safe map backed by *m*. |
| static <K, V> NavigableMap<K, V> synchronizedNavigableMap( NavigableMap<K, V> *nm*) | Returns a synchronized navigable map backed by *nm*. (Added by JDK 8.) |
| static <T> NavigableSet<T> synchronizedNavigableSet( NavigableSet<T> *ns*) | Returns a synchronized navigable set backed by *ns*. (Added by JDK 8.) |
| static <T> Set<T> synchronizedSet(Set<T> *s*) | Returns a thread-safe set backed by *s*. |
| static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> *sm*) | Returns a thread-safe sorted map backed by *sm*. |
| static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> *ss*) | Returns a thread-safe sorted set backed by *ss*. |
| static <T> Collection<T> unmodifiableCollection( Collection<? extends T> *c*) | Returns an unmodifiable collection backed by *c*. |
| static <T> List<T> unmodifiableList(List<? extends T> *list*) | Returns an unmodifiable list backed by *list*. |
| static <K, V> Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> *m*) | Returns an unmodifiable map backed by *m*. |
| static <K, V> NavigableMap<K, V> unmodifiableNavigableMap( NavigableMap<K, ? extends V> *nm*) | Returns an unmodifiable navigable map backed by *nm*. (Added by JDK 8.) |

**Table 18-15** The Algorithms Defined by **Collections** *(continued)*

| Method | Description |
|---|---|
| static <T> NavigableSet<T><br>unmodifiableNavigableSet(<br>NavigableSet<T> *ns*) | Returns an unmodifiable navigable set backed by *ns*. (Added by JDK 8.) |
| static <T> Set<T><br>unmodifiableSet(Set<? extends T> *s*) | Returns an unmodifiable set backed by *s*. |
| static <K, V> SortedMap<K, V><br>unmodifiableSortedMap(SortedMap<K,<br>? extends V> *sm*) | Returns an unmodifiable sorted map backed by *sm*. |
| static <T> SortedSet<T><br>unmodifiableSortedSet(SortedSet<T> *ss*) | Returns an unmodifiable sorted set backed by *ss*. |

**Table 18-15** The Algorithms Defined by **Collections** *(continued)*



The set of methods that begins with **unmodifiable** returns views of the various collections that cannot be modified. These will be useful when you want to grant some process read—but not write—capabilities on a collection.

**Collections** defines three static variables: **EMPTY_SET, EMPTY_LIST,** and **EMPTY_MAP**. All are immutable.

The following program demonstrates some of the algorithms. It creates and initializes a linked list. The **reverseOrder( )** method returns a **Comparator** that reverses the comparison of **Integer** objects. The list elements are sorted according to this comparator and then are displayed. Next, the list is randomized by calling **shuffle( )**, and then its minimum and maximum values are displayed.

```
// Demonstrate various algorithms.
import java.util.*;

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

    // Create and initialize linked list.
    LinkedList<Integer> ll = new LinkedList<Integer>();
    ll.add(-8);
    ll.add(20);
    ll.add(-20);
    ll.add(8);

    // Create a reverse order comparator.
    Comparator<Integer> r = Collections.reverseOrder();

    // Sort list by using the comparator.
    Collections.sort(ll, r);

    System.out.print("List sorted in reverse: ");
    for(int i : ll)
      System.out.print(i+ " ");
```

```
    System.out.println();

    // Shuffle list.
    Collections.shuffle(ll);

    // Display randomized list.
    System.out.print("List shuffled: ");
    for(int i : ll)
      System.out.print(i + " ");

    System.out.println();
    System.out.println("Minimum: " + Collections.min(ll));
    System.out.println("Maximum: " + Collections.max(ll));
  }
}
```

Output from this program is shown here:

```
List sorted in reverse: 20 8 -8 -20
List shuffled: 20 -20 8 -8
Minimum: -20
Maximum: 20
```

Notice that **min( )** and **max( )** operate on the list after it has been shuffled. Neither requires a sorted list for its operation.

## Arrays

The **Arrays** class provides various methods that are useful when working with arrays. These methods help bridge the gap between collections and arrays. Each method defined by **Arrays** is examined in this section.

The **asList( )** method returns a **List** that is backed by a specified array. In other words, both the list and the array refer to the same location. It has the following signature:

static <T> List asList(T... *array*)

Here, *array* is the array that contains the data.

The **binarySearch( )** method uses a binary search to find a specified value. This method must be applied to sorted arrays. Here are some of its forms. (Additional forms let you search a subrange):

static int binarySearch(byte *array*[ ], byte *value*)
static int binarySearch(char *array*[ ], char *value*)
static int binarySearch(double *array*[ ], double *value*)
static int binarySearch(float *array*[ ], float *value*)
static int binarySearch(int *array*[ ], int *value*)
static int binarySearch(long *array*[ ], long *value*)
static int binarySearch(short *array*[ ], short *value*)
static int binarySearch(Object *array*[ ], Object *value*)
static <T> int binarySearch(T[ ] *array*, T *value*, Comparator<? super T> *c*)

Here, *array* is the array to be searched, and *value* is the value to be located. The last two forms throw a **ClassCastException** if *array* contains elements that cannot be compared (for example, **Double** and **StringBuffer**) or if *value* is not compatible with the types in *array*. In the last form, the **Comparator** *c* is used to determine the order of the elements in *array*. In all cases, if *value* exists in *array*, the index of the element is returned. Otherwise, a negative value is returned.

The **copyOf( )** method returns a copy of an array and has the following forms:

```
static boolean[ ] copyOf(boolean[ ] source, int len)
static byte[ ] copyOf(byte[ ] source, int len)
static char[ ] copyOf(char[ ] source, int len)
static double[ ] copyOf(double[ ] source, int len)
static float[ ] copyOf(float[ ] source, int len)
static int[ ] copyOf(int[ ] source, int len)
static long[ ] copyOf(long[ ] source, int len)
static short[ ] copyOf(short[ ] source, int len)
static <T> T[ ] copyOf(T[ ] source, int len)
static <T,U> T[ ] copyOf(U[ ] source, int len, Class<? extends T[ ]> resultT)
```

The original array is specified by *source*, and the length of the copy is specified by *len*. If the copy is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays). If the copy is shorter than *source*, then the copy is truncated. In the last form, the type of *resultT* becomes the type of the array returned. If *len* is negative, a **NegativeArraySizeException** is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

The **copyOfRange( )** method returns a copy of a range within an array and has the following forms:

```
static boolean[ ] copyOfRange(boolean[ ] source, int start, int end)
static byte[ ] copyOfRange(byte[ ] source, int start, int end)
static char[ ] copyOfRange(char[ ] source, int start, int end)
static double[ ] copyOfRange(double[ ] source, int start, int end)
static float[ ] copyOfRange(float[ ] source, int start, int end)
static int[ ] copyOfRange(int[ ] source, int start, int end)
static long[ ] copyOfRange(long[ ] source, int start, int end)
static short[ ] copyOfRange(short[ ] source, int start, int end)
static <T> T[ ] copyOfRange(T[ ] source, int start, int end)
static <T,U> T[ ] copyOfRange(U[ ] source, int start, int end,
                              Class<? extends T[ ]> resultT)
```

The original array is specified by *source*. The range to copy is specified by the indices passed via *start* and *end*. The range runs from *start* to *end* – 1. If the range is longer than *source*, then the copy is padded with zeros (for numeric arrays), **null**s (for object arrays), or **false** (for boolean arrays). In the last form, the type of *resultT* becomes the type of the array returned. If *start* is negative or greater than the length of *source*, an **ArrayIndexOutOfBoundsException** is thrown. If *start* is greater than *end*, an

**IllegalArgumentException** is thrown. If *source* is **null**, a **NullPointerException** is thrown. If *resultT* is incompatible with the type of *source*, an **ArrayStoreException** is thrown.

The **equals( )** method returns **true** if two arrays are equivalent. Otherwise, it returns **false**. The **equals( )** method has the following forms:

```
static boolean equals(boolean array1[ ], boolean array2 [ ])
static boolean equals(byte array1[ ], byte array2 [ ])
static boolean equals(char array1[ ], char array2 [ ])
static boolean equals(double array1[ ], double array2 [ ])
static boolean equals(float array1[ ], float array2 [ ])
static boolean equals(int array1[ ], int array2 [ ])
static boolean equals(long array1[ ], long array2 [ ])
static boolean equals(short array1[ ], short array2 [ ])
static boolean equals(Object array1[ ], Object array2 [ ])
```

Here, *array1* and *array2* are the two arrays that are compared for equality.

The **deepEquals( )** method can be used to determine if two arrays, which might contain nested arrays, are equal. It has this declaration:

```
static boolean deepEquals(Object[ ] a, Object[ ] b)
```

It returns **true** if the arrays passed in *a* and *b* contain the same elements. If *a* and *b* contain nested arrays, then the contents of those nested arrays are also checked. It returns **false** if the arrays, or any nested arrays, differ.

The **fill( )** method assigns a value to all elements in an array. In other words, it fills an array with a specified value. The **fill( )** method has two versions. The first version, which has the following forms, fills an entire array:

```
static void fill(boolean array[ ], boolean value)
static void fill(byte array[ ], byte value)
static void fill(char array[ ], char value)
static void fill(double array[ ], double value)
static void fill(float array[ ], float value)
static void fill(int array[ ], int value)
static void fill(long array[ ], long value)
static void fill(short array[ ], short value)
static void fill(Object array[ ], Object value)
```

Here, *value* is assigned to all elements in *array*. The second version of the **fill( )** method assigns a value to a subset of an array.

The **sort( )** method sorts an array so that it is arranged in ascending order. The **sort( )** method has two versions. The first version, shown here, sorts the entire array:

```
static void sort(byte array[ ])
static void sort(char array[ ])
static void sort(double array[ ])
static void sort(float array[ ])
static void sort(int array[ ])
static void sort(long array[ ])
```

static void sort(short *array*[ ])
static void sort(Object *array*[ ])
static <T> void sort(T *array*[ ], Comparator<? super T> *c*)

Here, *array* is the array to be sorted. In the last form, *c* is a **Comparator** that is used to order the elements of *array*. The last two forms can throw a **ClassCastException** if elements of the array being sorted are not comparable. The second version of **sort( )** enables you to specify a range within an array that you want to sort.

JDK 8 adds several new methods to **Arrays**. Perhaps the most important is **parallelSort( )** because it sorts, into ascending order, portions of an array in parallel and then merges the results. This approach can greatly speed up sorting times. Like **sort( )**, there are two basic types of **parallelSort( )**, each with several overloads. The first type sorts the entire array. It is shown here:

static void parallelSort(byte *array*[ ])
static void parallelSort(char *array*[ ])
static void parallelSort(double *array*[ ])
static void parallelSort(float *array*[ ])
static void parallelSort(int *array*[ ])
static void parallelSort(long *array*[ ])
static void parallelSort(short *array*[ ])
static <T extends Comparable<? super T>> void parallelSort(T *array*[ ])
static <T> void parallelSort(T *array*[ ], Comparator<? super T> *c*)

Here, *array* is the array to be sorted. In the last form, *c* is a comparator that is used to order the elements in the array. The last two forms can throw a **ClassCastException** if the elements of the array being sorted are not comparable. The second version of **parallelSort( )** enables you to specify a range within the array that you want to sort.

JDK 8 gives **Arrays** support for spliterators by including the **spliterator( )** method. It has two basic forms. The first type returns a spliterator to an entire array. It is shown here:

static Spliterator.OfDouble spliterator(double *array*[ ])
static Spliterator.OfInt spliterator(int *array*[ ])
static Spliterator.OfLong spliterator(long *array*[ ])
static <T> Spliterator spliterator(T *array*[ ])

Here, *array* is the array that the spliterator will cycle through. The second version of **spliterator( )** enables you to specify a range to iterate within the array.

Beginning with JDK 8, **Arrays** supports the new **Stream** interface (see Chapter 29) by including the **stream( )** method. It has two forms. The first is shown here:

static DoubleStream stream(double *array*[ ])
static IntStream stream(int *array*[ ])
static LongStream stream(long *array*[ ])
static <T> Stream stream(T *array*[ ])

Here, *array* is the array to which the stream will refer. The second version of **stream( )** enables you to specify a range within the array.

In addition to those just discussed, JDK 8 adds three other new methods. Two are related: **setAll( )** and **parallelSetAll( )**. Both assign values to all of the elements, but **parallelSetAll( )** works in parallel. Here is an example of each:

```
static void setAll(double array[ ],
                   IntToDoubleFunction<? extends T> genVal)

static void parallelSetAll(double array[ ],
                           IntToDoubleFunction<? extends T> genVal)
```

Several overloads exist for each of these that handle types **int**, **long**, and generic.

Finally, JDK 8 includes one of the more intriguing additions to **Arrays**. It is called **parallelPrefix( )**, and it modifies an array so that each element contains the cumulative result of an operation applied to all previous elements. For example, if the operation is multiplication, then on return, the array elements will contain the values associated with the running product of the original values. It has several overloads. Here is one example:

```
static void parallelPrefix(double array[ ], DoubleBinaryOperator func)
```

Here, *array* is the array being acted upon, and *func* specifies the operation applied. (**DoubleBinaryOperator** is a functional interface defined in **java.util.function**.) Many other versions are provided, including those that operate on types **int**, **long**, and generic, and those that let you specify a range within the array on which to operate.

**Arrays** also provides **toString( )** and **hashCode( )** for the various types of arrays. In addition, **deepToString( )** and **deepHashCode( )** are provided, which operate effectively on arrays that contain nested arrays.

The following program illustrates how to use some of the methods of the **Arrays** class:

```
// Demonstrate Arrays
import java.util.*;

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

    // Allocate and initialize array.
    int array[] = new int[10];
    for(int i = 0; i < 10; i++)
      array[i] = -3 * i;

    // Display, sort, and display the array.
    System.out.print("Original contents: ");
    display(array);
    Arrays.sort(array);
    System.out.print("Sorted: ");
    display(array);

    // Fill and display the array.
    Arrays.fill(array, 2, 6, -1);
    System.out.print("After fill(): ");
    display(array);

    // Sort and display the array.
```

```
    Arrays.sort(array);
    System.out.print("After sorting again: ");
    display(array);

    // Binary search for -9.
    System.out.print("The value -9 is at location ");
    int index =
      Arrays.binarySearch(array, -9);

    System.out.println(index);
  }

  static void display(int array[]) {
    for(int i: array)
      System.out.print(i + " ");

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

The following is the output from this program:

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```

## The Legacy Classes and Interfaces

As explained at the start of this chapter, early versions of **java.util** did not include the Collections Framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing objects. When collections were added (by J2SE 1.2), several of the original classes were reengineered to support the collection interfaces. Thus, they are now technically part of the Collections Framework. However, where a modern collection duplicates the functionality of a legacy class, you will usually want to use the newer collection class. In general, the legacy classes are supported because there is still code that uses them.

One other point: none of the modern collection classes described in this chapter are synchronized, but all the legacy classes are synchronized. This distinction may be important in some situations. Of course, you can easily synchronize collections by using one of the algorithms provided by **Collections**.

The legacy classes defined by **java.util** are shown here:

| | | | | |
|---|---|---|---|---|
| Dictionary | Hashtable | Properties | Stack | Vector |

There is one legacy interface called **Enumeration**. The following sections examine **Enumeration** and each of the legacy classes, in turn.

## The Enumeration Interface

The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects. This legacy interface has been superseded by **Iterator**. Although not deprecated, **Enumeration** is considered obsolete for new code. However, it is used by several methods defined by the legacy classes (such as **Vector** and **Properties**) and is used by several other API classes. Because it is still in use, it was retrofitted for generics by JDK 5. It has this declaration:

```
interface Enumeration<E>
```

where **E** specifies the type of element being enumerated.

**Enumeration** specifies the following two methods:

```
boolean hasMoreElements( )
E nextElement( )
```

When implemented, **hasMoreElements( )** must return **true** while there are still more elements to extract, and **false** when all the elements have been enumerated. **nextElement( )** returns the next object in the enumeration. That is, each call to **nextElement( )** obtains the next object in the enumeration. It throws **NoSuchElementException** when the enumeration is complete.

## Vector

**Vector** implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that duplicate the functionality of methods defined by the Collections Framework. With the advent of collections, **Vector** was reengineered to extend **AbstractList** and to implement the **List** interface. With the release of JDK 5, it was retrofitted for generics and reengineered to implement **Iterable**. This means that **Vector** is fully compatible with collections, and a **Vector** can have its contents iterated by the enhanced **for** loop.

**Vector** is declared like this:

```
class Vector<E>
```

Here, **E** specifies the type of element that will be stored.

Here are the **Vector** constructors:

```
Vector( )
Vector(int size)
Vector(int size, int incr)
Vector(Collection<? extends E> c)
```

The first form creates a default vector, which has an initial size of 10. The second form creates a vector whose initial capacity is specified by *size*. The third form creates a vector whose initial capacity is specified by *size* and whose increment is specified by *incr*. The increment specifies the number of elements to allocate each time that a vector is resized upward. The fourth form creates a vector that contains the elements of collection *c*.

All vectors start with an initial capacity. After this initial capacity is reached, the next time that you attempt to store an object in the vector, the vector automatically allocates space for that object plus extra room for additional objects. By allocating more than just the required memory, the vector reduces the number of allocations that must take place as the vector grows. This reduction is important, because allocations are costly in terms of time. The amount of extra space allocated during each reallocation is determined by the increment that you specify when you create the vector. If you don't specify an increment, the vector's size is doubled by each allocation cycle.

**Vector** defines these protected data members:

```
int capacityIncrement;
int elementCount;
Object[ ] elementData;
```

The increment value is stored in **capacityIncrement**. The number of elements currently in the vector is stored in **elementCount**. The array that holds the vector is stored in **elementData**.

In addition to the collections methods specified by **List**, **Vector** defines several legacy methods, which are summarized in Table 18-16.

Because **Vector** implements **List**, you can use a vector just like you use an **ArrayList** instance. You can also manipulate one using its legacy methods. For example, after you instantiate a **Vector**, you can add an element to it by calling **addElement( )**. To obtain the element at a specific location, call **elementAt( )**. To obtain the first element in the vector, call **firstElement( )**. To retrieve the last element, call **lastElement( )**. You can obtain the index of an element by using **indexOf( )** and **lastIndexOf( )**. To remove an element, call **removeElement( )** or **removeElementAt( )**.

| Method | Description |
|---|---|
| void addElement(E *element*) | The object specified by *element* is added to the vector. |
| int capacity( ) | Returns the capacity of the vector. |
| Object clone( ) | Returns a duplicate of the invoking vector. |
| boolean contains(Object *element*) | Returns **true** if *element* is contained by the vector, and returns **false** if it is not. |
| void copyInto(Object *array*[ ]) | The elements contained in the invoking vector are copied into the array specified by *array*. |
| E elementAt(int *index*) | Returns the element at the location specified by *index*. |
| Enumeration<E> elements( ) | Returns an enumeration of the elements in the vector. |
| void ensureCapacity(int *size*) | Sets the minimum capacity of the vector to *size*. |
| E firstElement( ) | Returns the first element in the vector. |
| int indexOf(Object *element*) | Returns the index of the first occurrence of *element*. If the object is not in the vector, –1 is returned. |

**Table 18-16** The Legacy Methods Defined by **Vector**

| Method | Description |
|---|---|
| int indexOf(Object *element*, int *start*) | Returns the index of the first occurrence of *element* at or after *start*. If the object is not in that portion of the vector, –1 is returned. |
| void insertElementAt(E *element*, int *index*) | Adds *element* to the vector at the location specified by *index*. |
| boolean isEmpty( ) | Returns **true** if the vector is empty, and returns **false** if it contains one or more elements. |
| E lastElement( ) | Returns the last element in the vector. |
| int lastIndexOf(Object *element*) | Returns the index of the last occurrence of *element*. If the object is not in the vector, –1 is returned. |
| int lastIndexOf(Object *element*, int *start*) | Returns the index of the last occurrence of *element* before *start*. If the object is not in that portion of the vector, –1 is returned. |
| void removeAllElements( ) | Empties the vector. After this method executes, the size of the vector is zero. |
| boolean removeElement(Object *element*) | Removes *element* from the vector. If more than one instance of the specified object exists in the vector, then it is the first one that is removed. Returns **true** if successful and **false** if the object is not found. |
| void removeElementAt(int *index*) | Removes the element at the location specified by *index*. |
| void setElementAt(E *element*, int *index*) | The location specified by *index* is assigned *element*. |
| void setSize(int *size*) | Sets the number of elements in the vector to *size*. If the new size is less than the old size, elements are lost. If the new size is larger than the old size, **null** elements are added. |
| int size( ) | Returns the number of elements currently in the vector. |
| String toString( ) | Returns the string equivalent of the vector. |
| void trimToSize( ) | Sets the vector's capacity equal to the number of elements that it currently holds. |

**Table 18-16** The Legacy Methods Defined by **Vector** *(continued)*

The following program uses a vector to store various types of numeric objects. It demonstrates several of the legacy methods defined by **Vector**. It also demonstrates the **Enumeration** interface.

```
// Demonstrate various Vector operations.
import java.util.*;

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

```
    // initial size is 3, increment is 2
    Vector<Integer> v = new Vector<Integer>(3, 2);

    System.out.println("Initial size: " + v.size());
    System.out.println("Initial capacity: " +
                       v.capacity());

    v.addElement(1);
    v.addElement(2);
    v.addElement(3);
    v.addElement(4);

    System.out.println("Capacity after four additions: " +
                       v.capacity());

    v.addElement(5);
    System.out.println("Current capacity: " +
                       v.capacity());

    v.addElement(6);
    v.addElement(7);

    System.out.println("Current capacity: " +
                       v.capacity());

    v.addElement(9);
    v.addElement(10);

    System.out.println("Current capacity: " +
                       v.capacity());

    v.addElement(11);
    v.addElement(12);

    System.out.println("First element: " + v.firstElement());
    System.out.println("Last element: " + v.lastElement());

    if(v.contains(3))
      System.out.println("Vector contains 3.");

    // Enumerate the elements in the vector.
    Enumeration<Integer> vEnum = v.elements();

    System.out.println("\nElements in vector:");
    while(vEnum.hasMoreElements())
      System.out.print(vEnum.nextElement() + " ");
    System.out.println();
  }
}
```

The output from this program is shown here:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5 6 7 9 10 11 12
```

Instead of relying on an enumeration to cycle through the objects (as the preceding program does), you can use an iterator. For example, the following iterator-based code can be substituted into the program:

```
// Use an iterator to display contents.
Iterator<Integer> vItr = v.iterator();

System.out.println("\nElements in vector:");
while(vItr.hasNext())
  System.out.print(vItr.next() + " ");
System.out.println();
```

You can also use a for-each **for** loop to cycle through a **Vector**, as the following version of the preceding code shows:

```
// Use an enhanced for loop to display contents
System.out.println("\nElements in vector:");
for(int i : v)
  System.out.print(i + " ");

System.out.println();
```

Because the **Enumeration** interface is not recommended for new code, you will usually use an iterator or a for-each **for** loop to enumerate the contents of a vector. Of course, legacy code will employ **Enumeration**. Fortunately, enumerations and iterators work in nearly the same manner.

## Stack

**Stack** is a subclass of **Vector** that implements a standard last-in, first-out stack. **Stack** only defines the default constructor, which creates an empty stack. With the release of JDK 5, **Stack** was retrofitted for generics and is declared as shown here:

class Stack<E>

Here, **E** specifies the type of element stored in the stack.

**Stack** includes all the methods defined by **Vector** and adds several of its own, shown in Table 18-17.

| Method | Description |
|---|---|
| boolean empty( ) | Returns **true** if the stack is empty, and returns **false** if the stack contains elements. |
| E peek( ) | Returns the element on the top of the stack, but does not remove it. |
| E pop( ) | Returns the element on the top of the stack, removing it in the process. |
| E push(E *element*) | Pushes *element* onto the stack. *element* is also returned. |
| int search(Object *element*) | Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned. |

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



To put an object on the top of the stack, call **push( )**. To remove and return the top element, call **pop( )**. You can use **peek( )** to return, but not remove, the top object. An **EmptyStackException** is thrown if you call **pop( )** or **peek( )** when the invoking stack is empty. The **empty( )** method returns **true** if nothing is on the stack. The **search( )** method determines whether an object exists on the stack and returns the number of pops that are required to bring it to the top of the stack. Here is an example that creates a stack, pushes several **Integer** objects onto it, and then pops them off again:

```
// Demonstrate the Stack class.
import java.util.*;

class StackDemo {
  static void showpush(Stack<Integer> st, int a) {
    st.push(a);
    System.out.println("push(" + a + ")");
    System.out.println("stack: " + st);
  }

  static void showpop(Stack<Integer> st) {
    System.out.print("pop -> ");
    Integer a = st.pop();
    System.out.println(a);
    System.out.println("stack: " + st);
  }

  public static void main(String args[]) {
    Stack<Integer> st = new Stack<Integer>();

    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
```

```
    try {
      showpop(st);
    } catch (EmptyStackException e) {
      System.out.println("empty stack");
    }
  }
}
```

The following is the output produced by the program; notice how the exception handler for **EmptyStackException** is caught so that you can gracefully handle a stack underflow:

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

One other point: although **Stack** is not deprecated, **ArrayDeque** is a better choice.

## Dictionary

**Dictionary** is an abstract class that represents a key/value storage repository and operates much like **Map**. Given a key and value, you can store the value in a **Dictionary** object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs. Although not currently deprecated, **Dictionary** is classified as obsolete, because it is fully superseded by **Map**. However, **Dictionary** is still in use and thus is discussed here.

With the advent of JDK 5, **Dictionary** was made generic. It is declared as shown here:

```
class Dictionary<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values. The abstract methods defined by **Dictionary** are listed in Table 18-18.

To add a key and a value, use the **put( )** method. Use **get( )** to retrieve the value of a given key. The keys and values can each be returned as an **Enumeration** by the **keys( )** and **elements( )** methods, respectively. The **size( )** method returns the number of key/value pairs stored in a dictionary, and **isEmpty( )** returns **true** when the dictionary is empty. You can use the **remove( )** method to delete a key/value pair.

---

**REMEMBER** The **Dictionary** class is obsolete. You should implement the **Map** interface to obtain key/value storage functionality.

| Method | Purpose |
|---|---|
| Enumeration<V> elements( ) | Returns an enumeration of the values contained in the dictionary. |
| V get(Object *key*) | Returns the object that contains the value associated with *key*. If *key* is not in the dictionary, a **null** object is returned. |
| boolean isEmpty( ) | Returns **true** if the dictionary is empty, and returns **false** if it contains at least one key. |
| Enumeration<K> keys( ) | Returns an enumeration of the keys contained in the dictionary. |
| V put(K *key*, V *value*) | Inserts a key and its value into the dictionary. Returns **null** if *key* is not already in the dictionary; returns the previous value associated with *key* if *key* is already in the dictionary. |
| V remove(Object *key*) | Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the dictionary, a **null** is returned. |
| int size( ) | Returns the number of entries in the dictionary. |

**Table 18-18** The Abstract Methods Defined by **Dictionary**



## Hashtable

**Hashtable** was part of the original **java.util** and is a concrete implementation of a **Dictionary**. However, with the advent of collections, **Hashtable** was reengineered to also implement the **Map** interface. Thus, **Hashtable** is integrated into the Collections Framework. It is similar to **HashMap**, but is synchronized.

Like **HashMap**, **Hashtable** stores key/value pairs in a hash table. However, neither keys nor values can be **null**. When using a **Hashtable**, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

**Hashtable** was made generic by JDK 5. It is declared like this:

```
class Hashtable<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

A hash table can only store objects that override the **hashCode( )** and **equals( )** methods that are defined by **Object**. The **hashCode( )** method must compute and return the hash code for the object. Of course, **equals( )** compares two objects. Fortunately, many of Java's built-in classes already implement the **hashCode( )** method. For example, the most common type of **Hashtable** uses a **String** object as the key. **String** implements both **hashCode( )** and **equals( )**.

The **Hashtable** constructors are shown here:

```
Hashtable( )
Hashtable(int size)
Hashtable(int size, float fillRatio)
Hashtable(Map<? extends K, ? extends V> m)
```

The first version is the default constructor. The second version creates a hash table that has an initial size specified by *size*. (The default size is 11.) The third version creates a hash table that has an initial size specified by *size* and a fill ratio specified by *fillRatio*. This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash table multiplied by its fill ratio, the hash table is expanded. If you do not specify a fill ratio, then 0.75 is used. Finally, the fourth version creates a hash table that is initialized with the elements in *m*. The default load factor of 0.75 is used.

In addition to the methods defined by the **Map** interface, which **Hashtable** now implements, **Hashtable** defines the legacy methods listed in Table 18-19. Several methods throw **NullPointerException** if an attempt is made to use a **null** key or value.

| Method | Description |
|---|---|
| void clear( ) | Resets and empties the hash table. |
| Object clone( ) | Returns a duplicate of the invoking object. |
| boolean contains(Object *value*) | Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found. |
| boolean containsKey(Object *key*) | Returns **true** if some key equal to *key* exists within the hash table. Returns **false** if the key isn't found. |
| boolean containsValue(Object *value*) | Returns **true** if some value equal to *value* exists within the hash table. Returns **false** if the value isn't found. |
| Enumeration<V> elements( ) | Returns an enumeration of the values contained in the hash table. |
| V get(Object *key*) | Returns the object that contains the value associated with *key*. If *key* is not in the hash table, a **null** object is returned. |
| boolean isEmpty( ) | Returns **true** if the hash table is empty; returns **false** if it contains at least one key. |
| Enumeration<K> keys( ) | Returns an enumeration of the keys contained in the hash table. |
| V put(K *key*, V *value*) | Inserts a key and a value into the hash table. Returns **null** if *key* isn't already in the hash table; returns the previous value associated with *key* if *key* is already in the hash table. |
| void rehash( ) | Increases the size of the hash table and rehashes all of its keys. |
| V remove(Object *key*) | Removes *key* and its value. Returns the value associated with *key*. If *key* is not in the hash table, a **null** object is returned. |
| int size( ) | Returns the number of entries in the hash table. |
| String toString( ) | Returns the string equivalent of a hash table. |

**Table 18-19** The Legacy Methods Defined by **Hashtable**

The following example reworks the bank account program, shown earlier, so that it uses a **Hashtable** to store the names of bank depositors and their current balances:

```
// Demonstrate a Hashtable.
import java.util.*;

class HTDemo {
  public static void main(String args[]) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double>();

    Enumeration<String> names;
    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Show all balances in hashtable.
    names = balance.keys();
    while(names.hasMoreElements()) {
      str = names.nextElement();
      System.out.println(str + ": " +
                         balance.get(str));
    }

    System.out.println();

    // Deposit 1,000 into John Doe's account.
    bal = balance.get("John Doe");
    balance.put("John Doe", bal+1000);
    System.out.println("John Doe's new balance: " +
                       balance.get("John Doe"));
  }
}
```

The output from this program is shown here:

```
Todd Hall: 99.22
Ralph Smith: -19.08
John Doe: 3434.34
Jane Baker: 1378.0
Tom Smith: 123.22

John Doe's new balance: 4434.34
```

One important point: Like the map classes, **Hashtable** does not directly support iterators. Thus, the preceding program uses an enumeration to display the contents of **balance**. However, you can obtain set-views of the hash table, which permits the use of iterators. To do so, you simply use one of the collection-view methods defined by **Map**, such

as **entrySet( )** or **keySet( )**. For example, you can obtain a set-view of the keys and cycle through them using either an iterator or an enhanced **for** loop. Here is a reworked version of the program that shows this technique:

```
// Use iterators with a Hashtable.
import java.util.*;

class HTDemo2 {
  public static void main(String args[]) {
    Hashtable<String, Double> balance =
      new Hashtable<String, Double>();

    String str;
    double bal;

    balance.put("John Doe", 3434.34);
    balance.put("Tom Smith", 123.22);
    balance.put("Jane Baker", 1378.00);
    balance.put("Tod Hall", 99.22);
    balance.put("Ralph Smith", -19.08);

    // Show all balances in hashtable.
    // First, get a set view of the keys.
    Set<String> set = balance.keySet();

    // Get an iterator.
    Iterator<String> itr = set.iterator();
    while(itr.hasNext()) {
      str = itr.next();
      System.out.println(str + ": " +
                         balance.get(str));

    }

    System.out.println();

    // Deposit 1,000 into John Doe's account.
    bal = balance.get("John Doe");
    balance.put("John Doe", bal+1000);
    System.out.println("John Doe's new balance: " +
                       balance.get("John Doe"));
  }
}
```

## Properties

**Properties** is a subclass of **Hashtable**. It is used to maintain lists of values in which the key is a **String** and the value is also a **String**. The **Properties** class is used by some other Java classes. For example, it is the type of object returned by **System.getProperties( )** when obtaining environmental values. Although the **Properties** class, itself, is not generic, several of its methods are.

**Properties** defines the following instance variable:

Properties defaults;

This variable holds a default property list associated with a **Properties** object. **Properties** defines these constructors:

```
Properties( )
Properties(Properties propDefault)
```

The first version creates a **Properties** object that has no default values. The second creates an object that uses *propDefault* for its default values. In both cases, the property list is empty.

In addition to the methods that **Properties** inherits from **Hashtable**, **Properties** defines the methods listed in Table 18-20. **Properties** also contains one deprecated method: **save( )**. This was replaced by **store( )** because **save( )** did not handle errors correctly.



| Method | Description |
|---|---|
| String getProperty(String *key*) | Returns the value associated with *key*. A **null** object is returned if *key* is neither in the list nor in the default property list. |
| String getProperty(String *key*, String *defaultProperty*) | Returns the value associated with *key*. *defaultProperty* is returned if *key* is neither in the list nor in the default property list. |
| void list(PrintStream *streamOut*) | Sends the property list to the output stream linked to *streamOut*. |
| void list(PrintWriter *streamOut*) | Sends the property list to the output stream linked to *streamOut*. |
| void load(InputStream *streamIn*) throws IOException | Inputs a property list from the input stream linked to *streamIn*. |
| void load(Reader *streamIn*) throws IOException | Inputs a property list from the input stream linked to *streamIn*. |
| void loadFromXML(InputStream *streamIn*) throws IOException, InvalidPropertiesFormatException | Inputs a property list from an XML document linked to *streamIn*. |
| Enumeration<?> propertyNames( ) | Returns an enumeration of the keys. This includes those keys found in the default property list, too. |
| Object setProperty(String *key*, String *value*) | Associates *value* with *key*. Returns the previous value associated with *key*, or returns **null** if no such association exists. |
| void store(OutputStream *streamOut*, String *description*) throws IOException | After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*. |
| void store(Writer *streamOut*, String *description*) throws IOException | After writing the string specified by *description*, the property list is written to the output stream linked to *streamOut*. |
| void storeToXML(OutputStream *streamOut*, String *description*) throws IOException | After writing the string specified by *description*, the property list is written to the XML document linked to *streamOut*. |

**Table 18-20** The Methods Defined by **Properties**

| Method | Description |
|---|---|
| void storeToXML(OutputStream *streamOut*,<br>String *description*,<br>String *enc*) | The property list and the string specified by *description* is written to the XML document linked to *streamOut* using the specified character encoding. |
| Set<String> stringPropertyNames( ) | Returns a set of keys. |

**Table 18-20** The Methods Defined by **Properties** *(continued)*

One useful capability of the **Properties** class is that you can specify a default property that will be returned if no value is associated with a certain key. For example, a default value can be specified along with the key in the **getProperty( )** method—such as **getProperty( "name" ,"default value")**. If the "name" value is not found, then "default value" is returned. When you construct a **Properties** object, you can pass another instance of **Properties** to be used as the default properties for the new instance. In this case, if you call **getProperty("foo")** on a given **Properties** object, and "foo" does not exist, Java looks for "foo" in the default **Properties** object. This allows for arbitrary nesting of levels of default properties.

The following example demonstrates **Properties**. It creates a property list in which the keys are the names of states and the values are the names of their capitals. Notice that the attempt to find the capital for Florida includes a default value.

```
// Demonstrate a Property list.
import java.util.*;

class PropDemo {
  public static void main(String args[]) {
    Properties capitals = new Properties();

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Get a set-view of the keys.
    Set<?> states = capitals.keySet();

    // Show all of the states and capitals.
    for(Object name : states)
      System.out.println("The capital of " +
                         name + " is " +
                         capitals.getProperty((String)name)
                         + ".");

    System.out.println();

    // Look for state not in list -- specify default.
    String str = capitals.getProperty("Florida", "Not Found");
    System.out.println("The capital of Florida is " + str + ".");
  }
}
```

The output from this program is shown here:

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.
```

Since Florida is not in the list, the default value is used.

Although it is perfectly valid to use a default value when you call **getProperty( )**, as the preceding example shows, there is a better way of handling default values for most applications of property lists. For greater flexibility, specify a default property list when constructing a **Properties** object. The default list will be searched if the desired key is not found in the main list. For example, the following is a slightly reworked version of the preceding program, with a default list of states specified. Now, when Florida is sought, it will be found in the default list:

```
// Use a default property list.
import java.util.*;

class PropDemoDef {
  public static void main(String args[]) {
    Properties defList = new Properties();
    defList.put("Florida", "Tallahassee");
    defList.put("Wisconsin", "Madison");

    Properties capitals = new Properties(defList);

    capitals.put("Illinois", "Springfield");
    capitals.put("Missouri", "Jefferson City");
    capitals.put("Washington", "Olympia");
    capitals.put("California", "Sacramento");
    capitals.put("Indiana", "Indianapolis");

    // Get a set-view of the keys.
    Set<?> states = capitals.keySet();

    // Show all of the states and capitals.
    for(Object name : states)
      System.out.println("The capital of " +
                         name + " is " +
                         capitals.getProperty((String)name)
                         + ".");

    System.out.println();

    // Florida will now be found in the default list.
    String str = capitals.getProperty("Florida");
    System.out.println("The capital of Florida is "
                       + str + ".");
  }
}
```

## Using store( ) and load( )

One of the most useful aspects of **Properties** is that the information contained in a **Properties** object can be easily stored to or loaded from disk with the **store( )** and **load( )** methods. At any time, you can write a **Properties** object to a stream or read it back. This makes property lists especially convenient for implementing simple databases. For example, the following program uses a property list to create a simple computerized telephone book that stores names and phone numbers. To find a person's number, you enter his or her name. The program uses the **store( )** and **load( )** methods to store and retrieve the list. When the program executes, it first tries to load the list from a file called **phonebook.dat**. If this file exists, the list is loaded. You can then add to the list. If you do, the new list is saved when you terminate the program. Notice how little code is required to implement a small, but functional, computerized phone book.

```
/* A simple telephone number database that uses
   a property list. */
import java.io.*;
import java.util.*;

class Phonebook {
  public static void main(String args[])
    throws IOException
  {
    Properties ht = new Properties();
    BufferedReader br =
      new BufferedReader(new InputStreamReader(System.in));
    String name, number;
    FileInputStream fin = null;
    boolean changed = false;

    // Try to open phonebook.dat file.
    try {
      fin = new FileInputStream("phonebook.dat");
    } catch(FileNotFoundException e) {
      // ignore missing file
    }

    /* If phonebook file already exists,
       load existing telephone numbers. */
    try {
      if(fin != null) {
        ht.load(fin);
        fin.close();
      }
    } catch(IOException e) {
      System.out.println("Error reading file.");
    }

    // Let user enter new names and numbers.
    do {
      System.out.println("Enter new name" +
                         " ('quit' to stop): ");
```

```
      name = br.readLine();
      if(name.equals("quit")) continue;

      System.out.println("Enter number: ");
      number = br.readLine();

      ht.put(name, number);
      changed = true;
    } while(!name.equals("quit"));

    // If phone book data has changed, save it.
    if(changed) {
      FileOutputStream fout = new FileOutputStream("phonebook.dat");

      ht.store(fout, "Telephone Book");
      fout.close();
    }

    // Look up numbers given a name.
    do {
      System.out.println("Enter name to find" +
                         " ('quit' to quit): ");
      name = br.readLine();
      if(name.equals("quit")) continue;

      number = (String) ht.get(name);
      System.out.println(number);
    } while(!name.equals("quit"));
  }
}
```



## Parting Thoughts on Collections

The Collections Framework gives you, the programmer, a powerful set of well-engineered solutions to some of programming's most common tasks. Consider using a collection the next time that you need to store and retrieve information. Remember, collections need not be reserved for only the "large jobs," such as corporate databases, mailing lists, or inventory systems. They are also effective when applied to smaller jobs. For example, a **TreeMap** might make an excellent collection to hold the directory structure of a set of files. A **TreeSet** could be quite useful for storing project-management information. Frankly, the types of problems that will benefit from a collections-based solution are limited only by your imagination. One last point: In Chapter 29, the new stream API is discussed. Because streams are now integrated with collections, consider using a stream when operating on a collection.

This page has been intentionally left blank

CHAPTER

# 19 java.util Part 2: More Utility Classes

This chapter continues our discussion of **java.util** by examining those classes and interfaces that are not part of the Collections Framework. These include classes that tokenize strings, work with dates, compute random numbers, bundle resources, and observe events. Also covered are the **Formatter** and **Scanner** classes which make it easy to write and read formatted data, and the new **Optional** class, which makes it easier to handle situations in which a value may be absent. Finally, the subpackages of **java.util** are summarized at the end of this chapter. Of particular interest is **java.util.function**, which defines several standard functional interfaces.

## StringTokenizer

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

To use **StringTokenizer**, you specify an input string and a string that contains delimiters. *Delimiters* are characters that separate tokens. Each character in the delimiters string is considered a valid delimiter—for example, **",;:"** sets the delimiters to a comma, semicolon, and colon. The default set of delimiters consists of the whitespace characters: space, tab, form feed, newline, and carriage return.

The **StringTokenizer** constructors are shown here:

StringTokenizer(String *str*)
StringTokenizer(String *str*, String *delimiters*)
StringTokenizer(String *str*, String *delimiters*, boolean *delimAsToken*)

In all versions, *str* is the string that will be tokenized. In the first version, the default delimiters are used. In the second and third versions, *delimiters* is a string that specifies the delimiters. In the third version, if *delimAsToken* is **true**, then the delimiters are also returned

as tokens when the string is parsed. Otherwise, the delimiters are not returned. Delimiters are not returned as tokens by the first two forms.

Once you have created a **StringTokenizer** object, the **nextToken( )** method is used to extract consecutive tokens. The **hasMoreTokens( )** method returns **true** while there are more tokens to be extracted. Since **StringTokenizer** implements **Enumeration**, the **hasMoreElements( )** and **nextElement( )** methods are also implemented, and they act the same as **hasMoreTokens( )** and **nextToken( )**, respectively. The **StringTokenizer** methods are shown in Table 19-1.

Here is an example that creates a **StringTokenizer** to parse "key=value" pairs. Consecutive sets of "key=value" pairs are separated by a semicolon.

```
// Demonstrate StringTokenizer.
import java.util.StringTokenizer;

class STDemo {
  static String in = "title=Java: The Complete Reference;" +
    "author=Schildt;" +
    "publisher=McGraw-Hill;" +
    "copyright=2014";

  public static void main(String args[]) {
    StringTokenizer st = new StringTokenizer(in, "=;");

    while(st.hasMoreTokens()) {
      String key = st.nextToken();
      String val = st.nextToken();
      System.out.println(key + "\t" + val);
    }
  }
}
```

The output from this program is shown here:

```
title  Java: The Complete Reference
author  Schildt
publisher  McGraw-Hill
copyright  2014
```

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

**Table 19-1** The Methods Defined by **StringTokenizer**

## BitSet

A **BitSet** class creates a special type of array that holds bit values in the form of **boolean** values. This array can increase in size as needed. This makes it similar to a vector of bits. The **BitSet** constructors are shown here:

BitSet( )
BitSet(int *size*)

The first version creates a default object. The second version allows you to specify its initial size (that is, the number of bits that it can hold). All bits are initialized to **false**.

**BitSet** defines the methods listed in Table 19-2.



| Method | Description |
|---|---|
| void and(BitSet *bitSet*) | ANDs the contents of the invoking **BitSet** object with those specified by *bitSet*. The result is placed into the invoking object. |
| void andNot(BitSet *bitSet*) | For each set bit in *bitSet*, the corresponding bit in the invoking **BitSet** is cleared. |
| int cardinality( ) | Returns the number of set bits in the invoking object. |
| void clear( ) | Zeros all bits. |
| void clear(int *index*) | Zeros the bit specified by *index*. |
| void clear(int *startIndex*, int *endIndex*) | Zeros the bits from *startIndex* to *endIndex*–1. |
| Object clone( ) | Duplicates the invoking **BitSet** object. |
| boolean equals(Object *bitSet*) | Returns **true** if the invoking bit set is equivalent to the one passed in *bitSet*. Otherwise, the method returns **false**. |
| void flip(int *index*) | Reverses the bit specified by *index*. |
| void flip(int *startIndex*, int *endIndex*) | Reverses the bits from *startIndex* to *endIndex*–1. |
| boolean get(int *index*) | Returns the current state of the bit at the specified index. |
| BitSet get(int *startIndex*, int *endIndex*) | Returns a **BitSet** that consists of the bits from *startIndex* to *endIndex*–1. The invoking object is not changed. |
| int hashCode( ) | Returns the hash code for the invoking object. |
| boolean intersects(BitSet *bitSet*) | Returns **true** if at least one pair of corresponding bits within the invoking object and *bitSet* are set. |
| boolean isEmpty( ) | Returns **true** if all bits in the invoking object are cleared. |
| int length( ) | Returns the number of bits required to hold the contents of the invoking **BitSet**. This value is determined by the location of the last set bit. |
| int nextClearBit(int *startIndex*) | Returns the index of the next cleared bit (that is, the next **false** bit), starting from the index specified by *startIndex*. |

**Table 19-2** The Methods Defined by **BitSet**

| Method | Description |
|---|---|
| int nextSetBit(int *startIndex*) | Returns the index of the next set bit (that is, the next **true** bit), starting from the index specified by *startIndex*. If no bit is set, –1 is returned. |
| void or(BitSet *bitSet*) | ORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object. |
| int previousClearBit(int *startIndex*) | Returns the index of the next cleared bit (that is, the next **false** bit) at or prior to the index specified by *startIndex*. If no cleared bit is found, –1 is returned. |
| int previousSetBit(int *startIndex*) | Returns the index of the next set bit (that is, the next **true** bit) at or prior to the index specified by *startIndex*. If no set bit is found, –1 is returned. |
| void set(int *index*) | Sets the bit specified by *index*. |
| void set(int *index*, boolean *v*) | Sets the bit specified by *index* to the value passed in *v*. **true** sets the bit; **false** clears the bit. |
| void set(int *startIndex*, int *endIndex*) | Sets the bits from *startIndex* to *endIndex*–1. |
| void set(int *startIndex*, int *endIndex*, boolean *v*) | Sets the bits from *startIndex* to *endIndex*–1 to the value passed in *v*. **true** sets the bits; **false** clears the bits. |
| int size( ) | Returns the number of bits in the invoking **BitSet** object. |
| IntStream stream( ) | Returns a stream that contains the bit positions, from low to high, that have set bits. (Added by JDK 8.) |
| byte[ ] toByteArray( ) | Returns a **byte** array that contains the invoking **BitSet** object. |
| long[ ] toLongArray( ) | Returns a **long** array that contains the invoking **BitSet** object. |
| String toString( ) | Returns the string equivalent of the invoking **BitSet** object. |
| static BitSet valueOf(byte[ ] *v*) | Returns a **BitSet** that contains the bits in *v*. |
| static BitSet valueOf(ByteBuffer *v*) | Returns a **BitSet** that contains the bits in *v*. |
| static BitSet valueOf(long[ ] *v*) | Returns a **BitSet** that contains the bits in *v*. |
| static BitSet valueOf(LongBuffer *v*) | Returns a **BitSet** that contains the bits in *v*. |
| void xor(BitSet *bitSet*) | XORs the contents of the invoking **BitSet** object with that specified by *bitSet*. The result is placed into the invoking object. |

**Table 19-2** The Methods Defined by **BitSet** *(continued)*

Here is an example that demonstrates **BitSet**:

```
// BitSet Demonstration.
import java.util.BitSet;

class BitSetDemo {
  public static void main(String args[]) {
    BitSet bits1 = new BitSet(16);
    BitSet bits2 = new BitSet(16);

    // set some bits
    for(int i=0; i<16; i++) {
      if((i%2) == 0) bits1.set(i);
      if((i%5) != 0) bits2.set(i);
    }

    System.out.println("Initial pattern in bits1: ");
    System.out.println(bits1);
    System.out.println("\nInitial pattern in bits2: ");
    System.out.println(bits2);

    // AND bits
    bits2.and(bits1);
    System.out.println("\nbits2 AND bits1: ");
    System.out.println(bits2);

    // OR bits
    bits2.or(bits1);
    System.out.println("\nbits2 OR bits1: ");
    System.out.println(bits2);

    // XOR bits
    bits2.xor(bits1);
    System.out.println("\nbits2 XOR bits1: ");
    System.out.println(bits2);
  }
}
```

The output from this program is shown here. When **toString( )** converts a **BitSet** object to its string equivalent, each set bit is represented by its bit position. Cleared bits are not shown.

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

## Optional, OptionalDouble, OptionalInt, and OptionalLong

JDK 8 adds classes called **Optional**, **OptionalDouble**, **OptionalInt**, and **OptionalLong** that offer a way to handle situations in which a value may or may not be present. In the past, you would normally use the value **null** to indicate that no value is present. However, this can lead to null pointer exceptions if an attempt is made to dereference a null reference. As a result, frequent checks for a **null** value were necessary to avoid generating an exception. These classes provide a better way to handle such situations.

The first and most general of these classes is **Optional**. For this reason, it is the primary focus of this discussion. It is shown here:

```
class Optional<T>
```

Here, **T** specifies the type of value stored. It is important to understand that an **Optional** instance can either contain a value of type **T** or be empty. In other words, an **Optional** object does not necessarily contain a value. **Optional** does not define any constructors, but it does define several methods that let you work with **Optional** objects. For example, you can determine if a value is present, obtain the value if it is present, obtain a default value when no value is present, and construct an **Optional** value. The **Optional** methods are shown in Table 19-3.

| Method | Description |
|---|---|
| static <T> Optional<T> empty( ) | Returns an object for which **isPresent( )** returns **false**. |
| boolean equals(Object *optional*) | Returns **true** if the invoking object equals *optional*. Otherwise, returns **false**. |
| Optional<T> filter( Predicate<? super T> *condition*) | Returns an **Optional** instance that contains the same value as the invoking object if that value satisfies *condition*. Otherwise, an empty object is returned. |
| U Optional<U> flatMap( Function<? super T, Optional<U>> *mapFunc*) | Applies the mapping function specified by *mapFunc* to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise. |
| T get( ) | Returns the value in the invoking object. However, if no value is present, **NoSuchElementException** is thrown. |
| int hashCode( ) | Returns a hashcode for the invoking object. |
| void ifPresent( Consumer<? super T> *func*) | Calls *func* if a value is present in the invoking object, passing the object to *func*. If no value is present, no action occurs. |
| boolean isPresent( ) | Returns **true** if the invoking object contains a value. Returns **false** if no value is present. |
| U Optional<U> map( Function<? super T, ? extends U>> *mapFunc*) | Applies the mapping function specified by *mapFunc* to the invoking object if that object contains a value and returns the result. Returns an empty object otherwise. |
| static <T> Optional<T> of(T *val*) | Creates an **Optional** instance that contains *val* and returns the result. The value of *val* must not be **null**. |

**Table 19-3** The Methods Defined by **Optional**

| Method | Description |
|---|---|
| static <T> Optional<T> ofNullable(T *val*) | Creates an **Optional** instance that contains *val* and returns the result. However, if *val* is **null**, then an empty **Optional** instance is returned. |
| T orElse(T *defVal*) | If the invoking object contains a value, the value is returned. Otherwise, the value specified by *defVal* is returned. |
| T orElseGet( Supplier<? extends T> *getFunc*) | If the invoking object contains a value, the value is returned. Otherwise, the value obtained from *getFunc* is returned. |
| <X extends Throwable> T orElseThrow( Supplier<? extends X> *excFunc*) throws X extends Throwable | Returns the value in the invoking object. However, if no value is present, the exception generated by *excFunc* is thrown. |
| String toString( ) | Returns a string corresponding to the invoking object. |

**Table 19-3** The Methods Defined by **Optional** *(continued)*



The best way to understand **Optional** is to work through an example that uses its core methods. At the foundation of **Optional** are **isPresent( )** and **get( )**. You can determine if a value is present by calling **isPresent( )**. If a value is available, it will return **true**. Otherwise, **false** is returned. If a value is present in an **Optional** instance, you can obtain it by calling **get( )**. However, if you call **get( )** on an object that does not contain a value, **NoSuchElementException** is thrown. For this reason, you should always first confirm that a value is present before calling **get( )** on an **Optional** object.

Of course, having to call two methods to retrieve a value adds overhead to each access. Fortunately, **Optional** defines methods that combine the check for a value with the retrieval of the value. One such method is **orElse( )**. If the object on which it is called contains a value, the value is returned. Otherwise, a default value is returned.

**Optional** does not define any constructors. Instead, you will use one of its methods to create an instance. For example, you can create an **Optional** instance with a specified value by using **of( )**. You can create an instance of **Optional** that does not contain a value by using **empty( )**.

The following program demonstrates these methods:

```
// Demonstrate several Optional<T> methods

import java.util.*;

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

    Optional<String> noVal = Optional.empty();

    Optional<String> hasVal = Optional.of("ABCDEFG");
```

```
    if(noVal.isPresent()) System.out.println("This won't be displayed");
    else System.out.println("noVal has no value");

    if(hasVal.isPresent()) System.out.println("The string in hasVal is: " +
                                              hasVal.get());

    String defStr = noVal.orElse("Default String");
    System.out.println(defStr);
  }
}
```

The output is shown here:

```
noVal has no value
The string in hasVal is: ABCDEFG
Default String
```

As the output shows, a value can be obtained from an **Optional** object only if one is present. This basic mechanism enables **Optional** to prevent null pointer exceptions.

The **OptionalDouble**, **OptionalInt**, and **OptionalLong** classes work much like **Optional**, except that they are designed expressly for use on **double**, **int**, and **long** values, respectively. As such, they specify the methods **getAsDouble( )**, **getAsInt( )**, and **getAsLong( )**, respectively, rather than **get( )**. Also, they do not support the **filter( )**, **ofNullable( )**, **map( )** and **flatMap( )** methods.

## Date

The **Date** class encapsulates the current date and time. Before beginning our examination of **Date**, it is important to point out that it has changed substantially from its original version defined by Java 1.0. When Java 1.1 was released, many of the functions carried out by the original **Date** class were moved into the **Calendar** and **DateFormat** classes, and as a result, many of the original 1.0 **Date** methods were deprecated. Since the deprecated 1.0 methods should not be used for new code, they are not described here.

**Date** supports the following non-deprecated constructors:

Date( )
Date(long *millisec*)

The first constructor initializes the object with the current date and time. The second constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970. The nondeprecated methods defined by **Date** are shown in Table 19-4. **Date** also implements the **Comparable** interface.

| Method | Description |
|---|---|
| boolean after(Date *date*) | Returns **true** if the invoking **Date** object contains a date that is later than the one specified by *date*. Otherwise, it returns **false**. |
| boolean before(Date *date*) | Returns **true** if the invoking **Date** object contains a date that is earlier than the one specified by *date*. Otherwise, it returns **false**. |
| Object clone( ) | Duplicates the invoking **Date** object. |
| int compareTo(Date *date*) | Compares the value of the invoking object with that of *date*. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than *date*. Returns a positive value if the invoking object is later than *date*. |
| boolean equals(Object *date*) | Returns **true** if the invoking **Date** object contains the same time and date as the one specified by *date*. Otherwise, it returns **false**. |
| static Date from(Instant t) | Returns a **Date** object corresponding to the **Instant** object passed in *t*. (Added by JDK 8.) |
| long getTime( ) | Returns the number of milliseconds that have elapsed since January 1, 1970. |
| int hashCode( ) | Returns a hash code for the invoking object. |
| void setTime(long *time*) | Sets the time and date as specified by *time*, which represents an elapsed time in milliseconds from midnight, January 1, 1970. |
| Instant toInstant( ) | Returns an **Instant** object corresponding to the invoking **Date** object. (Added by JDK 8.) |
| String toString( ) | Converts the invoking **Date** object into a string and returns the result. |

**Table 19-4** The Nondeprecated Methods Defined by **Date**



As you can see by examining Table 19-4, the non-deprecated **Date** features do not allow you to obtain the individual components of the date or time. As the following program demonstrates, you can only obtain the date and time in terms of milliseconds, in its default string representation as returned by **toString( )**, or (beginning with JDK 8) as an **Instant** object. To obtain more-detailed information about the date and time, you will use the **Calendar** class.

```
// Show date and time using only Date methods.
import java.util.Date;

class DateDemo {
  public static void main(String args[]) {
    // Instantiate a Date object
    Date date = new Date();

    // display time and date using toString()
    System.out.println(date);

    // Display number of milliseconds since midnight, January 1, 1970 GMT
    long msec = date.getTime();
    System.out.println("Milliseconds since Jan. 1, 1970 GMT = " + msec);
  }
}
```

Sample output is shown here:

```
Wed Jan 01 11:11:44 CST 2014
Milliseconds since Jan. 1, 1970 GMT = 1388596304803
```

## Calendar

The abstract **Calendar** class provides a set of methods that allows you to convert a time in milliseconds to a number of useful components. Some examples of the type of information that can be provided are year, month, day, hour, minute, and second. It is intended that subclasses of **Calendar** will provide the specific functionality to interpret time information according to their own rules. This is one aspect of the Java class library that enables you to write programs that can operate in international environments. An example of such a subclass is **GregorianCalendar**.

---

**NOTE** JDK 8 defines a new date and time API in **java.time**, which new applications may want to employ. See Chapter 30.

**Calendar** provides no public constructors. **Calendar** defines several protected instance variables. **areFieldsSet** is a **boolean** that indicates if the time components have been set. **fields** is an array of **int**s that holds the components of the time. **isSet** is a **boolean** array that indicates if a specific time component has been set. **time** is a **long** that holds the current time for this object. **isTimeSet** is a **boolean** that indicates if the current time has been set.

A sampling of methods defined by **Calendar** are shown in Table 19-5.

| Method | Description |
|---|---|
| abstract void add(int *which*, int *val*) | Adds *val* to the time or date component specified by *which*. To subtract, add a negative value. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**. |
| boolean after(Object *calendarObj*) | Returns **true** if the invoking **Calendar** object contains a date that is later than the one specified by *calendarObj*. Otherwise, it returns **false**. |
| boolean before(Object *calendarObj*) | Returns **true** if the invoking **Calendar** object contains a date that is earlier than the one specified by *calendarObj*. Otherwise, it returns **false**. |
| final void clear( ) | Zeros all time components in the invoking object. |
| final void clear(int *which*) | Zeros the time component specified by *which* in the invoking object. |
| Object clone( ) | Returns a duplicate of the invoking object. |
| boolean equals(Object *calendarObj*) | Returns **true** if the invoking **Calendar** object contains a date that is equal to the one specified by *calendarObj*. Otherwise, it returns **false**. |

**Table 19-5** A Sampling of the Methods Defined by **Calendar**

| Method | Description |
|---|---|
| int get(int *calendarField*) | Returns the value of one component of the invoking object. The component is indicated by *calendarField*. Examples of the components that can be requested are **Calendar.YEAR**, **Calendar.MONTH**, **Calendar.MINUTE**, and so forth. |
| static Locale[ ] getAvailableLocales( ) | Returns an array of **Locale** objects that contains the locales for which calendars are available. |
| static Calendar getInstance( ) | Returns a **Calendar** object for the default locale and time zone. |
| static Calendar getInstance(TimeZone *tz*) | Returns a **Calendar** object for the time zone specified by *tz*. The default locale is used. |
| static Calendar getInstance(Locale *locale*) | Returns a **Calendar** object for the locale specified by *locale*. The default time zone is used. |
| static Calendar getInstance(TimeZone *tz*, Locale *locale*) | Returns a **Calendar** object for the time zone specified by *tz* and the locale specified by *locale*. |
| final Date getTime( ) | Returns a **Date** object equivalent to the time of the invoking object. |
| TimeZone getTimeZone( ) | Returns the time zone for the invoking object. |
| final boolean isSet(int *which*) | Returns **true** if the specified time component is set. Otherwise, it returns **false**. |
| void set(int *which*, int *val*) | Sets the date or time component specified by *which* to the value specified by *val* in the invoking object. *which* must be one of the fields defined by **Calendar**, such as **Calendar.HOUR**. |
| final void set(int *year*, int *month*, int *dayOfMonth*) | Sets various date and time components of the invoking object. |
| final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*) | Sets various date and time components of the invoking object. |
| final void set(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int *seconds*) | Sets various date and time components of the invoking object. |
| final void setTime(Date *d*) | Sets various date and time components of the invoking object. This information is obtained from the **Date** object *d*. |
| void setTimeZone(TimeZone *tz*) | Sets the time zone for the invoking object to that specified by *tz*. |
| final Instant toInstant( ) | Returns an **Instant** object corresponding to the invoking **Calendar** instance. (Added by JDK 8.) |

**Table 19-5** A Sampling of the Methods Defined by **Calendar** *(continued)*

**Calendar** defines the following **int** constants, which are used when you get or set components of the calendar. (The ones with the suffix **FORMAT** or **STANDALONE** were added by JDK 8.)

| | | |
|---|---|---|
| ALL_STYLES | HOUR_OF_DAY | PM |
| AM | JANUARY | SATURDAY |
| AM_PM | JULY | SECOND |
| APRIL | JUNE | SEPTEMBER |
| AUGUST | LONG | SHORT |
| DATE | LONG_FORMAT | SHORT_FORMAT |
| DAY_OF_MONTH | LONG_STANDALONE | SHORT_STANDALONE |
| DAY_OF_WEEK | MARCH | SUNDAY |
| DAY_OF_WEEK_IN_MONTH | MAY | THURSDAY |
| DAY_OF_YEAR | MILLISECOND | TUESDAY |
| DECEMBER | MINUTE | UNDECIMBER |
| DST_OFFSET | MONDAY | WEDNESDAY |
| ERA | MONTH | WEEK_OF_MONTH |
| FEBRUARY | NARROW_FORMAT | WEEK_OF_YEAR |
| FIELD_COUNT | NARROW_STANDALONE | YEAR |
| FRIDAY | NOVEMBER | ZONE_OFFSET |
| HOUR | OCTOBER | |

The following program demonstrates several **Calendar** methods:

```
// Demonstrate Calendar
import java.util.Calendar;

class CalendarDemo {
  public static void main(String args[]) {
    String months[] = {
              "Jan", "Feb", "Mar", "Apr",
              "May", "Jun", "Jul", "Aug",
              "Sep", "Oct", "Nov", "Dec"};

    // Create a calendar initialized with the
    // current date and time in the default
    // locale and timezone.
    Calendar calendar = Calendar.getInstance();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[calendar.get(Calendar.MONTH)]);
    System.out.print(" " + calendar.get(Calendar.DATE) + " ");
    System.out.println(calendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
```

```
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));

    // Set the time and date information and display it.
    calendar.set(Calendar.HOUR, 10);
    calendar.set(Calendar.MINUTE, 29);
    calendar.set(Calendar.SECOND, 22);
    System.out.print("Updated time: ");
    System.out.print(calendar.get(Calendar.HOUR) + ":");
    System.out.print(calendar.get(Calendar.MINUTE) + ":");
    System.out.println(calendar.get(Calendar.SECOND));
  }
}
```

Sample output is shown here:

```
Date: Jan 1 2014
Time: 11:29:39
Updated time: 10:29:22
```



## GregorianCalendar

**GregorianCalendar** is a concrete implementation of a **Calendar** that implements the normal Gregorian calendar with which you are familiar. The **getInstance( )** method of **Calendar** will typically return a **GregorianCalendar** initialized with the current date and time in the default locale and time zone.

**GregorianCalendar** defines two fields: **AD** and **BC**. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for **GregorianCalendar** objects. The default, **GregorianCalendar( )**, initializes the object with the current date and time in the default locale and time zone. Three more constructors offer increasing levels of specificity:

GregorianCalendar(int *year*, int *month*, int *dayOfMonth*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*)
GregorianCalendar(int *year*, int *month*, int *dayOfMonth*, int *hours*, int *minutes*, int *seconds*)

All three versions set the day, month, and year. Here, *year* specifies the year. The month is specified by *month*, with zero indicating January. The day of the month is specified by *dayOfMonth*. The first version sets the time to midnight. The second version also sets the hours and the minutes. The third version adds seconds.

You can also construct a **GregorianCalendar** object by specifying the locale and/or time zone. The following constructors create objects initialized with the current date and time using the specified time zone and/or locale:

GregorianCalendar(Locale *locale*)
GregorianCalendar(TimeZone *timeZone*)
GregorianCalendar(TimeZone *timeZone*, Locale *locale*)

**GregorianCalendar** provides an implementation of all the abstract methods in **Calendar**. It also provides some additional methods. Perhaps the most interesting is **isLeapYear( )**, which tests if the year is a leap year. Its form is

boolean isLeapYear(int *year*)

This method returns **true** if *year* is a leap year and **false** otherwise. JDK 8 also adds the following methods: **from( )** and **toZonedDateTime( )**, which support the new date and time API, and **getCalendarType( )**, which returns the calendar type as a string, which is "gregory".

The following program demonstrates **GregorianCalendar**:

```
// Demonstrate GregorianCalendar
import java.util.*;

class GregorianCalendarDemo {
  public static void main(String args[]) {
    String months[] = {
              "Jan", "Feb", "Mar", "Apr",
              "May", "Jun", "Jul", "Aug",
              "Sep", "Oct", "Nov", "Dec"};
    int year;

    // Create a Gregorian calendar initialized
    // with the current date and time in the
    // default locale and timezone.
    GregorianCalendar gcalendar = new GregorianCalendar();

    // Display current time and date information.
    System.out.print("Date: ");
    System.out.print(months[gcalendar.get(Calendar.MONTH)]);
    System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
    System.out.println(year = gcalendar.get(Calendar.YEAR));

    System.out.print("Time: ");
    System.out.print(gcalendar.get(Calendar.HOUR) + ":");
    System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
    System.out.println(gcalendar.get(Calendar.SECOND));

    // Test if the current year is a leap year
    if(gcalendar.isLeapYear(year)) {
      System.out.println("The current year is a leap year");
    }
    else {
      System.out.println("The current year is not a leap year");
    }
  }
}
```

Sample output is shown here:

```
Date: Jan 1 2014
Time: 1:45:5
The current year is not a leap year
```

## TimeZone

Another time-related class is **TimeZone**. The abstract **TimeZone** class allows you to work with time zone offsets from Greenwich mean time (GMT), also referred to as Coordinated Universal Time (UTC). It also computes daylight saving time. **TimeZone** only supplies the default constructor.

A sampling of methods defined by **TimeZone** is given in Table 19-6.



| Method | Description |
|---|---|
| Object clone( ) | Returns a **TimeZone**-specific version of **clone( )**. |
| static String[ ] getAvailableIDs( ) | Returns an array of **String** objects representing the names of all time zones. |
| static String[ ] getAvailableIDs(int *timeDelta*) | Returns an array of **String** objects representing the names of all time zones that are *timeDelta* offset from GMT. |
| static TimeZone getDefault( ) | Returns a **TimeZone** object that represents the default time zone used on the host computer. |
| String getID( ) | Returns the name of the invoking **TimeZone** object. |
| abstract int getOffset(int *era*, int *year*, int *month*, int *dayOfMonth*, int *dayOfWeek*, int *millisec*) | Returns the offset that should be added to GMT to compute local time. This value is adjusted for daylight saving time. The parameters to the method represent date and time components. |
| abstract int getRawOffset( ) | Returns the raw offset (in milliseconds) that should be added to GMT to compute local time. This value is not adjusted for daylight saving time. |
| static TimeZone getTimeZone(String *tzName*) | Returns the **TimeZone** object for the time zone named *tzName*. |
| abstract boolean inDaylightTime(Date *d*) | Returns **true** if the date represented by *d* is in daylight saving time in the invoking object. Otherwise, it returns **false**. |
| static void setDefault(TimeZone *tz*) | Sets the default time zone to be used on this host. *tz* is a reference to the **TimeZone** object to be used. |
| void setID(String *tzName*) | Sets the name of the time zone (that is, its ID) to that specified by *tzName*. |
| abstract void setRawOffset(int *millis*) | Sets the offset in milliseconds from GMT. |
| ZoneId toZoneId( ) | Converts the invoking object into a **ZoneId** and returns the result. **ZoneId** is packaged in **java.time.** (Added by JDK 8.) |
| abstract boolean useDaylightTime( ) | Returns **true** if the invoking object uses daylight saving time. Otherwise, it returns **false**. |

**Table 19-6** A Sampling of the Methods Defined by **TimeZone**

## SimpleTimeZone

The **SimpleTimeZone** class is a convenient subclass of **TimeZone**. It implements **TimeZone**'s abstract methods and allows you to work with time zones for a Gregorian calendar. It also computes daylight saving time.

**SimpleTimeZone** defines four constructors. One is

SimpleTimeZone(int *timeDelta*, String *tzName*)

This constructor creates a **SimpleTimeZone** object. The offset relative to Greenwich mean time (GMT) is *timeDelta*. The time zone is named *tzName*.

The second **SimpleTimeZone** constructor is

SimpleTimeZone(int *timeDelta*, String *tzId*, int *dstMonth0*,
int *dstDayInMonth0*, int *dstDay0*, int *time0*,
int *dstMonth1*, int *dstDayInMonth1*, int *dstDay1*,
int *time1*)

Here, the offset relative to GMT is specified in *timeDelta*. The time zone name is passed in *tzId*. The start of daylight saving time is indicated by the parameters *dstMonth0*, *dstDayInMonth0*, *dstDay0*, and *time0*. The end of daylight saving time is indicated by the parameters *dstMonth1*, *dstDayInMonth1*, *dstDay1*, and *time1*.

The third **SimpleTimeZone** constructor is

SimpleTimeZone(int *timeDelta*, String *tzId*, int *dstMonth0*,
int *dstDayInMonth0*, int *dstDay0*, int *time0*,
int *dstMonth1*, int *dstDayInMonth1*,
int *dstDay1*, int *time1*, int *dstDelta*)

Here, *dstDelta* is the number of milliseconds saved during daylight saving time.

The fourth **SimpleTimeZone** constructor is:

SimpleTimeZone(int *timeDelta*, String *tzId*, int *dstMonth0*,
int *dstDayInMonth0*, int *dstDay0*, int *time0*,
int *time0mode*, int *dstMonth1*, int *dstDayInMonth1*,
int *dstDay1*, int *time1*, int *time1mode*, int *dstDelta*)

Here, *time0mode* specifies the mode of the starting time, and *time1mode* specifies the mode of the ending time. Valid mode values include:

| | | |
|---|---|---|
| STANDARD_TIME | WALL_TIME | UTC_TIME |

The time mode indicates how the time values are interpreted. The default mode used by the other constructors is **WALL_TIME**.

## Locale

The **Locale** class is instantiated to produce objects that describe a geographical or cultural region. It is one of several classes that provide you with the ability to write programs that can execute in different international environments. For example, the formats used to display dates, times, and numbers are different in various regions.

Internationalization is a large topic that is beyond the scope of this book. However, many programs will only need to deal with its basics, which include setting the current locale.

The **Locale** class defines the following constants that are useful for dealing with several common locales:

| | | |
|---|---|---|
| CANADA | GERMAN | KOREAN |
| CANADA_FRENCH | GERMANY | PRC |
| CHINA | ITALIAN | SIMPLIFIED_CHINESE |
| CHINESE | ITALY | TAIWAN |
| ENGLISH | JAPAN | TRADITIONAL_CHINESE |
| FRANCE | JAPANESE | UK |
| FRENCH | KOREA | US |



For example, the expression **Locale.CANADA** represents the **Locale** object for Canada.

The constructors for **Locale** are

Locale(String *language*)
Locale(String *language*, String *country*)
Locale(String *language*, String *country*, String *variant*)

These constructors build a **Locale** object to represent a specific *language* and in the case of the last two, *country*. These values must contain standard language and country codes. Auxiliary variant information can be provided in *variant*.

**Locale** defines several methods. One of the most important is **setDefault( )**, shown here:

static void setDefault(Locale *localeObj*)

This sets the default locale used by the JVM to that specified by *localeObj*.

Some other interesting methods are the following:

final String getDisplayCountry( )
final String getDisplayLanguage( )
final String getDisplayName( )

These return human-readable strings that can be used to display the name of the country, the name of the language, and the complete description of the locale.

The default locale can be obtained using **getDefault( )**, shown here:

static Locale getDefault( )

JDK 7 added significant upgrades to the **Locale** class that support Internet Engineering Task Force (IETF) BCP 47, which defines tags for identifying languages, and Unicode Technical Standard (UTS) 35, which defines the Locale Data Markup Language (LDML). Support for BCP 47 and UTS 35 caused several features to be added to **Locale**, including several new methods and the **Locale.Builder** class. Among others, new methods include **getScript( )**, which obtains the locale's script, and **toLanguageTag( )**, which obtains a string that contains the locale's language tag. The **Locale.Builder** class constructs **Locale** instances. It ensures that a locale specification is well-formed as defined by BCP 47. (The **Locale** constructors do not provide such a check.) Several new methods have also been added to **Locale** by JDK 8. Among these are methods that support filtering, extensions, and lookups.

**Calendar** and **GregorianCalendar** are examples of classes that operate in a locale-sensitive manner. **DateFormat** and **SimpleDateFormat** also depend on the locale.

## Random

The **Random** class is a generator of pseudorandom numbers. These are called *pseudorandom* numbers because they are simply uniformly distributed sequences. **Random** defines the following constructors:

Random( )
Random(long *seed*)

The first version creates a number generator that uses a reasonably unique seed. The second form allows you to specify a seed value manually.

If you initialize a **Random** object with a seed, you define the starting point for the random sequence. If you use the same seed to initialize another **Random** object, you will extract the same random sequence. If you want to generate different sequences, specify different seed values. One way to do this is to use the current time to seed a **Random** object. This approach reduces the possibility of getting repeated sequences.

The core public methods defined by **Random** are shown in Table 19-7. These are the methods that have been available in **Random** for several years (many since Java 1.0) and are widely used.

As you can see, there are seven types of random numbers that you can extract from a **Random** object. Random Boolean values are available from **nextBoolean( )**. Random bytes can be obtained by calling **nextBytes( )**. Integers can be extracted via the **nextInt( )** method. Long integers, uniformly distributed over their range, can be obtained with **nextLong( )**. The **nextFloat( )** and **nextDouble( )** methods return a uniformly distributed **float** and **double**, respectively, between 0.0 and 1.0. Finally, **nextGaussian( )** returns a **double** value centered at 0.0 with a standard deviation of 1.0. This is what is known as a *bell curve*.

Here is an example that demonstrates the sequence produced by **nextGaussian( )**. It obtains 100 random Gaussian values and averages these values. The program also counts the

| Method | Description |
|---|---|
| boolean nextBoolean( ) | Returns the next **boolean** random number. |
| void nextBytes(byte *vals*[ ]) | Fills *vals* with randomly generated values. |
| double nextDouble( ) | Returns the next **double** random number. |
| float nextFloat( ) | Returns the next **float** random number. |
| double nextGaussian( ) | Returns the next Gaussian random number. |
| int nextInt( ) | Returns the next **int** random number. |
| int nextInt(int *n*) | Returns the next **int** random number within the range zero to *n*. |
| long nextLong( ) | Returns the next **long** random number. |
| void setSeed(long *newSeed*) | Sets the seed value (that is, the starting point for the random number generator) to that specified by *newSeed*. |

**Table 19-7** The Core Methods Defined by **Random**

number of values that fall within two standard deviations, plus or minus, using increments of 0.5 for each category. The result is graphically displayed sideways on the screen.

```
// Demonstrate random Gaussian values.
import java.util.Random;
class RandDemo {
  public static void main(String args[]) {
    Random r = new Random();
    double val;
    double sum = 0;
    int bell[] = new int[10];

    for(int i=0; i<100; i++) {
      val = r.nextGaussian();
      sum += val;
      double t = -2;

      for(int x=0; x<10; x++, t += 0.5)
        if(val < t) {
          bell[x]++;
          break;
        }
    }
    System.out.println("Average of values: " +
                       (sum/100));

    // display bell curve, sideways
    for(int i=0; i<10; i++) {
      for(int x=bell[i]; x>0; x--)
        System.out.print("*");
      System.out.println();
    }
  }
}
```

Here is a sample program run. As you can see, a bell-like distribution of numbers is obtained.

```
Average of values: 0.0702235271133344
**
*******
******
***************
******************
*****************
*************
**********
********
***
```

JDK 8 adds three new methods to **Random** that support the new stream API (see Chapter 29). They are called **doubles( )**, **ints( )**, and **longs( )**, and each returns a reference

to a stream that contains a sequence of pseudorandom values of the specified type. Each method defines several overloads. Here are their simplest forms:

```
DoubleStream doubles( )
IntStream ints( )
LongStream longs( )
```

The **doubles( )** method returns a stream that contains pseudorandom **double** values. (The range of these values will be less than 1.0 but greater than 0.0.) The **ints( )** method returns a stream that contains pseudorandom **int** values. The **longs( )** method returns a stream that contains pseudorandom **long** values. For these three methods, the stream returned is effectively infinite. Several overloads of each method are provided that let you specify the size of the stream, an origin, and an upper bound.

## Observable

The **Observable** class is used to create subclasses that other parts of your program can observe. When an object of such a subclass undergoes a change, observing classes are notified. Observing classes must implement the **Observer** interface, which defines the **update( )** method. The **update( )** method is called when an observer is notified of a change in an observed object.

**Observable** defines the methods shown in Table 19-8. An object that is being observed must follow two simple rules. First, if it has changed, it must call **setChanged( )**. Second, when it is ready to notify observers of this change, it must call **notifyObservers( )**. This causes the **update( )** method in the observing object(s) to be called. Be careful—if the

| Method | Description |
|---|---|
| void addObserver(Observer *obj*) | Adds *obj* to the list of objects observing the invoking object. |
| protected void clearChanged( ) | Calling this method returns the status of the invoking object to "unchanged." |
| int countObservers( ) | Returns the number of objects observing the invoking object. |
| void deleteObserver(Observer *obj*) | Removes *obj* from the list of objects observing the invoking object. |
| void deleteObservers( ) | Removes all observers for the invoking object. |
| boolean hasChanged( ) | Returns **true** if the invoking object has been modified and **false** if it has not. |
| void notifyObservers( ) | Notifies all observers of the invoking object that it has changed by calling **update( )**. A **null** is passed as the second argument to **update( )**. |
| void notifyObservers(Object *obj*) | Notifies all observers of the invoking object that it has changed by calling **update( )**. *obj* is passed as an argument to **update( )**. |
| protected void setChanged( ) | Called when the invoking object has changed. |

**Table 19-8** The Methods Defined by **Observable**

object calls **notifyObservers( )** without having previously called **setChanged( )**, no action will take place. The observed object must call both **setChanged( )** and **notifyObservers( )** before **update( )** will be called.

Notice that **notifyObservers( )** has two forms: one that takes an argument and one that does not. If you call **notifyObservers( )** with an argument, this object is passed to the observer's **update( )** method as its second parameter. Otherwise, **null** is passed to **update( )**. You can use the second parameter for passing any type of object that is appropriate for your application.



## The Observer Interface

To observe an observable object, you must implement the **Observer** interface. This interface defines only the one method shown here:

void update(Observable *observOb*, Object *arg*)

Here, *observOb* is the object being observed, and *arg* is the value passed by **notifyObservers( )**. The **update( )** method is called when a change in the observed object takes place.

## An Observer Example

Here is an example that demonstrates an observable object. It creates an observer class, called **Watcher**, that implements the **Observer** interface. The class being monitored is called **BeingWatched**. It extends **Observable**. Inside **BeingWatched** is the method **counter( )**, which simply counts down from a specified value. It uses **sleep( )** to wait a tenth of a second between counts. Each time the count changes, **notifyObservers( )** is called with the current count passed as its argument. This causes the **update( )** method inside **Watcher** to be called, which displays the current count. Inside **main( )**, a **Watcher** and a **BeingWatched** object, called **observing** and **observed**, respectively, are created. Then, **observing** is added to the list of observers for **observed**. This means that **observing.update( )** will be called each time **counter( )** calls **notifyObservers( )**.

```
/* Demonstrate the Observable class and the
   Observer interface.
*/
import java.util.*;

// This is the observing class.
class Watcher implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update() called, count is " +
                       ((Integer)arg).intValue());
  }
}

// This is the class being observed.
class BeingWatched extends Observable {
  void counter(int period) {
    for( ; period >=0; period--) {
      setChanged();
      notifyObservers(new Integer(period));
```

```
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println("Sleep interrupted");
      }
    }
  }

}

class ObserverDemo {
  public static void main(String args[]) {
    BeingWatched observed = new BeingWatched();
    Watcher observing = new Watcher();

    /* Add the observing to the list of observers for
       observed object.  */
    observed.addObserver(observing);

    observed.counter(10);
  }
}
```

The output from this program is shown here:

```
update() called, count is 10
update() called, count is 9
update() called, count is 8
update() called, count is 7
update() called, count is 6
update() called, count is 5
update() called, count is 4
update() called, count is 3
update() called, count is 2
update() called, count is 1
update() called, count is 0
```

More than one object can be an observer. For example, the following program implements two observing classes and adds an object of each class to the **BeingWatched** observer list. The second observer waits until the count reaches zero and then rings the bell.

```
/* An object may be observed by two or more
   observers.
*/

import java.util.*;


// This is the first observing class.
class Watcher1 implements Observer {
  public void update(Observable obj, Object arg) {
    System.out.println("update() called, count is " +
                       ((Integer)arg).intValue());
```

```
  }
}


// This is the second observing class.
class Watcher2 implements Observer {
  public void update(Observable obj, Object arg) {
    // Ring bell when done
    if(((Integer)arg).intValue() == 0)
      System.out.println("Done" + '\7');
  }
}


// This is the class being observed.
class BeingWatched extends Observable {
  void counter(int period) {
    for( ; period >=0; period--) {
      setChanged();
      notifyObservers(new Integer(period));
      try {
        Thread.sleep(100);
      } catch(InterruptedException e) {
        System.out.println("Sleep interrupted");
      }
    }
  }
}


class TwoObservers {
  public static void main(String args[]) {
    BeingWatched observed = new BeingWatched();
    Watcher1 observing1 = new Watcher1();
    Watcher2 observing2 = new Watcher2();


    // add both observers
    observed.addObserver(observing1);
    observed.addObserver(observing2);


    observed.counter(10);
  }
}
```

The **Observable** class and the **Observer** interface allow you to implement sophisticated program architectures based on the document/view methodology.

## Timer and TimerTask

An interesting and useful feature offered by **java.util** is the ability to schedule a task for execution at some future time. The classes that support this are **Timer** and **TimerTask**. Using these classes, you can create a thread that runs in the background, waiting for a specific time. When the time arrives, the task linked to that thread is executed. Various options allow you to schedule a task for repeated execution, and to schedule a task to run on a specific date. Although it was always possible to manually create a task that would be executed at a specific time using the **Thread** class, **Timer** and **TimerTask** greatly simplify this process.

**Timer** and **TimerTask** work together. **Timer** is the class that you will use to schedule a task for execution. The task being scheduled must be an instance of **TimerTask**. Thus, to schedule a task, you will first create a **TimerTask** object and then schedule it for execution using an instance of **Timer**.

**TimerTask** implements the **Runnable** interface; thus, it can be used to create a thread of execution. Its constructor is shown here:

```
protected TimerTask( )
```

**TimerTask** defines the methods shown in Table 19-9. Notice that **run( )** is abstract, which means that it must be overridden. The **run( )** method, defined by the **Runnable** interface, contains the code that will be executed. Thus, the easiest way to create a timer task is to extend **TimerTask** and override **run( )**.

Once a task has been created, it is scheduled for execution by an object of type **Timer**. The constructors for **Timer** are shown here:

```
Timer( )
Timer(boolean DThread)
Timer(String tName)
Timer(String tName, boolean DThread)
```

The first version creates a **Timer** object that runs as a normal thread. The second uses a daemon thread if *DThread* is **true**. A daemon thread will execute only as long as the rest of the program continues to execute. The third and fourth constructors allow you to specify a name for the **Timer** thread. The methods defined by **Timer** are shown in Table 19-9.

Once a **Timer** has been created, you will schedule a task by calling **schedule( )** on the **Timer** that you created. As Table 19-10 shows, there are several forms of **schedule( )** which allow you to schedule tasks in a variety of ways.

| Method | Description |
|---|---|
| boolean cancel( ) | Terminates the task. Returns **true** if an execution of the task is prevented. Otherwise, returns **false**. |
| abstract void run( ) | Contains the code for the timer task. |
| long scheduledExecutionTime( ) | Returns the time at which the last execution of the task was scheduled to have occurred. |

**Table 19-9** The Methods Defined by **TimerTask**

| Method | Description |
|---|---|
| void cancel( ) | Cancels the timer thread. |
| int purge( ) | Deletes canceled tasks from the timer's queue. |
| void schedule(TimerTask *TTask*, long *wait*) | *TTask* is scheduled for execution after the period passed in *wait* has elapsed. The *wait* parameter is specified in milliseconds. |
| void schedule(TimerTask *TTask*, long *wait*, long *repeat*) | *TTask* is scheduled for execution after the period passed in *wait* has elapsed. The task is then executed repeatedly at the interval specified by *repeat*. Both *wait* and *repeat* are specified in milliseconds. |
| void schedule(TimerTask *TTask*, Date *targetTime*) | *TTask* is scheduled for execution at the time specified by *targetTime*. |
| void schedule(TimerTask *TTask*, Date *targetTime*, long *repeat*) | *TTask* is scheduled for execution at the time specified by *targetTime*. The task is then executed repeatedly at the interval passed in *repeat*. The *repeat* parameter is specified in milliseconds. |
| void scheduleAtFixedRate( TimerTask *TTask*, long *wait*, long *repeat*) | *TTask* is scheduled for execution after the period passed in *wait* has elapsed. The task is then executed repeatedly at the interval specified by *repeat*. Both *wait* and *repeat* are specified in milliseconds. The time of each repetition is relative to the first execution, not the preceding execution. Thus, the overall rate of execution is fixed. |
| void scheduleAtFixedRate( TimerTask *TTask*, Date *targetTime*, long *repeat*) | *TTask* is scheduled for execution at the time specified by *targetTime*. The task is then executed repeatedly at the interval passed in *repeat*. The *repeat* parameter is specified in milliseconds. The time of each repetition is relative to the first execution, not the preceding execution. Thus, the overall rate of execution is fixed. |

**Table 19-10** The Methods Defined by **Timer**

If you create a non-daemon task, then you will want to call **cancel( )** to end the task when your program ends. If you don't do this, then your program may "hang" for a period of time.

The following program demonstrates **Timer** and **TimerTask**. It defines a timer task whose **run( )** method displays the message "Timer task executed." This task is scheduled to run once every half second after an initial delay of one second.

```
// Demonstrate Timer and TimerTask.

import java.util.*;

class MyTimerTask extends TimerTask {
  public void run() {
    System.out.println("Timer task executed.");
  }
}

class TTest {
```

```
  public static void main(String args[]) {
    MyTimerTask myTask = new MyTimerTask();
    Timer myTimer = new Timer();

    /* Set an initial delay of 1 second,
       then repeat every half second.
    */
    myTimer.schedule(myTask, 1000, 500);

    try {
      Thread.sleep(5000);
    } catch (InterruptedException exc) {}

    myTimer.cancel();
  }
}
```

## Currency

The **Currency** class encapsulates information about a currency. It defines no constructors. The methods supported by **Currency** are shown in Table 19-11. The following program demonstrates **Currency**:

```
// Demonstrate Currency.
import java.util.*;
```

| Method | Description |
|---|---|
| static Set<Currency> getAvailableCurrencies( ) | Returns a set of the supported currencies. |
| String getCurrencyCode( ) | Returns the code (as defined by ISO 4217) that describes the invoking currency. |
| int getDefaultFractionDigits( ) | Returns the number of digits after the decimal point that are normally used by the invoking currency. For example, there are two fractional digits normally used for dollars. |
| String getDisplayName( ) | Returns the name of the invoking currency for the default locale. |
| String getDisplayName(Locale *loc*) | Returns the name of the invoking currency for the specified locale. |
| static Currency getInstance(Locale *localeObj*) | Returns a **Currency** object for the locale specified by *localeObj*. |
| static Currency getInstance(String *code*) | Returns a **Currency** object associated with the currency code passed in *code*. |
| int getNumericCode( ) | Returns the numeric code (as defined by ISO 4217) for the invoking currency. |
| String getSymbol( ) | Returns the currency symbol (such as $) for the invoking object. |
| String getSymbol(Locale *localeObj*) | Returns the currency symbol (such as $) for the locale passed in *localeObj*. |
| String toString( ) | Returns the currency code for the invoking object. |

**Table 19-11** The Methods Defined by **Currency**

```
class CurDemo {
  public static void main(String args[]) {
    Currency c;

    c = Currency.getInstance(Locale.US);

    System.out.println("Symbol: " + c.getSymbol());
    System.out.println("Default fractional digits: " +
                       c.getDefaultFractionDigits());
  }
}
```

The output is shown here:

```
Symbol: $
Default fractional digits: 2
```



# Formatter

At the core of Java's support for creating formatted output is the **Formatter** class. It provides *format conversions* that let you display numbers, strings, and time and date in virtually any format you like. It operates in a manner similar to the C/C++ **printf( )** function, which means that if you are familiar with C/C++, then learning to use **Formatter** will be very easy. It also further streamlines the conversion of C/C++ code to Java. If you are not familiar with C/C++, it is still quite easy to format data.

**NOTE** Although Java's **Formatter** class operates in a manner very similar to the C/C++ **printf( )** function, there are some differences, and some new features. Therefore, if you have a C/C++ background, a careful reading is advised.

## The Formatter Constructors

Before you can use **Formatter** to format output, you must create a **Formatter** object. In general, **Formatter** works by converting the binary form of data used by a program into formatted text. It stores the formatted text in a buffer, the contents of which can be obtained by your program whenever they are needed. It is possible to let **Formatter** supply this buffer automatically, or you can specify the buffer explicitly when a **Formatter** object is created. It is also possible to have **Formatter** output its buffer to a file.

The **Formatter** class defines many constructors, which enable you to construct a **Formatter** in a variety of ways. Here is a sampling:

Formatter( )

Formatter(Appendable *buf*)

Formatter(Appendable *buf*, Locale *loc*)

Formatter(String *filename*)
  throws FileNotFoundException

Formatter(String *filename*, String *charset*)
  throws FileNotFoundException, UnsupportedEncodingException

Formatter(File *outF*)
throws FileNotFoundException

Formatter(OutputStream *outStrm*)

Here, *buf* specifies a buffer for the formatted output. If *buf* is null, then **Formatter** automatically allocates a **StringBuilder** to hold the formatted output. The *loc* parameter specifies a locale. If no locale is specified, the default locale is used. The *filename* parameter specifies the name of a file that will receive the formatted output. The *charset* parameter specifies the character set. If no character set is specified, then the default character set is used. The *outF* parameter specifies a reference to an open file that will receive output. The *outStrm* parameter specifies a reference to an output stream that will receive output. When using a file, output is also written to the file.

Perhaps the most widely used constructor is the first, which has no parameters. It automatically uses the default locale and allocates a **StringBuilder** to hold the formatted output.

## The Formatter Methods

**Formatter** defines the methods shown in Table 19-12.

| Method | Description |
|---|---|
| void close( ) | Closes the invoking **Formatter**. This causes any resources used by the object to be released. After a **Formatter** has been closed, it cannot be reused. An attempt to use a closed **Formatter** results in a **FormatterClosedException**. |
| void flush( ) | Flushes the format buffer. This causes any output currently in the buffer to be written to the destination. This applies mostly to a **Formatter** tied to a file. |
| Formatter format(String *fmtString*, Object ... *args*) | Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. Returns the invoking object. |
| Formatter format(Locale *loc*, String *fmtString*, Object ... *args*) | Formats the arguments passed via *args* according to the format specifiers contained in *fmtString*. The locale specified by *loc* is used for this format. Returns the invoking object. |
| IOException ioException( ) | If the underlying object that is the destination for output throws an **IOException**, then this exception is returned. Otherwise, null is returned. |
| Locale locale( ) | Returns the invoking object's locale. |
| Appendable out( ) | Returns a reference to the underlying object that is the destination for output. |
| String toString( ) | Returns a **String** containing the formatted output. |

**Table 19-12** The Methods Defined by **Formatter**

## Formatting Basics

After you have created a **Formatter**, you can use it to create a formatted string. To do so, use the **format( )** method. The most commonly used version is shown here:

Formatter format(String *fmtString*, Object ... *arg*s)

The *fmtSring* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains *format specifiers* that define the way the subsequent arguments are displayed.

In its simplest form, a format specifier begins with a percent sign followed by the format *conversion specifier*. All format conversion specifiers consist of a single character. For example, the format specifier for floating-point data is **%f**. In general, there must be the same number of arguments as there are format specifiers, and the format specifiers and the arguments are matched in order from left to right. For example, consider this fragment:

```
Formatter fmt = new Formatter();
fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);
```

This sequence creates a **Formatter** that contains the following string:

```
Formatting with Java is easy 10 98.600000
```

In this example, the format specifiers, **%s**, **%d**, and **%f**, are replaced with the arguments that follow the format string. Thus, **%s** is replaced by "with Java", **%d** is replaced by 10, and **%f** is replaced by 98.6. All other characters are simply used as-is. As you might guess, the format specifier **%s** specifies a string, and **%d** specifies an integer value. As mentioned earlier, the **%f** specifies a floating-point value.

The **format( )** method accepts a wide variety of format specifiers, which are shown in Table 19-13. Notice that many specifiers have both upper- and lowercase forms. When an uppercase specifier is used, then letters are shown in uppercase. Otherwise, the upper- and

| Format Specifier | Conversion Applied |
|---|---|
| %a<br>%A | Floating-point hexadecimal |
| %b<br>%B | Boolean |
| %c | Character |
| %d | Decimal integer |
| %h<br>%H | Hash code of the argument |
| %e<br>%E | Scientific notation |
| %f | Decimal floating-point |

**Table 19-13** The Format Specifiers

| Format Specifier | Conversion Applied |
|---|---|
| %g<br>%G | Uses **%e** or **%f**, based on the value being formatted and the precision |
| %o | Octal integer |
| %n | Inserts a newline character |
| %s<br>%S | String |
| %t<br>%T | Time and date |
| %x<br>%X | Integer hexadecimal |
| %% | Inserts a % sign |

**Table 19-13** The Format Specifiers *(continued)*

lowercase specifiers perform the same conversion. It is important to understand that Java type-checks each format specifier against its corresponding argument. If the argument doesn't match, an **IllegalFormatException** is thrown.

Once you have formatted a string, you can obtain it by calling **toString( )**. For example, continuing with the preceding example, the following statement obtains the formatted string contained in **fmt**:

```
String str = fmt.toString();
```

Of course, if you simply want to display the formatted string, there is no reason to first assign it to a **String** object. When a **Formatter** object is passed to **println( )**, for example, its **toString( )** method is automatically called.

Here is a short program that puts together all of the pieces, showing how to create and display a formatted string:

```
// A very simple example that uses Formatter.
import java.util.*;

class FormatDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Formatting %s is easy %d %f", "with Java", 10, 98.6);

    System.out.println(fmt);
    fmt.close();
  }
}
```

One other point: You can obtain a reference to the underlying output buffer by calling **out( )**. It returns a reference to an **Appendable** object.

Now that you know the general mechanism used to create a formatted string, the remainder of this section discusses in detail each conversion. It also describes various options, such as justification, minimum field width, and precision.

## Formatting Strings and Characters

To format an individual character, use **%c**. This causes the matching character argument to be output, unmodified. To format a string, use **%s**.

## Formatting Numbers

To format an integer in decimal format, use **%d**. To format a floating-point value in decimal format, use **%f**. To format a floating-point value in scientific notation, use **%e**. Numbers represented in scientific notation take this general form:

*x.ddddd*e+/–yy

The **%g** format specifier causes **Formatter** to use either **%f** or **%e**, based on the value being formatted and the precision, which is 6 by default. The following program demonstrates the effect of the **%f** and **%e** format specifiers:

```
// Demonstrate the %f and %e format specifiers.
import java.util.*;

class FormatDemo2 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    for(double i=1.23; i < 1.0e+6; i *= 100) {
      fmt.format("%f %e", i, i);
      System.out.println(fmt);
    }
    fmt.close();

  }
}
```

It produces the following output:

```
1.230000 1.230000e+00
1.230000 1.230000e+00 123.000000 1.230000e+02
1.230000 1.230000e+00 123.000000 1.230000e+02 12300.000000 1.230000e+04
```

You can display integers in octal or hexadecimal format by using **%o** and **%x**, respectively. For example, this fragment:

```
fmt.format("Hex: %x, Octal: %o", 196, 196);
```

produces this output:

```
Hex: c4, Octal: 304
```

You can display floating-point values in hexadecimal format by using **%a**. The format produced by **%a** appears a bit strange at first glance. This is because its representation uses a form similar to scientific notation that consists of a hexadecimal significand and a decimal exponent of powers of 2. Here is the general format:

0x1.*sig*p*exp*

Here, *sig* contains the fractional portion of the significand and *exp* contains the exponent. The **p** indicates the start of the exponent. For example, this call:

```
fmt.format("%a", 512.0);
```

produces this output:

```
  0x1.0p9
```

## Formatting Time and Date

One of the more powerful conversion specifiers is **%t**. It lets you format time and date information. The **%t** specifier works a bit differently than the others because it requires the use of a suffix to describe the portion and precise format of the time or date desired. The suffixes are shown in Table 19-14. For example, to display minutes, you would use **%tM**, where **M** indicates minutes in a two-character field. The argument corresponding to the **%t** specifier must be of type **Calendar**, **Date**, **Long**, or **long**.

Here is a program that demonstrates several of the formats:

```
// Formatting time and date.
import java.util.*;

class TimeDateFormat {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    // Display standard 12-hour time format.
    fmt.format("%tr", cal);
    System.out.println(fmt);
    fmt.close();

    // Display complete time and date information.
    fmt = new Formatter();
    fmt.format("%tc", cal);
    System.out.println(fmt);
    fmt.close();

    // Display just hour and minute.
    fmt = new Formatter();
    fmt.format("%tl:%tM", cal, cal);
    System.out.println(fmt);
    fmt.close();

    // Display month by name and number.
    fmt = new Formatter();
    fmt.format("%tB %tb %tm", cal, cal, cal);
    System.out.println(fmt);
    fmt.close();
  }
}
```

| Suffix | Replaced By |
|---|---|
| a | Abbreviated weekday name |
| A | Full weekday name |
| b | Abbreviated month name |
| B | Full month name |
| c | Standard date and time string formatted as *day month date hh::mm:ss tzone year* |
| C | First two digits of year |
| d | Day of month as a decimal (01—31) |
| D | month/day/year |
| e | Day of month as a decimal (1—31) |
| F | year-month-day |
| h | Abbreviated month name |
| H | Hour (00 to 23) |
| I | Hour (01 to 12) |
| j | Day of year as a decimal (001 to 366) |
| k | Hour (0 to 23) |
| l | Hour (1 to 12) |
| L | Millisecond (000 to 999) |
| m | Month as decimal (01 to 13) |
| M | Minute as decimal (00 to 59) |
| N | Nanosecond (000000000 to 999999999) |
| p | Locale's equivalent of AM or PM in lowercase |
| Q | Milliseconds from 1/1/1970 |
| r | *hh:mm:ss* (12-hour format) |
| R | *hh:mm* (24-hour format) |
| S | Seconds (00 to 60) |
| s | Seconds from 1/1/1970 UTC |
| T | *hh:mm:ss* (24-hour format) |
| y | Year in decimal without century (00 to 99) |
| Y | Year in decimal including century (0001 to 9999) |
| z | Offset from UTC |
| Z | Time zone name |

**Table 19-14** The Time and Date Format Suffixes

Sample output is shown here:

```
03:15:34 PM
Wed Jan 01 15:15:34 CST 2014
3:15
January Jan 01
```

## The %n and %% Specifiers

The **%n** and**%%** format specifiers differ from the others in that they do not match an argument. Instead, they are simply escape sequences that insert a character into the output sequence. The **%n** inserts a newline. The **%%** inserts a percent sign. Neither of these characters can be entered directly into the format string. Of course, you can also use the standard escape sequence **\n** to embed a newline character.

Here is an example that demonstrates the **%n** and **%%** format specifiers:

```
// Demonstrate the %n and %% format specifiers.
import java.util.*;

class FormatDemo3 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("Copying file%nTransfer is %d%% complete", 88);
    System.out.println(fmt);
    fmt.close();
  }
}
```

It displays the following output:

```
Copying file
Transfer is 88% complete
```

## Specifying a Minimum Field Width

An integer placed between the **%** sign and the format conversion code acts as a *minimum field-width specifier*. This pads the output with spaces to ensure that it reaches a certain minimum length. If the string or number is longer than that minimum, it will still be printed in full. The default padding is done with spaces. If you want to pad with 0's, place a 0 before the field-width specifier. For example, **%05d** will pad a number of less than five digits with 0's so that its total length is five. The field-width specifier can be used with all format specifiers except **%n**.

The following program demonstrates the minimum field-width specifier by applying it to the **%f** conversion:

```
// Demonstrate a field-width specifier.
import java.util.*;

class FormatDemo4 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
```

```
    fmt.format("|%f|%n|%12f|%n|%012f|",
               10.12345, 10.12345, 10.12345);

    System.out.println(fmt);
    fmt.close();

  }
}
```

This program produces the following output:

```
|10.123450|
|   10.123450|
|00010.123450|
```

The first line displays the number 10.12345 in its default width. The second line displays that value in a 12-character field. The third line displays the value in a 12-character field, padded with leading zeros.

The minimum field-width modifier is often used to produce tables in which the columns line up. For example, the next program produces a table of squares and cubes for the numbers between 1 and 10:

```
// Create a table of squares and cubes.
import java.util.*;

class FieldWidthDemo {
  public static void main(String args[]) {
    Formatter fmt;

    for(int i=1; i <= 10; i++) {
      fmt = new Formatter();
      fmt.format("%4d %4d %4d", i, i*i, i*i*i);
      System.out.println(fmt);
      fmt.close();
   }

  }
}
```

Its output is shown here:

```
   1    1    1
   2    4    8
   3    9   27
   4   16   64
   5   25  125
   6   36  216
   7   49  343
   8   64  512
   9   81  729
  10  100 1000
```

## Specifying Precision

A *precision specifier* can be applied to the **%f**, **%e**, **%g**, and **%s** format specifiers. It follows the minimum field-width specifier (if there is one) and consists of a period followed by an integer. Its exact meaning depends upon the type of data to which it is applied.

When you apply the precision specifier to floating-point data using the **%f** or **%e** specifiers, it determines the number of decimal places displayed. For example, **%10.4f** displays a number at least ten characters wide with four decimal places. When using **%g**, the precision determines the number of significant digits. The default precision is 6.

Applied to strings, the precision specifier specifies the maximum field length. For example, **%5.7s** displays a string of at least five and not exceeding seven characters long. If the string is longer than the maximum field width, the end characters will be truncated.

The following program illustrates the precision specifier:

```
// Demonstrate the precision modifier.
import java.util.*;

class PrecisionDemo {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Format 4 decimal places.
    fmt.format("%.4f", 123.1234567);
    System.out.println(fmt);
    fmt.close();

    // Format to 2 decimal places in a 16 character field
    fmt = new Formatter();
    fmt.format("%16.2e", 123.1234567);
    System.out.println(fmt);
    fmt.close();

    // Display at most 15 characters in a string.
    fmt = new Formatter();
    fmt.format("%.15s", "Formatting with Java is now easy.");
    System.out.println(fmt);
    fmt.close();
  }
}
```

It produces the following output:

```
123.1235
        1.23e+02
Formatting with
```

## Using the Format Flags

**Formatter** recognizes a set of format *flags* that lets you control various aspects of a conversion. All format flags are single characters, and a format flag follows the **%** in a format specification. The flags are shown here:

| Flag | Effect |
|---|---|
| – | Left justification |
| # | Alternate conversion format |
| 0 | Output is padded with zeros rather than spaces |
| *space* | Positive numeric output is preceded by a space |
| + | Positive numeric output is preceded by a + sign |
| , | Numeric values include grouping separators |
| ( | Negative numeric values are enclosed within parentheses |

Not all flags apply to all format specifiers. The following sections explain each in detail.

## Justifying Output

By default, all output is right-justified. That is, if the field width is larger than the data printed, the data will be placed on the right edge of the field. You can force output to be left-justified by placing a minus sign directly after the %. For instance, **%–10.2f** left-justifies a floating-point number with two decimal places in a 10-character field. For example, consider this program:

```
// Demonstrate left justification.
import java.util.*;

class LeftJustify {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    // Right justify by default
    fmt.format("|%10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();

    // Now, left justify.
    fmt = new Formatter();
    fmt.format("|%-10.2f|", 123.123);
    System.out.println(fmt);
    fmt.close();
  }
}
```

It produces the following output:

```
|    123.12|
|123.12    |
```

As you can see, the second line is left-justified within a 10-character field.

## The Space, +, 0, and ( Flags

To cause a + sign to be shown before positive numeric values, add the + flag. For example,

```
fmt.format("%+d", 100);
```

creates this string:

```
+100
```

When creating columns of numbers, it is sometimes useful to output a space before positive values so that positive and negative values line up. To do this, add the space flag. For example,

```
// Demonstrate the space format specifiers.
import java.util.*;

class FormatDemo5 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();

    fmt.format("% d", -100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", 100);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", -200);
    System.out.println(fmt);
    fmt.close();

    fmt = new Formatter();
    fmt.format("% d", 200);
    System.out.println(fmt);
    fmt.close();
  }
}
```

The output is shown here:

```
-100
 100
-200
 200
```

Notice that the positive values have a leading space, which causes the digits in the column to line up properly.

To show negative numeric output inside parentheses, rather than with a leading –, use the **(** flag. For example,

```
fmt.format("%(d", -100);
```

creates this string:

```
(100)
```

The 0 flag causes output to be padded with zeros rather than spaces.



## The Comma Flag

When displaying large numbers, it is often useful to add grouping separators, which in English are commas. For example, the value 1234567 is more easily read when formatted as 1,234,567. To add grouping specifiers, use the comma **(,)** flag. For example,

```
fmt.format("%,.2f", 4356783497.34);
```

creates this string:

```
4,356,783,497.34
```

## The # Flag

The **#** can be applied to **%o**, **%x**, **%e,** and **%f**. For **%e,** and **%f**, the **#** ensures that there will be a decimal point even if there are no decimal digits. If you precede the **%x** format specifier with a **#**, the hexadecimal number will be printed with a **0x** prefix. Preceding the **%o** specifier with **#** causes the number to be printed with a leading zero.

## The Uppercase Option

As mentioned earlier, several of the format specifiers have uppercase versions that cause the conversion to use uppercase where appropriate. The following table describes the effect.

| Specifier | Effect |
|---|---|
| %A | Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F*. Also, the prefix **0x** is displayed as **0X**, and the **p** will be displayed as **P**. |
| %B | Uppercases the values **true** and **false**. |
| %E | Causes the *e* symbol that indicates the exponent to be displayed in uppercase. |
| %G | Causes the *e* symbol that indicates the exponent to be displayed in uppercase. |
| %H | Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F*. |
| %S | Uppercases the corresponding string. |
| %T | Causes all alphabetical output to be displayed in uppercase. |
| %X | Causes the hexadecimal digits *a* through *f* to be displayed in uppercase as *A* through *F*. Also, the optional prefix **0x** is displayed as **0X**, if present. |

For example, this call:

```
fmt.format("%X", 250);
```

creates this string:

```
FA
```

This call:

```
fmt.format("%E", 123.1234);
```

creates this string:

```
1.231234E+02
```

## Using an Argument Index

**Formatter** includes a very useful feature that lets you specify the argument to which a format specifier applies. Normally, format specifiers and arguments are matched in order, from left to right. That is, the first format specifier matches the first argument, the second format specifier matches the second argument, and so on. However, by using an *argument index*, you can explicitly control which argument a format specifier matches.

An argument index immediately follows the % in a format specifier. It has the following format:

*n*$

where *n* is the index of the desired argument, beginning with 1. For example, consider this example:

```
fmt.format("%3$d %1$d %2$d", 10, 20, 30);
```

It produces this string:

```
30 10 20
```

In this example, the first format specifier matches 30, the second matches 10, and the third matches 20. Thus, the arguments are used in an order other than strictly left to right.

One advantage of argument indexes is that they enable you to reuse an argument without having to specify it twice. For example, consider this line:

```
fmt.format("%d in hex is %1$x", 255);
```

It produces the following string:

```
255 in hex is ff
```

As you can see, the argument 255 is used by both format specifiers.

There is a convenient shorthand called a *relative index* that enables you to reuse the argument matched by the preceding format specifier. Simply specify < for the argument index. For example, the following call to **format( )** produces the same results as the previous example:

```
fmt.format("%d in hex is %<x", 255);
```

Relative indexes are especially useful when creating custom time and date formats. Consider the following example:

```
// Use relative indexes to simplify the
// creation of a custom time and date format.
import java.util.*;

class FormatDemo6 {
  public static void main(String args[]) {
    Formatter fmt = new Formatter();
    Calendar cal = Calendar.getInstance();

    fmt.format("Today is day %te of %<tB, %<tY", cal);
    System.out.println(fmt);
    fmt.close();
  }
}
```

Here is sample output:

```
Today is day 1 of January, 2014
```

Because of relative indexing, the argument **cal** need only be passed once, rather than three times.

## Closing a Formatter

In general, you should close a **Formatter** when you are done using it. Doing so frees any resources that it was using. This is especially important when formatting to a file, but it can be important in other cases, too. As the previous examples have shown, one way to close a **Formatter** is to explicitly call **close( )**. However, beginning with JDK 7, **Formatter** implements the **AutoCloseable** interface. This means that it supports the **try**-with-resources statement. Using this approach, the **Formatter** is automatically closed when it is no longer needed.

The **try**-with-resources statement is described in Chapter 13, in connection with files, because files are some of the most commonly used resources that must be closed. However, the same basic techniques apply here. For example, here is the first **Formatter** example reworked to use automatic resource management:

```
// Use automatic resource management with Formatter.
import java.util.*;

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

    try (Formatter fmt = new Formatter())
    {
      fmt.format("Formatting %s is easy %d %f", "with Java",
                 10, 98.6);
      System.out.println(fmt);
```

```
    }
  }
}
```

The output is the same as before.

### The Java printf( ) Connection

Although there is nothing technically wrong with using **Formatter** directly (as the preceding examples have done) when creating output that will be displayed on the console, there is a more convenient alternative: the **printf( )** method. The **printf( )** method automatically uses **Formatter** to create a formatted string. It then displays that string on **System.out**, which is the console by default. The **printf( )** method is defined by both **PrintStream** and **PrintWriter**. The **printf( )** method is described in Chapter 20.

## Scanner

**Scanner** is the complement of **Formatter**. It reads formatted input and converts it into its binary form. **Scanner** can be used to read input from the console, a file, a string, or any source that implements the **Readable** interface or **ReadableByteChannel**. For example, you can use **Scanner** to read a number from the keyboard and assign its value to a variable. As you will see, given its power, **Scanner** is surprisingly easy to use.

### The Scanner Constructors

**Scanner** defines the constructors shown in Table 19-15. In general, a **Scanner** can be created for a **String**, an **InputStream**, a **File**, or any object that implements the **Readable** or **ReadableByteChannel** interfaces. Here are some examples.

The following sequence creates a **Scanner** that reads the file **Test.txt**:

```
FileReader fin = new FileReader("Test.txt");
Scanner src = new Scanner(fin);
```

This works because **FileReader** implements the **Readable** interface. Thus, the call to the constructor resolves to **Scanner(Readable)**.

This next line creates a **Scanner** that reads from standard input, which is the keyboard by default:

```
Scanner conin = new Scanner(System.in);
```

This works because **System.in** is an object of type **InputStream**. Thus, the call to the constructor maps to **Scanner(InputStream)**.

The next sequence creates a **Scanner** that reads from a string.

```
String instr = "10 99.88 scanning is easy.";
Scanner conin = new Scanner(instr);
```

### Scanning Basics

Once you have created a **Scanner**, it is a simple matter to use it to read formatted input. In general, a **Scanner** reads *tokens* from the underlying source that you specified when the **Scanner** was created. As it relates to **Scanner**, a token is a portion of input that is delineated

| Method | Description |
|---|---|
| Scanner(File *from*)<br>throws FileNotFoundException | Creates a **Scanner** that uses the file specified by *from* as a source for input. |
| Scanner(File *from*, String *charset*)<br>throws FileNotFoundException | Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input. |
| Scanner(InputStream *from*) | Creates a **Scanner** that uses the stream specified by *from* as a source for input. |
| Scanner(InputStream *from*, String *charset*) | Creates a **Scanner** that uses the stream specified by *from* with the encoding specified by *charset* as a source for input. |
| Scanner(Path *from*)<br>throws IOException | Creates a **Scanner** that uses the file specified by *from* as a source for input. |
| Scanner(Path *from*, String *charset*)<br>throws IOException | Creates a **Scanner** that uses the file specified by *from* with the encoding specified by *charset* as a source for input. |
| Scanner(Readable *from*) | Creates a **Scanner** that uses the **Readable** object specified by *from* as a source for input. |
| Scanner (ReadableByteChannel *from*) | Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* as a source for input. |
| Scanner(ReadableByteChannel *from*,<br>String *charset*) | Creates a **Scanner** that uses the **ReadableByteChannel** specified by *from* with the encoding specified by *charset* as a source for input. |
| Scanner(String *from*) | Creates a **Scanner** that uses the string specified by *from* as a source for input. |

**Table 19-15** The **Scanner** Constructors

by a set of delimiters, which is whitespace by default. A token is read by matching it with a particular *regular expression*, which defines the format of the data. Although **Scanner** allows you to define the specific type of expression that its next input operation will match, it includes many predefined patterns, which match the primitive types, such as **int** and **double**, and strings. Thus, often you won't need to specify a pattern to match.

In general, to use **Scanner**, follow this procedure:

1. Determine if a specific type of input is available by calling one of **Scanner**'s **hasNext***X* methods, where *X* is the type of data desired.
2. If input is available, read it by calling one of **Scanner**'s **next***X* methods.
3. Repeat the process until input is exhausted.
4. Close the **Scanner** by calling **close( )**.

As the preceding indicates, **Scanner** defines two sets of methods that enable you to read input. The first are the **hasNext***X* methods, which are shown in Table 19-16. These methods determine if the specified type of input is available. For example, calling **hasNextInt( )** returns **true** only if the next token to be read is an integer. If the desired data is available, then you read it by calling one of **Scanner**'s **next***X* methods, which are shown in Table 19-17.

| Method | Description |
|---|---|
| boolean hasNext( ) | Returns **true** if another token of any type is available to be read. Returns **false** otherwise. |
| boolean hasNext(Pattern *pattern*) | Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise. |
| boolean hasNext(String *pattern*) | Returns **true** if a token that matches the pattern passed in *pattern* is available to be read. Returns **false** otherwise. |
| boolean hasNextBigDecimal( ) | Returns **true** if a value that can be stored in a **BigDecimal** object is available to be read. Returns **false** otherwise. |
| boolean hasNextBigInteger( ) | Returns **true** if a value that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.) |
| boolean hasNextBigInteger(int *radix*) | Returns **true** if a value in the specified radix that can be stored in a **BigInteger** object is available to be read. Returns **false** otherwise. |
| boolean hasNextBoolean( ) | Returns **true** if a **boolean** value is available to be read. Returns **false** otherwise. |
| boolean hasNextByte( ) | Returns **true** if a **byte** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.) |
| boolean hasNextByte(int *radix*) | Returns **true** if a **byte** value in the specified radix is available to be read. Returns **false** otherwise. |
| boolean hasNextDouble( ) | Returns **true** if a **double** value is available to be read. Returns **false** otherwise. |
| boolean hasNextFloat( ) | Returns **true** if a **float** value is available to be read. Returns **false** otherwise. |
| boolean hasNextInt( ) | Returns **true** if an **int** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.) |
| boolean hasNextInt(int *radix*) | Returns **true** if an **int** value in the specified radix is available to be read. Returns **false** otherwise. |
| boolean hasNextLine( ) | Returns **true** if a line of input is available. |
| boolean hasNextLong( ) | Returns **true** if a **long** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.) |
| boolean hasNextLong(int *radix*) | Returns **true** if a **long** value in the specified radix is available to be read. Returns **false** otherwise. |
| boolean hasNextShort( ) | Returns **true** if a **short** value is available to be read. Returns **false** otherwise. The default radix is used. (Unless changed, the default radix is 10.) |
| boolean hasNextShort(int *radix*) | Returns **true** if a **short** value in the specified radix is available to be read. Returns **false** otherwise. |

**Table 19-16** The **Scanner hasNext** Methods

| Method | Description |
|---|---|
| String next( ) | Returns the next token of any type from the input source. |
| String next(Pattern *pattern*) | Returns the next token that matches the pattern passed in *pattern* from the input source. |
| String next(String *pattern*) | Returns the next token that matches the pattern passed in *pattern* from the input source. |
| BigDecimal nextBigDecimal( ) | Returns the next token as a **BigDecimal** object. |
| BigInteger nextBigInteger( ) | Returns the next token as a **BigInteger** object. The default radix is used. (Unless changed, the default radix is 10.) |
| BigInteger nextBigInteger(int *radix*) | Returns the next token (using the specified radix) as a **BigInteger** object. |
| boolean nextBoolean( ) | Returns the next token as a **boolean** value. |
| byte nextByte( ) | Returns the next token as a **byte** value. The default radix is used. (Unless changed, the default radix is 10.) |
| byte nextByte(int *radix*) | Returns the next token (using the specified radix) as a **byte** value. |
| double nextDouble( ) | Returns the next token as a **double** value. |
| float nextFloat( ) | Returns the next token as a **float** value. |
| int nextInt( ) | Returns the next token as an **int** value. The default radix is used. (Unless changed, the default radix is 10.) |
| int nextInt(int *radix*) | Returns the next token (using the specified radix) as an **int** value. |
| String nextLine( ) | Returns the next line of input as a string. |
| long nextLong( ) | Returns the next token as a **long** value. The default radix is used. (Unless changed, the default radix is 10.) |
| long nextLong(int *radix*) | Returns the next token (using the specified radix) as a **long** value. |
| short nextShort( ) | Returns the next token as a **short** value. The default radix is used. (Unless changed, the default radix is 10.) |
| short nextShort(int *radix*) | Returns the next token (using the specified radix) as a **short** value. |

**Table 19-17** The **Scanner next** Methods

For example, to read the next integer, call **nextInt( )**. The following sequence shows how to read a list of integers from the keyboard.

```
Scanner conin = new Scanner(System.in);
int i;

// Read a list of integers.
while(conin.hasNextInt()) {
```

```
  i = conin.nextInt();
  // ...
}
```

The **while** loop stops as soon as the next token is not an integer. Thus, the loop stops reading integers as soon as a non-integer is encountered in the input stream.

If a **next** method cannot find the type of data it is looking for, it throws an **InputMismatchException**. A **NoSuchElementException** is thrown if no more input is available. For this reason, it is best to first confirm that the desired type of data is available by calling a **hasNext** method before calling its corresponding **next** method.

## Some Scanner Examples

**Scanner** makes what could be a tedious task into an easy one. To understand why, let's look at some examples. The following program averages a list of numbers entered at the keyboard:

```
// Use Scanner to compute an average of the values.
import java.util.*;

class AvgNums {
  public static void main(String args[]) {
    Scanner conin = new Scanner(System.in);

    int count = 0;
    double sum = 0.0;

    System.out.println("Enter numbers to average.");

    // Read and sum numbers.
    while(conin.hasNext()) {
      if(conin.hasNextDouble()) {
        sum += conin.nextDouble();
        count++;
      }
      else {
        String str = conin.next();
        if(str.equals("done")) break;
        else {
          System.out.println("Data format error.");
          return;
        }
      }
    }

    conin.close();
    System.out.println("Average is " + sum / count);
  }
}
```

The program reads numbers from the keyboard, summing them in the process, until the user enters the string "done". It then stops input and displays the average of the numbers. Here is a sample run:

```
Enter numbers to average.
1.2
2
3.4
4
done
Average is 2.65
```

The program reads numbers until it encounters a token that does not represent a valid **double** value. When this occurs, it confirms that the token is the string "done". If it is, the program terminates normally. Otherwise, it displays an error.

Notice that the numbers are read by calling **nextDouble( )**. This method reads any number that can be converted into a **double** value, including an integer value, such as 2, and a floating-point value like 3.4. Thus, a number read by **nextDouble( )** need not specify a decimal point. This same general principle applies to all **next** methods. They will match and read any data format that can represent the type of value being requested.

One thing that is especially nice about **Scanner** is that the same technique used to read from one source can be used to read from another. For example, here is the preceding program reworked to average a list of numbers contained in a text file:

```
// Use Scanner to compute an average of the values in a file.
import java.util.*;
import java.io.*;

class AvgFile {
  public static void main(String args[])
    throws IOException {

    int count = 0;
    double sum = 0.0;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("2 3.4 5 6 7.4 9.1 10.5 done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Read and sum numbers.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
```

```
        String str = src.next();
        if(str.equals("done")) break;
        else {
          System.out.println("File format error.");
          return;
        }
      }
    }

    src.close();
    System.out.println("Average is " + sum / count);
  }
}
```

Here is the output:

```
Average is 6.2
```

The preceding program illustrates another important feature of **Scanner**. Notice that the file reader referred to by **fin** is not closed directly. Rather, it is closed automatically when **src** calls **close( )**. When you close a **Scanner**, the **Readable** associated with it is also closed (if that **Readable** implements the **Closeable** interface). Therefore, in this case, the file referred to by **fin** is automatically closed when **src** is closed.

Beginning with JDK 7, **Scanner** also implements the **AutoCloseable** interface. This means that it can be managed by a **try**-with-resources block. As explained in Chapter 13, when **try**-with-resources is used, the scanner is automatically closed when the block ends. For example, **src** in the preceding program could have been managed like this:

```
try (Scanner src = new Scanner(fin))
{
  // Read and sum numbers.
  while(src.hasNext()) {
    if(src.hasNextDouble()) {
      sum += src.nextDouble();
      count++;
    }
    else {
      String str = src.next();
      if(str.equals("done")) break;
      else {
        System.out.println("File format error.");
        return;
      }
    }
  }
}
```

To clearly demonstrate the closing of a **Scanner**, the following examples will call **close( )** explicitly. (Doing so also allows them to be compiled by versions of Java prior to JDK 7.) However, the **try**-with-resources approach is more streamlined and can help prevent errors. Its use is recommended for new code.

One other point: To keep this and the other examples in this section compact, I/O exceptions are simply thrown out of **main( )**. However, your real-world code will normally handle I/O exceptions itself.

You can use **Scanner** to read input that contains several different types of data—even if the order of that data is unknown in advance. You must simply check what type of data is available before reading it. For example, consider this program:

```
// Use Scanner to read various types of data from a file.
import java.util.*;
import java.io.*;

class ScanMixed {
  public static void main(String args[])
    throws IOException {

    int i;
    double d;
    boolean b;
    String str;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");
    fout.write("Testing Scanner 10 12.2 one true two false");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);

    // Read to end.
    while(src.hasNext()) {
      if(src.hasNextInt()) {
        i = src.nextInt();
        System.out.println("int: " + i);
      }
      else if(src.hasNextDouble()) {
        d = src.nextDouble();
        System.out.println("double: " + d);
      }
      else if(src.hasNextBoolean()) {
        b = src.nextBoolean();
        System.out.println("boolean: " + b);
      }
      else {
        str = src.next();
        System.out.println("String: " + str);
      }
    }

    src.close();
  }
}
```

Here is the output:

```
String: Testing
String: Scanner
int: 10
double: 12.2
String: one
boolean: true
String: two
boolean: false
```

When reading mixed data types, as the preceding program does, you need to be a bit careful about the order in which you call the **next** methods. For example, if the loop reversed the order of the calls to **nextInt( )** and **nextDouble( )**, both numeric values would have been read as **double**s, because **nextDouble( )** matches any numeric string that can be represented as a **double**.

## Setting Delimiters

**Scanner** defines where a token starts and ends based on a set of *delimiters*. The default delimiters are the whitespace characters, and this is the delimiter set that the preceding examples have used. However, it is possible to change the delimiters by calling the **useDelimiter( )** method, shown here:

Scanner useDelimiter(String *pattern*)

Scanner useDelimiter(Pattern *pattern*)

Here, *pattern* is a regular expression that specifies the delimiter set.

Here is the program that reworks the average program shown earlier so that it reads a list of numbers that are separated by commas, and any number of spaces:

```
// Use Scanner to compute an average a list of
// comma-separated values.
import java.util.*;
import java.io.*;

class SetDelimiters {
  public static void main(String args[])
    throws IOException {

    int count = 0;
    double sum = 0.0;

    // Write output to a file.
    FileWriter fout = new FileWriter("test.txt");

    // Now, store values in comma-separated list.
    fout.write("2, 3.4,     5,6, 7.4, 9.1, 10.5, done");
    fout.close();

    FileReader fin = new FileReader("Test.txt");

    Scanner src = new Scanner(fin);
```

```
    // Set delimiters to space and comma.
    src.useDelimiter(", *");

    // Read and sum numbers.
    while(src.hasNext()) {
      if(src.hasNextDouble()) {
        sum += src.nextDouble();
        count++;
      }
      else {
        String str = src.next();
        if(str.equals("done")) break;
        else {
          System.out.println("File format error.");
          return;
        }
      }
    }

    src.close();
    System.out.println("Average is " + sum / count);
  }
}
```

In this version, the numbers written to **test.txt** are separated by commas and spaces. The use of the delimiter pattern **", * "** tells **Scanner** to match a comma and zero or more spaces as delimiters. The output is the same as before.

You can obtain the current delimiter pattern by calling **delimiter( )**, shown here:

Pattern delimiter( )

## Other Scanner Features

**Scanner** defines several other methods in addition to those already discussed. One that is particularly useful in some circumstances is **findInLine( )**. Its general forms are shown here:

String findInLine(Pattern *pattern*)
String findInLine(String *pattern*)

This method searches for the specified pattern within the next line of text. If the pattern is found, the matching token is consumed and returned. Otherwise, null is returned. It operates independently of any delimiter set. This method is useful if you want to locate a specific pattern. For example, the following program locates the Age field in the input string and then displays the age:

```
// Demonstrate findInLine().
import java.util.*;

class FindInLineDemo {
  public static void main(String args[]) {
    String instr = "Name: Tom Age: 28 ID: 77";
```

```
    Scanner conin = new Scanner(instr);

    // Find and display age.
    conin.findInLine("Age:"); // find Age

    if(conin.hasNext())
      System.out.println(conin.next());
    else
      System.out.println("Error!");

    conin.close();
  }
}
```

The output is **28**. In the program, **findInLine( )** is used to find an occurrence of the pattern "Age". Once found, the next token is read, which is the age.

Related to **findInLine( )** is **findWithinHorizon( )**. It is shown here:

String findWithinHorizon(Pattern *pattern*, int *count*)

String findWithinHorizon(String *pattern*, int *count*)

This method attempts to find an occurrence of the specified pattern within the next *count* characters. If successful, it returns the matching pattern. Otherwise, it returns **null**. If *count* is zero, then all input is searched until either a match is found or the end of input is encountered.

You can bypass a pattern using **skip( )**, shown here:

Scanner skip(Pattern *pattern*)

Scanner skip(String *pattern*)

If *pattern* is matched, **skip( )** simply advances beyond it and returns a reference to the invoking object. If pattern is not found, **skip( )** throws **NoSuchElementException**.

Other **Scanner** methods include **radix( )**, which returns the default radix used by the **Scanner**; **useRadix( )**, which sets the radix; **reset( )**, which resets the scanner; and **close( )**, which closes the scanner.

## The ResourceBundle, ListResourceBundle, and PropertyResourceBundle Classes

The **java.util** package includes three classes that aid in the internationalization of your program. The first is the abstract class **ResourceBundle**. It defines methods that enable you to manage a collection of locale-sensitive resources, such as the strings that are used to label the user interface elements in your program. You can define two or more sets of translated strings that support various languages, such as English, German, or Chinese, with each translation set residing in its own bundle. You can then load the bundle appropriate to the current locale and use the strings to construct the program's user interface.

Resource bundles are identified by their *family name* (also called their *base name*). To the family name can be added a two-character lowercase *language code* which specifies the language. In this case, if a requested locale matches the language code, then that version of the resource bundle is used. For example, a resource bundle with a family name of **SampleRB** could have a German version called **SampleRB_de** and a Russian version called **SampleRB_ru**. (Notice that an underscore links the family name to the language code.) Therefore, if the locale is **Locale.GERMAN**, **SampleRB_de** will be used.

It is also possible to indicate specific variants of a language that relate to a specific country by specifying a *country code* after the language code. A country code is a two-character uppercase identifier, such as **AU** for Australia or **IN** for India. A country code is also preceded by an underscore when linked to the resource bundle name. A resource bundle that has only the family name is the default bundle. It is used when no language-specific bundles are applicable.



**NOTE** The language codes are defined by ISO standard 639 and the country codes by ISO standard 3166.

The methods defined by **ResourceBundle** are summarized in Table 19-18. One important point: **null** keys are not allowed and several of the methods will throw a **NullPointerException** if **null** is passed as the key. Notice the nested class **ResourceBundle.Control**. It is used to control the resource-bundle loading process.

| Method | Description |
|---|---|
| static final void clearCache( ) | Deletes all resource bundles from the cache that were loaded by the default class loader. |
| static final void clearCache(ClassLoader *ldr*) | Deletes all resource bundles from the cache that were loaded by *ldr*. |
| boolean containsKey(String *k*) | Returns **true** if *k* is a key within the invoking resource bundle (or its parent). |
| String getBaseBundleName( ) | Returns the resource bundle's base name if available. Returns **null** otherwise. (Added by JDK 8.) |
| static final ResourceBundle getBundle(String *familyName*) | Loads the resource bundle with a family name of *familyName* using the default locale and the default class loader. Throws **MissingResourceException** if no resource bundle matching the family name specified by *familyName* is available. |
| static final ResourceBundle getBundle(String *familyName*, Locale *loc*) | Loads the resource bundle with a family name of *familyName* using the specified locale and the default class loader. Throws **MissingResourceException** if no resource bundle matching the family name specified by *familyName* is available. |
| static ResourceBundle getBundle(String *familyName*, Locale *loc*, ClassLoader *ldr*) | Loads the resource bundle with a family name of *familyName* using the specified locale and the specified class loader. Throws **MissingResourceException** if no resource bundle matching the family name specified by *familyName* is available. |

**Table 19-18** The Methods Defined by **ResourceBundle**

| Method | Description |
|---|---|
| static final ResourceBundle getBundle(String *familyName*, ResourceBundle.Control *cntl*) | Loads the resource bundle with a family name of *familyName* using the default locale and the default class loader. The loading process is under the control of *cntl*. Throws **MissingResourceException** if no resource bundle matching the family name specified by *familyName* is available. |
| static final ResourceBundle getBundle(String *familyName*, Locale *loc*, ResourceBundle.Control *cntl*) | Loads the resource bundle with a family name of *familyName* using the specified locale and the default class loader. The loading process is under the control of *cntl*. Throws **MissingResourceException** if no resource bundle matching the family name specified by *familyName* is available. |
| static ResourceBundle getBundle(String *familyName*, Locale *loc*, ClassLoader *ldr*, ResourceBundle.Control *cntl*) | Loads the resource bundle with a family name of *familyName* using the specified locale and the specified class loader. The loading process is under the control of *cntl*. Throws **MissingResourceException** if no resource bundle matching the family name specified by *familyName* is available. |
| abstract Enumeration<String> getKeys( ) | Returns the resource bundle keys as an enumeration of strings. Any parent's keys are also obtained. |
| Locale getLocale( ) | Returns the locale supported by the resource bundle. |
| final Object getObject(String *k*) | Returns the object associated with the key passed via *k*. Throws **MissingResourceException** if *k* is not in the resource bundle. |
| final String getString(String *k*) | Returns the string associated with the key passed via *k*. Throws **MissingResourceException** if *k* is not in the resource bundle. Throws **ClassCastException** if the object associated with *k* is not a string. |
| final String[ ] getStringArray(String *k*) | Returns the string array associated with the key passed via *k*. Throws **MissingResourceException** if *k* is not in the resource bundle. Throws **MissingResourceException** if the object associated with *k* is not a string array. |
| protected abstract Object handleGetObject(String *k*) | Returns the object associated with the key passed via *k*. Returns **null** if *k* is not in the resource bundle. |
| protected Set<String> handleKeySet( ) | Returns the resource bundle keys as a set of strings. No parent's keys are obtained. Also, keys with **null** values are not obtained. |
| Set<String> keySet( ) | Returns the resource bundle keys as a set of strings. Any parent keys are also obtained. |
| protected void setParent(ResourceBundle *parent*) | Sets *parent* as the parent bundle for the resource bundle. When a key is looked up, the parent will be searched if the key is not found in the invoking resource object. |

**Table 19-18** The Methods Defined by **ResourceBundle** *(continued)*

There are two subclasses of **ResourceBundle**. The first is **PropertyResourceBundle**, which manages resources by using property files. **PropertyResourceBundle** adds no methods of its own. The second is the abstract class **ListResourceBundle**, which manages resources in an array of key/value pairs. **ListResourceBundle** adds the method **getContents( )**, which all subclasses must implement. It is shown here:

protected abstract Object[ ][ ] getContents( )

It returns a two-dimensional array that contains key/value pairs that represent resources. The keys must be strings. The values are typically strings, but can be other types of objects.

Here is an example that demonstrates using a resource bundle. The resource bundle has the family name **SampleRB**. Two resource bundle classes of this family are created by extending **ListResourceBundle**. The first is called **SampleRB**, and it is the default bundle (which uses English). It is shown here:

```
import java.util.*;
public class SampleRB extends ListResourceBundle {
  protected Object[][] getContents() {
    Object[][] resources = new Object[3][2];

    resources[0][0] = "title";
    resources[0][1] = "My Program";

    resources[1][0] = "StopText";
    resources[1][1] = "Stop";

    resources[2][0] = "StartText";
    resources[2][1] = "Start";

    return resources;
  }
}
```

The second resource bundle, shown next, is called **SampleRB_de**. It contains the German translation.

```
import java.util.*;

// German version.
public class SampleRB_de extends ListResourceBundle {
  protected Object[][] getContents() {
    Object[][] resources = new Object[3][2];

    resources[0][0] = "title";
    resources[0][1] = "Mein Programm";

    resources[1][0] = "StopText";
    resources[1][1] = "Anschlag";

    resources[2][0] = "StartText";
    resources[2][1] = "Anfang";
```

```
    return resources;
  }
}
```

The following program demonstrates these two resource bundles by displaying the string associated with each key for both the default (English) version and the German version:

```
// Demonstrate a resource bundle.
import java.util.*;

class LRBDemo {
  public static void main(String args[]) {
    // Load the default bundle.
    ResourceBundle rd = ResourceBundle.getBundle("SampleRB");

    System.out.println("English version: ");
    System.out.println("String for Title key : " +
                       rd.getString("title"));

    System.out.println("String for StopText key: " +
                       rd.getString("StopText"));

    System.out.println("String for StartText key: " +
                       rd.getString("StartText"));

    // Load the German bundle.
    rd = ResourceBundle.getBundle("SampleRB", Locale.GERMAN);

    System.out.println("\nGerman version: ");
    System.out.println("String for Title key : " +
                       rd.getString("title"));

    System.out.println("String for StopText key: " +
                       rd.getString("StopText"));

    System.out.println("String for StartText key: " +
                       rd.getString("StartText"));
  }
}
```

The output from the program is shown here:

```
English version:
String for Title key : My Program
String for StopText key: Stop
String for StartText key: Start

German version:
String for Title key : Mein Programm
String for StopText key: Anschlag
String for StartText key: Anfang
```

## Miscellaneous Utility Classes and Interfaces

In addition to the classes already discussed, **java.util** includes the following classes:

| | |
|---|---|
| Base64 | Supports Base64 encoding. **Encoder** and **Decoder** nested classes are also defined. (Added by JDK 8.) |
| DoubleSummaryStatistics | Supports the compilation of **double** values. The following statistics are available: average, minimum, maximum, count, and sum. (Added by JDK 8.) |
| EventListenerProxy | Extends the **EventListener** class to allow additional parameters. See Chapter 24 for a discussion of event listeners. |
| EventObject | The superclass for all event classes. Events are discussed in Chapter 24. |
| FormattableFlags | Defines formatting flags that are used with the **Formattable** interface. |
| IntSummaryStatistics | Supports the compilation of **int** values. The following statistics are available: average, minimum, maximum, count, and sum. (Added by JDK 8.) |
| Objects | Various methods that operate on objects. |
| PropertyPermission | Manages property permissions. |
| ServiceLoader | Provides a means of finding service providers. |
| StringJoiner | Supports the concatenation of **CharSequence**s, which may include a separator, a prefix, and a suffix. (Added by JDK 8.) |
| UUID | Encapsulates and manages Universally Unique Identifiers (UUIDs). |

The following interfaces are also packaged in **java.util**:

| | |
|---|---|
| EventListener | Indicates that a class is an event listener. Events are discussed in Chapter 24. |
| Formattable | Enables a class to provide custom formatting. |

## The java.util Subpackages

Java defines the following subpackages of **java.util**:

- java.util.concurrent
- java.util.concurrent.atomic
- java.util.concurrent.locks
- java.util.function
- java.util.jar
- java.util.logging
- java.util.prefs

- java.util.regex
- java.util.spi
- java.util.stream
- java.util.zip

Each is briefly examined here.

## java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks

The **java.util.concurrent** package along with its two subpackages, **java.util.concurrent.atomic** and **java.util.concurrent.locks**, support concurrent programming. These packages provide a high-performance alternative to using Java's built-in synchronization features when thread-safe operation is required. Beginning with JDK 7, **java.util.concurrent** also provides the Fork/Join Framework. These packages are examined in detail in Chapter 28.

## java.util.function

The **java.util.function** package defines several predefined functional interfaces that you can use when creating lambda expressions or method references. They are also widely used throughout the Java API. The functional interfaces defined by **java.util.function** are shown in Table 19-19 along with a synopsis of their abstract methods. Be aware that some of these interfaces also define default or static methods that supply additional functionality. You will want to explore them fully on your own. (For a discussion of the use of functional interfaces, see Chapter 15.)

| Interface | Abstract Method |
|---|---|
| BiConsumer<T, U> | void accept(T *tVal*, U *uVal*)<br>**Description**: Acts on *tVal* and *uVal*. |
| BiFunction<T, U, R> | R apply(T *tVal*, U *uVal*)<br>**Description**: Acts on *tVal* and *uVal* and returns the result. |
| BinaryOperator<T> | T apply(T *val1*, T *val2*)<br>**Description**: Acts on two objects of the same type and returns the result, which is also of the same type. |
| BiPredicate<T, U> | boolean test(T *tVal*, U *uVal*)<br>**Description**: Returns **true** if *tVal* and *uVal* satisfy the condition defined by **test( )** and **false** otherwise. |
| BooleanSupplier | boolean getAsBoolean( )<br>**Description**: Returns a **boolean** value. |
| Consumer<T> | void accept(T *val*)<br>**Description**: Acts on *val*. |

**Table 19-19** Functional Interfaces Defined by **java.util.function** and Their Abstract Methods

| Interface | Abstract Method |
|---|---|
| DoubleBinaryOperator | double applyAsDouble(double *val1*, double *val2*)<br>**Description**: Acts on two **double** values and returns a **double** result. |
| DoubleConsumer | void accept(double *val*)<br>**Description**: Acts on *val*. |
| DoubleFunction<R> | R apply(double *val*)<br>**Description**: Acts on a **double** value and returns the result. |
| DoublePredicate | boolean test(double *val*)<br>**Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise. |
| DoubleSupplier | double getAsDouble( )<br>**Description**: Returns a **double** result. |
| DoubleToIntFunction | int applyAsInt(double *val*)<br>**Description**: Acts on a **double** value and returns the result as an **int**. |
| DoubleToLongFunction | long applyAsLong(double *val*)<br>**Description**: Acts on a **double** value and returns the result as a **long**. |
| DoubleUnaryOperator | double applyAsDouble(double *val*)<br>**Description**: Acts on a **double** and returns a **double** result. |
| Function<T, R> | R apply(T *val*)<br>**Description**: Acts on *val* and returns the result. |
| IntBinaryOperator | int applyAsInt(int *val1*, int *val2*)<br>**Description**: Acts on two **int** values and returns an **int** result. |
| IntConsumer | int accept(int *val*)<br>**Description**: Acts on *val*. |
| IntFunction<R> | R apply(int *val*)<br>**Description**: Acts on an **int** value and returns the result. |
| IntPredicate | boolean test(int *val*)<br>**Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise. |
| IntSupplier | int getAsInt( )<br>**Description**: Returns an **int** result. |
| IntToDoubleFunction | double applyAsDouble(int *val*)<br>**Description**: Acts on an **int** value and returns the result as a **double**. |
| IntToLongFunction | long applyAsLong(int *val*)<br>**Description**: Acts on an **int** value and returns the result as a **long**. |

**Table 19-19** Functional Interfaces Defined by **java.util.function** and Their Abstract Methods *(continued)*

| Interface | Abstract Method |
|---|---|
| IntUnaryOperator | int applyAsInt(int *val*)<br>**Description**: Acts on an **int** and returns an **int** result. |
| LongBinaryOperator | long applyAsLong(long *val1*, long *val2*)<br>**Description**: Acts on two **long** values and returns a **long** result. |
| LongConsumer | void accept(long *val*)<br>**Description**: Acts on *val*. |
| LongFunction<R> | R apply(long *val*)<br>**Description**: Acts on a **long** value and returns the result. |
| LongPredicate | boolean test(long *val*)<br>**Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise. |
| LongSupplier | long getAsLong( )<br>**Description**: Returns a **long** result. |
| LongToDoubleFunction | double applyAsDouble(long *val*)<br>**Description**: Acts on a **long** value and returns the result as a **double**. |
| LongToIntFunction | int applyAsInt(long *val*)<br>**Description**: Acts on a **long** value and returns the result as an **int**. |
| LongUnaryOperator | long applyAsLong(long *val*)<br>**Description**: Acts on a **long** and returns a **long** result. |
| ObjDoubleConsumer<T> | void accept(T *val1*, double *val2*)<br>**Description**: Acts on *val1* and the **double** value *val2*. |
| ObjIntConsumer<T> | void accept(T *val1*, int *val2*)<br>**Description**: Acts on *val1* and the **int** value *val2*. |
| ObjLongConsumer<T> | void accept(T *val1*, long *val2*)<br>**Description**: Acts on *val1* and the **long** value *val2*. |
| Predicate<T> | boolean test(T val)<br>**Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise. |
| Supplier<T> | T get( )<br>**Description**: Returns an object of type **T**. |
| ToDoubleBiFunction<T, U> | double applyAsDouble(T *tVal*, U *uVal*)<br>**Description**: Acts on *tVal* and *uVal* and returns the result as a **double**. |
| ToDoubleFunction<T> | double applyAsDouble(T *val*)<br>**Description**: Acts on *val* and returns the result as a **double**. |

**Table 19-19** Functional Interfaces Defined by **java.util.function** and Their Abstract Methods *(continued)*

| Interface | Abstract Method |
|---|---|
| ToIntBiFunction<T, U> | int applyAsInt(T *tVal*, U *uVal*)<br>**Description**: Acts on *tVal* and *uVal* and returns the result as an **int**. |
| ToIntFunction<T> | int applyAsInt(T *val*)<br>**Description**: Acts on *val* and returns the result as an **int**. |
| ToLongBiFunction<T, U> | long applyAsLong(T *tVal*, U *uVal*)<br>**Description**: Acts on *tVal* and *uVal* and returns the result as a **long**. |
| ToLongFunction<T> | long applyAsLong(T *val*)<br>**Description**: Acts on *val* and returns the result as a **long**. |
| UnaryOperator<T> | T apply(T *val*)<br>**Description**: Acts on *val* and returns the result |

**Table 19-19** Functional Interfaces Defined by **java.util.function** and Their Abstract Methods *(continued)*



## java.util.jar

The **java.util.jar** package provides the ability to read and write Java Archive (JAR) files.

## java.util.logging

The **java.util.logging** package provides support for program activity logs, which can be used to record program actions, and to help find and debug problems.

## java.util.prefs

The **java.util.prefs** package provides support for user preferences. It is typically used to support program configuration.

## java.util.regex

The **java.util.regex** package provides support for regular expression handling. It is described in detail in Chapter 30.

## java.util.spi

The **java.util.spi** package provides support for service providers.

## java.util.stream

The **java.util.stream** package contains Java's stream API, which was added by JDK 8. A discussion of the stream API is found in Chapter 29.

## java.util.zip

The **java.util.zip** package provides the ability to read and write files in the popular ZIP and GZIP formats. Both ZIP and GZIP input and output streams are available.

This page has been intentionally left blank

CHAPTER

# 20 Input/Output: Exploring java.io

This chapter explores **java.io**, which provides support for I/O operations. Chapter 13 presented an overview of Java's I/O system, including basic techniques for reading and writing files, handling I/O exceptions, and closing a file. Here, we will examine the Java I/O system in greater detail.

As all programmers learn early on, most programs cannot accomplish their goals without accessing external data. Data is retrieved from an *input* source. The results of a program are sent to an *output* destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the *stream*. An I/O stream, as explained in Chapter 13, is a logical entity that either produces or consumes information. An I/O stream is linked to a physical device by the Java I/O system. All I/O streams behave in the same manner, even if the actual physical devices they are linked to differ.

**NOTE** The stream-based I/O system packaged in **java.io** and described in this chapter has been part of Java since its original release and is widely used. However, beginning with version 1.4, a second I/O system was added to Java. It is called NIO (which was originally an acronym for New I/O). NIO is packaged in **java.nio** and its subpackages. The NIO system is described in Chapter 21.

**NOTE** It is important not to confuse the I/O streams used by the I/O system discussed here with the new stream API added by JDK 8. Although conceptually related, they are two different things. Therefore, when the term *stream* is used in this chapter, it refers to an I/O stream.

## The I/O Classes and Interfaces

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

| | | |
|---|---|---|
| BufferedInputStream | FileWriter | PipedOutputStream |
| BufferedOutputStream | FilterInputStream | PipedReader |

| BufferedReader | FilterOutputStream | PipedWriter |
|---|---|---|
| BufferedWriter | FilterReader | PrintStream |
| ByteArrayInputStream | FilterWriter | PrintWriter |
| ByteArrayOutputStream | InputStream | PushbackInputStream |
| CharArrayReader | InputStreamReader | PushbackReader |
| CharArrayWriter | LineNumberReader | RandomAccessFile |
| Console | ObjectInputStream | Reader |
| DataInputStream | ObjectInputStream.GetField | SequenceInputStream |
| DataOutputStream | ObjectOutputStream | SerializablePermission |
| File | ObjectOutputStream.PutField | StreamTokenizer |
| FileDescriptor | ObjectStreamClass | StringReader |
| FileInputStream | ObjectStreamField | StringWriter |
| FileOutputStream | OutputStream | Writer |
| FilePermission | OutputStreamWriter | |
| FileReader | PipedInputStream | |

The **java.io** package also contains two deprecated classes that are not shown in the preceding table: **LineNumberInputStream** and **StringBufferInputStream**. These classes should not be used for new code.

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

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

As you can see, there are many classes and interfaces in the **java.io** package. These include byte and character streams, and object serialization (the storage and retrieval of objects). This chapter examines several commonly used I/O components. We begin our discussion with one of the most distinctive I/O classes: **File**.

## File

Although most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

**NOTE** The **Path** interface and **Files** class, which are part of the NIO system, offer a powerful alternative to **File** in many cases. See Chapter 21 for details.

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

Here, *directoryPath* is the path name of the file; *filename* is the name of the file or subdirectory; *dirObj* is a **File** object that specifies a directory; and *uriObj* is a **URI** object that describes a file.

The following example creates three files: **f1**, **f2**, and **f3**. The first **File** object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to **f1** and a filename; **f3** refers to the same file as **f2**.

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

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

**File** defines many methods that obtain the standard properties of a **File** object. For example, **getName( )** returns the name of the file; **getParent( )** returns the name of the parent directory; and **exists( )** returns **true** if the file exists, **false** if it does not. The following example demonstrates several of the **File** methods. It assumes that a directory called **java** exists off the root directory and that it contains a file called **COPYRIGHT**.

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");

    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
```

```
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

This program will produce output similar to this:

```
File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1282832030047
File size: 695 Bytes
```

Most of the **File** methods are self-explanatory. **isFile( )** and **isAbsolute( )** are not. **isFile( )** returns **true** if called on a file and **false** if called on a directory. Also, **isFile( )** returns **false** for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file. The **isAbsolute( )** method returns **true** if the file has an absolute path and **false** if its path is relative.

**File** includes two useful utility methods of special interest. The first is **renameTo( )**, shown here:

boolean renameTo(File *newName*)

Here, the filename specified by *newName* becomes the new name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed (if you attempt to rename a file so that it uses an existing filename, for example).

The second utility method is **delete( )**, which deletes the disk file represented by the path of the invoking **File** object. It is shown here:

boolean delete( )

You can also use **delete( )** to delete a directory if the directory is empty. **delete( )** returns **true** if it deletes the file and **false** if the file cannot be removed.

Here are some other **File** methods that you will find helpful:

| Method | Description |
|---|---|
| void deleteOnExit( ) | Removes the file associated with the invoking object when the Java Virtual Machine terminates. |
| long getFreeSpace( ) | Returns the number of free bytes of storage available on the partition associated with the invoking object. |
| long getTotalSpace( ) | Returns the storage capacity of the partition associated with the invoking object. |
| long getUsableSpace( ) | Returns the number of usable free bytes of storage available on the partition associated with the invoking object. |
| boolean isHidden( ) | Returns **true** if the invoking file is hidden. Returns **false** otherwise. |
| boolean setLastModified(long *millisec*) | Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC). |
| boolean setReadOnly( ) | Sets the invoking file to read-only. |

Methods also exist to mark files as readable, writable, and executable. Because **File** implements the **Comparable** interface, the method **compareTo( )** is also supported.

JDK 7 added a method to **File** called **toPath( )**, which is shown here:

```
Path toPath( )
```

**toPath( )** returns a **Path** object that represents the file encapsulated by the invoking **File** object. (In other words, **toPath( )** converts a **File** into a **Path**.) **Path** is packaged in **java.nio.file** and is part of NIO. Thus, **toPath( )** forms a bridge between the older **File** class and the newer **Path** interface. (See Chapter 21 for a discussion of **Path**.)

## Directories

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

```
String[ ] list( )
```

The list of files is returned in an array of **String** objects.

The program shown here illustrates how to use **list( )** to examine the contents of a directory:

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

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
```

```
    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

## Using FilenameFilter

You will often want to limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*. To do this, you must use a second form of **list( )**, shown here:

String[ ] list(FilenameFilter *FFObj*)

In this form, *FFObj* is an object of a class that implements the **FilenameFilter** interface.

**FilenameFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *directory*, String *filename*)

The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument) and returns **false** for those files that should be excluded.

The **OnlyExt** class, shown next, implements **FilenameFilter**. It will be used to modify the preceding program to restrict the visibility of the filenames returned by **list( )** to files with names that end in the file extension specified when the object is constructed.

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```



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

```
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

## The listFiles( ) Alternative

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

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

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

The third version of **listFiles( )** returns those files with path names that satisfy the specified **FileFilter**. **FileFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *path*)

The **accept( )** method returns **true** for files that should be included in the list (that is, those that match the *path* argument) and **false** for those that should be excluded.

### Creating Directories

Another two useful **File** utility methods are **mkdir( )** and **mkdirs( )**. The **mkdir( )** method creates a directory, returning **true** on success and **false** on failure. Failure can occur for various reasons, such as the path specified in the **File** object already exists, or the directory cannot be created because the entire path does not exist yet. To create a directory for which no path exists, use the **mkdirs( )** method. It creates both a directory and all the parents of the directory.

## The AutoCloseable, Closeable, and Flushable Interfaces

There are three interfaces that are quite important to the stream classes. Two are **Closeable** and **Flushable**. They are defined in **java.io** and were added by JDK 5. The third, **AutoCloseable**, was added by JDK 7. It is packaged in **java.lang**.

**AutoCloseable** provides support for the **try**-with-resources statement, which automates the process of closing a resource. (See Chapter 13.) Only objects of classes that implement **AutoCloseable** can be managed by **try**-with-resources. **AutoCloseable** is discussed in Chapter 17, but it is reviewed here for convenience. The **AutoCloseable** interface defines only the **close( )** method:

void close( ) throws Exception

This method closes the invoking object, releasing any resources that it may hold. It is called automatically at the end of a **try**-with-resources statement, thus eliminating the need to explicitly call **close( )**. Because this interface is implemented by all of the I/O classes that open a stream, all such streams can be automatically closed by a **try**-with-resources statement. Automatically closing a stream ensures that it is properly closed when it is no longer needed, thus preventing memory leaks and other problems.

The **Closeable** interface also defines the **close( )** method. Objects of a class that implement **Closeable** can be closed. Beginning with JDK 7, **Closeable** extends **AutoCloseable**. Therefore, any class that implements **Closeable** also implements **AutoCloseable**.

Objects of a class that implements **Flushable** can force buffered output to be written to the stream to which the object is attached. It defines the **flush( )** method, shown here:

void flush( ) throws IOException

Flushing a stream typically causes buffered output to be physically written to the underlying device. This interface is implemented by all of the I/O classes that write to a stream.

## I/O Exceptions

Two exceptions play an important role in I/O handling. The first is **IOException**. As it relates to most of the I/O classes described in this chapter, if an I/O error occurs, an **IOException** is thrown. In many cases, if a file cannot be opened, a **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single **catch** that catches **IOException**. For brevity, this is the approach used by most of the sample code in this chapter. However, in your own applications, you might find it useful to **catch** each exception separately.

Another exception class that is sometimes important when performing I/O is **SecurityException**. As explained in Chapter 13, in situations in which a security manager is present, several of the file classes will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, applets will use the security manager provided by the browser, and file I/O performed by an applet could generate a **SecurityException**. In such a case, you will need to handle this exception.



## Two Ways to Close a Stream

In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation. The techniques used to close a stream were described in Chapter 13, but because of their importance, they warrant a brief review here before the stream classes are examined.

Beginning with JDK 7, there are two basic ways in which you can close a stream. The first is to explicitly call **close( )** on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, **close( )** is typically called within a **finally** block. Thus, a simplified skeleton for the traditional approach is shown here:

```
try {
  // open and access file
} catch( I/O-exception) {
  // ...
} finally {
  // close the file
}
```

This general technique (or variation thereof) is common in code that predates JDK 7.

The second approach to closing a stream is to automate the process by using the **try**-with-resources statement that was added by JDK 7 (and, of course, supported by JDK 8). The **try**-with-resources statement is an enhanced form of **try** that has the following form:

```
try (resource-specification) {
  // use the resource
}
```

Here, *resource-specification* is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the

**try** block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call **close( )** explicitly.

Here are three key points about the **try**-with-resources statement:

- Resources managed by **try**-with-resources must be objects of classes that implement **AutoCloseable**.
- The resource declared in the **try** is implicitly **final**.
- You can manage more than one resource by separating each declaration by a semicolon.

Also, remember that the scope of the declared resource is limited to the **try**-with-resources statement.

The principal advantage of **try**-with-resources is that the resource (in this case, a stream) is closed automatically when the **try** block ends. Thus, it is not possible to forget to close the stream, for example. The **try**-with-resources approach also typically results in shorter, clearer, easier-to-maintain source code.

Because of its advantages, **try**-with-resources is expected to be used extensively in new code. As a result, most of the code in this chapter (and in this book) will use it. However, because a large amount of older code still exists, it is important for all programmers to also be familiar with the traditional approach to closing a stream. For example, you will quite likely have to work on legacy code that uses the traditional approach or in an environment that uses an older version of Java. There may also be times when the automated approach is not appropriate because of other aspects of your code. For this reason, a few I/O examples in this book will demonstrate the traditional approach so you can see it in action.

One last point: The examples that use **try**-with-resources must be compiled by a modern version of Java. They won't work with an older compiler. The examples that use the traditional approach can be compiled by older versions of Java.

---

**REMEMBER** Because **try**-with-resources streamlines the process of releasing a resource and eliminates the possibility of accidentally forgetting to release a resource, it is the approach recommended for new code when its use is appropriate.

## The Stream Classes

Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**. These classes were briefly discussed in Chapter 13. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

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

In the remainder of this chapter, both the byte- and character-oriented streams are examined.

# The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by **InputStream** and **OutputStream**, our discussion begins with them.

## InputStream

**InputStream** is an abstract class that defines Java's model of streaming byte input. It implements the **AutoCloseable** and **Closeable** interfaces. Most of the methods in this class will throw an **IOException** when an I/O error occurs. (The exceptions are **mark( )** and **markSupported( )**.) Table 20-1 shows the methods in **InputStream**.



**NOTE** Most of the methods described in Table 20-1 are implemented by the subclasses of **InputStream**. The **mark( )** and **reset( )** methods are exceptions; notice their use, or lack thereof, by each subclass in the discussions that follow.

## OutputStream

**OutputStream** is an abstract class that defines streaming byte output. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors. Table 20-2 shows the methods in **OutputStream**.

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

**Table 20-1** The Methods Defined by **InputStream**

| Method | Description |
|---|---|
| void close( ) | Closes the output stream. Further write attempts will generate an **IOException**. |
| void flush( ) | Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. |
| void write(int *b*) | Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with an expression without having to cast it back to **byte**. |
| void write(byte *buffer*[ ]) | Writes a complete array of bytes to an output stream. |
| void write(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*]. |

**Table 20-2** The Methods Defined by **OutputStream**

## FileInputStream

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Two commonly used constructors are shown here:

FileInputStream(String *filePath*)
FileInputStream(File *fileObj*)

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

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

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

Although the first constructor is probably more commonly used, the second allows you to closely examine the file using the **File** methods, before attaching it to an input stream. When a **FileInputStream** is created, it is also opened for reading. **FileInputStream** overrides six of the methods in the abstract class **InputStream**. The **mark( )** and **reset( )** methods are not overridden, and any attempt to use **reset( )** on a **FileInputStream** will generate an **IOException**.

The next example shows how to read a single byte, an array of bytes, and a subrange of an array of bytes. It also illustrates how to use **available( )** to determine the number of bytes remaining and how to use the **skip( )** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory. Notice that it uses the **try**-with-resources statement to automatically close the file when it is no longer needed.

```
// Demonstrate FileInputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;
```

```
class FileInputStreamDemo {
  public static void main(String args[]) {
    int size;

    // Use try-with-resources to close the stream.
    try ( FileInputStream f =
           new FileInputStream("FileInputStreamDemo.java") ) {

      System.out.println("Total Available Bytes: " +
                         (size = f.available()));

      int n = size/40;
      System.out.println("First " + n +
                         " bytes of the file one read() at a time");
      for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
      }

      System.out.println("\nStill Available: " + f.available());

      System.out.println("Reading the next " + n +
                         " with one read(b[])");
      byte b[] = new byte[n];
      if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
      }

      System.out.println(new String(b, 0, n));
      System.out.println("\nStill Available: " + (size = f.available()));
      System.out.println("Skipping half of remaining bytes with skip()");
      f.skip(size/2);
      System.out.println("Still Available: " + f.available());

      System.out.println("Reading " + n/2 + " into the end of array");
      if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("couldn't read " + n/2 + " bytes.");
      }

      System.out.println(new String(b, 0, b.length));
      System.out.println("\nStill Available: " + f.available());
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
// Demonstrate FileInputStream.
// This pr
Still Available: 1741
```

```
Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J

Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
      if (

Still Available: 827
```

This somewhat contrived example demonstrates how to read three ways, to skip input, and to inspect the amount of data available on a stream.

---

**NOTE** The preceding example and the other examples in this chapter handle any I/O exceptions that might occur as described in Chapter 13. See Chapter 13 for details and alternatives.

## FileOutputStream

**FileOutputStream** creates an **OutputStream** that you can use to write bytes to a file. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Four of its constructors are shown here:

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

They can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode.

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

The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes( )** method to extract the byte array equivalent. It then creates three files. The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileOutputStream.
// This program uses the traditional approach to closing a file.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
```

```
    byte buf[] = source.getBytes();
    FileOutputStream f0 = null;
    FileOutputStream f1 = null;
    FileOutputStream f2 = null;

    try {
      f0 = new FileOutputStream("file1.txt");
      f1 = new FileOutputStream("file2.txt");
      f2 = new FileOutputStream("file3.txt");

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

      // write to second file
      f1.write(buf);

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    } finally {
      try {
        if(f0 != null) f0.close();
      } catch(IOException e) {
        System.out.println("Error Closing file1.txt");
      }
      try {
        if(f1 != null) f1.close();
      } catch(IOException e) {
        System.out.println("Error Closing file2.txt");
      }
      try {
        if(f2 != null) f2.close();
      } catch(IOException e) {
        System.out.println("Error Closing file3.txt");
      }
    }
  }
}
```

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

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

Next, **file2.txt**:

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

Finally, **file3.txt**:

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

As the comment at the top of the program states, the preceding program shows an example that uses the traditional approach to closing a file when it is no longer needed. This approach is required by all versions of Java prior to JDK 7 and is widely used in legacy code. As you can see, quite a bit of rather awkward code is required to explicitly call **close( )** because each call could generate an **IOException** if the close operation fails. This program can be substantially improved by using the new **try**-with-resources statement. For comparison, here is the revised version. Notice that it is much shorter and streamlined:

```
// Demonstrate FileOutputStream.
// This version uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();

    // Use try-with-resources to close the files.
    try (FileOutputStream f0 = new FileOutputStream("file1.txt");
         FileOutputStream f1 = new FileOutputStream("file2.txt");
         FileOutputStream f2 = new FileOutputStream("file3.txt") )
    {

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

      // write to second file
      f1.write(buf);

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

## ByteArrayInputStream

**ByteArrayInputStream** is an implementation of an input stream that uses a byte array as the source. This class has two constructors, each of which requires a byte array to provide the data source:

ByteArrayInputStream(byte *array* [ ])
ByteArrayInputStream(byte *array* [ ], int *start*, int *numBytes*)

Here, *array* is the input source. The second constructor creates an **InputStream** from a subset of the byte array that begins with the character at the index specified by *start* and is *numBytes* long.

The **close( )** method has no effect on a **ByteArrayInputStream**. Therefore, it is not necessary to call **close( )** on a **ByteArrayInputStream**, but doing so is not an error.

The following example creates a pair of **ByteArrayInputStream**s, initializing them with the byte representation of the alphabet:

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

class ByteArrayInputStreamDemo {
  public static void main(String args[]) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    byte b[] = tmp.getBytes();

    ByteArrayInputStream input1 = new ByteArrayInputStream(b);
    ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
  }
}
```

The **input1** object contains the entire lowercase alphabet, whereas **input2** contains only the first three letters.

A **ByteArrayInputStream** implements both **mark( )** and **reset( )**. However, if **mark( )** has not been called, then **reset( )** sets the stream pointer to the start of the stream—which, in this case, is the start of the byte array passed to the constructor. The next example shows how to use the **reset( )** method to read the same input twice. In this case, the program reads and prints the letters "abc" once in lowercase and then again in uppercase.

```
import java.io.*;

class ByteArrayInputStreamReset {
  public static void main(String args[]) {
    String tmp = "abc";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(b);

    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

This example first reads each character from the stream and prints it as-is in lowercase. It then resets the stream and begins reading again, this time converting each character to uppercase before printing. Here's the output:

```
abc
ABC
```

## ByteArrayOutputStream

**ByteArrayOutputStream** is an implementation of an output stream that uses a byte array as the destination. **ByteArrayOutputStream** has two constructors, shown here:

ByteArrayOutputStream( )
ByteArrayOutputStream(int *numBytes*)

In the first form, a buffer of 32 bytes is created. In the second, a buffer is created with a size equal to that specified by *numBytes*. The buffer is held in the protected **buf** field of **ByteArrayOutputStream**. The buffer size will be increased automatically, if needed. The number of bytes held by the buffer is contained in the protected **count** field of **ByteArrayOutputStream**.

The **close( )** method has no effect on a **ByteArrayOutputStream**. Therefore, it is not necessary to call **close( )** on a **ByteArrayOutputStream**, but doing so is not an error.

The following example demonstrates **ByteArrayOutputStream**:

```
// Demonstrate ByteArrayOutputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class ByteArrayOutputStreamDemo {
  public static void main(String args[]) {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "This should end up in the array";
    byte buf[] = s.getBytes();

    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Error Writing to Buffer");
      return;
    }

    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");
    byte b[] = f.toByteArray();
    for (int i=0; i<b.length; i++) System.out.print((char) b[i]);

    System.out.println("\nTo an OutputStream()");

    // Use try-with-resources to manage the file stream.
    try ( FileOutputStream f2 = new FileOutputStream("test.txt") )
    {
      f.writeTo(f2);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
      return;
    }
```

```
    System.out.println("Doing a reset");
    f.reset();

    for (int i=0; i\<3; i++) f.write('X');

    System.out.println(f.toString());
  }
}
```

When you run the program, you will create the following output. Notice how after the call to **reset( )**, the three X's end up at the beginning.

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

This example uses the **writeTo( )** convenience method to write the contents of **f** to **test.txt**. Examining the contents of the **test.txt** file created in the preceding example shows the result we expected:

```
This should end up in the array
```

## Filtered Byte Streams

*Filtered streams* are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. These streams are typically accessed by methods that are expecting a generic stream, which is a superclass of the filtered streams. Typical extensions are buffering, character translation, and raw data translation. The filtered byte streams are **FilterInputStream** and **FilterOutputStream**. Their constructors are shown here:

FilterOutputStream(OutputStream *os*)
FilterInputStream(InputStream *is*)

The methods provided in these classes are identical to those in **InputStream** and **OutputStream**.

## Buffered Byte Streams

For the byte-oriented streams, a *buffered stream* extends a filtered stream class by attaching a memory buffer to the I/O stream. This buffer allows Java to do I/O operations on more than a byte at a time, thereby improving performance. Because the buffer is available, skipping, marking, and resetting of the stream become possible. The buffered byte stream classes are **BufferedInputStream** and **BufferedOutputStream**. **PushbackInputStream** also implements a buffered stream.

## BufferedInputStream

Buffering I/O is a very common performance optimization. Java's **BufferedInputStream** class allows you to "wrap" any **InputStream** into a buffered stream to improve performance.

**BufferedInputStream** has two constructors:

BufferedInputStream(InputStream *inputStream*)
BufferedInputStream(InputStream *inputStream*, int *bufSize*)

The first form creates a buffered stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*. Use of sizes that are multiples of a memory page, a disk block, and so on, can have a significant positive impact on performance. This is, however, implementation-dependent. An optimal buffer size is generally dependent on the host operating system, the amount of memory available, and how the machine is configured. To make good use of buffering doesn't necessarily require quite this degree of sophistication. A good guess for a size is around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the **InputStream**, you will be manipulating fast memory most of the time.

Buffering an input stream also provides the foundation required to support moving backward in the stream of the available buffer. Beyond the **read( )** and **skip( )** methods implemented in any **InputStream**, **BufferedInputStream** also supports the **mark( )** and **reset( )** methods. This support is reflected by **BufferedInputStream.markSupported( )** returning **true**.

The following example contrives a situation where we can use **mark( )** to remember where we are in an input stream and later use **reset( )** to get back there. This example is parsing a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset( )** happens and where it does not.

```
// Use buffered input.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    byte buf[] = s.getBytes();

    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;
    boolean marked = false;

    // Use try-with-resources to manage the file.
    try ( BufferedInputStream f = new BufferedInputStream(in) )
```

```
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '&':
          if (!marked) {
            f.mark(32);
            marked = true;
          } else {
            marked = false;
          }
          break;
        case ';':
          if (marked) {
            marked = false;
            System.out.print("(c)");
          } else
            System.out.print((char) c);
          break;
        case ' ':
          if (marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
            System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Notice that this example uses **mark(32)**, which preserves the mark for the next 32 bytes read (which is enough for all entity references). Here is the output produced by this program:

```
This is a (c) copyright symbol but this is &copy not.
```

## BufferedOutputStream

A **BufferedOutputStream** is similar to any **OutputStream** with the exception that the **flush( )** method is used to ensure that data buffers are written to the stream being buffered. Since the point of a **BufferedOutputStream** is to improve performance by reducing the number of times the system actually writes data, you may need to call **flush( )** to cause any data that is in the buffer to be immediately written.

Unlike buffered input, buffering output does not provide additional functionality. Buffers for output in Java are there to increase performance. Here are the two available constructors:

BufferedOutputStream(OutputStream *outputStream*)
BufferedOutputStream(OutputStream *outputStream*, int *bufSize*)

The first form creates a buffered stream using the default buffer size. In the second form, the size of the buffer is passed in *bufSize*.

## PushbackInputStream

One of the novel uses of buffering is the implementation of pushback. *Pushback* is used on an input stream to allow a byte to be read and then returned (that is, "pushed back") to the stream. The **PushbackInputStream** class implements this idea. It provides a mechanism to "peek" at what is coming from an input stream without disrupting it.

**PushbackInputStream** has the following constructors:

PushbackInputStream(InputStream *inputStream*)
PushbackInputStream(InputStream *inputStream*, int *numBytes*)

The first form creates a stream object that allows one byte to be returned to the input stream. The second form creates a stream that has a pushback buffer that is *numBytes* long. This allows multiple bytes to be returned to the input stream.

Beyond the familiar methods of **InputStream**, **PushbackInputStream** provides **unread( )**, shown here:

void unread(int *b*)
void unread(byte *buffer*[ ])
void unread(byte *buffer*, int *offset*, int *numBytes*)

The first form pushes back the low-order byte of *b*. This will be the next byte returned by a subsequent call to **read( )**. The second form pushes back the bytes in *buffer*. The third form pushes back *numBytes* bytes beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to push back a byte when the pushback buffer is full.

Here is an example that shows how a programming language parser might use a **PushbackInputStream** and **unread( )** to deal with the difference between the == operator for comparison and the = operator for assignment:

```
// Demonstrate unread().
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class PushbackInputStreamDemo {
  public static void main(String args[]) {
    String s = "if (a == 4) a = 0;\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;
```

```
    try ( PushbackInputStream f = new PushbackInputStream(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '=':
          if ((c = f.read()) == '=')
            System.out.print(".eq.");
          else {
            System.out.print("<-");
            f.unread(c);
          }
          break;
        default:
          System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is the output for this example. Notice that == was replaced by **".eq."** and **=** was replaced by **"<–"**.

```
    if (a .eq. 4) a <- 0;
```

**CAUTION** **PushbackInputStream** has the side effect of invalidating the **mark( )** or **reset( )** methods of the **InputStream** used to create it. Use **markSupported( )** to check any stream on which you are going to use **mark( )/reset( )**.

## SequenceInputStream

The **SequenceInputStream** class allows you to concatenate multiple **InputStream**s. The construction of a **SequenceInputStream** is different from any other **InputStream**. A **SequenceInputStream** constructor uses either a pair of **InputStream**s or an **Enumeration** of **InputStream**s as its argument:

SequenceInputStream(InputStream *first*, InputStream *second*)
SequenceInputStream(Enumeration <? extends InputStream> *streamEnum*)

Operationally, the class fulfills read requests from the first **InputStream** until it runs out and then switches over to the second one. In the case of an **Enumeration**, it will continue through all of the **InputStream**s until the end of the last one is reached. When the end of each file is reached, its associated stream is closed. Closing the stream created by **SequenceInputStream** causes all unclosed streams to be closed.

Here is a simple example that uses a **SequenceInputStream** to output the contents of two files. For demonstration purposes, this program uses the traditional technique used to

close a file. As an exercise, you might want to try changing it to use the **try**-with-resources statement.

```
// Demonstrate sequenced input.
// This program uses the traditional approach to closing a file.

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

class InputStreamEnumerator implements Enumeration<FileInputStream> {
  private Enumeration<String> files;

  public InputStreamEnumerator(Vector<String> files) {
    this.files = files.elements();
  }

  public boolean hasMoreElements() {
    return files.hasMoreElements();
  }

  public FileInputStream nextElement() {
    try {
      return new FileInputStream(files.nextElement().toString());
    } catch (IOException e) {
      return null;
    }
  }
}

class SequenceInputStreamDemo {
  public static void main(String args[]) {
    int c;
    Vector<String> files = new Vector<String>();

    files.addElement("file1.txt");
    files.addElement("file2.txt");
    files.addElement("file3.txt");
    InputStreamEnumerator ise = new InputStreamEnumerator(files);
    InputStream input = new SequenceInputStream(ise);

    try {
      while ((c = input.read()) != -1)
        System.out.print((char) c);
    } catch(NullPointerException e) {
      System.out.println("Error Opening File.");
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    } finally {
      try {
        input.close();
```

```
      } catch(IOException e) {
        System.out.println("Error Closing SequenceInputStream");
      }
    }
  }
}
```

This example creates a **Vector** and then adds three filenames to it. It passes that vector of names to the **InputStreamEnumerator** class, which is designed to provide a wrapper on the vector where the elements returned are not the filenames but, rather, open **FileInputStream**s on those names. The **SequenceInputStream** opens each file in turn, and this example prints the contents of the files.

Notice in **nextElement( )** that if a file cannot be opened, **null** is returned. This results in a **NullPointerException**, which is caught in **main( )**.



## PrintStream

The **PrintStream** class provides all of the output capabilities we have been using from the **System** file handle, **System.out**, since the beginning of the book. This makes **PrintStream** one of Java's most often used classes. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces.

**PrintStream** defines several constructors. The ones shown next have been specified from the start:

PrintStream(OutputStream *outputStream*)
PrintStream(OutputStream *outputStream*, boolean *autoFlushingOn*)
PrintStream(OutputStream *outputStream*, boolean *autoFlushingOn* String *charSet*)
    throws UnsupportedEncodingException

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *autoFlushingOn* parameter controls whether the output buffer is automatically flushed every time a newline (**\n**) character or a byte array is written or when **println( )** is called. If *autoFlushingOn* is **true**, flushing automatically takes place. If it is **false**, flushing is not automatic. The first constructor does not automatically flush. You can specify a character encoding by passing its name in *charSet*.

The next set of constructors gives you an easy way to construct a **PrintStream** that writes its output to a file:

PrintStream(File *outputFile*) throws FileNotFoundException
PrintStream(File *outputFile*, String *charSet*)
    throws FileNotFoundException, UnsupportedEncodingException
PrintStream(String *outputFileName*) throws FileNotFoundException
PrintStream(String *outputFileName*, String *charSet*) throws FileNotFoundException,
    UnsupportedEncodingException

These allow a **PrintStream** to be created from a **File** object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintStream** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet*.

**NOTE** If a security manager is present, some **PrintStream** constructors will throw a **SecurityException** if a security violation occurs.

**PrintStream** supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintStream** methods will call the object's **toString( )** method and then display the result.

Somewhat recently (with the release of JDK 5), the **printf( )** method was added to **PrintStream**. It allows you to specify the precise format of the data to be written. The **printf( )** method uses the **Formatter** class (described in Chapter 19) to format data. It then writes this data to the invoking stream. Although formatting can be done manually, by using **Formatter** directly, **printf( )** streamlines the process. It also parallels the C/C++ **printf( )** function, which makes it easy to convert existing C/C++ code into Java. Frankly, **printf( )** was a much welcome addition to the Java API because it greatly simplified the output of formatted data to the console.

The **printf( )** method has the following general forms:

PrintStream printf(String *fmtString*, Object … *args*)
PrintStream printf(Locale *loc*, String *fmtString*, Object … *args*)

The first version writes *args* to standard output in the format specified by *fmtString*, using the default locale. The second lets you specify a locale. Both return the invoking **PrintStream**.

In general, **printf( )** works in a manner similar to the **format( )** method specified by **Formatter**. The *fmtString* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains format specifiers that define the way the subsequent arguments, specified by *args*, are displayed. For complete information on formatting output, including a description of the format specifiers, see the **Formatter** class in Chapter 19.

Because **System.out** is a **PrintStream**, you can call **printf( )** on **System.out**. Thus, **printf( )** can be used in place of **println( )** when writing to the console whenever formatted output is desired. For example, the following program uses **printf( )** to output numeric values in various formats. Prior to JDK 5, such formatting required a bit of work. With the addition of **printf( )**, this is now an easy task.

```
// Demonstrate printf().

class PrintfDemo {
  public static void main(String args[]) {
    System.out.println("Here are some numeric values " +
                       "in different formats.\n");

    System.out.printf("Various integer formats: ");
    System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

    System.out.println();
    System.out.printf("Default floating-point format: %f\n",
                      1234567.123);
    System.out.printf("Floating-point with commas: %,f\n",
                      1234567.123);
```

```
    System.out.printf("Negative floating-point default: %,f\n",
                      -1234567.123);
    System.out.printf("Negative floating-point option: %,(f\n",
                      -1234567.123);

    System.out.println();

    System.out.printf("Line up positive and negative values:\n");
    System.out.printf("% ,.2f\n% ,.2f\n",
                      1234567.123, -1234567.123);
  }
}
```

The output is shown here:

```
Here are some numeric values in different formats.

Various integer formats: 3 (3) +3 00003

Default floating-point format: 1234567.123000
Floating-point with commas: 1,234,567.123000
Negative floating-point default: -1,234,567.123000
Negative floating-point option: (1,234,567.123000)

Line up positive and negative values:
  1,234,567.12
 -1,234,567.12
```

**PrintStream** also defines the **format( )** method. It has these general forms:

PrintStream format(String *fmtString*, Object … *args*)
PrintStream format(Locale *loc*, String *fmtString*, Object … *args*)

It works exactly like **printf( )**.

## DataOutputStream and DataInputStream

**DataOutputStream** and **DataInputStream** enable you to write or read primitive data to or from a stream. They implement the **DataOutput** and **DataInput** interfaces, respectively. These interfaces define methods that convert primitive values to or from a sequence of bytes. These streams make it easy to store binary data, such as integers or floating-point values, in a file. Each is examined here.

**DataOutputStream** extends **FilterOutputStream**, which extends **OutputStream**. In addition to implementing **DataOutput**, **DataOutputStream** also implements **AutoCloseable**, **Closeable**, and **Flushable**. **DataOutputStream** defines the following constructor:

DataOutputStream(OutputStream *outputStream*)

Here, *outputStream* specifies the output stream to which data will be written. When a **DataOutputStream** is closed (by calling **close( )**), the underlying stream specified by *outputStream* is also closed automatically.

**DataOutputStream** supports all of the methods defined by its superclasses. However, it is the methods defined by the **DataOutput** interface, which it implements, that make it interesting. **DataOutput** defines methods that convert values of a primitive type into a byte sequence and then writes it to the underlying stream. Here is a sampling of these methods:

final void writeDouble(double *value*) throws IOException
final void writeBoolean(boolean *value*) throws IOException
final void writeInt(int *value*) throws IOException

Here, *value* is the value written to the stream.

**DataInputStream** is the complement of **DataOuputStream**. It extends **FilterInputStream**, which extends **InputStream**. In addition to implementing the **DataInput** interface, **DataInputStream** also implements **AutoCloseable** and **Closeable**. Here is its only constructor:

DataInputStream(InputStream *inputStream*)

Here, *inputStream* specifies the input stream from which data will be read. When a **DataInputStream** is closed (by calling **close( )**), the underlying stream specified by *inputStream* is also closed automatically.

Like **DataOutputStream**, **DataInputStream** supports all of the methods of its superclasses, but it is the methods defined by the **DataInput** interface that make it unique. These methods read a sequence of bytes and convert them into values of a primitive type. Here is a sampling of these methods:

final double readDouble( ) throws IOException
final boolean readBoolean( ) throws IOException
final int readInt( ) throws IOException

The following program demonstrates the use of **DataOutputStream** and **DataInputStream**:

```
// Demonstrate DataInputStream and DataOutputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class DataIODemo {
  public static void main(String args[]) throws IOException {

    // First, write the data.
    try ( DataOutputStream dout =
            new DataOutputStream(new FileOutputStream("Test.dat")) )
    {
      dout.writeDouble(98.6);
      dout.writeInt(1000);
      dout.writeBoolean(true);
```

```
    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open Output File");
      return;
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    // Now, read the data back.
    try ( DataInputStream din =
            new DataInputStream(new FileInputStream("Test.dat")) )
    {

      double d = din.readDouble();
      int i = din.readInt();
      boolean b = din.readBoolean();

      System.out.println("Here are the values: " +
                         d + " " + i + " " + b);
    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open Input File");
      return;
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

The output is shown here:

```
Here are the values: 98.6 1000 true
```

## RandomAccessFile

**RandomAccessFile** encapsulates a random-access file. It is not derived from **InputStream** or **OutputStream**. Instead, it implements the interfaces **DataInput** and **DataOutput**, which define the basic I/O methods. It also implements the **AutoCloseable** and **Closeable** interfaces. **RandomAccessFile** is special because it supports positioning requests—that is, you can position the file pointer within the file. It has these two constructors:

RandomAccessFile(File *fileObj*, String *access*)
  throws FileNotFoundException

RandomAccessFile(String *filename*, String *access*)
  throws FileNotFoundException

In the first form, *fileObj* specifies the file to open as a **File** object. In the second form, the name of the file is passed in *filename*. In both cases, *access* determines what type of file access is permitted. If it is "r", then the file can be read, but not written. If it is "rw", then the file is opened in read-write mode. If it is "rws", the file is opened for read-write operations and

every change to the file's data or metadata will be immediately written to the physical device. If it is "rwd", the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device.

The method **seek( )**, shown here, is used to set the current position of the file pointer within the file:

```
void seek(long newPos) throws IOException
```

Here, *newPos* specifies the new position, in bytes, of the file pointer from the beginning of the file. After a call to **seek( )**, the next read or write operation will occur at the new file position.

**RandomAccessFile** implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods. One is **setLength( )**. It has this signature:

```
void setLength(long len) throws IOException
```

This method sets the length of the invoking file to that specified by *len*. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

# The Character Streams

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes. We will begin with them.

## Reader

**Reader** is an abstract class that defines Java's model of streaming character input. It implements the **AutoCloseable**, **Closeable**, and **Readable** interfaces. All of the methods in this class (except for **markSupported( )**) will throw an **IOException** on error conditions. Table 20-3 provides a synopsis of the methods in **Reader**.

## Writer

**Writer** is an abstract class that defines streaming character output. It implements the **AutoCloseable**, **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in this class throw an **IOException** in the case of errors. Table 20-4 shows a synopsis of the methods in **Writer**.

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

**Table 20-3** The Methods Defined by **Reader**

| Method | Description |
|---|---|
| Writer append(char *ch*) | Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream. |
| Writer append(CharSequence *chars*) | Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream. |
| Writer append(CharSequence *chars*, int *begin*, int *end*) | Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking output stream. Returns a reference to the invoking stream. |
| abstract void close( ) | Closes the output stream. Further write attempts will generate an **IOException**. |
| abstract void flush( ) | Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. |

**Table 20-4** The Methods Defined by **Writer**

| Method | Description |
|---|---|
| void write(int *ch*) | Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with an expression without having to cast it back to **char**. However, only the low-order 16 bits are written. |
| void write(char *buffer*[ ]) | Writes a complete array of characters to the invoking output stream. |
| abstract void write(char *buffer*[ ], int *offset*, int *numChars*) | Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream. |
| void write(String *str*) | Writes *str* to the invoking output stream. |
| void write(String *str*, int *offset*, int *numChars*) | Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*. |

**Table 20-4** The Methods Defined by **Writer** *(continued)*

## FileReader

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

FileReader(String *filePath*)
FileReader(File *fileObj*)

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

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

```
// Demonstrate FileReader.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

    try ( FileReader fr = new FileReader("FileReaderDemo.java") )
    {
      int c;

      // Read and display the file.
      while((c = fr.read()) != -1) System.out.print((char) c);

    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

## FileWriter

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

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

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

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

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

```
// Demonstrate FileWriter.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileWriterDemo {
  public static void main(String args[]) throws IOException {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    char buffer[] = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);

    try ( FileWriter f0 = new FileWriter("file1.txt");
          FileWriter f1 = new FileWriter("file2.txt");
          FileWriter f2 = new FileWriter("file3.txt") )
    {
      // write to first file
      for (int i=0; i < buffer.length; i += 2) {
        f0.write(buffer[i]);
      }

      // write to second file
      f1.write(buffer);

      // write to third file
      f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
```

```
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

## CharArrayReader

**CharArrayReader** is an implementation of an input stream that uses a character array as the source. This class has two constructors, each of which requires a character array to provide the data source:

CharArrayReader(char *array* [ ])
CharArrayReader(char *array* [ ], int *start*, int *numChars*)

Here, *array* is the input source. The second constructor creates a **Reader** from a subset of your character array that begins with the character at the index specified by *start* and is *numChars* long.

The **close( )** method implemented by **CharArrayReader** does not throw any exceptions. This is because it cannot fail.

The following example uses a pair of **CharArrayReader**s:

```
// Demonstrate CharArrayReader.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

public class CharArrayReaderDemo {
  public static void main(String args[]) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    int length = tmp.length();
    char c[] = new char[length];

    tmp.getChars(0, length, c, 0);
    int i;

    try (CharArrayReader input1 = new CharArrayReader(c) )
    {
      System.out.println("input1 is:");
      while((i = input1.read()) != -1) {
        System.out.print((char)i);
      }
      System.out.println();
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    try ( CharArrayReader input2 = new CharArrayReader(c, 0, 5) )
    {
      System.out.println("input2 is:");
      while((i = input2.read()) != -1) {
        System.out.print((char)i);
      }
```

```
      System.out.println();
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

The **input1** object is constructed using the entire lowercase alphabet, whereas **input2** contains only the first five letters. Here is the output:

```
input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is:
abcde
```



## CharArrayWriter

**CharArrayWriter** is an implementation of an output stream that uses an array as the destination. **CharArrayWriter** has two constructors, shown here:

CharArrayWriter( )
CharArrayWriter(int *numChars*)

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by *numChars*. The buffer is held in the **buf** field of **CharArrayWriter**. The buffer size will be increased automatically, if needed. The number of characters held by the buffer is contained in the **count** field of **CharArrayWriter**. Both **buf** and **count** are protected fields.

The **close( )** method has no effect on a **CharArrayWriter**.

The following example demonstrates **CharArrayWriter** by reworking the sample program shown earlier for **ByteArrayOutputStream**. It produces the same output as the previous version.

```
// Demonstrate CharArrayWriter.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class CharArrayWriterDemo {
  public static void main(String args[]) throws IOException {
    CharArrayWriter f = new CharArrayWriter();
    String s = "This should end up in the array";
    char buf[] = new char[s.length()];

    s.getChars(0, s.length(), buf, 0);

    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Error Writing to Buffer");
      return;
    }
```

```
    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");

    char c[] = f.toCharArray();
    for (int i=0; i<c.length; i++) {
      System.out.print(c[i]);
    }

    System.out.println("\nTo a FileWriter()");

    // Use try-with-resources to manage the file stream.
    try ( FileWriter f2 = new FileWriter("test.txt") )
    {
      f.writeTo(f2);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    System.out.println("Doing a reset");
    f.reset();

    for (int i=0; i<3; i++) f.write('X');

    System.out.println(f.toString());
  }
}
```

## BufferedReader

**BufferedReader** improves performance by buffering input. It has two constructors:

BufferedReader(Reader *inputStream*)
BufferedReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*.

Closing a **BufferedReader** also causes the underlying stream specified by *inputStream* to be closed.

As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer. To support this, **BufferedReader** implements the **mark( )** and **reset( )** methods, and **BufferedReader.markSupported( )** returns **true**. JDK 8 adds a new method to **BufferedReader** called **lines( )**. It returns a **Stream** reference to the sequence of lines read by the reader. (**Stream** is part of the new stream API discussed in Chapter 29.)

The following example reworks the **BufferedInputStream** example, shown earlier, so that it uses a **BufferedReader** character stream rather than a buffered byte stream. As before, it uses the **mark( )** and **reset( )** methods to parse a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset( )** happens and where it does not. Output is the same as that shown earlier.

```
// Use buffered input.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class BufferedReaderDemo {
  public static void main(String args[]) throws IOException {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);

    CharArrayReader in = new CharArrayReader(buf);
    int c;
    boolean marked = false;

    try ( BufferedReader f = new BufferedReader(in) )
    {

      while ((c = f.read()) != -1) {
        switch(c) {
        case '&':
          if (!marked) {
            f.mark(32);
            marked = true;
          } else {
            marked = false;
          }
          break;
        case ';':
          if (marked) {
            marked = false;
            System.out.print("(c)");
          } else
            System.out.print((char) c);
          break;
        case ' ':
          if (marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
            System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

## BufferedWriter

A **BufferedWriter** is a **Writer** that buffers output. Using a **BufferedWriter** can improve performance by reducing the number of times data is actually physically written to the output device.

A **BufferedWriter** has these two constructors:

BufferedWriter(Writer *outputStream*)
BufferedWriter(Writer *outputStream*, int *bufSize*)

The first form creates a buffered stream using a buffer with a default size. In the second, the size of the buffer is passed in *bufSize*.

## PushbackReader

The **PushbackReader** class allows one or more characters to be returned to the input stream. This allows you to look ahead in the input stream. Here are its two constructors:

PushbackReader(Reader *inputStream*)
PushbackReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered stream that allows one character to be pushed back. In the second, the size of the pushback buffer is passed in *bufSize*.

Closing a **PushbackReader** also closes the underlying stream specified by *inputStream*.

**PushbackReader** provides **unread( )**, which returns one or more characters to the invoking input stream. It has the three forms shown here:

void unread(int *ch*) throws IOException
void unread(char *buffer*[ ]) throws IOException
void unread(char *buffer*[ ], int *offset*, int *numChars*) throws IOException

The first form pushes back the character passed in *ch*. This will be the next character returned by a subsequent call to **read( )**. The second form returns the characters in *buffer*. The third form pushes back *numChars* characters beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to return a character when the pushback buffer is full.

The following program reworks the earlier **PushbackInputStream** example by replacing **PushbackInputStream** with **PushbackReader**. As before, it shows how a programming language parser can use a pushback stream to deal with the difference between the == operator for comparison and the = operator for assignment.

```
// Demonstrate unread().
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class PushbackReaderDemo {
  public static void main(String args[]) {
    String s = "if (a == 4) a = 0;\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);

    int c;
```

```
    try ( PushbackReader f = new PushbackReader(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '=':
          if ((c = f.read()) == '=')
            System.out.print(".eq.");
          else {
            System.out.print("<-");
            f.unread(c);
          }
          break;
        default:
          System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```



## PrintWriter

**PrintWriter** is essentially a character-oriented version of **PrintStream**. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces. **PrintWriter** has several constructors. The following have been supplied by **PrintWriter** from the start:

PrintWriter(OutputStream *outputStream*)
PrintWriter(OutputStream *outputStream*, boolean *autoFlushingOn*)
PrintWriter(Writer *outputStream*)
PrintWriter(Writer *outputStream*, boolean *autoFlushingOn*)

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *autoFlushingOn* parameter controls whether the output buffer is automatically flushed every time **println( )**, **printf( )**, or **format( )** is called. If *autoFlushingOn* is **true**, flushing automatically takes place. If **false**, flushing is not automatic. Constructors that do not specify the *autoFlushingOn* parameter do not automatically flush.

The next set of constructors gives you an easy way to construct a **PrintWriter** that writes its output to a file.

PrintWriter(File *outputFile*) throws FileNotFoundException
PrintWriter(File *outputFile*, String *charSet*)
 throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String *outputFileName*) throws FileNotFoundException
PrintWriter(String *outputFileName*, String *charSet*)
 throws FileNotFoundException, UnsupportedEncodingException

These allow a **PrintWriter** to be created from a **File** object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintWriter** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet*.

**PrintWriter** supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintWriter** methods will call the object's **toString( )** method and then output the result.

**PrintWriter** also supports the **printf( )** method. It works the same way it does in the **PrintStream** class described earlier: It allows you to specify the precise format of the data. Here is how **printf( )** is declared in **PrintWriter**:

PrintWriter printf(String *fmtString*, Object … *args*)
PrintWriter printf(Locale *loc*, String *fmtString*, Object …*args*)

The first version writes *args* to standard output in the format specified by *fmtString*, using the default locale. The second lets you specify a locale. Both return the invoking **PrintWriter**.

The **format( )** method is also supported. It has these general forms:

PrintWriter format(String *fmtString*, Object … *args*)
PrintWriter format(Locale *loc*, String *fmtString*, Object … *args*)

It works exactly like **printf( )**.

## The Console Class

The **Console** class was added to **java.io** by JDK 6. It is used to read from and write to the console, if one exists. It implements the **Flushable** interface. **Console** is primarily a convenience class because most of its functionality is available through **System.in** and **System.out**. However, its use can simplify some types of console interactions, especially when reading strings from the console.

**Console** supplies no constructors. Instead, a **Console** object is obtained by calling **System.console( )**, which is shown here:

static Console console( )

If a console is available, then a reference to it is returned. Otherwise, **null** is returned. A console will not be available in all cases. Thus, if **null** is returned, no console I/O is possible.

**Console** defines the methods shown in Table 20-5. Notice that the input methods, such as **readLine( )**, throw **IOError** if an input error occurs. **IOError** is a subclass of **Error**. It indicates an I/O failure that is beyond the control of your program. Thus, you will not normally catch an **IOError**. Frankly, if an **IOError** is thrown while accessing the console, it usually means there has been a catastrophic system failure.

Also notice the **readPassword( )** methods. These methods let your application read a password without echoing what is typed. When reading passwords, you should "zero-out" both the array that holds the string entered by the user and the array that holds the password that the string is tested against. This reduces the chance that a malicious program will be able to obtain a password by scanning memory.

| Method | Description |
|---|---|
| void flush( ) | Causes buffered output to be written physically to the console. |
| Console format(String *fmtString*, Object...*args*) | Writes *args* to the console using the format specified by *fmtString*. |
| Console printf(String *fmtString*, Object...*args*) | Writes *args* to the console using the format specified by *fmtString*. |
| Reader reader( ) | Returns a reference to a **Reader** connected to the console. |
| String readLine( ) | Reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| String readLine(String *fmtString*, Object…*args*) | Displays a prompting string formatted as specified by *fmtString* and *args*, and then reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| char[ ] readPassword( ) | Reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| char[ ] readPassword(String *fmtString*, Object… *args*) | Displays a prompting string formatted as specified by *fmtString* and *args*, and then reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| PrintWriter writer( ) | Returns a reference to a **Writer** connected to the console. |

**Table 20-5** The Methods Defined by **Console**

Here is an example that demonstrates the **Console** class:

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

class ConsoleDemo {
  public static void main(String args[]) {
    String str;
    Console con;
```

```
    // Obtain a reference to the console.
    con = System.console();
    // If no console available, exit.
    if(con == null) return;

    // Read a string and then display it.
    str = con.readLine("Enter a string: ");
    con.printf("Here is your string: %s\n", str);
  }
}
```

Here is sample output:

```
Enter a string: This is a test.
Here is your string: This is a test.
```

# Serialization

*Serialization* is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement *Remote Method Invocation (RMI)*. RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it. (More information about RMI appears in Chapter 30.)

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

An overview of the interfaces and classes that support serialization follows.

## Serializable

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.

## Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.

The **Externalizable** interface defines these two methods:

```
void readExternal(ObjectInput inStream)
  throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
  throws IOException
```

In these methods, *inStream* is the byte stream from which the object is to be read, and *outStream* is the byte stream to which the object is to be written.

## ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** and **AutoCloseable** interfaces and supports object serialization. It defines the methods shown in Table 20-6. Note especially the **writeObject( )** method. This is called to serialize an object. All of these methods will throw an **IOException** on error conditions.

| Method | Description |
|---|---|
| void close( ) | Closes the invoking stream. Further write attempts will generate an **IOException**. |
| void flush( ) | Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers. |
| void write(byte *buffer*[ ]) | Writes an array of bytes to the invoking stream. |
| void write(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*]. |
| void write(int *b*) | Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*. |
| void writeObject(Object *obj*) | Writes object *obj* to the invoking stream. |

**Table 20-6** The Methods Defined by **ObjectOutput**

## ObjectOutputStream

The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:

ObjectOutputStream(OutputStream *outStream*) throws IOException

The argument *outStream* is the output stream to which serialized objects will be written. Closing an **ObjectOutputStream** automatically closes the underlying stream specified by *outStream*.

Several commonly used methods in this class are shown in Table 20-7. They will throw an **IOException** on error conditions. There is also an inner class to **ObjectOuputStream** called **PutField**. It facilitates the writing of persistent fields, and its use is beyond the scope of this book.

| Method | Description |
|---|---|
| void close( ) | Closes the invoking stream. Further write attempts will generate an **IOException**. The underlying stream is also closed. |
| void flush( ) | Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers. |
| void write(byte *buffer*[ ]) | Writes an array of bytes to the invoking stream. |
| void write(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*]. |
| void write(int *b*) | Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*. |
| void writeBoolean(boolean *b*) | Writes a **boolean** to the invoking stream. |
| void writeByte(int *b*) | Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*. |
| void writeBytes(String *str*) | Writes the bytes representing *str* to the invoking stream. |
| void writeChar(int *c*) | Writes a **char** to the invoking stream. |
| void writeChars(String *str*) | Writes the characters in *str* to the invoking stream. |
| void writeDouble(double *d*) | Writes a **double** to the invoking stream. |
| void writeFloat(float *f*) | Writes a **float** to the invoking stream. |
| void writeInt(int *i*) | Writes an **int** to the invoking stream. |
| void writeLong(long *l*) | Writes a **long** to the invoking stream. |
| final void writeObject(Object *obj*) | Writes *obj* to the invoking stream. |
| void writeShort(int *i*) | Writes a **short** to the invoking stream. |

**Table 20-7** A Sampling of Commonly Used Methods Defined by **ObjectOutputStream**

## ObjectInput

The **ObjectInput** interface extends the **DataInput** and **AutoCloseable** interfaces and defines the methods shown in Table 20-8. It supports object serialization. Note especially the **readObject( )** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**.

## ObjectInputStream

The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream. One constructor of this class is shown here:

ObjectInputStream(InputStream *inStream*) throws IOException

The argument *inStream* is the input stream from which serialized objects should be read. Closing an **ObjectInputStream** automatically closes the underlying stream specified by *inStream*.

Several commonly used methods in this class are shown in Table 20-9. They will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**. There is also an inner class to **ObjectInputStream** called **GetField**. It facilitates the reading of persistent fields, and its use is beyond the scope of this book.



| Method | Description |
|---|---|
| int available( ) | Returns the number of bytes that are now available in the input buffer. |
| void close( ) | Closes the invoking stream. Further read attempts will generate an **IOException**. |
| int read( ) | Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ]) | Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered. |
| Object readObject( ) | Reads an object from the invoking stream. |
| long skip(long *numBytes*) | Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored. |

**Table 20-8** The Methods Defined by **ObjectInput**

| Method | Description |
|---|---|
| int available( ) | Returns the number of bytes that are now available in the input buffer. |
| void close( ) | Closes the invoking stream. Further read attempts will generate an **IOException**. The underlying stream is also closed. |
| int read( ) | Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered. |
| Boolean readBoolean( ) | Reads and returns a **boolean** from the invoking stream. |
| byte readByte( ) | Reads and returns a **byte** from the invoking stream. |
| char readChar( ) | Reads and returns a **char** from the invoking stream. |
| double readDouble( ) | Reads and returns a **double** from the invoking stream. |
| float readFloat( ) | Reads and returns a **float** from the invoking stream. |
| void readFully(byte *buffer*[ ]) | Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read. |
| void readFully(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read. |
| int readInt( ) | Reads and returns an **int** from the invoking stream. |
| long readLong( ) | Reads and returns a **long** from the invoking stream. |
| final Object readObject( ) | Reads and returns an object from the invoking stream. |
| short readShort( ) | Reads and returns a **short** from the invoking stream. |
| int readUnsignedByte( ) | Reads and returns an unsigned **byte** from the invoking stream. |
| int readUnsignedShort( ) | Reads and returns an unsigned **short** from the invoking stream. |

**Table 20-9** Commonly Used Methods Defined by **ObjectInputStream**

## A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String**, **int**, and **double**. This is the information we want to save and restore.

A **FileOutputStream** is created that refers to a file named "serial", and an **ObjectOutputStream** is created for that file stream. The **writeObject( )** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed.

A **FileInputStream** is then created that refers to the file named "serial", and an **ObjectInputStream** is created for that file stream. The **readObject( )** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.

Note that **MyClass** is defined to implement the **Serializable** interface. If this is not done, a **NotSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be **transient**. That data is then not saved during serialization.

```
// A serialization demo.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

    // Object serialization

    try ( ObjectOutputStream objOStrm =
            new ObjectOutputStream(new FileOutputStream("serial")) )
    {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);

      objOStrm.writeObject(object1);
    }
    catch(IOException e) {
      System.out.println("Exception during serialization: " + e);
    }

    // Object deserialization

    try ( ObjectInputStream objIStrm =
            new ObjectInputStream(new FileInputStream("serial")) )
    {
      MyClass object2 = (MyClass)objIStrm.readObject();
      System.out.println("object2: " + object2);
    }
    catch(Exception e) {
      System.out.println("Exception during deserialization: " + e);
    }
  }
}

class MyClass implements Serializable {
  String s;
  int i;
  double d;

  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }
```

```
  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical. The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

## Stream Benefits

The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task. The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements. Java programs written to adhere to the abstract, high-level **InputStream**, **OutputStream**, **Reader**, and **Writer** classes will function properly in the future even when new and improved concrete stream classes are invented. As you will see in Chapter 22, this model works very well when we switch from a file system–based set of streams to the network and socket streams. Finally, serialization of objects plays an important role in many types of Java programs. Java's serialization I/O classes provide a portable solution to this sometimes tricky task.

CHAPTER

# 21 Exploring NIO

Beginning with version 1.4, Java has provided a second I/O system called NIO (which is short for *New I/O*). It supports a buffer-oriented, channel-based approach to I/O operations. With the release of JDK 7, the NIO system was greatly expanded, providing enhanced support for file-handling and file system features. In fact, so significant were the changes that the term *NIO.2* is often used. Because of the capabilities supported by the NIO file classes, NIO is expected to become an increasingly important approach to file handling. This chapter explores several of the key features of the NIO system.

## The NIO Classes

The NIO classes are contained in the packages shown here:

| Package | Purpose |
|---|---|
| java.nio | Top-level package for the NIO system. Encapsulates various types of buffers that contain data operated upon by the NIO system. |
| java.nio.channels | Supports channels, which are essentially open I/O connections. |
| java.nio.channels.spi | Supports service providers for channels. |
| java.nio.charset | Encapsulates character sets. Also supports encoders and decoders that convert characters to bytes and bytes to characters, respectively. |
| java.nio.charset.spi | Supports service providers for character sets. |
| java.nio.file | Provides support for files. |
| java.nio.file.attribute | Provides support for file attributes. |
| java.nio.file.spi | Supports service providers for file systems. |

Before we begin, it is important to emphasize that the NIO subsystem does not replace the stream-based I/O classes found in **java.io**, which are discussed in Chapter 20, and good working knowledge of the stream-based I/O in **java.io** is helpful to understanding NIO.

**NOTE** This chapter assumes that you have read the overview of I/O given in Chapter 13 and the discussion of stream-based I/O supplied in Chapter 20.

# NIO Fundamentals

The NIO system is built on two foundational items: buffers and channels. A *buffer* holds data. A *channel* represents an open connection to an I/O device, such as a file or a socket. In general, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold data. You then operate on the buffer, inputting or outputting data as needed. The following sections examine buffers and channels in more detail.

## Buffers

Buffers are defined in the **java.nio** package. All buffers are subclasses of the **Buffer** class, which defines the core functionality common to all buffers: current position, limit, and capacity. The *current position* is the index within the buffer at which the next read or write operation will take place. The current position is advanced by most read or write operations. The *limit* is the index value one past the last valid location in the buffer. The *capacity* is the number of elements that the buffer can hold. Often the limit equals the capacity of the buffer. **Buffer** also supports mark and reset. **Buffer** defines several methods, which are shown in Table 21-1.

| Method | Description |
|---|---|
| abstract Object array( ) | If the invoking buffer is backed by an array, returns a reference to the array. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown. |
| abstract int arrayOffset( ) | If the invoking buffer is backed by an array, returns the index of the first element. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown. |
| final int capacity( ) | Returns the number of elements that the invoking buffer is capable of holding. |
| final Buffer clear( ) | Clears the invoking buffer and returns a reference to the buffer. |
| final Buffer flip( ) | Sets the invoking buffer's limit to the current position and resets the current position to 0. Returns a reference to the buffer. |
| abstract boolean hasArray( ) | Returns **true** if the invoking buffer is backed by a read/write array and **false** otherwise. |
| final boolean hasRemaining( ) | Returns **true** if there are elements remaining in the invoking buffer. Returns **false** otherwise. |

**Table 21-1** The Methods Defined by **Buffer**

| Method | Description |
|---|---|
| abstract boolean isDirect( ) | Returns **true** if the invoking buffer is direct, which means I/O operations act directly upon it. Returns **false** otherwise. |
| abstract boolean isReadOnly( ) | Returns **true** if the invoking buffer is read-only. Returns **false** otherwise. |
| final int limit( ) | Returns the invoking buffer's limit. |
| final Buffer limit(int *n*) | Sets the invoking buffer's limit to *n*. Returns a reference to the buffer. |
| final Buffer mark( ) | Sets the mark and returns a reference to the invoking buffer. |
| final int position( ) | Returns the current position. |
| final Buffer position(int *n*) | Sets the invoking buffer's current position to *n*. Returns a reference to the buffer. |
| int remaining( ) | Returns the number of elements available before the limit is reached. In other words, it returns the limit minus the current position. |
| final Buffer reset( ) | Resets the current position of the invoking buffer to the previously set mark. Returns a reference to the buffer. |
| final Buffer rewind( ) | Sets the position of the invoking buffer to 0. Returns a reference to the buffer. |

**Table 21-1** The Methods Defined by **Buffer** *(continued)*



From **Buffer**, the following specific buffer classes are derived, which hold the type of data that their names imply:

| | | | |
|---|---|---|---|
| ByteBuffer | CharBuffer | DoubleBuffer | FloatBuffer |
| IntBuffer | LongBuffer | MappedByteBuffer | ShortBuffer |

**MappedByteBuffer** is a subclass of **ByteBuffer** and is used to map a file to a buffer.

All of the aforementioned buffers provide various **get( )** and **put( )** methods, which allow you to get data from a buffer or put data into a buffer. (Of course, if a buffer is read-only, then **put( )** operations are not available.) Table 21-2 shows the **get( )** and **put( )** methods defined by **ByteBuffer**. The other buffer classes have similar methods. All buffer classes also support methods that perform various buffer operations. For example, you can allocate a buffer manually using **allocate( )**. You can wrap an array inside a buffer using **wrap( )**. You can create a subsequence of a buffer using **slice( )**.

## Channels

Channels are defined in **java.nio.channels**. A channel represents an open connection to an I/O source or destination. Channels implement the **Channel** interface. It extends **Closeable**, and it extends **AutoCloseable**. By implementing **AutoCloseable**, channels can be managed

| Method | Description |
|---|---|
| abstract byte get( ) | Returns the byte at the current position. |
| ByteBuffer get(byte *vals*[ ]) | Copies the invoking buffer into the array referred to by *vals*. Returns a reference to the buffer. If there are not *vals*.**length** elements remaining in the buffer, a **BufferUnderflowException** is thrown. |
| ByteBuffer get(byte *vals*[ ], int *start*, int *num*) | Copies *num* elements from the invoking buffer into the array referred to by *vals*, beginning at the index specified by *start*. Returns a reference to the buffer. If there are not *num* elements remaining in the buffer, a **BufferUnderflowException** is thrown. |
| abstract byte get(int *idx*) | Returns the byte at the index specified by *idx* within the invoking buffer. |
| abstract ByteBuffer put(byte *b*) | Copies *b* into the invoking buffer at the current position. Returns a reference to the buffer. If the buffer is full, a **BufferOverflowException** is thrown. |
| final ByteBuffer put(byte *vals*[ ] ) | Copies all elements of *vals* into the invoking buffer, beginning at the current position. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. |
| ByteBuffer put(byte *vals*[ ], int *start*, int *num*) | Copies *num* elements from *vals*, beginning at *start*, into the invoking buffer. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. |
| ByteBuffer put(ByteBuffer *bb*) | Copies the elements in *bb* to the invoking buffer, beginning at the current position. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. Returns a reference to the buffer. |
| abstract ByteBuffer put(int *idx*, byte *b*) | Copies *b* into the invoking buffer at the location specified by *idx*. Returns a reference to the buffer. |

**Table 21-2** The **get( )** and **put( )** Methods Defined for **ByteBuffer**

with a **try**-with-resources statement. When used in a **try**-with-resources block, a channel is closed automatically when it is no longer needed. (See Chapter 13 for a discussion of **try**-with-resources.)

One way to obtain a channel is by calling **getChannel( )** on an object that supports channels. For example, **getChannel( )** is supported by the following I/O classes:

| | | |
|---|---|---|
| DatagramSocket | FileInputStream | FileOutputStream |
| RandomAccessFile | ServerSocket | Socket |

The specific type of channel returned depends upon the type of object **getChannel( )** is called on. For example, when called on a **FileInputStream**, **FileOutputStream**, or **RandomAccessFile**, **getChannel( )** returns a channel of type **FileChannel**. When called on a **Socket**, **getChannel( )** returns a **SocketChannel**.

Another way to obtain a channel is to use one of the **static** methods defined by the **Files** class. For example, using **Files**, you can obtain a byte channel by calling **newByteChannel( )**. It returns a **SeekableByteChannel**, which is an interface implemented by **FileChannel**. (The **Files** class is examined in detail later in this chapter.)

Channels such as **FileChannel** and **SocketChannel** support various **read( )** and **write( )** methods that enable you to perform I/O operations through the channel. For example, here are a few of the **read( )** and **write( )** methods defined for **FileChannel**.

| Method | Description |
|---|---|
| abstract int read(ByteBuffer *bb*) throws IOException | Reads bytes from the invoking channel into *bb* until the buffer is full or there is no more input. Returns the number of bytes actually read. Returns –1 at end-of-stream. |
| abstract int read(ByteBuffer *bb*, long *start*) throws IOException | Beginning at the file location specified by *start*, reads bytes from the invoking channel into *bb* until the buffer is full or there is no more input. The current position is unchanged. Returns the number of bytes actually read or –1 if *start* is beyond the end of the file. |
| abstract int write(ByteBuffer *bb*) throws IOException | Writes the contents of *bb* to the invoking channel, starting at the current position. Returns the number of bytes written. |
| abstract int write(ByteBuffer *bb*, long *start*) throws IOException | Beginning at the file location specified by *start*, writes the contents of *bb* to the invoking channel. The current position is unchanged. Returns the number of bytes written. |

All channels support additional methods that give you access to and control over the channel. For example, **FileChannel** supports methods to get or set the current position, transfer information between file channels, obtain the current size of the channel, and lock the channel, among others. **FileChannel** provides a **static** method called **open( )**, which opens a file and returns a channel to it. This provides another way to obtain a channel. **FileChannel** also provides the **map( )** method, which lets you map a file to a buffer.

## Charsets and Selectors

Two other entities used by NIO are charsets and selectors. A *charset* defines the way that bytes are mapped to characters. You can encode a sequence of characters into bytes using an *encoder*. You can decode a sequence of bytes into characters using a *decoder*. Charsets, encoders, and decoders are supported by classes defined in the **java.nio.charset** package. Because default encoders and decoders are provided, you will not often need to work explicitly with charsets.

A *selector* supports key-based, non-blocking, multiplexed I/O. In other words, selectors enable you to perform I/O through multiple channels. Selectors are supported by classes defined in the **java.nio.channels** package. Selectors are most applicable to socket-backed channels.

We will not use charsets or selectors in this chapter, but you might find them useful in your own applications.

# Enhancements Added to NIO by JDK 7

Beginning with JDK 7, the NIO system was substantially expanded and enhanced. In addition to support for the **try**-with-resources statement (which provides automatic resource management), the improvements included three new packages (**java.nio.file**, **java.nio.file.attribute**, and **java.nio.file.spi**); several new classes, interfaces, and methods; and direct support for stream-based I/O. The additions have greatly expanded the ways in which NIO can be used, especially with files. Several of the key additions are described in the following sections.

## The Path Interface

Perhaps the single most important addition to the NIO system is the **Path** interface because it encapsulates a path to a file. As you will see, **Path** is the glue that binds together many of the NIO.2 file-based features. It describes a file's location within the directory structure. **Path** is packaged in **java.nio.file**, and it inherits the following interfaces: **Watchable**, **Iterable<Path>**, and **Comparable<Path>**. **Watchable** describes an object that can be monitored for changes. The **Iterable** and **Comparable** interfaces were described earlier in this book.

**Path** declares a number of methods that operate on the path. A sampling is shown in Table 21-3. Pay special attention to the **getName( )** method. It is used to obtain an element in a path. It works using an index. At index zero is the part of the path nearest the root, which is the leftmost element in a path. Subsequent indexes specify elements to the right of the root. The number of elements in a path can be obtained by calling **getNameCount( )**. If you want to obtain a string representation of the entire path, simply call **toString( )**. Notice that you can resolve a relative path into an absolute path by using the **resolve( )** method.

| Method | Description |
|---|---|
| boolean endsWith(String *path*) | Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**. |
| boolean endsWith(Path *path*) | Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**. |
| Path getFileName( ) | Returns the filename associated with the invoking **Path**. |
| Path getName(int *idx*) | Returns a **Path** object that contains the name of the path element specified by *idx* within the invoking object. The leftmost element is at index 0. This is the element nearest the root. The rightmost element is at **getNameCount( )** – 1. |
| int getNameCount( ) | Returns the number of elements beyond the root directory in the invoking **Path**. |
| Path getParent( ) | Returns a **Path** that contains the entire path except for the name of the file specified by the invoking **Path**. |
| Path getRoot( ) | Returns the root of the invoking **Path**. |

**Table 21-3** A Sampling of Methods Specified by **Path**

| Method | Description |
|---|---|
| boolean isAbsolute( ) | Returns **true** if the invoking **Path** is absolute. Otherwise, returns **false**. |
| Path resolve(Path *path*) | If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified. |
| Path resolve(String *path*) | If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified. |
| boolean startsWith(String *path*) | Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**. |
| boolean startsWith(Path *path*) | Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**. |
| Path toAbsolutePath( ) | Returns the invoking **Path** as an absolute path. |
| String toString( ) | Returns a string representation of the invoking **Path**. |

**Table 21-3** A Sampling of Methods Specified by **Path** *(continued)*



One other point: When updating legacy code that uses the **File** class defined by **java.io**, it is possible to convert a **File** instance into a **Path** instance by calling **toPath( )** on the **File** object. Furthermore, it is possible to obtain a **File** instance by calling the **toFile( )** method defined by **Path**.

## The Files Class

Many of the actions that you perform on a file are provided by **static** methods within the **Files** class. The file to be acted upon is specified by its **Path**. Thus, the **Files** methods use a **Path** to specify the file that is being operated upon. **Files** contains a wide array of functionality. For example, it has methods that let you open or create a file that has the specified path. You can obtain information about a **Path**, such as whether it is executable, hidden, or read-only. **Files** also supplies methods that let you copy or move files. A sampling is shown in Table 21-4. In addition to **IOException**, several other exceptions are possible. JDK 8 adds these four methods to **Files**: **list( )**, **walk( )**, **lines( )**, and **find( )**. All return a **Stream** object. These methods help integrate NIO with the new stream API defined by JDK 8 and described in Chapter 29.

Notice that several of the methods in Table 21-4 take an argument of type **OpenOption**. This is an interface that describes how to open a file. It is implemented by the **StandardOpenOption** class, which defines an enumeration that has the values shown in Table 21-5.

| Method | Description |
|---|---|
| static Path copy(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException | Copies the file specified by *src* to the location specified by *dest*. The *how* parameter specifies how the copy will take place. |
| static Path createDirectory(Path *path*, FileAttribute<?> ... *attribs*) throws IOException | Creates the directory whose path is specified by *path*. The directory attributes are specified by *attribs*. |
| static Path createFile(Path *path*, FileAttribute<?> ... *attribs*) throws IOException | Creates the file whose path is specified by *path*. The file attributes are specified by *attribs*. |
| static void delete(Path *path*) throws IOException | Deletes the file whose path is specified by *path*. |
| static boolean exists(Path *path*, LinkOption ... *opts*) | Returns **true** if the file specified by *path* exists and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static boolean isDirectory(Path *path*, LinkOption ... *opts*) | Returns **true** if *path* specifies a directory and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static boolean isExecutable(Path *path*) | Returns **true** if the file specified by *path* is executable and **false** otherwise. |
| static boolean isHidden(Path *path*) throws IOException | Returns **true** if the file specified by *path* is hidden and **false** otherwise. |
| static boolean isReadable(Path *path*) | Returns **true** if the file specified by *path* can be read from and **false** otherwise. |
| static boolean isRegularFile(Path *path*, LinkOption ... *opts*) | Returns **true** if *path* specifies a file and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static boolean isWritable(Path *path*) | Returns **true** if the file specified by *path* can be written to and **false** otherwise. |
| static Path move(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException | Moves the file specified by *src* to the location specified by *dest*. The *how* parameter specifies how the move will take place. |
| static SeekableByteChannel newByteChannel(Path *path*, OpenOption ... *how*) throws IOException | Opens the file specified by *path*, as specified by *how*. Returns a **SeekableByteChannel** to the file. This is a byte channel whose current position can be changed. **SeekableByteChannel** is implemented by **FileChannel**. |
| static DirectoryStream<Path> newDirectoryStream(Path *path*) throws IOException | Opens the directory specified by *path*. Returns a **DirectoryStream** linked to the directory. |

**Table 21-4** A Sampling of Methods Defined by **Files**

| Method | Description |
|---|---|
| static InputStream<br>newInputStream(Path *path*,<br>OpenOption ... *how*)<br>throws IOException | Opens the file specified by *path*, as specified by *how*. Returns an **InputStream** linked to the file. |
| static OutputStream<br>newOutputStream(Path *path*,<br>OpenOption ... *how*)<br>throws IOException | Opens the file specified by the invoking object, as specified by *how*. Returns an **OutputStream** linked to the file. |
| static boolean<br>notExists(Path *path*,<br>LinkOption ... *opts*) | Returns **true** if the file specified by *path* does *not* exist and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static <A extends BasicFileAttributes> A<br>readAttributes(Path *path*,<br>Class<A> *attribType*,<br>LinkOption ... *opts*)<br>throws IOException | Obtains the attributes associated with the file specified by *path*. The type of attributes to obtain is passed in *attribType*. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static long size(Path *path*)<br>throws IOException | Returns the size of the file specified by *path*. |

**Table 21-4** A Sampling of Methods Defined by **Files** *(continued)*

| Value | Meaning |
|---|---|
| APPEND | Causes output to be written to the end of the file. |
| CREATE | Creates the file if it does not already exist. |
| CREATE_NEW | Creates the file only if it does not already exist. |
| DELETE_ON_CLOSE | Deletes the file when it is closed. |
| DSYNC | Causes changes to the file to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed. |
| READ | Opens the file for input operations. |
| SPARSE | Indicates to the file system that the file is sparse, meaning that it may not be completely filled with data. If the file system does not support sparse files, this option is ignored. |
| SYNC | Causes changes to the file or its metadata to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed. |
| TRUNCATE_EXISTING | Causes a preexisting file opened for output to be reduced to zero length. |
| WRITE | Opens the file for output operations. |

**Table 21-5** The Standard Open Options

## The Paths Class

Because **Path** is an interface, not a class, you can't create an instance of **Path** directly through the use of a constructor. Instead, you obtain a **Path** by a calling a method that returns one. Frequently, you do this by using the **get( )** method defined by the **Paths** class. There are two forms of **get( )**. The one used in this chapter is shown here:

static Path get(String *pathname*, String ... *parts*)

It returns a **Path** that encapsulates the specified path. The path can be specified in two ways. First, if *parts* is not used, then the path must be specified in its entirety by *pathname*. Alternatively, you can pass the path in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. In either case, if the path specified is syntactically invalid, **get( )** will throw an **InvalidPathException**.

The second form of **get( )** creates a **Path** from a **URI**. It is shown here:

static Path get(URI *uri*)

The **Path** corresponding to *uri* is returned.

It is important to understand that creating a **Path** to a file does not open or create a file. It simply creates an object that encapsulates the file's directory path.

## The File Attribute Interfaces

Associated with a file is a set of attributes. These attributes include such things as the file's time of creation, the time of its last modification, whether the file is a directory, and its size. NIO organizes file attributes into several different interfaces. Attributes are represented by a hierarchy of interfaces defined in **java.nio.file.attribute**. At the top is **BasicFileAttributes**. It encapsulates the set of attributes that are commonly found in a variety of file systems. The methods defined by **BasicFileAttributes** are shown in Table 21-6.

| Method | Description |
|---|---|
| FileTime creationTime( ) | Returns the time at which the file was created. If creation time is not provided by the file system, then an implementation-dependent value is returned. |
| Object fileKey( ) | Returns the file key. If not supported, **null** is returned. |
| boolean isDirectory( ) | Returns **true** if the file represents a directory. |
| boolean isOther( ) | Returns **true** if the file is not a file, symbolic link, or a directory. |
| boolean isRegularFile( ) | Returns **true** if the file is a normal file, rather than a directory or symbolic link. |
| boolean isSymbolicLink( ) | Returns **true** if the file is a symbolic link. |
| FileTime lastAccessTime( ) | Returns the time at which the file was last accessed. If the time of last access is not provided by the file system, then an implementation-dependent value is returned. |
| FileTime lastModifiedTime( ) | Returns the time at which the file was last modified. If the time of last modification is not provided by the file system, then an implementation-dependent value is returned. |
| long size( ) | Returns the size of the file. |

**Table 21-6** The Methods Defined by **BasicFileAttributes**

From **BasicFileAttributes** two interfaces are derived: **DosFileAttributes** and **PosixFileAttributes**. **DosFileAttributes** describes those attributes related to the FAT file system as first defined by DOS. It defines the methods shown here:

| Method | Description |
|---|---|
| boolean isArchive( ) | Returns **true** if the file is flagged for archiving and **false** otherwise. |
| boolean isHidden( ) | Returns **true** if the file is hidden and **false** otherwise. |
| boolean isReadOnly( ) | Returns **true** if the file is read-only and **false** otherwise. |
| boolean isSystem( ) | Returns **true** if the file is flagged as a system file and **false** otherwise. |

**PosixFileAttributes** encapsulates attributes defined by the POSIX standards. (POSIX stands for *Portable Operating System Interface*.) It defines the methods shown here:

| Method | Description |
|---|---|
| GroupPrincipal group( ) | Returns the file's group owner. |
| UserPrincipal owner( ) | Returns the file's owner. |
| Set<PosixFilePermission> permissions( ) | Returns the file's permissions. |

There are various ways to access a file's attributes. First, you can obtain an object that encapsulates a file's attributes by calling **readAttributes( )**, which is a **static** method defined by **Files**. One of its forms is shown here:

```
static <A extends BasicFileAttributes>
    A readAttributes(Path path, Class<A> attrType, LinkOption... opts)
            throws IOException
```

This method returns a reference to an object that specifies the attributes associated with the file passed in *path*. The specific type of attributes is specified as a **Class** object in the *attrType* parameter. For example, to obtain the basic file attributes, pass **BasicFileAttributes.class** to *attrType*. For DOS attributes, use **DosFileAttributes.class**, and for POSIX attributes, use **PosixFileAttributes.class**. Optional link options are passed via *opts*. If not specified, symbolic links are followed. The method returns a reference to requested attributes. If the requested attribute type is not available, **UnsupportedOperationException** is thrown. Using the object returned, you can access the file's attributes.

A second way to gain access to a file's attributes is to call **getFileAttributeView( )** defined by **Files**. NIO defines several attribute view interfaces, including **AttributeView**, **BasicFileAttributeView**, **DosFileAttributeView**, and **PosixFileAttributeView**, among others. Although we won't be using attribute views in this chapter, they are a feature that you may find helpful in some situations.

In some cases, you won't need to use the file attribute interfaces directly because the **Files** class offers **static** convenience methods that access several of the attributes. For example, **Files** includes methods such as **isHidden( )** and **isWritable( )**.

It is important to understand that not all file systems support all possible attributes. For example, the DOS file attributes apply to the older FAT file system as first defined by DOS. The attributes that will apply to a wide variety of file systems are described by **BasicFileAttributes**. For this reason, these attributes are used in the examples in this chapter.

### The FileSystem, FileSystems, and FileStore Classes

You can easily access the file system through the **FileSystem** and **FileSystems** classes packaged in **java.nio.file**. In fact, by using the **newFileSystem( )** method defined by **FileSystems**, it is even possible to obtain a new file system. The **FileStore** class encapsulates the file storage system. Although these classes are not used directly in this chapter, you may find them helpful in your own applications.

## Using the NIO System

This section illustrates how to apply the NIO system to a variety of tasks. Before beginning, it is important to emphasize that with the release of JDK 7, the NIO subsystem was greatly expanded. As a result, its uses have also been greatly expanded. As mentioned, the enhanced version is sometimes referred to as NIO.2. Because the features added by NIO.2 are so substantial, they have changed the way that much NIO-based code is written and have increased the types of tasks to which NIO can be applied. Because of its importance, most of the remaining discussion and examples in this chapter utilize NIO.2 features and, therefore, require JDK 7, JDK 8, or later. However, at the end of the chapter is a brief description of pre-JDK 7 code. This will be of aid to those programmers working in pre-JDK 7 environments or maintaining older code.

---

**REMEMBER** Most of the examples in this chapter require JDK 7 or later.

In the past, the primary purpose of NIO was channel-based I/O, and this is still a very important use. However, you can now use NIO for stream-based I/O and for performing file-system operations. As a result, the discussion of using NIO is divided into three parts:

- Using NIO for channel-based I/O
- Using NIO for stream-based I/O
- Using NIO for path and file system operations

Because the most common I/O device is the disk file, the rest of this chapter uses disk files in the examples. Because all file channel operations are byte-based, the type of buffers that we will be using are of type **ByteBuffer**.

Before you can open a file for access via the NIO system, you must obtain a **Path** that describes the file. One way to do this is to call the **Paths.get( )** factory method, which was described earlier. The form of **get( )** used in the examples is shown here:

static Path get(String *pathname*, String ... *parts*)

Recall that the path can be specified in two ways. It can be passed in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. Alternatively, the entire path can be specified in *pathname* and *parts* is not used. This is the approach used by the examples.

### Use NIO for Channel-Based I/O

An important use of NIO is to access a file via a channel and buffers. The following sections demonstrate some techniques that use a channel to read from and write to a file.

## Reading a File via a Channel

There are several ways to read data from a file using a channel. Perhaps the most common way is to manually allocate a buffer and then perform an explicit read operation that loads that buffer with data from the file. It is with this approach that we begin.

Before you can read from a file, you must open it. To do this, first create a **Path** that describes the file. Then use this **Path** to open the file. There are various ways to open the file depending on how it will be used. In this example, the file will be opened for byte-based input via explicit input operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel( )**. The **newByteChannel( )** method has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. The **Path** that describes the file is passed in *path*. The *how* parameter specifies how the file will be opened. Because it is a varargs parameter, you can specify zero or more comma-separated arguments. (The valid values were discussed earlier and shown in Table 21-5.) If no arguments are specified, the file is opened for input operations. **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel** class. When the default file system is used, the returned object can be cast to **FileChannel**. You must close the channel after you have finished with it. Since all channels, including **FileChannel**, implement **AutoCloseable**, you can use a **try**-with-resources statement to close the file automatically instead of calling **close( )** explicitly. This approach is used in the examples.

Next, you must obtain a buffer that will be used by the channel either by wrapping an existing array or by allocating the buffer dynamically. The examples use allocation, but the choice is yours. Because file channels operate on byte buffers, we will use the **allocate( )** method defined by **ByteBuffer** to obtain the buffer. It has this general form:

```
static ByteBuffer allocate(int cap)
```

Here, *cap* specifies the capacity of the buffer. A reference to the buffer is returned.

After you have created the buffer, call **read( )** on the channel, passing a reference to the buffer. The version of **read( )** that we will use is shown next:

```
int read(ByteBuffer buf) throws IOException
```

Each time it is called, **read( )** fills the buffer specified by *buf* with data from the file. The reads are sequential, meaning that each call to **read( )** reads the next buffer's worth of bytes from the file. The **read( )** method returns the number of bytes actually read. It returns –1 when there is an attempt to read at the end of the file.

The following program puts the preceding discussion into action by reading a file called **test.txt** through a channel using explicit input operations:

```
// Use Channel I/O to read a file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
```

```
public class ExplicitChannelRead {
  public static void main(String args[]) {
    int count;
    Path filepath = null;

    // First, obtain a path to the file.
    try {
      filepath = Paths.get("test.txt");
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
      return;
    }

    // Next, obtain a channel to that file within a try-with-resources block.
    try ( SeekableByteChannel fChan = Files.newByteChannel(filepath) )
    {

      // Allocate a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen as characters.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);

      System.out.println();
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

Here is how the program works. First, a **Path** object is obtained that contains the relative path to a file called **test.txt**. A reference to this object is assigned to **filepath**. Next, a channel connected to the file is obtained by calling **newByteChannel( )**, passing in **filepath**. Because no open option is specified, the file is opened for reading. Notice that this channel is the object managed by the **try**-with-resources statement. Thus, the channel is automatically closed when the block ends. The program then calls the **allocate( )** method of **ByteBuffer** to allocate a buffer that will hold the contents of the file when it is read. A reference to this buffer is stored in **mBuf**. The contents of the file are then read, one buffer at a time, into **mBuf** through a call to **read( )**. The number of bytes read is stored in **count**. Next, the

buffer is rewound through a call to **rewind( )**. This call is necessary because the current position is at the end of the buffer after the call to **read( )**. It must be reset to the start of the buffer in order for the bytes in **mBuf** to be read by calling **get( )**. (Recall that **get( )** is defined by **ByteBuffer**.) Because **mBuf** is a byte buffer, the values returned by **get( )** are bytes. They are cast to **char** so the file can be displayed as text. (Alternatively, it is possible to create a buffer that encodes the bytes into characters and then read that buffer.) When the end of the file has been reached, the value returned by **read( )** will be –1. When this occurs, the program ends, and the channel is automatically closed.

As a point of interest, notice that the program obtains the **Path** within one **try** block and then uses another **try** block to obtain and manage a channel linked to that path. Although there is nothing wrong, per se, with this approach, in many cases, it can be streamlined so that only one **try** block is needed. In this approach, the calls to **Paths.get( )** and **newByteChannel( )** are sequenced together. For example, here is a reworked version of the program that uses this approach:



```
// A more compact way to open a channel. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Here, the channel is opened on the Path returned by Paths.get().
    // There is no need for the filepath variable.
    try ( SeekableByteChannel fChan =
            Files.newByteChannel(Paths.get("test.txt")) )
    {
      // Allocate a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen as characters.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);
```

```
      System.out.println();
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

In this version, the variable **filepath** is not needed and both exceptions are handled by the same **try** statement. Because this approach is more compact, it is the approach used in the rest of the examples in this chapter. Of course, in your own code, you may encounter situations in which the creation of a **Path** object needs to be separate from the acquisition of a channel. In these cases, the previous approach can be used.

Another way to read a file is to map it to a buffer. The advantage is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file, follow this general procedure. First, obtain a **Path** object that encapsulates the file as previously described. Next, obtain a channel to that file by calling **Files.newByteChannel( )**, passing in the **Path** and casting the returned object to **FileChannel**. As explained, **newByteChannel( )** returns a **SeekableByteChannel**. When using the default file system, this object can be cast to **FileChannel**. Then, map the channel to a buffer by calling **map( )** on the channel. The **map( )** method is defined by **FileChannel**. This is why the cast to **FileChannel** is needed. The **map( )** function is shown here:

```
MappedByteBuffer map(FileChannel.MapMode how,
                     long pos, long size) throws IOException
```

The **map( )** method causes the data in the file to be mapped into a buffer in memory. The value in *how* determines what type of operations are allowed. It must be one of these values:

| MapMode.READ_ONLY | MapMode.READ_WRITE | MapMode.PRIVATE |
|---|---|---|

For reading a file, use **MapMode.READ_ONLY**. To read and write, use **MapMode.READ_WRITE**. **MapMode.PRIVATE** causes a private copy of the file to be made, and changes to the buffer do not affect the underlying file. The location within the file to begin mapping is specified by *pos*, and the number of bytes to map are specified by *size*. A reference to this buffer is returned as a **MappedByteBuffer**, which is a subclass of **ByteBuffer**. Once the file has been mapped to a buffer, you can read the file from that buffer. Here is an example that illustrates this approach:

```
// Use a mapped file to read a file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

```
    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan =
         (FileChannel) Files.newByteChannel(Paths.get("test.txt")) )
    {

      // Get the size of the file.
      long fSize = fChan.size();

      // Now, map the file into a buffer.
      MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Read and display bytes from buffer.
      for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());

      System.out.println();

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

In the program, a **Path** to the file is created and then opened via **newByteChannel( )**. The channel is cast to **FileChannel** and stored in **fChan**. Next, the size of the file is obtained by calling **size( )** on the channel. Then, the entire file is mapped into memory by calling **map( )** on **fChan** and a reference to the buffer is stored in **mBuf**. Notice that **mBuf** is declared as a reference to a **MappedByteBuffer**. The bytes in **mBuf** are read by calling **get( )**.

## Writing to a File via a Channel

As is the case when reading from a file, there are also several ways to write data to a file using a channel. We will begin with one of the most common. In this approach, you manually allocate a buffer, write data to that buffer, and then perform an explicit write operation to write that data to a file.

Before you can write to a file, you must open it. To do this, first obtain a **Path** that describes the file and then use this **Path** to open the file. In this example, the file will be opened for byte-based output via explicit output operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel( )**. As shown in the previous section, the **newByteChannel( )** method has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. To open a file for output, the *how* parameter must specify **StandardOpenOption.WRITE**. If you want to create the file if it does not already exist, then you must also specify **StandardOpenOption.CREATE**. (Other options, which are shown in Table 21-5, are also available.) As explained in the previous section, **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel**

class. When the default file system is used, the return object can be cast to **FileChannel**. You must close the channel after you have finished with it.

Here is one way to write to a file through a channel using explicit calls to **write( )**. First, obtain a **Path** to the file and then open it with a call to **newByteChannel( )**, casting the result to **FileChannel**. Next, allocate a byte buffer and write data to that buffer. Before the data is written to the file, call **rewind( )** on the buffer to set its current position to zero. (Each output operation on the buffer increases the current position. Thus, it must be reset prior to writing to the file.) Then, call **write( )** on the channel, passing in the buffer. The following program demonstrates this procedure. It writes the alphabet to a file called **test.txt**.

```
// Write to a file using NIO. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan = (FileChannel)
            Files.newByteChannel(Paths.get("test.txt"),
                                 StandardOpenOption.WRITE,
                                 StandardOpenOption.CREATE) )
    {
      // Create a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

      // Reset the buffer so that it can be written.
      mBuf.rewind();

      // Write the buffer to the output file.
      fChan.write(mBuf);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
      System.exit(1);
    }
  }
}
```

It is useful to emphasize an important aspect of this program. As mentioned, after data is written to **mBuf**, but before it is written to the file, a call to **rewind( )** on **mBuf** is made. This is necessary in order to reset the current position to zero after data has been written to **mBuf**. Remember, each call to **put( )** on **mBuf** advances the current position. Therefore,

it is necessary for the current position to be reset to the start of the buffer before calling **write( )**. If this is not done, **write( )** will think that there is no data in the buffer.

Another way to handle the resetting of the buffer between input and output operations is to call **flip( )** instead of **rewind( )**. The **flip( )** method sets the value of the current position to zero and the limit to the previous current position. In the preceding example, because the capacity of the buffer equals its limit, **flip( )** could have been used instead of **rewind( )**. However, the two methods are not interchangeable in all cases.

In general, you must reset the buffer between read and write operations. For example, assuming the preceding example, the following loop will write the alphabet to the file three times. Pay special attention to the calls to **rewind( )**.

```
for(int h=0; h<3; h++) {
  // Write some bytes to the buffer.
  for(int i=0; i<26; i++)
    mBuf.put((byte)('A' + i));

  // Rewind the buffer so that it can be written.
  mBuf.rewind();

  // Write the buffer to the output file.
  fChan.write(mBuf);

  // Rewind the buffer so that it can be written to again.
  mBuf.rewind();
}
```

Notice that **rewind( )** is called between each read and write operation.

One other thing about the program warrants mentioning: When the buffer is written to the file, the first 26 bytes in the file will contain the output. If the file **test.txt** was preexisting, then after the program executes, the first 26 bytes of **test.txt** will contain the alphabet, but the remainder of the file will remain unchanged.

Another way to write to a file is to map it to a buffer. The advantage to this approach is that the data written to the buffer will automatically be written to the file. No explicit write operation is necessary. To map and write the contents of a file, we will use this general procedure. First, obtain a **Path** object that encapsulates the file and then create a channel to that file by calling **Files.newByteChannel( )**, passing in the **Path**. Cast the reference returned by **newByteChannel( )** to **FileChannel**. Next, map the channel to a buffer by calling **map( )** on the channel. The **map( )** method was described in detail in the previous section. It is summarized here for your convenience. Here is its general form:

```
MappedByteBuffer map(FileChannel.MapMode how,
                     long pos, long size) throws IOException
```

The **map( )** method causes the data in the file to be mapped into a buffer in memory. The value in *how* determines what type of operations are allowed. For writing to a file, *how* must be **MapMode.READ_WRITE**. The location within the file to begin mapping is specified by *pos*, and the number of bytes to map are specified by *size*. A reference to this buffer is returned. Once the file has been mapped to a buffer, you can write data to that buffer, and it will automatically be written to the file. Therefore, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used. Notice that in the call to **newByteChannel( )**, the open option **StandardOpenOption.READ** has been added. This is because a mapped buffer can either be read-only or read/write. Thus, to write to the mapped buffer, the channel must be opened as read/write.

```
// Write to a mapped file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan = (FileChannel)
          Files.newByteChannel(Paths.get("test.txt"),
                   StandardOpenOption.WRITE,
                   StandardOpenOption.READ,
                   StandardOpenOption.CREATE) )
    {

      // Then, map the file into a buffer.
      MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

As you can see, there are no explicit write operations to the channel itself. Because **mBuf** is mapped to the file, changes to **mBuf** are automatically reflected in the underlying file.

## Copying a File Using NIO

NIO simplifies several types of file operations. Although we can't examine them all, an example will give you an idea of what is available. The following program copies a file using a call to a single NIO method: **copy( )**, which is a **static** method defined by **Files**. It has several forms. Here is the one we will be using:

static Path copy(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException

The file specified by *src* is copied to the file specified by *dest*. How the copy is performed is specified by *how*. Because it is a varargs parameter, it can be missing. If specified, it can be one or more of these values, which are valid for all file systems:

| | |
|---|---|
| StandardCopyOption.COPY_ATTRIBUTES | Request that the file's attributes be copied. |
| StandardLinkOption.NOFOLLOW_LINKS | Do not follow symbolic links. |
| StandardCopyOption.REPLACE_EXISTING | Overwrite a preexisting file. |

Other options may be supported, depending on the implementation.

The following program demonstrates **copy( )**. The source and destination files are specified on the command line, with the source file specified first. Notice how short the program is. You might want to compare this version of the file copy program to the one found in Chapter 13. As you will find, the part of the program that actually copies the file is substantially shorter in the NIO version shown here.



```
// Copy a file using NIO. Requires JDK 7 or later.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

  public static void main(String args[]) {

    if(args.length != 2) {
      System.out.println("Usage: Copy from to");
      return;
    }

    try {
      Path source = Paths.get(args[0]);
      Path target = Paths.get(args[1]);

      // Copy the file.
      Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

## Use NIO for Stream-Based I/O

Beginning with NIO.2, you can use NIO to open an I/O stream. Once you have a **Path**, open a file by calling **newInputStream( )** or **newOutputStream( )**, which are **static** methods defined by **Files**. These methods return a stream connected to the specified file. In either case, the stream can then be operated on in the way described in Chapter 20, and the same techniques apply. The advantage of using **Path** to open a file is that all of the features defined by NIO are available for your use.

To open a file for stream-based input, use **Files.newInputStream( )**. It is shown here:

static InputStream newInputStream(Path *path*, OpenOption ... *how*)
throws IOException

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an input stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.READ** were passed.

Once opened, you can use any of the methods defined by **InputStream**. For example, you can use **read( )** to read bytes from the file.

The following program demonstrates the use of NIO-based stream I/O. It reworks the **ShowFile** program from Chapter 13 so that it uses NIO features to open the file and obtain a stream. As you can see, it is very similar to the original, except for the use of **Path** and **newInputStream( )**.

```
/* Display a text file using stream-based, NIO code.
   Requires JDK 7 or later.

   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   java ShowFile TEST.TXT
*/

import java.io.*;
import java.nio.file.*;

class ShowFile {
  public static void main(String args[])
  {
    int i;

    // First, confirm that a filename has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // Open the file and obtain a stream linked to it.
    try ( InputStream fin = Files.newInputStream(Paths.get(args[0])) )
    {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(IOException e) {
```

```
      System.out.println("I/O Error "  + e);
    }
  }
}
```

Because the stream returned by **newInputStream( )** is a normal stream, it can be used like any other stream. For example, you can wrap the stream inside a buffered stream, such as a **BufferedInputStream**, to provide buffering, as shown here:

```
new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))
```

Now, all reads will be automatically buffered.

To open a file for output, use **Files.newOutputStream( )**. It is shown here:

static OutputStream newOutputStream(Path *path*, OpenOption ... *how*)
throws IOException

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an output stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.WRITE**, **StandardOpenOption.CREATE**, and **StandardOpenOption.TRUNCATE_EXISTING** were passed.

The methodology for using **newOutputStream( )** is similar to that shown previously for **newInputStream( )**. Once opened, you can use any of the methods defined by **OutputStream**. For example, you can use **write( )** to write bytes to the file. You can also wrap the stream inside a **BufferedOutputStream** to buffer the stream.

The following program shows **newOutputStream( )** in action. It writes the alphabet to a file called **test.txt**. Notice the use of buffered I/O.

```
// Demonstrate NIO-based, stream output. Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
  public static void main(String args[])
  {
    // Open the file and obtain a stream linked to it.
    try ( OutputStream fout =
          new BufferedOutputStream(
                Files.newOutputStream(Paths.get("test.txt"))) )
    {
      // Write some bytes to the stream.
      for(int i=0; i < 26; i++)
        fout.write((byte)('A' + i));

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

## Use NIO for Path and File System Operations

At the beginning of Chapter 20, the **File** class in the **java.io** package was examined. As explained there, the **File** class deals with the file system and with the various attributes associated with a file, such as whether a file is read-only, hidden, and so on. It was also used to obtain information about a file's path. Although the **File** class is still perfectly acceptable, the interfaces and classes defined by NIO.2 offer a better way to perform these functions. The benefits include support for symbolic links, better support for directory tree traversal, and improved handling of metadata, among others. The following sections show samples of two common file system operations: obtaining information about a path and file and getting the contents of a directory.

---

**REMEMBER** If you want to update older code that uses **java.io.File** to the new **Path** interface, you can use the **toPath( )** method to obtain a **Path** instance from a **File** instance.

### Obtain Information About a Path and a File

Information about a path can be obtained by using methods defined by **Path**. Some attributes associated with the file described by a **Path** (such as whether or not the file is hidden) are obtained by using methods defined by **Files**. The **Path** methods used here are **getName( )**, **getParent( )**, and **toAbsolutePath( )**. Those provided by **Files** are **isExecutable( )**, **isHidden( )**, **isReadable( )**, **isWritable( )**, and **exists( )**. These are summarized in Tables 21-3 and 21-4, shown earlier.

---

**CAUTION** Methods such as **isExecutable( ), isReadable( ), isWritable( ),** and **exists( )** must be used with care because the state of the file system may change after the call, in which case a program malfunction could occur. Such a situation could have security implications.

Other file attributes are obtained by requesting a list of attributes by calling **Files.readAttributes( )**. In the program, this method is called to obtain the **BasicFileAttributes** associated with a file, but the general approach applies to other types of attributes.

The following program demonstrates several of the **Path** and **Files** methods, along with several methods provided by **BasicFileAttributes**. This program assumes that a file called **test.txt** exists in a directory called **examples**, which must be a subdirectory of the current directory.

```
// Obtain information about a path and a file.
// Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
  public static void main(String args[]) {
    Path filepath = Paths.get("examples\\test.txt");
```

```
    System.out.println("File Name: " + filepath.getName(1));
    System.out.println("Path: " + filepath);
    System.out.println("Absolute Path: " + filepath.toAbsolutePath());
    System.out.println("Parent: " + filepath.getParent());

    if(Files.exists(filepath))
      System.out.println("File exists");
    else
      System.out.println("File does not exist");

    try {
      if(Files.isHidden(filepath))
        System.out.println("File is hidden");
      else
        System.out.println("File is not hidden");
    } catch(IOException e) {
        System.out.println("I/O Error: " + e);
    }

    Files.isWritable(filepath);
    System.out.println("File is writable");


    Files.isReadable(filepath);
    System.out.println("File is readable");

    try {
      BasicFileAttributes attribs =
        Files.readAttributes(filepath, BasicFileAttributes.class);

      if(attribs.isDirectory())
        System.out.println("The file is a directory");
      else
        System.out.println("The file is not a directory");

      if(attribs.isRegularFile())
        System.out.println("The file is a normal file");
      else
        System.out.println("The file is not a normal file");

      if(attribs.isSymbolicLink())
        System.out.println("The file is a symbolic link");
      else
        System.out.println("The file is not a symbolic link");

      System.out.println("File last modified: " + attribs.lastModifiedTime());
      System.out.println("File size: " + attribs.size() + " Bytes");
    } catch(IOException e) {
      System.out.println("Error reading attributes: " + e);
    }
  }
}
```

If you execute this program from a directory called **MyDir**, which has a subdirectory called **examples**, and the **examples** directory contains the **test.txt** file, then you will see output similar to that shown here. (Of course, the information you see will differ.)

```
File Name: test.txt
Path: examples\test.txt
Absolute Path: C:\MyDir\examples\test.txt
Parent: examples
File exists
File is not hidden
File is writable
File is readable
The file is not a directory
The file is a normal file
The file is not a symbolic link
File last modified: 2014-01-01T18:20:46.380445Z
File size: 18 Bytes
```

If you are using a computer that supports the FAT file system (i.e., the DOS file system), then you might want to try using the methods defined by **DosFileAttributes**. If you are using a POSIX-compatible system, then try using **PosixFileAttributes**.

## List the Contents of a Directory

If a path describes a directory, then you can read the contents of that directory by using **static** methods defined by **Files**. To do this, you first obtain a directory stream by calling **newDirectoryStream( )**, passing in a **Path** that describes the directory. One form of **newDirectoryStream( )** is shown here:

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath)
                throws IOException
```

Here, *dirPath* encapsulates the path to the directory. The method returns a **DirectoryStream<Path>** object that can be used to obtain the contents of the directory. It will throw an **IOException** if an I/O error occurs and a **NotDirectoryException** (which is a subclass of **IOException**) if the specified path is not a directory. A **SecurityException** is also possible if access to the directory is not permitted.

**DirectoryStream<Path>** implements **AutoCloseable**, so it can be managed by a **try**-with-resources statement. It also implements **Iterable<Path>**. This means that you can obtain the contents of the directory by iterating over the **DirectoryStream** object. When iterating, each directory entry is represented by a **Path** instance. An easy way to iterate over a **DirectoryStream** is to use a for-each style **for** loop. It is important to understand, however, that the iterator implemented by **DirectoryStream<Path>** can be obtained only once for each instance. Thus, the **iterator( )** method can be called only once, and a for-each loop can be executed only once.

The following program displays the contents of a directory called **MyDir**:

```
// Display a directory. Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
```

```
class DirList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    // Obtain and manage a directory stream within a try block.
    try ( DirectoryStream<Path> dirstrm =
            Files.newDirectoryStream(Paths.get(dirname)) )
    {
      System.out.println("Directory of " + dirname);

      // Because DirectoryStream implements Iterable, we
      // can use a "foreach" loop to display the directory.
      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
             Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
          System.out.print("<DIR> ");
        else
          System.out.print("      ");

        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " is not a directory.");
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is sample output from the program:

```
Directory of \MyDir
      DirList.class
      DirList.java
<DIR> examples
      Test.txt
```

You can filter the contents of a directory in two ways. The easiest is to use this version of **newDirectoryStream( )**:

static DirectoryStream<Path> newDirectoryStream(Path *dirPath*, String *wildcard*)
throws IOException

In this version, only files that match the wildcard filename specified by *wildcard* will be obtained. For *wildcard*, you can specify either a complete filename or a *glob*. A *glob* is a string that defines a general pattern that will match one or more files using the familiar * and ? wildcard characters. These match zero or more of any character and any one character, respectively. The following are also recognized within a glob.

| | |
|---|---|
| ** | Matches zero or more of any character across directories. |
| [*chars*] | Matches any one character in *chars*. A * or ? within *chars* will be treated as a normal character, not a wildcard. A range can be specified by use of a hyphen, such as [x-z]. |
| {*globlist*} | Matches any one of the globs specified in a comma-separated list of globs in *globlist*. |

You can specify a * or ? character, using * and \?. To specify a \, use \\. You can experiment with a glob by substituting this call to **newDirectoryStream( )** into the previous program:

```
Files.newDirectoryStream(Paths.get(dirname), "{Path,Dir}*.{java,class}")
```

This obtains a directory stream that contains only those files whose names begin with either "Path" or "Dir" and use either the "java" or "class" extension. Thus, it would match names like **DirList.java** and **PathDemo.java**, but not **MyPathDemo.java**, for example.

Another way to filter a directory is to use this version of **newDirectoryStream( )**:

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath,
                DirectoryStream.Filter<? super Path> filefilter)
                        throws IOException
```

Here, **DirectoryStream.Filter** is an interface that specifies the following method:

```
boolean accept(T entry) throws IOException
```

In this case, **T** will be **Path**. If you want to include *entry* in the list, return **true**. Otherwise, return **false**. This form of **newDirectoryStream( )** offers the advantage of being able to filter a directory based on something other than a filename. For example, you can filter based on size, creation date, modification date, or attribute, to name a few.

The following program demonstrates the process. It will list only those files that are writable.

```
// Display a directory of only those files that are writable.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class DirList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    // Create a filter that returns true only for writable files.
    DirectoryStream.Filter<Path> how = new DirectoryStream.Filter<Path>() {
      public boolean accept(Path filename) throws IOException {
        if(Files.isWritable(filename)) return true;
        return false;
      }
    };
```

```
    // Obtain and manage a directory stream of writable files.
    try (DirectoryStream<Path> dirstrm =
           Files.newDirectoryStream(Paths.get(dirname), how) )
    {
      System.out.println("Directory of " + dirname);

      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
          Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
          System.out.print("<DIR> ");
        else
          System.out.print("      ");

        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " is not a directory.");
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

## Use walkFileTree( ) to List a Directory Tree

The preceding examples have obtained the contents of only a single directory. However, sometimes you will want to obtain a list of the files in a directory tree. In the past, this was quite a chore, but NIO.2 makes it easy because now you can use the **walkFileTree( )** method defined by **Files** to process a directory tree. It has two forms. The one used in this chapter is shown here:

```
static Path walkFileTree(Path root, FileVisitor<? extends Path> fv)
              throws IOException
```

The path to the starting point of the directory walk is passed in *root*. An instance of **FileVisitor** is passed in *fv*. The implementation of **FileVisitor** determines how the directory tree is traversed, and it gives you access to the directory information. If an I/O error occurs, an **IOException** is thrown. A **SecurityException** is also possible.

**FileVisitor** is an interface that defines how files are visited when a directory tree is traversed. It is a generic interface that is declared like this:

```
interface FileVisitor<T>
```

For use in **walkFileTree( )**, **T** will be **Path** (or any type derived from **Path**). **FileVisitor** defines the following methods.

| Method | Description |
|---|---|
| FileVisitResult<br>postVisitDirectory(T *dir*, IOException *exc*)<br>throws IOException | Called after a directory has been visited. The directory is passed in *dir*, and any **IOException** is passed in *exc*. If *exc* is **null**, no exception occurred. The result is returned. |
| FileVisitResult<br>preVisitDirectory(T *dir*,<br>BasicFileAttributes *attribs*)<br>throws IOException | Called before a directory is visited. The directory is passed in *dir*, and the attributes associated with the directory are passed in *attribs*. The result is returned. To examine the directory, return **FileVisitResult.CONTINUE**. |
| FileVisitResult<br>visitFile(T *file*, BasicFileAttributes *attribs*)<br>throws IOException | Called when a file is visited. The file is passed in *file*, and the attributes associated with the file are passed in *attribs*. The result is returned. |
| FileVisitResult<br>visitFileFailed(T *file*, IOException *exc*)<br>throws IOException | Called when an attempt to visit a file fails. The file that failed is passed in *file*, and the **IOException** is passed in *exc*. The result is returned. |

Notice that each method returns a **FileVisitResult**. This enumeration defines the following values:

| | | | |
|---|---|---|---|
| CONTINUE | SKIP_SIBLINGS | SKIP_SUBTREE | TERMINATE |

In general, to continue traversing the directory and subdirectories, a method should return **CONTINUE**. For **preVisitDirectory( )**, return **SKIP_SIBLINGS** to bypass the directory and its siblings and prevent **postVisitDirectory( )** from being called. To bypass just the directory and subdirectories, return **SKIP_SUBTREE**. To stop the directory traversal, return **TERMINATE**.

Although it is certainly possible to create your own visitor class that implements these methods defined by **FileVisitor**, you won't normally do so because a simple implementation is provided by **SimpleFileVisitor**. You can just override the default implementation of the method or methods in which you are interested. Here is a short example that illustrates the process. It displays all files in the directory tree that has **\MyDir** as its root. Notice how short this program is.

```
// A simple example that uses walkFileTree( ) to display a directory tree.
// Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;


// Create a custom version of SimpleFileVisitor that overrides
// the visitFile( ) method.
class MyFileVisitor extends SimpleFileVisitor<Path> {
  public FileVisitResult visitFile(Path path, BasicFileAttributes attribs)
```

```
    throws IOException
  {
    System.out.println(path);
    return FileVisitResult.CONTINUE;
  }
}

class DirTreeList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    System.out.println("Directory tree starting with " + dirname + ":\n");

    try {
      Files.walkFileTree(Paths.get(dirname), new MyFileVisitor());
    } catch (IOException exc) {
      System.out.println("I/O Error");
    }
  }
}
```



Here is sample output produced by the program when used on the same **MyDir** directory shown earlier. In this example, the subdirectory called **examples** contains one file called **MyProgram.java**.

```
Directory tree starting with \MyDir:

\MyDir\DirList.class
\MyDir\DirList.java
\MyDir\examples\MyProgram.java
\MyDir\Test.txt
```

In the program, the class **MyFileVisitor** extends **SimpleFileVisitor**, overriding only the **visitFile( )** method. In this example, **visitFile( )** simply displays the files, but more sophisticated functionality is easy to achieve. For example, you could filter the files or perform actions on the files, such as copying them to a backup device. For the sake of clarity, a named class was used to override **visitFile( )**, but you could also use an anonymous inner class.

One last point: It is possible to watch a directory for changes by using **java.nio.file.WatchService**.

## Pre-JDK 7 Channel-Based Examples

Before concluding this chapter, one more aspect of NIO needs to be covered. The preceding sections have used several of the new features added to NIO by JDK 7. However, you may encounter pre-JDK 7 code that will need to be maintained or possibly converted to use the new features. For this reason, the following sections show how to read and write files using the pre-JDK 7 NIO system. They do so by reworking some of the examples shown earlier so that they use the original NIO features, rather than those supported by NIO.2. This means that the examples in this section will work with versions of Java prior to JDK 7.

The key difference between pre-JDK 7 NIO code and newer NIO code is the **Path** interface, which was added by JDK 7. Thus, pre-JDK 7 code does not use **Path** to describe a file or open a channel to it. Also, pre-JDK 7 code does not use **try**-with-resource statements since automatic resource management was also added by JDK 7.

**REMEMBER** The examples in this section describe how legacy NIO code works. This section is strictly for the benefit of those programmers working on pre-JDK 7 code or using pre-JDK 7 compilers. New code should take advantage of the NIO features added by JDK 7.

## Read a File, Pre-JDK 7

This section reworks the two channel-based file input examples shown earlier so they use only pre-JDK 7 features. The first example reads a file by manually allocating a buffer and then performing an explicit read operation. The second example uses a mapped file, which automates the process.

When using a pre-JDK 7 version of Java to read a file using a channel and a manually allocated buffer, you first open the file for input using **FileInputStream**, using the same mechanism explained in Chapter 20. Next, obtain a channel to this file by calling **getChannel( )** on the **FileInputStream** object. It has this general form:

```
FileChannel getChannel( )
```

It returns a **FileChannel** object, which encapsulates the channel for file operations. Then, call **allocate( )** to allocate a buffer. Because file channels operate on byte buffers, you will use the **allocate( )** method defined by **ByteBuffer**, which works as previously described.

The following program shows how to read and display a file called **test.txt** through a channel using explicit input operations for versions of Java prior to JDK 7:

```
// Use Channels to read a file. Pre-JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelRead {
  public static void main(String args[]) {
    FileInputStream fIn = null;
    FileChannel fChan = null;
    ByteBuffer mBuf;
    int count;

    try {
      // First, open a file for input.
      fIn = new FileInputStream("test.txt");

      // Next, obtain a channel to that file.
      fChan = fIn.getChannel();

      // Allocate a buffer.
      mBuf = ByteBuffer.allocate(128);

      do {
```

```
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);

      System.out.println();

    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fIn != null) fIn.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

In this program, notice that the file is opened by using the **FileInputStream** constructor, and a reference to that object is assigned to **fIn**. Next, a channel connected to the file is obtained by calling **getChannel( )** on **fIn**. After this point, the program works like the NIO.2 version shown previously. To synopsize: The program then calls the **allocate( )** method of **ByteBuffer** to allocate a buffer that will hold the contents of the file when it is read. A byte buffer is used because **FileChannel** operates on bytes. A reference to this buffer is stored in **mBuf**. The contents of the file are then read, one buffer at a time, into **mBuf** through a call to **read( )**. The number of bytes read is stored in **count**. Next, the buffer is rewound through a call to **rewind( )**. This call is necessary because the current position is at the end of the buffer after the call to **read( )**, and it must be reset to the start of the buffer in order for the bytes in **mBuf** to be read by calling **get( )**. When the end of the file has been reached, the value returned by **read( )** will be –1. When this occurs, the program ends, explicitly closing the channel and the file.

Another way to read a file is to map it to a buffer. As explained earlier, a principal advantage to this approach is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file using

pre-JDK 7 NIO, first open the file using **FileInputStream**. Next, obtain a channel to that file by calling **getChannel( )** on the file object. Then, map the channel to a buffer by calling **map( )** on the **FileChannel** object. The **map( )** method works as described earlier.

The following program reworks the preceding example so that it uses only pre-JDK 7 features to create a mapped file:

```
// Use a mapped file to read a file. Pre-JDK 7 version.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelRead {
  public static void main(String args[]) {
    FileInputStream fIn = null;
    FileChannel fChan = null;
    long fSize;
    MappedByteBuffer mBuf;

    try {
      // First, open a file for input.
      fIn = new FileInputStream("test.txt");

      // Next, obtain a channel to that file.
      fChan = fIn.getChannel();

      // Get the size of the file.
      fSize = fChan.size();

      // Now, map the file into a buffer.
      mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Read and display bytes from buffer.
      for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());

    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fIn != null) fIn.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

In the program, the file is opened by using the **FileInputStream** constructor, and a reference to that object is assigned to **fIn**. A channel connected to the file is obtained by calling **getChannel( )** on **fIn**. Next, the size of the file is obtained. Then, the entire file is mapped into memory by calling **map( )**, and a reference to the buffer is stored in **mBuf**. The bytes in **mBuf** are read by calling **get( )**.

## Write to a File, Pre-JDK 7

This section reworks the two channel-based file output examples shown earlier so that they use only pre-JDK 7 features. The first example writes to a file by manually allocating a buffer and then performing an explicit output operation. The second example uses a mapped file, which automates the process. In both cases, neither **Path** nor **try**-with-resources is used. This is because neither were part of Java until JDK 7.

When using a pre-JDK 7 version of Java to write a file using a channel and a manually allocated buffer, first open the file for output. This is done by creating a **FileOutputStream**, as described in Chapter 20. Next, obtain a channel to the file by calling **getChannel( )** and then allocate a byte buffer by calling **allocate( )**, as described in the previous section. Next, put the data you want to write into that buffer, and then call **write( )** on the channel. The following program demonstrates this procedure. It writes the alphabet to a file called **test.txt**.

```
// Write to a file using NIO. Pre-JDK 7 Version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class ExplicitChannelWrite {
  public static void main(String args[]) {
    FileOutputStream fOut = null;
    FileChannel fChan = null;
    ByteBuffer mBuf;

    try {
      // First, open the output file.
      fOut = new FileOutputStream("test.txt");

      // Next, get a channel to the output file.
      fChan = fOut.getChannel();

      // Create a buffer.
      mBuf = ByteBuffer.allocate(26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

      // Rewind the buffer so that it can be written.
      mBuf.rewind();

      // Write the buffer to the output file.
      fChan.write(mBuf);
```

```
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fOut != null) fOut.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

The call to **rewind( )** on **mBuf** is necessary in order to reset the current position to zero after data has been written to **mBuf**. Remember, each call to **put( )** advances the current position. Therefore, it is necessary for the current position to be reset to the start of the buffer before calling **write( )**. If this is not done, **write( )** will think that there is no data in the buffer.

When using a pre-JDK 7 version of Java to write to a file using a mapped file, follow these steps. First, open the file for read/write operations by creating a **RandomAccessFile** object. This is necessary to enable the file to be both read from and written to. Next, map that file to a buffer by calling **map( )** on that object. Then, write to the buffer. Because the buffer is mapped to the file, any changes to that buffer are automatically reflected in the file. Thus, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used:

```
// Write to a mapped file. Pre JDK 7 version.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MappedChannelWrite {
  public static void main(String args[]) {
    RandomAccessFile fOut = null;
    FileChannel fChan = null;
    ByteBuffer mBuf;

    try {
      fOut = new RandomAccessFile("test.txt", "rw");

      // Next, obtain a channel to that file.
      fChan = fOut.getChannel();

      // Then, map the file into a buffer.
      mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);
```

```
      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    } finally {
      try {
        if(fChan != null) fChan.close(); // close channel
      } catch(IOException e) {
        System.out.println("Error Closing Channel.");
      }
      try {
        if(fOut != null) fOut.close(); // close file
      } catch(IOException e) {
        System.out.println("Error Closing File.");
      }
    }
  }
}
```

As you can see, there are no explicit write operations to the channel itself. Because **mBuf** is mapped to the file, changes to **mBuf** are automatically reflected in the underlying file.

This page has been intentionally left blank

CHAPTER

# 22 Networking

As all readers know, Java is practically a synonym for Internet programming. There are a number of reasons for this, not the least of which is its ability to generate secure, cross-platform, portable code. However, one of the most important reasons that Java is the premier language for network programming are the classes defined in the **java.net** package. They provide an easy-to-use means by which programmers of all skill levels can access network resources.

This chapter explores the **java.net** package. It is important to emphasize that networking is a very large and at times complicated topic. It is not possible for this book to discuss all of the capabilities contained in **java.net.** Instead, this chapter focuses on several of its core classes and interfaces.

## Networking Basics

Before we begin, it will be useful to review some key networking concepts and terms. At the core of Java's networking support is the concept of a *socket.* A socket identifies an endpoint in a network. The socket paradigm was part of the 4.2BSD Berkeley UNIX release in the early 1980s. Because of this, the term *Berkeley socket* is also used. Sockets are at the foundation of modern networking because a socket allows a single computer to serve many different clients at once, as well as to serve many different types of information. This is accomplished through the use of a *port,* which is a numbered socket on a particular machine. A server process is said to "listen" to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

Socket communication takes place via a protocol. *Internet Protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination. *Transmission Control Protocol* (TCP) is a higher-level protocol that manages to robustly string together these packets, sorting and retransmitting them as necessary to reliably transmit data. A third protocol, *User Datagram Protocol (UDP),* sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

Once a connection has been established, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower 1,024 ports for specific protocols. Many of these will seem familiar to you if you have spent any time surfing the Internet. Port number 21 is for FTP; 23 is for Telnet; 25 is for e-mail; 43 is for whois; 80 is for HTTP; 119 is for netnews—and the list goes on. It is up to each protocol to determine how a client should interact with the port.

For example, HTTP is the protocol that web browsers and servers use to transfer hypertext pages and images. It is a quite simple protocol for a basic page-browsing web server. Here's how it works. When a client requests a file from an HTTP server, an action known as a *hit,* it simply sends the name of the file in a special format to a predefined port and reads back the contents of the file. The server also responds with a status code to tell the client whether or not the request can be fulfilled and why.

A key component of the Internet is the *address*. Every computer on the Internet has one. An Internet address is a number that uniquely identifies each computer on the Net. Originally, all Internet addresses consisted of 32-bit values, organized as four 8-bit values. This address type was specified by IPv4 (Internet Protocol, version 4). However, a new addressing scheme, called IPv6 (Internet Protocol, version 6) has come into play. IPv6 uses a 128-bit value to represent an address, organized into eight 16-bit chunks. Although there are several reasons for and advantages to IPv6, the main one is that it supports a much larger address space than does IPv4. Fortunately, when using Java, you won't normally need to worry about whether IPv4 or IPv6 addresses are used because Java handles the details for you.

Just as the numbers of an IP address describe a network hierarchy, the name of an Internet address, called its *domain name,* describes a machine's location in a name space. For example, **www.HerbSchildt.com** is in the *COM* top-level domain (reserved for U.S. commercial sites); it is called *HerbSchildt*, and *www* identifies the server for web requests. An Internet domain name is mapped to an IP address by the *Domain Naming Service (DNS).* This enables users to work with domain names, but the Internet operates on IP addresses.

## The Networking Classes and Interfaces

Java supports TCP/IP both by extending the already established stream I/O interface introduced in Chapter 20 and by adding the features required to build I/O objects across the network. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model. The classes contained in the **java.net** package are shown here:

| | | |
|---|---|---|
| Authenticator | InetAddress | SocketAddress |
| CacheRequest | InetSocketAddress | SocketImpl |
| CacheResponse | InterfaceAddress | SocketPermission |
| ContentHandler | JarURLConnection | StandardSocketOption |
| CookieHandler | MulticastSocket | URI |
| CookieManager | NetPermission | URL |
| DatagramPacket | NetworkInterface | URLClassLoader |

| | | |
|---|---|---|
| DatagramSocket | PasswordAuthentication | URLConnection |
| DatagramSocketImpl | Proxy | URLDecoder |
| HttpCookie | ProxySelector | URLEncoder |
| HttpURLConnection | ResponseCache | URLPermission (Added by JDK 8.) |
| IDN | SecureCacheResponse | URLStreamHandler |
| Inet4Address | ServerSocket | |
| Inet6Address | Socket | |

The **java.net** package's interfaces are listed here:

| | | |
|---|---|---|
| ContentHandlerFactory | FileNameMap | SocketOptions |
| CookiePolicy | ProtocolFamily | URLStreamHandlerFactory |
| CookieStore | SocketImplFactory | |
| DatagramSocketImplFactory | SocketOption | |

In the sections that follow, we will examine the main networking classes and show several examples that apply to them. Once you understand these core networking classes, you will be able to easily explore the others on your own.

# InetAddress

The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.

## Factory Methods

The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods. *Factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used **InetAddress** factory methods are shown here:

```
static InetAddress getLocalHost( )
  throws UnknownHostException

static InetAddress getByName(String hostName)
  throws UnknownHostException

static InetAddress[ ] getAllByName(String hostName)
  throws UnknownHostException
```

The **getLocalHost( )** method simply returns the **InetAddress** object that represents the local host. The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException**.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The **getAllByName( )** factory method returns an array of **InetAddress**es that represent all of the addresses that a particular name resolves to. It will also throw an **UnknownHostException** if it can't resolve the name to at least one address.

**InetAddress** also includes the factory method **getByAddress( ),** which takes an IP address and returns an **InetAddress** object. Either an IPv4 or an IPv6 address can be used.

The following example prints the addresses and names of the local machine and two Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
  public static void main(String args[]) throws UnknownHostException {
    InetAddress Address = InetAddress.getLocalHost();
    System.out.println(Address);

    Address = InetAddress.getByName("www.HerbSchildt.com");
    System.out.println(Address);

    InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

Here is the output produced by this program. (Of course, the output you see may be slightly different.)

```
default/166.203.115.212
www.HerbSchildt.com/216.92.65.4
www.nba.com/216.66.31.161
www.nba.com/216.66.31.179
```

## Instance Methods

The **InetAddress** class has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the more commonly used methods:

| | |
|---|---|
| boolean equals(Object *other*) | Returns **true** if this object has the same Internet address as *other*. |
| byte[ ] getAddress( ) | Returns a byte array that represents the object's IP address in network byte order. |
| String getHostAddress( ) | Returns a string that represents the host address associated with the **InetAddress** object. |

| | |
|---|---|
| String getHostName( ) | Returns a string that represents the host name associated with the **InetAddress** object. |
| boolean isMulticastAddress( ) | Returns **true** if this address is a multicast address. Otherwise, it returns **false.** |
| String toString( ) | Returns a string that lists the host name and the IP address for convenience. |

Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping automatically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn't have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server. This process might take a long time, so it is wise to structure your code so that you cache IP address information locally rather than look it up repeatedly.



## Inet4Address and Inet6Address

Java includes support for both IPv4 and IPv6 addresses. Because of this, two subclasses of **InetAddress** were created: **Inet4Address** and **Inet6Address**. **Inet4Address** represents a traditional-style IPv4 address. **Inet6Address** encapsulates a newer IPv6 address. Because they are subclasses of **InetAddress,** an **InetAddress** reference can refer to either. This is one way that Java was able to add IPv6 functionality without breaking existing code or adding many more classes. For the most part, you can simply use **InetAddress** when working with IP addresses because it can accommodate both styles.

## TCP/IP Client Sockets

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

**NOTE** As a general rule, applets may only establish socket connections back to the host from which the applet was downloaded. This restriction exists because it would be dangerous for applets loaded through a firewall to have access to any arbitrary machine.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, **ServerSocket** is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges. Because client sockets are the most commonly used by Java applications, they are examined here.

The creation of a **Socket** object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

| | |
|---|---|
| Socket(String *hostName,* int *port*) throws UnknownHostException, IOException | Creates a socket connected to the named host and port. |
| Socket(InetAddress *ipAddress,* int *port*) throws IOException | Creates a socket using a preexisting **InetAddress** object and a port. |

**Socket** defines several instance methods. For example, a **Socket** can be examined at any time for the address and port information associated with it, by use of the following methods:

| | |
|---|---|
| InetAddress getInetAddress( ) | Returns the **InetAddress** associated with the **Socket** object. It returns **null** if the socket is not connected. |
| int getPort( ) | Returns the remote port to which the invoking **Socket** object is connected. It returns 0 if the socket is not connected. |
| int getLocalPort( ) | Returns the local port to which the invoking **Socket** object is bound. It returns –1 if the socket is not bound. |

You can gain access to the input and output streams associated with a **Socket** by use of the **getInputStream( )** and **getOuptutStream( )** methods, as shown here. Each can throw an **IOException** if the socket has been invalidated by a loss of connection. These streams are used exactly like the I/O streams described in Chapter 20 to send and receive data.

| | |
|---|---|
| InputStream getInputStream( ) throws IOException | Returns the **InputStream** associated with the invoking socket. |
| OutputStream getOutputStream( ) throws IOException | Returns the **OutputStream** associated with the invoking socket. |

Several other methods are available, including **connect( ),** which allows you to specify a new connection; **isConnected( ),** which returns true if the socket is connected to a server; **isBound( ),** which returns true if the socket is bound to an address; and **isClosed( ),** which returns true if the socket is closed. To close a socket, call **close( )**. Closing a socket also closes the I/O streams associated with the socket. Beginning with JDK 7, **Socket** also implements **AutoCloseable**, which means that you can use a **try**-with-resources block to manage a socket.

The following program provides a simple **Socket** example. It opens a connection to a "whois" port (port 43) on the InterNIC server, sends the command-line argument down the socket, and then prints the data that is returned. InterNIC will try to look up the argument as a registered Internet domain name, and then send back the IP address and contact information for that site.

```
// Demonstrate Sockets.
import java.net.*;
import java.io.*;

class Whois {
  public static void main(String args[]) throws Exception {
    int c;

    // Create a socket connected to internic.net, port 43.
    Socket s = new Socket("whois.internic.net", 43);

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.

    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
    s.close();
  }
}
```



If, for example, you obtained information about **MHProfessional.com**, you'd get something similar to the following:

```
Whois Server Version 2.0

Domain names in the .com and .net domains can now be registered
with many different competing registrars. Go to http://www.internic.net
for detailed information.

   Domain Name: MHPROFESSIONAL.COM
   Registrar: CSC CORPORATE DOMAINS, INC.
   Whois Server: whois.corporatedomains.com
   Referral URL: http://www.cscglobal.com
   Name Server: NS1.MHEDU.COM
   Name Server: NS2.MHEDU.COM
.
.
.
```

Here is how the program works. First, a **Socket** is constructed that specifies the host name "whois.internic.net" and the port number 43. **Internic.net** is the InterNIC web site that handles whois requests. Port 43 is the whois port. Next, both input and output streams are opened on the socket. Then, a string is constructed that contains the name of the web site you want to obtain information about. In this case, if no web site is specified on the command line, then "MHProfessional.com" is used. The string is converted into a **byte** array and then sent out of the socket. The response is read by inputting from the socket, and the results are displayed. Finally, the socket is closed, which also closes the I/O streams.

In the preceding example, the socket was closed manually by calling **close( )**. If you are using JDK 7 or later, then you can use a **try**-with-resources block to automatically close the socket. For example, here is another way to write the **main( )** method of the previous program:

```
// Use try-with-resources to close a socket.
public static void main(String args[]) throws Exception {
  int c;

  // Create a socket connected to internic.net, port 43. Manage this
  // socket with a try-with-resources block.
  try ( Socket s = new Socket("whois.internic.net", 43) ) {

    // Obtain input and output streams.
    InputStream in = s.getInputStream();
    OutputStream out = s.getOutputStream();

    // Construct a request string.
    String str = (args.length == 0 ? "MHProfessional.com" : args[0]) + "\n";
    // Convert to bytes.
    byte buf[] = str.getBytes();

    // Send request.
    out.write(buf);

    // Read and display response.
    while ((c = in.read()) != -1) {
      System.out.print((char) c);
    }
  }
  // The socket is now closed.
}
```

In this version, the socket is automatically closed when the **try** block ends.

So the examples will work with earlier versions of Java and to clearly illustrate when a network resource can be closed, subsequent examples will continue to call **close( )** explicitly. However, in your own code, you should consider using automatic resource management since it offers a more streamlined approach. One other point: In this version, exceptions are still thrown out of **main( )**, but they could be handled by adding **catch** clauses to the end of the **try**-with-resources block.

**NOTE** For simplicity, the examples in this chapter simply throw all exceptions out of **main( )**. This allows the logic of the network code to be clearly illustrated. However, in real-world code, you will normally need to handle the exceptions in an appropriate way.

## URL

The preceding example was rather obscure because the modern Internet is not about the older protocols such as whois, finger, and FTP. It is about WWW, the World Wide Web. The Web is a loose collection of higher-level protocols and file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scalable way to locate all of the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. Within Java's network class library, the **URL** class provides a simple, concise API to access information across the Internet using URLs.

All URLs share the same basic format, although some variation is allowed. Here are two examples: **http://www.MHProfessional.com/** and **http://www.MHProfessional.com:80/index.htm**. A URL specification is based on four components. The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are HTTP, FTP, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the "http://" from your URL specification). The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:). The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus, ":80" is redundant.) The fourth part is the actual file path. Most HTTP servers will append a file named **index.html** or **index.htm** to URLs that refer directly to a directory resource. Thus, **http://www.MHProfessional.com/** is the same as **http://www.MHProfessional.com/index.htm**.

Java's **URL** class has several constructors; each can throw a **MalformedURLException**. One commonly used form specifies the URL with a string that is identical to what you see displayed in a browser:

URL(String *urlSpecifier*) throws MalformedURLException

The next two forms of the constructor allow you to break up the URL into its component parts:

URL(String *protocolName,* String *hostName,* int *port,* String *path* )
  throws MalformedURLException

URL(String *protocolName,* String *hostName,* String *path*)
  throws MalformedURLException

Another frequently used constructor allows you to use an existing URL as a reference context and then create a new URL from that context. Although this sounds a little contorted, it's really quite easy and useful.

URL(URL *urlObj,* String *urlSpecifier*) throws MalformedURLException

The following example creates a URL to a page on **HerbSchildt.com** and then examines its properties:

```
// Demonstrate URL.
import java.net.*;
class URLDemo {
  public static void main(String args[]) throws MalformedURLException {
    URL hp = new URL(http://www.HerbSchildt.com/WhatsNew");

    System.out.println("Protocol: " + hp.getProtocol());
    System.out.println("Port: " + hp.getPort());

    System.out.println("Host: " + hp.getHost());
    System.out.println("File: " + hp.getFile());
    System.out.println("Ext:" + hp.toExternalForm());
  }
}
```

When you run this, you will get the following output:

```
Protocol: http
Port: -1
Host:  www.HerbSchildt.com
File: /WhatsNew
Ext:http://www.HerbSchildt.com/WhatsNew
```

Notice that the port is –1; this means that a port was not explicitly set. Given a **URL** object, you can retrieve the data associated with it. To access the actual bits or content information of a **URL,** create a **URLConnection** object from it, using its **openConnection( )** method, like this:

```
urlc = url.openConnection()
```

**openConnection( )** has the following general form:

URLConnection openConnection( ) throws IOException

It returns a **URLConnection** object associated with the invoking **URL** object. Notice that it may throw an **IOException**.

## URLConnection

**URLConnection** is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use **URLConnection** to inspect the properties of the remote object before actually transporting it locally. These attributes

are exposed by the HTTP protocol specification and, as such, only make sense for **URL** objects that are using the HTTP protocol.

**URLConnection** defines several methods. Here is a sampling:

| | |
|---|---|
| int getContentLength( ) | Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned. |
| long getContentLengthLong( ) | Returns the size in bytes of the content associated with the resource. If the length is unavailable, –1 is returned. |
| String getContentType( ) | Returns the type of content found in the resource. This is the value of the **content-type** header field. Returns **null** if the content type is not available. |
| long getDate( ) | Returns the time and date of the response represented in terms of milliseconds since January 1, 1970 GMT. |
| long getExpiration( ) | Returns the expiration time and date of the resource represented in terms of milliseconds since January 1, 1970 GMT. Zero is returned if the expiration date is unavailable. |
| String getHeaderField(int *idx*) | Returns the value of the header field at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields. |
| String getHeaderField(String *fieldName*) | Returns the value of header field whose name is specified by *fieldName*. Returns **null** if the specified name is not found. |
| String getHeaderFieldKey(int *idx*) | Returns the header field key at index *idx*. (Header field indexes begin at 0.) Returns **null** if the value of *idx* exceeds the number of fields. |
| Map<String, List<String>> getHeaderFields( ) | Returns a map that contains all of the header fields and values. |
| long getLastModified( ) | Returns the time and date, represented in terms of milliseconds since January 1, 1970 GMT, of the last modification of the resource. Zero is returned if the last-modified date is unavailable. |
| InputStream getInputStream( ) throws IOException | Returns an **InputStream** that is linked to the resource. This stream can be used to obtain the content of the resource. |

Notice that **URLConnection** defines several methods that handle header information. A header consists of pairs of keys and values represented as strings. By using **getHeaderField( )**, you can obtain the value associated with a header key. By calling **getHeaderFields( )**, you can obtain a map that contains all of the headers. Several standard header fields are available directly through methods such as **getDate( )** and **getContentType( )**.

The following example creates a **URLConnection** using the **openConnection( )** method of a **URL** object and then uses it to examine the document's properties and content:

```
// Demonstrate URLConnection.
import java.net.*;
import java.io.*;
import java.util.Date;

class UCDemo
{
  public static void main(String args[]) throws Exception {
    int c;
    URL hp = new URL(http://www.internic.net");
    URLConnection hpCon = hp.openConnection();

    // get date
    long d = hpCon.getDate();
    if(d==0)
      System.out.println("No date information.");
    else
      System.out.println("Date: " + new Date(d));

    // get content type
    System.out.println("Content-Type: " + hpCon.getContentType());

    // get expiration date
    d = hpCon.getExpiration();
    if(d==0)
      System.out.println("No expiration information.");
    else
      System.out.println("Expires: " + new Date(d));

    // get last-modified date
    d = hpCon.getLastModified();
    if(d==0)
      System.out.println("No last-modified information.");
    else
      System.out.println("Last-Modified: " + new Date(d));

    // get content length
    long len = hpCon.getContentLengthLong();
    if(len == -1)
      System.out.println("Content length unavailable.");
    else
      System.out.println("Content-Length: " + len);

    if(len != 0) {
      System.out.println("=== Content ===");
      InputStream input = hpCon.getInputStream();
      while (((c = input.read()) != -1)) {
        System.out.print((char) c);
      }
      input.close();
```

```
    } else {
      System.out.println("No content available.");
    }
  }
}
```

The program establishes an HTTP connection to **www.internic.net** over port 80. It then displays several header values and retrieves the content. You might find it interesting to try this example, observing the results, and then for comparison purposes try a different web site of your own choosing.



## HttpURLConnection

Java provides a subclass of **URLConnection** that provides support for HTTP connections. This class is called **HttpURLConnection.** You obtain an **HttpURLConnection** in the same way just shown, by calling **openConnection( )** on a **URL** object, but you must cast the result to **HttpURLConnection.** (Of course, you must make sure that you are actually opening an HTTP connection.) Once you have obtained a reference to an **HttpURLConnection** object, you can use any of the methods inherited from **URLConnection.** You can also use any of the several methods defined by **HttpURLConnection.** Here is a sampling:

| | |
|---|---|
| static boolean getFollowRedirects( ) | Returns **true** if redirects are automatically followed and **false** otherwise. This feature is on by default. |
| String getRequestMethod( ) | Returns a string representing how URL requests are made. The default is GET. Other options, such as POST, are available. |
| int getResponseCode( )<br>throws IOException | Returns the HTTP response code. –1 is returned if no response code can be obtained. An **IOException** is thrown if the connection fails. |
| String getResponseMessage( )<br>throws IOException | Returns the response message associated with the response code. Returns **null** if no message is available. An **IOException** is thrown if the connection fails. |
| static void setFollowRedirects(boolean *how*) | If *how* is **true,** then redirects are automatically followed. If *how* is **false,** redirects are not automatically followed. By default, redirects are automatically followed. |
| void setRequestMethod(String *how*)<br>throws ProtocolException | Sets the method by which HTTP requests are made to that specified by *how*. The default method is GET, but other options, such as POST, are available. If *how* is invalid, a **ProtocolException** is thrown. |

The following program demonstrates **HttpURLConnection.** It first establishes a connection to **www.google.com**. Then it displays the request method, the response code, and the response message. Finally, it displays the keys and values in the response header.

```
// Demonstrate HttpURLConnection.
import java.net.*;
import java.io.*;
import java.util.*;

class HttpURLDemo
{

public static void main(String args[]) throws Exception {
  URL hp = new URL(http://www.google.com");

  HttpURLConnection hpCon = (HttpURLConnection) hp.openConnection();

  // Display request method.
  System.out.println("Request method is " +
                     hpCon.getRequestMethod());

  // Display response code.
  System.out.println("Response code is " +
                     hpCon.getResponseCode());

  // Display response message.
  System.out.println("Response Message is " +
                     hpCon.getResponseMessage());

  // Get a list of the header fields and a set
  // of the header keys.
  Map<String, List<String>> hdrMap = hpCon.getHeaderFields();
  Set<String> hdrField = hdrMap.keySet();

  System.out.println("\nHere is the header:");

   // Display all header keys and values.
   for(String k : hdrField) {
     System.out.println("Key: " + k +
                        "  Value: " + hdrMap.get(k));
   }
  }
}
```

The output produced by the program is shown here. (Of course, the exact response returned by **www.google.com** will vary over time.)

```
Request method is GET
Response code is 200
Response Message is OK

Here is the header:
Key: Transfer-Encoding  Value: [chunked]
```

```
Key: X-Frame-Options  Value: [SAMEORIGIN]
Key: null  Value: [HTTP/1.1 200 OK]
Key: Server  Value: [gws]
Key: Cache-Control  Value: [private, max-age=0]
Key: Set-Cookie  Value:
[NID=67=rMTQWvn5eVIYA2d8F5Iu_8L-68wiMACyaXYqeSe1bvR8SzQQ_PaDCy5mNbxuw5XtdcjY
KIwmy3oVJM1Y0qZdibBOkQfJmtHpAtO61GVwumQ1ApgSXWjZ67yHxQX3g3-h; expires=Wed,
23-Apr-2014 18:31:09 GMT; path=/; domain=.google.com; HttpOnly,
PREF=ID=463b5df7b9ced9d8:FF=0:TM=1382466669:LM=1382466669:S=3LI-oT-Dzi46U1On
; expires=Thu, 22-Oct-2015 18:31:09 GMT; path=/; domain=.google.com]
Key: Expires  Value: [-1]
Key: X-XSS-Protection  Value: [1; mode=block]
Key: P3P  Value: [CP="This is not a P3P policy! See
http://www.google.com/support/accounts/bin/answer.py?hl=en&answer=151657 for
more info."]
Key: Date  Value: [Tue, 22 Oct 2013 18:31:09 GMT]
Key: Content-Type  Value: [text/html;charset=ISO-8859-1]
```

Notice how the header keys and values are displayed. First, a map of the header keys and values is obtained by calling **getHeaderFields( )** (which is inherited from **URLConnection**). Next, a set of the header keys is retrieved by calling **keySet( )** on the map. Then, the key set is cycled through by using a for-each style **for** loop. The value associated with each key is obtained by calling **get( )** on the map.

## The URI Class

The **URI** class encapsulates a *Uniform Resource Identifier (URI)*. URIs are similar to URLs. In fact, URLs constitute a subset of URIs. A URI represents a standard way to identify a resource. A URL also describes how to access the resource.

## Cookies

The **java.net** package includes classes and interfaces that help manage cookies and can be used to create a stateful (as opposed to stateless) HTTP session. The classes are **CookieHandler, CookieManager,** and **HttpCookie.** The interfaces are **CookiePolicy** and **CookieStore.** The creation of a stateful HTTP session is beyond the scope of this book.

---

**NOTE** For information about using cookies with servlets, see Chapter 38.

## TCP/IP Server Sockets

As mentioned earlier, Java has a different socket class that must be used for creating server applications. The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports. **ServerSocket**s are quite different from normal **Socket**s. When you create a **ServerSocket,** it will register itself with the system as having an interest in client connections. The constructors for **ServerSocket** reflect the port number that you want to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default

is 50. The constructors might throw an **IOException** under adverse conditions. Here are three of its constructors:

| | |
|---|---|
| ServerSocket(int *port*) throws IOException | Creates server socket on the specified port with a queue length of 50. |
| ServerSocket(int *port,* int *maxQueue*) throws IOException | Creates a server socket on the specified port with a maximum queue length of *maxQueue*. |
| ServerSocket(int *port,* int *maxQueue,* InetAddress *localAddress*) throws IOException | Creates a server socket on the specified port with a maximum queue length of *maxQueue*. On a multihomed host, *localAddress* specifies the IP address to which this socket binds. |

**ServerSocket** has a method called **accept( ),** which is a blocking call that will wait for a client to initiate communications and then return with a normal **Socket** that is then used for communication with the client.

# Datagrams

TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

*Datagrams* are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s. Each is examined here.

## DatagramSocket

**DatagramSocket** defines four public constructors. They are shown here:

DatagramSocket( ) throws SocketException

DatagramSocket(int *port*) throws SocketException

DatagramSocket(int *port,* InetAddress *ipAddress*) throws SocketException

DatagramSocket(SocketAddress *address*) throws SocketException

The first creates a **DatagramSocket** bound to any unused port on the local computer. The second creates a **DatagramSocket** bound to the port specified by *port*. The third constructs a **DatagramSocket** bound to the specified port and **InetAddress**. The fourth constructs a **DatagramSocket** bound to the specified **SocketAddress**. **SocketAddress** is an abstract

class that is implemented by the concrete class **InetSocketAddress**. **InetSocketAddress** encapsulates an IP address with a port number. All can throw a **SocketException** if an error occurs while creating the socket.

**DatagramSocket** defines many methods. Two of the most important are **send( )** and **receive( )**, which are shown here:

void send(DatagramPacket *packet*) throws IOException

void receive(DatagramPacket *packet*) throws IOException

The **send( )** method sends a packet to the port specified by *packet*. The **receive( )** method waits for a packet to be received and returns the result.

**DatagramSocket** also defines the **close( )**method, which closes the socket. Beginning with JDK 7, **DatagramSocket** implements **AutoCloseable**, which means that a **DatagramSocket** can be managed by a **try**-with-resources block.

Other methods give you access to various attributes associated with a **DatagramSocket**. Here is a sampling:

| | |
|---|---|
| InetAddress getInetAddress( ) | If the socket is connected, then the address is returned. Otherwise, **null** is returned. |
| int getLocalPort( ) | Returns the number of the local port. |
| int getPort( ) | Returns the number of the port to which the socket is connected. It returns –1 if the socket is not connected to a port. |
| boolean isBound( ) | Returns **true** if the socket is bound to an address. Returns **false** otherwise. |
| boolean isConnected( ) | Returns **true** if the socket is connected to a server. Returns **false** otherwise. |
| void setSoTimeout(int *millis*) throws SocketException | Sets the time-out period to the number of milliseconds passed in *millis*. |

## DatagramPacket

**DatagramPacket** defines several constructors. Four are shown here:

DatagramPacket(byte *data* [ ], int *size*)
DatagramPacket(byte *data* [ ], int *offset,* int *size*)
DatagramPacket(byte *data* [ ], int *size,* InetAddress *ipAddress,* int *port*)
DatagramPacket(byte *data* [ ], int *offset,* int *size,* InetAddress *ipAddress,* int *port*)

The first constructor specifies a buffer that will receive data and the size of a packet. It is used for receiving data over a **DatagramSocket.** The second form allows you to specify an offset into the buffer at which data will be stored. The third form specifies a target address and port, which are used by a **DatagramSocket** to determine where the data in the packet will be sent. The fourth form transmits packets beginning at the specified offset into the data. Think of the first two forms as building an "in box," and the second two forms as stuffing and addressing an envelope.

**DatagramPacket** defines several methods, including those shown here, that give access to the address and port number of a packet, as well as the raw data and its length.

| | |
|---|---|
| InetAddress getAddress( ) | Returns the address of the source (for datagrams being received) or destination (for datagrams being sent). |
| byte[ ] getData( ) | Returns the byte array of data contained in the datagram. Mostly used to retrieve data from the datagram after it has been received. |
| int getLength( ) | Returns the length of the valid data contained in the byte array that would be returned from the **getData( )** method. This may not equal the length of the whole byte array. |
| int getOffset( ) | Returns the starting index of the data. |
| int getPort( ) | Returns the port number. |
| void setAddress(InetAddress *ipAddress*) | Sets the address to which a packet will be sent. The address is specified by *ipAddress*. |
| void setData(byte[ ] *data*) | Sets the data to *data*, the offset to zero, and the length to number of bytes in *data*. |
| void setData(byte[ ] *data*, int *idx*, int *size*) | Sets the data to *data*, the offset to *idx*, and the length to *size*. |
| void setLength(int *size*) | Sets the length of the packet to *size*. |
| void setPort(int *port*) | Sets the port to *port*. |

## A Datagram Example

The following example implements a very simple networked communications client and server. Messages are typed into the window at the server and written across the network to the client side, where they are displayed.

```
// Demonstrate datagrams.
import java.net.*;

class WriteServer {
  public static int serverPort = 998;
  public static int clientPort = 999;
  public static int buffer_size = 1024;
  public static DatagramSocket ds;
  public static byte buffer[] = new byte[buffer_size];

  public static void TheServer() throws Exception {
    int pos=0;
    while (true) {
      int c = System.in.read();
      switch (c) {
        case -1:
          System.out.println("Server Quits.");
```

```
          ds.close();
          return;
        case '\r':
          break;
        case '\n':
          ds.send(new DatagramPacket(buffer,pos,
             InetAddress.getLocalHost(),clientPort));
          pos=0;
          break;
        default:
          buffer[pos++] = (byte) c;
      }
    }
  }

  public static void TheClient() throws Exception {
    while(true) {
      DatagramPacket p = new DatagramPacket(buffer, buffer.length);
      ds.receive(p);
      System.out.println(new String(p.getData(), 0, p.getLength()));
    }
  }

  public static void main(String args[]) throws Exception {
    if(args.length == 1) {
      ds = new DatagramSocket(serverPort);
      TheServer();
    } else {
      ds = new DatagramSocket(clientPort);
      TheClient();
    }
  }
}
```

This sample program is restricted by the **DatagramSocket** constructor to running between two ports on the local machine. To use the program, run

```
java WriteServer
```

in one window; this will be the client. Then run

```
java WriteServer 1
```

This will be the server. Anything that is typed in the server window will be sent to the client window after a newline is received.

---

**NOTE** The use of datagrams may not be allowed on your computer. (For example, a firewall may prevent their use.) If this is the case, the preceding example cannot be used. Also, the port numbers used in the program work on the author's system, but may have to be adjusted for your environment.

This page has been intentionally left blank

CHAPTER

# 23 The Applet Class

This chapter examines the **Applet** class, which provides the foundation for applets. The **Applet** class is contained in the **java.applet** package. **Applet** contains several methods that give you detailed control over the execution of your applet. In addition, **java.applet** also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

## Two Types of Applets

It is important to state at the outset that there are two varieties of applets based on **Applet**. The first are those based directly on the **Applet** class described in this chapter. These applets use the Abstract Window Toolkit (AWT) to provide the graphical user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class **JApplet**, which inherits **Applet**. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. However, traditional AWT-based applets are still used, especially when only a very simple user interface is required. Thus, both AWT- and Swing-based applets are valid.

This chapter describes AWT-based applets. However, because **JApplet** inherits **Applet**, all the features of **Applet** are also available in **JApplet**, and much of the information in this chapter applies to both types of applets. Therefore, even if you are interested in only Swing applets, the information in this chapter is still relevant and necessary. Understand, however, that when creating Swing-based applets, some additional constraints apply and these are described later in this book, when Swing is covered.

**NOTE** For information on building applets when using Swing, see Chapter 31.

## Applet Basics

Chapter 13 introduced the general form of an applet and the steps necessary to compile and run one. Let's begin by reviewing this information.

AWT-based applets are subclasses of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer. The illustrations shown in this chapter were created with the standard applet viewer, called **appletviewer**, provided by the JDK.

Execution of an applet does not begin at **main( )**. Actually, few applets even have **main( )** methods. Instead, execution of an applet is started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by **System.out.println( )**. Rather, in an AWT-based applet, output is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location. Input is also handled differently than in a console application.

Before an applet can be used, a deployment strategy must be chosen. There are two basic approaches. The first is to use the Java Network Launch Protocol (JNLP). This approach offers the most flexibility, especially as it relates to rich Internet applications. For real-world applets that you create, JNLP will often be the best choice. However, a detailed discussion of JNLP is beyond the scope of this book. (See the JDK documentation for the latest details on JNLP.) Fortunately, JNLP is not required for the example applets shown here.

The second basic approach to deploying an applet is to specify the applet directly in an HTML file, without the use of JNLP. This is the original way that applets were launched when Java was created, and it is still used today—especially for simple applets. Furthermore, because of its inherent simplicity, it is the appropriate method for the applet examples described in this book. At the time of this writing, Oracle recommends the APPLET tag for this purpose. Therefore, the APPLET tag is used in this book. (Be aware that the APPLET tag is currently deprecated by the HTML specification. The alternative is the OBJECT tag. You should check the JDK documentation in this regard for the latest recommendations.) When an APPLET tag is encountered in the HTML file, the specified applet will be executed by a Java-enabled web browser.

The use of the APPLET tag offers a secondary advantage when developing applets because it enables you to easily view and test the applet. To do so, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is an example of such a comment:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called **MyApplet** in a window that is 200 pixels wide and 60 pixels high. Because the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

---

**NOTE** As noted in Chapter 13, beginning with the release of Java 7, update 21, Java applets must be signed to prevent security warnings when run in a browser. In fact, in some cases, the applet may be prevented from running. Applets stored in the local file system, such as you would create when compiling the examples in this book, are especially sensitive to this change. You may need to adjust the security settings in the Java Control Panel to run a local applet in a browser. At the time of this writing, Oracle recommends against the use of local applets, recommending instead that applets be executed through a web server. Furthermore, unsigned local applets may be blocked from execution in the future. In general, for applets that will be distributed via the Internet, such as commercial

applications, signing is a virtual necessity. The concepts and techniques required to sign applets (and other types of Java programs) are beyond the scope of this book. However, extensive information is found on Oracle's website. Finally, as mentioned, the easiest way to try the applet examples is to use **appletviewer**.

## The Applet Class

The **Applet** class defines the methods shown in Table 23-1. **Applet** provides all necessary support for applet execution, such as starting and stopping. It also provides methods that load and display images, and methods that load and play audio clips. **Applet** extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities. (An overview of the AWT is presented in subsequent chapters.)



| Method | Description |
|---|---|
| void destroy( ) | Called by the browser just before an applet is terminated. Your applet will override this method if it needs to perform any cleanup prior to its destruction. |
| AccessibleContext getAccessibleContext( ) | Returns the accessibility context for the invoking object. |
| AppletContext getAppletContext( ) | Returns the context associated with the applet. |
| String getAppletInfo( ) | Overrides of this method should return a string that describes the applet. The default implementation returns **null**. |
| AudioClip getAudioClip(URL *url*) | Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. |
| AudioClip getAudioClip(URL *url*, String *clipName*) | Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url* and having the name specified by *clipName*. |
| URL getCodeBase( ) | Returns the URL associated with the invoking applet. |
| URL getDocumentBase( ) | Returns the URL of the HTML document that invokes the applet. |
| Image getImage(URL *url*) | Returns an **Image** object that encapsulates the image found at the location specified by *url*. |
| Image getImage(URL *url*, String *imageName*) | Returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*. |
| Locale getLocale( ) | Returns a **Locale** object that is used by various locale-sensitive classes and methods. |
| String getParameter(String *paramName*) | Returns the parameter associated with *paramName*. **null** is returned if the specified parameter is not found. |

**Table 23-1** The Methods Defined by **Applet**

| Method | Description |
|---|---|
| String[ ] [ ] getParameterInfo( ) | Overrides of this method should return a **String** table that describes the parameters recognized by the applet. Each entry in the table must consist of three strings that contain the name of the parameter, a description of its type and/or range, and an explanation of its purpose. The default implementation returns **null**. |
| void init( ) | Called when an applet begins execution. It is the first method called for any applet. |
| boolean isActive( ) | Returns **true** if the applet has been started. It returns **false** if the applet has been stopped. |
| boolean isValidateRoot( ) | Returns **true**, which indicates that an applet is a validate root. |
| static final AudioClip newAudioClip(URL *url*) | Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. This method is similar to **getAudioClip( )** except that it is static and can be executed without the need for an **Applet** object. |
| void play(URL *url*) | If an audio clip is found at the location specified by *url*, the clip is played. |
| void play(URL *url*, String *clipName*) | If an audio clip is found at the location specified by *url* with the name specified by *clipName*, the clip is played. |
| void resize(Dimension *dim*) | Resizes the applet according to the dimensions specified by *dim*. **Dimension** is a class stored inside **java.awt**. It contains two integer fields: **width** and **height**. |
| void resize(int *width*, int *height*) | Resizes the applet according to the dimensions specified by *width* and *height*. |
| final void setStub(AppletStub *stubObj*) | Makes *stubObj* the stub for the applet. This method is used by the run-time system and is not usually called by your applet. A *stub* is a small piece of code that provides the linkage between your applet and the browser. |
| void showStatus(String *str*) | Displays *str* in the status window of the browser or applet viewer. If the browser does not support a status window, then no action takes place. |
| void start( ) | Called by the browser when an applet should start (or resume) execution. It is automatically called after **init( )** when an applet first begins. |
| void stop( ) | Called by the browser to suspend execution of the applet. Once stopped, an applet is restarted when the browser calls **start( )**. |

**Table 23-1** The Methods Defined by **Applet** *(continued)*

## Applet Architecture

As a general rule, an applet is a GUI-based program. As such, its architecture is different from the console-based programs shown in the first part of this book. If you are already familiar with GUI programming, you will be right at home writing applets. If not, then there are a few key concepts you must understand.

First, applets are event driven. Although we won't examine event handling until the following chapter, it is important to understand in a general way how the event-driven architecture impacts the design of an applet. An applet resembles a set of interrupt service routines. Here is how the process works. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return. This is a crucial point. For the most part, your applet should not enter a "mode" of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the run-time system. In those situations in which your applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), you must start an additional thread of execution. (You will see an example later in this chapter.)

Second, the user initiates interaction with an applet—not the other way around. As you know, in a console-based program, when the program needs input, it will prompt the user and then call some input method, such as **readLine( )**. This is not the way it works in an applet. Instead, the user interacts with the applet as he or she wants, when he or she wants. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. As you will see in later chapters, applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

While the architecture of an applet is not as easy to understand as that of a console-based program, Java makes it as simple as possible. If you have written programs for Windows (or other GUI-based operating systems), you know how intimidating that environment can be. Fortunately, Java provides a much cleaner approach that is more quickly mastered.

## An Applet Skeleton

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them.

AWT-based applets (such as those discussed in this chapter) will also often override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when

the applet's output must be redisplayed. (Swing-based applets use a different mechanism to accomplish this task.) These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/

public class AppletSkel extends Applet {
  // Called first.
  public void init() {
    // initialization
  }

  /* Called second, after init().  Also called whenever
     the applet is restarted. */
  public void start() {
    // start or resume execution
  }

  // Called when the applet is stopped.
  public void stop() {
    // suspends execution
  }

  /* Called when applet is terminated.  This is the last
     method executed. */
  public void destroy() {
    // perform shutdown activities
  }

  // Called when an applet's window must be restored.
  public void paint(Graphics g) {
    // redisplay contents of window
  }
}
```

Although this skeleton does not do anything, it can be compiled and run. When run, it generates the following empty window when viewed with **appletviewer**. Of course, in this and all subsequent examples, the precise look of the **appletviewer** frame may differ based on your execution environment. To help illustrate this fact, a variety of environments were used to generate the screen captures shown throughout this book.

## Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:

1. **init( )**
2. **start( )**
3. **paint( )**

When an applet is terminated, the following sequence of method calls takes place:

1. **stop( )**
2. **destroy( )**

Let's look more closely at these methods.

### init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

### start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

### paint( )

The **paint( )** method is called each time an AWT-based applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

### stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

### destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

### Overriding update( )

In some situations, an AWT-based applet may need to override another method defined by the AWT, called **update( )**. This method is called when your applet has requested that a portion of its window be redrawn. The default version of **update( )** simply calls **paint( )**. However, you can override the **update( )** method so that it performs more subtle repainting. In general, overriding **update( )** is a specialized technique that is not applicable to all applets, and the examples in this chapter do not override **update( )**.

## Simple Applet Display Methods

As we've mentioned, applets are displayed in a window, and AWT-based applets use the AWT to perform input and output. Although we will examine the methods, procedures, and techniques related to the AWT in subsequent chapters, a few are described here, because we will use them to write sample applets. (Remember, Swing-based applets are described later in this book.)

As described in Chapter 13, to output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class. Typically, it is called from within either **update( )** or **paint( )**. It has the following general form:

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output beginning at *x,y*. In a Java window, the upper-left corner is location 0,0. The **drawString( )** method will not recognize newline characters. If you want to start a line of text on another line, you must do so manually, specifying the precise X,Y location where you want the line to begin. (As you will see in later chapters, there are techniques that make this process easy.)

To set the background color of an applet's window, use **setBackground( )**. To set the foreground color (the color in which text is shown, for example), use **setForeground( )**. These methods are defined by **Component**, and they have the following general forms:

void setBackground(Color *newColor*)
void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class **Color** defines the constants shown here that can be used to specify colors:

| | |
|---|---|
| Color.black | Color.magenta |
| Color.blue | Color.orange |
| Color.cyan | Color.pink |
| Color.darkGray | Color.red |
| Color.gray | Color.white |
| Color.green | Color.yellow |
| Color.lightGray | |

Uppercase versions of the constants are also defined.

The following example sets the background color to green and the text color to red:

```
setBackground(Color.green);
setForeground(Color.red);
```

A good place to set the foreground and background colors is in the **init( )** method. Of course, you can change these colors as often as necessary during the execution of your applet.

You can obtain the current settings for the background and foreground colors by calling **getBackground( )** and **getForeground( )**, respectively. They are also defined by **Component** and are shown here:

Color getBackground( )
Color getForeground( )

Here is a very simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the **init( )**, **start( )**, and **paint( )** methods are called when an applet starts up:

```
/* A simple applet that sets the foreground and
   background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/

public class Sample extends Applet{
  String msg;

  // set the foreground and background colors.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
    msg = "Inside init( ) --";
  }

  // Initialize the string to be displayed.
  public void start() {
    msg += " Inside start( ) --";
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    msg += " Inside paint( ).";
    g.drawString(msg, 10, 30);
  }
}
```

Sample output is shown here:


The methods **stop( )** and **destroy( )** are not overridden, because they are not needed by this simple applet.

## Requesting Repainting

As a general rule, an applet writes to its window only when its **paint( )** method is called by the AWT. This raises an interesting question: How can the applet itself cause its window to be updated when its information changes? For example, if an applet is displaying a moving banner, what mechanism does the applet use to update the window each time this banner scrolls? Remember, one of the fundamental architectural constraints imposed on an applet is that it must quickly return control to the run-time system. It cannot create a loop inside **paint( )** that repeatedly scrolls the banner, for example. This would prevent control from passing back to the AWT. Given this constraint, it may seem that output to your applet's window will be difficult at best. Fortunately, this is not the case. Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.

The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**. Thus, for another part of your applet to output to its window, simply store the output and then call **repaint( )**. The AWT will then execute a call to **paint( )**, which can display the stored information. For example, if part of your applet needs to output a string, it can store this string in a **String** variable and then call **repaint( )**. Inside **paint( )**, you will output the string using **drawString( )**.

The **repaint( )** method has four forms. Let's look at each one, in turn. The simplest version of **repaint( )** is shown here:

```
void repaint( )
```

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

```
void repaint(int left, int top, int width, int height)
```

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top,* and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels. You save time by specifying a region to repaint. Window updates are costly in terms of time. If you need to update only a small portion of the window, it is more efficient to repaint only that region.

Calling **repaint( )** is essentially a request that your applet be repainted sometime soon. However, if your system is slow or busy, **update( )** might not be called immediately. Multiple requests for repainting that occur within a short time can be collapsed by the AWT in a manner such that **update( )** is only called sporadically. This can be a problem in many situations, including animation, in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( )**:

void repaint(long *maxDelay*)
void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called. Beware, though. If the time elapses before **update( )** can be called, it isn't called. There's no return value or exception thrown, so you must be careful.



**NOTE** It is possible for a method other than **paint( )** or **update( )** to output to an applet's window. To do so, it must obtain a graphics context by calling **getGraphics( )** (defined by **Component**) and then use this context to output to the window. However, for most applications, it is better and easier to route window output through **paint( )** and to call **repaint( )** when the contents of the window change.

## A Simple Banner Applet

To demonstrate **repaint( )**, a simple banner applet is developed. This applet scrolls a message, from right to left, across the applet's window. Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initialized. The banner applet is shown here:

```
/* A simple banner applet.

   This applet creates a thread that scrolls
   the message contained in msg right to left
   across the applet's window.
*/
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleBanner" width=300 height=50>
</applet>
*/

public class SimpleBanner extends Applet implements Runnable {
  String msg = " A Simple Moving Banner.";
  Thread t = null;
  int state;
  volatile boolean stopFlag;

  // Set colors and initialize thread.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
  }
```

```
  // Start thread
  public void start() {
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {

    // Redisplay banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        if(stopFlag)
          break;
      } catch(InterruptedException e) {}
    }
  }

  // Pause the banner.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
  }
}
```

Following is sample output:


Let's take a close look at how this applet operates. First, notice that **SimpleBanner** extends **Applet**, as expected, but it also implements **Runnable**. This is necessary, since the

applet will be creating a second thread of execution that will be used to scroll the banner. Inside **init( )**, the foreground and background colors of the applet are set.

After initialization, the run-time system calls **start( )** to start the applet running. Inside **start( )**, a new thread of execution is created and assigned to the **Thread** variable **t**. Then, the **boolean** variable **stopFlag**, which controls the execution of the applet, is set to **false**. Next, the thread is started by a call to **t.start( )**. Remember that **t.start( )** calls a method defined by **Thread**, which causes **run( )** to begin executing. It does not cause a call to the version of **start( )** defined by **Applet**. These are two separate methods.

Inside **run( )**, a call to **repaint( )** is made. This eventually causes the **paint( )** method to be called, and the rotated contents of **msg** are displayed. Between each iteration, **run( )** sleeps for a quarter of a second. The net effect is that the contents of **msg** are scrolled right to left in a constantly moving display. The **stopFlag** variable is checked on each iteration. When it is **true**, the **run( )** method terminates.

If a browser is displaying the applet when a new page is viewed, the **stop( )** method is called, which sets **stopFlag** to **true**, causing **run( )** to terminate. This is the mechanism used to stop the thread when its page is no longer in view. When the applet is brought back into view, **start( )** is once again called, which starts a new thread to execute the banner.



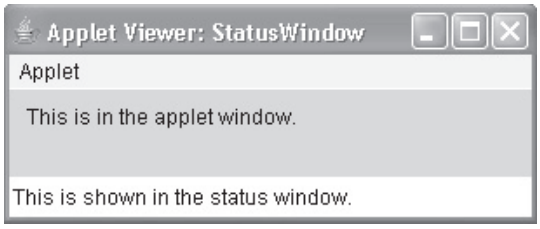
## Using the Status Window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus( )** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

The following applet demonstrates **showStatus( )**:

```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*

<applet code="StatusWindow" width=300 height=50>
</applet>
*/

public class StatusWindow extends Applet {
  public void init() {
    setBackground(Color.cyan);
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    g.drawString("This is in the applet window.", 10, 20);
    showStatus("This is shown in the status window.");
  }
}
```

Sample output from this program is shown here:




## The HTML APPLET Tag

As mentioned earlier, at the time of this writing, Oracle recommends that the APPLET tag be used to manually start an applet when JNLP is not used. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers will allow many applets on a single page. So far, we have been using only a simplified form of the APPLET tag. Now it is time to take a closer look at it.

The syntax for a fuller form of the APPLET tag is shown here. Bracketed items are optional.

```
< APPLET
  [CODEBASE = codebaseURL]
  CODE = appletFile
  [ALT = alternateText]
  [NAME = appletInstanceName]
  WIDTH = pixels HEIGHT = pixels
  [ALIGN = alignment ]
  [VSPACE = pixels] [HSPACE = pixels]
>
[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
. . .
[HTML Displayed in the absence of Java]
</APPLET>
```

Let's take a look at each part now.

**CODEBASE** CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag). The HTML document's URL directory is used as the CODEBASE if this attribute is not specified.

**CODE** CODE is a required attribute that gives the name of the file containing your applet's compiled **.class** file. This file is relative to the code base URL of the applet, which is the directory that the HTML file was in or the directory indicated by CODEBASE if set.

**ALT** The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser recognizes the APPLET tag but can't currently run Java applets. This is distinct from the alternate HTML you provide for browsers that don't support applets.

**NAME** NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet( )**, which is defined by the **AppletContext** interface.

**WIDTH and HEIGHT** WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

**ALIGN** ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

**VSPACE and HSPACE** These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

**PARAM NAME and VALUE** The PARAM tag allows you to specify applet-specific arguments. Applets access their attributes with the **getParameter( )** method.

Other valid APPLET attributes include ARCHIVE, which lets you specify one or more archive files, and OBJECT, which specifies a saved version of the applet. In general, an APPLET tag should include only a CODE or an OBJECT attribute, but not both.

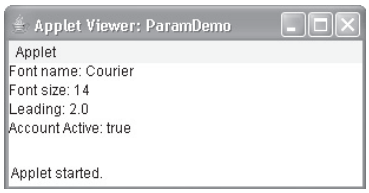
## Passing Parameters to Applets

As just discussed, the APPLET tag allows you to pass parameters to your applet. To retrieve a parameter, use the **getParameter( )** method. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **boolean** values, you will need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/

public class ParamDemo extends Applet {
  String fontName;
```

```
  int fontSize;
  float leading;
  boolean active;

  // Initialize the string to be displayed.
  public void start() {
    String param;

    fontName = getParameter("fontName");
    if(fontName == null)
      fontName = "Not Found";

    param = getParameter("fontSize");
    try {
      if(param != null)
        fontSize = Integer.parseInt(param);
      else
        fontSize = 0;
    } catch(NumberFormatException e) {
      fontSize = -1;
    }

    param = getParameter("leading");
    try {
      if(param != null)
        leading = Float.valueOf(param).floatValue();
      else
        leading = 0;
    } catch(NumberFormatException e) {
      leading = -1;
    }

    param = getParameter("accountEnabled");
    if(param != null)
      active = Boolean.valueOf(param).booleanValue();
  }

  // Display parameters.
  public void paint(Graphics g) {
    g.drawString("Font name: " + fontName, 0, 10);
    g.drawString("Font size: " + fontSize, 0, 26);
    g.drawString("Leading: " + leading, 0, 42);
    g.drawString("Account Active: " + active, 0, 58);
  }
}
```

Sample output from this program is shown here:




As the program shows, you should test the return values from **getParameter( )**. If a parameter isn't available, **getParameter( )** will return **null**. Also, conversions to numeric types must be attempted in a **try** statement that catches **NumberFormatException**. Uncaught exceptions should never occur within an applet.

## Improving the Banner Applet

It is possible to use a parameter to enhance the banner applet shown earlier. In the previous version, the message being scrolled was hard-coded into the applet. However, passing the message as a parameter allows the banner applet to display a different message each time it is executed. This improved version is shown here. Notice that the APPLET tag at the top of the file now specifies a parameter called **message** that is linked to a quoted string.



```
// A parameterized banner
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamBanner" width=300 height=50>
<param name=message value="Java makes the Web move!">
</applet>
*/

public class ParamBanner extends Applet implements Runnable {
  String msg;
  Thread t = null;
  int state;
  volatile boolean stopFlag;

  // Set colors and initialize thread.
  public void init() {
    setBackground(Color.cyan);
    setForeground(Color.red);
  }

  // Start thread
  public void start() {
    msg = getParameter("message");
    if(msg == null) msg = "Message not found.";
    msg = " " + msg;
    t = new Thread(this);
    stopFlag = false;
    t.start();
  }

  // Entry point for the thread that runs the banner.
  public void run() {

    // Redisplay banner
    for( ; ; ) {
      try {
        repaint();
        Thread.sleep(250);
        if(stopFlag)
          break;
      } catch(InterruptedException e) {}
    }
  }
```

```
  // Pause the banner.
  public void stop() {
    stopFlag = true;
    t = null;
  }

  // Display the banner.
  public void paint(Graphics g) {
    char ch;

    ch = msg.charAt(0);
    msg = msg.substring(1, msg.length());
    msg += ch;

    g.drawString(msg, 50, 30);
  }
}
```

## getDocumentBase( ) and getCodeBase( )

Often, you will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet (the *document base*) and the directory from which the applet's class file was loaded (the *code base*). These directories are returned as **URL** objects (described in Chapter 22) by **getDocumentBase( )** and **getCodeBase( )**. They can be concatenated with a string that names the file you want to load. To actually load another file, you will use the **showDocument( )** method defined by the **AppletContext** interface, discussed in the next section.

The following applet illustrates these methods:

```
// Display code and document bases.
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="Bases" width=300 height=50>
</applet>
*/

public class Bases extends Applet {
  // Display code and document bases.
  public void paint(Graphics g) {
    String msg;

    URL url = getCodeBase(); // get code base
    msg = "Code base: " + url.toString();
    g.drawString(msg, 10, 20);

    url = getDocumentBase(); // get document base
    msg = "Document base: " + url.toString();
    g.drawString(msg, 10, 40);
  }
}
```

Sample output from this program is shown here:




## AppletContext and showDocument( )

One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than simple text-based links. To allow your applet to transfer control to another URL, you must use the **showDocument( )** method defined by the **AppletContext** interface. **AppletContext** is an interface that lets you get information from the applet's execution environment. The methods defined by **AppletContext** are shown in Table 23-2. The context of the currently executing applet is obtained by a call to the **getAppletContext( )** method defined by **Applet**.

| Method | Description |
|---|---|
| Applet getApplet(String *appletName*) | Returns the applet specified by *appletName* if it is within the current applet context. Otherwise, **null** is returned. |
| Enumeration<Applet> getApplets( ) | Returns an enumeration that contains all of the applets within the current applet context. |
| AudioClip getAudioClip(URL *url*) | Returns an **AudioClip** object that encapsulates the audio clip found at the location specified by *url*. |
| Image getImage(URL *url*) | Returns an **Image** object that encapsulates the image found at the location specified by *url*. |
| InputStream getStream(String *key*) | Returns the stream linked to *key*. Keys are linked to streams by using the **setStream( )** method. A **null** reference is returned if no stream is linked to *key*. |
| Iterator<String> getStreamKeys( ) | Returns an iterator for the keys associated with the invoking object. The keys are linked to streams. See **getStream( )** and **setStream( )**. |
| void setStream(String *key*, InputStream *strm*) throws IOException | Links the stream specified by *strm* to the key passed in *key*. The *key* is deleted from the invoking object if *strm* is **null**. |
| void showDocument(URL *url*) | Brings the document at the **URL** specified by *url* into view. This method may not be supported by applet viewers. |

**Table 23-2** The Methods Defined by the **AppletContext** Interface

| Method | Description |
|---|---|
| void showDocument(URL *url*, String *where*) | Brings the document at the **URL** specified by *url* into view. This method may not be supported by applet viewers. The placement of the document is specified by *where* as described in the text. |
| void showStatus(String *str*) | Displays *str* in the status window. |

**Table 23-2** The Methods Defined by the **AppletContext** Interface *(continued)*

Within an applet, once you have obtained the applet's context, you can bring another document into view by calling **showDocument( )**. This method has no return value and throws no exception if it fails, so use it carefully. There are two **showDocument( )** methods. The method **showDocument(URL)** displays the document at the specified **URL**. The method **showDocument(URL, String)** displays the specified document at the specified location within the browser window. Valid arguments for *where* are "_self" (show in current frame), "_parent" (show in parent frame), "_top" (show in topmost frame), and "_blank" (show in new browser window). You can also specify a name, which causes the document to be shown in a new browser window by that name.

The following applet demonstrates **AppletContext** and **showDocument( )**. Upon execution, it obtains the current applet context and uses that context to transfer control to a file called **Test.html**. This file must be in the same directory as the applet. **Test.html** can contain any valid hypertext that you like.

```
/* Using an applet context, getCodeBase(),
   and showDocument() to display an HTML file.
*/

import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="ACDemo" width=300 height=50>
</applet>
*/

public class ACDemo extends Applet {
  public void start() {
    AppletContext ac = getAppletContext();
    URL url = getCodeBase(); // get url of this applet

    try {
      ac.showDocument(new URL(url+"Test.html"));
    } catch(MalformedURLException e) {
      showStatus("URL not found");
    }
  }
}
```

## The AudioClip Interface

The **AudioClip** interface defines these methods: **play( )** (play a clip from the beginning), **stop( )** (stop playing the clip), and **loop( )** (play the loop continuously). After you have loaded an audio clip using **getAudioClip( )**, you can use these methods to play it.

## The AppletStub Interface

The **AppletStub** interface provides the means by which an applet and the browser (or applet viewer) communicate. Your code will not typically implement this interface.



## Outputting to the Console

Although output to an applet's window must be accomplished through GUI-based methods, such as **drawString( )**, it is still possible to use console output in your applet—especially for debugging purposes. In an applet, when you call a method such as **System.out.println( )**, the output is not sent to your applet's window. Instead, it appears either in the console session in which you launched the applet viewer or in the Java console that is available in some browsers. Use of console output for purposes other than debugging is discouraged, since it violates the design principles of the graphical interface most users will expect.

This page has been intentionally left blank

CHAPTER

# 24 Event Handling

This chapter examines an important aspect of Java: the event. Event handling is fundamental to Java programming because it is integral to the creation of many kinds of applications, including applets and other types of GUI-based programs. As explained in Chapter 23, applets are event-driven programs that use a graphical user interface to interact with the user. Furthermore, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Thus, you cannot write these types of programs without a solid command of event handling. Events are supported by a number of packages, including **java.util**, **java.awt**, and **java.awt.event**.

Most events to which your program will respond are generated when the user interacts with a GUI-based program. These are the types of events examined in this chapter. They are passed to your program in a variety of ways, with the specific method dependent upon the actual event. There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.

This chapter begins with an overview of Java's event handling mechanism. It then examines the main event classes and interfaces used by the AWT and develops several examples that demonstrate the fundamentals of event processing. This chapter also explains how to use adapter classes, inner classes, and anonymous inner classes to streamline event handling code. The examples provided in the remainder of this book make frequent use of these techniques.

**NOTE** This chapter focuses on events related to GUI-based programs. However, events are also occasionally used for purposes not directly related to GUI-based programs. In all cases, the same basic event handling techniques apply.

## Two Event Handling Mechanisms

Before beginning our discussion of event handling, an important historical point must be made: The way in which events are handled changed significantly between the original version of Java (1.0) and all subsequent versions of Java, beginning with version 1.1.

Although the 1.0 method of event handling is still supported, it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs in this book.

# The Delegation Event Model

The modern approach to handling events is based on the *delegation event model*, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners*. In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the original Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

The following sections define events and describe the roles of sources and listeners.

## Events

In the delegation model, an *event* is an object that describes a state change in a source. Among other causes, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

## Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void add*Type*Listener (*Type*Listener *el*)

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event

occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

```
public void add Type Listener( Type Listener el )
  throws java.util.TooManyListenersException
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

```
public void remove Type Listener( Type Listener el )
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.



## Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Other listener interfaces are discussed later in this and other chapters.

# Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, a discussion of event handling must begin with the event classes. It is important to understand, however, that Java defines several types of events and that not all event classes can be discussed in this chapter. Arguably, the most widely used events at the time of this writing are those defined by the AWT and those defined by Swing. This chapter focuses on the AWT events. (Most of these events also apply to Swing.) Several Swing-specific events are described in Chapter 31, when Swing is covered.

At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

```
EventObject(Object src)
```

Here, *src* is the object that generates this event.

**EventObject** defines two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

```
Object getSource( )
```

As expected, **toString( )** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

```
int getID( )
```

Additional details about **AWTEvent** are provided at the end of Chapter 26. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**.

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The package **java.awt.event** defines many types of events that are generated by various user interface elements. Table 24-1 shows several commonly used event classes and provides a brief description of when they are generated. Commonly used constructors and methods in each class are described in the following sections.

| Event Class | Description |
|---|---|
| ActionEvent | Generated when a button is pressed, a list item is double-clicked, or a menu item is selected. |
| AdjustmentEvent | Generated when a scroll bar is manipulated. |
| ComponentEvent | Generated when a component is hidden, moved, resized, or becomes visible. |
| ContainerEvent | Generated when a component is added to or removed from a container. |
| FocusEvent | Generated when a component gains or loses keyboard focus. |
| InputEvent | Abstract superclass for all component input event classes. |
| ItemEvent | Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected. |
| KeyEvent | Generated when input is received from the keyboard. |
| MouseEvent | Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component. |
| MouseWheelEvent | Generated when the mouse wheel is moved. |
| TextEvent | Generated when the value of a text area or text field is changed. |
| WindowEvent | Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**Table 24-1** Commonly Used Event Classes in **java.awt.event**

## The ActionEvent Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

**ActionEvent** has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type,* and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

String getActionCommand( )

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

int getModifiers( )

The method **getWhen( )** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen( )** method is shown here:

long getWhen( )

## The AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

| | |
|---|---|
| BLOCK_DECREMENT | The user clicked inside the scroll bar to decrease its value. |
| BLOCK_INCREMENT | The user clicked inside the scroll bar to increase its value. |
| TRACK | The slider was dragged. |
| UNIT_DECREMENT | The button at the end of the scroll bar was clicked to decrease its value. |
| UNIT_INCREMENT | The button at the end of the scroll bar was clicked to increase its value. |

In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred.

Here is one **AdjustmentEvent** constructor:

AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *val*)

Here, *src* is a reference to the object that generated this event. The *id* specifies the event. The type of the adjustment is specified by *type*, and its associated value is *val*.

The **getAdjustable( )** method returns the object that generated the event. Its form is shown here:

Adjustable getAdjustable( )

The type of the adjustment event may be obtained by the **getAdjustmentType( )** method. It returns one of the constants defined by **AdjustmentEvent**. The general form is shown here:

int getAdjustmentType( )

The amount of the adjustment can be obtained from the **getValue( )** method, shown here:

int getValue( )

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

## The ComponentEvent Class

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

| | |
|---|---|
| COMPONENT_HIDDEN | The component was hidden. |
| COMPONENT_MOVED | The component was moved. |
| COMPONENT_RESIZED | The component was resized. |
| COMPONENT_SHOWN | The component became visible. |

**ComponentEvent** has this constructor:

ComponentEvent(Component *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

**ComponentEvent** is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**, among others.

The **getComponent( )** method returns the component that generated the event. It is shown here:

Component getComponent( )

## The ContainerEvent Class

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT_ADDED** and

**COMPONENT_REMOVED**. They indicate that a component has been added to or removed from the container.

**ContainerEvent** is a subclass of **ComponentEvent** and has this constructor:

ContainerEvent(Component *src*, int *type*, Component *comp*)

Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type,* and the component that has been added to or removed from the container is *comp*.

You can obtain a reference to the container that generated this event by using the **getContainer ( )** method, shown here:

Container getContainer( )

The **getChild( )** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

Component getChild( )



## The FocusEvent Class

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.

**FocusEvent** is a subclass of **ComponentEvent** and has these constructors:

FocusEvent(Component *src*, int *type*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type*. The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)

The other component involved in the focus change, called the *opposite component*, is passed in *other*. Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus. Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus.

You can determine the other component by calling **getOppositeComponent( )**, shown here:

Component getOppositeComponent( )

The opposite component is returned.

The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary( )

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

## The InputEvent Class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

**InputEvent** defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers:

| | | |
|---|---|---|
| ALT_MASK | BUTTON2_MASK | META_MASK |
| ALT_GRAPH_MASK | BUTTON3_MASK | SHIFT_MASK |
| BUTTON1_MASK | CTRL_MASK | |

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

| | | |
|---|---|---|
| ALT_DOWN_MASK | BUTTON2_DOWN_MASK | META_DOWN_MASK |
| ALT_GRAPH_DOWN_MASK | BUTTON3_DOWN_MASK | SHIFT_DOWN_MASK |
| BUTTON1_DOWN_MASK | CTRL_DOWN_MASK | |

When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers.

To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

You can obtain a value that contains all of the original modifier flags by calling the **getModifiers( )** method. It is shown here:

```
int getModifiers( )
```

You can obtain the extended modifiers by calling **getModifiersEx( )**, which is shown here:

```
int getModifiersEx( )
```

## The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this book.) There are two types of item events, which are identified by the following integer constants:

| | |
|---|---|
| DESELECTED | The user deselected an item. |
| SELECTED | The user selected an item. |

In addition, **ItemEvent** defines one integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state.

**ItemEvent** has this constructor:

ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type*. The specific item that generated the item event is passed in *entry*. The current state of that item is in *state*.

The **getItem( )** method can be used to obtain a reference to the item that changed. Its signature is shown here:

Object getItem( )

The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

ItemSelectable getItemSelectable( )

Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.

The **getStateChange( )** method returns the state change (that is, **SELECTED** or **DESELECTED**) for the event. It is shown here:

int getStateChange( )

## The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

| | | | |
|---|---|---|---|
| VK_ALT | VK_DOWN | VK_LEFT | VK_RIGHT |
| VK_CANCEL | VK_ENTER | VK_PAGE_DOWN | VK_SHIFT |
| VK_CONTROL | VK_ESCAPE | VK_PAGE_UP | VK_UP |

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.

**KeyEvent** is a subclass of **InputEvent**. Here is one of its constructors:

KeyEvent(Component *src*, int *type*, long *when*, int *modifiers*, int *code*, char *ch*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The

*modifiers* argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**.

The **KeyEvent** class defines several methods, but probably the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

```
char getKeyChar( )
int getKeyCode( )
```

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

## The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

| | |
|---|---|
| MOUSE_CLICKED | The user clicked the mouse. |
| MOUSE_DRAGGED | The user dragged the mouse. |
| MOUSE_ENTERED | The mouse entered a component. |
| MOUSE_EXITED | The mouse exited from a component. |
| MOUSE_MOVED | The mouse moved. |
| MOUSE_PRESSED | The mouse was pressed. |
| MOUSE_RELEASED | The mouse was released. |
| MOUSE_WHEEL | The mouse wheel was moved. |

**MouseEvent** is a subclass of **InputEvent**. Here is one of its constructors:

```
MouseEvent(Component src, int type, long when, int modifiers,
           int x, int y, int clicks, boolean triggersPopup)
```

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

```
int getX( )
int getY( )
```

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:

```
Point getPoint( )
```

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.

The **translatePoint( )** method changes the location of the event. Its form is shown here:

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.

The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

boolean isPopupTrigger( )

Also available is the **getButton( )** method, shown here:

int getButton( )

It returns a value that represents the button that caused the event. For most cases, the return value will be one of these constants defined by **MouseEvent**:

| | | | |
|---|---|---|---|
| NOBUTTON | BUTTON1 | BUTTON2 | BUTTON3 |

The **NOBUTTON** value indicates that no button was pressed or released.

Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )

The **getLocationOnScreen( )** method returns a **Point** object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

## The MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**. Not all mice have wheels. If a mouse has a wheel, it is typically located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants:

| | |
|---|---|
| WHEEL_BLOCK_SCROLL | A page-up or page-down scroll event occurred. |
| WHEEL_UNIT_SCROLL | A line-up or line-down scroll event occurred. |

Here is one of the constructors defined by **MouseWheelEvent**:

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*,
int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*.

The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_ SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

**MouseWheelEvent** defines methods that give you access to the wheel event. To obtain the number of rotational units, call **getWheelRotation( )**, shown here:

```
int getWheelRotation( )
```

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. JDK 7 added a method called **getPreciseWheelRotation( )**, which supports high-resolution wheels. It works like **getWheelRotation( )**, but returns a **double**.

To obtain the type of scroll, call **getScrollType( )**, shown next:

```
int getScrollType( )
```

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**.

If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount( )**. It is shown here:

```
int getScrollAmount( )
```

## The TextEvent Class

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.

The one constructor for this class is shown here:

```
TextEvent(Object src, int type)
```

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

## The WindowEvent Class

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

| | |
|---|---|
| WINDOW_ACTIVATED | The window was activated. |
| WINDOW_CLOSED | The window has been closed. |

| | |
|---|---|
| WINDOW_CLOSING | The user requested that the window be closed. |
| WINDOW_DEACTIVATED | The window was deactivated. |
| WINDOW_DEICONIFIED | The window was deiconified. |
| WINDOW_GAINED_FOCUS | The window gained input focus. |
| WINDOW_ICONIFIED | The window was iconified. |
| WINDOW_LOST_FOCUS | The window lost input focus. |
| WINDOW_OPENED | The window was opened. |
| WINDOW_STATE_CHANGED | The state of the window changed. |



**WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors. The first is

WindowEvent(Window *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is *type*.

The next three constructors offer more detailed control:

WindowEvent(Window *src*, int *type*, Window *other*)
WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Here, *other* specifies the opposite window when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *toState* specifies the new state that the window will have when a window state change occurs.

A commonly used method in this class is **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:

Window getWindow( )

**WindowEvent** also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

Window getOppositeWindow( )
int getOldState( )
int getNewState( )

## Sources of Events

Table 24-2 lists some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, any class derived from **Component**, such as **Applet**, can generate events. For example, you can receive key and mouse events from an applet. (You may also build your own components that generate events.) In this chapter, we will be handling only mouse and keyboard events, but the following two chapters will be handling events from the sources shown in Table 24-2.

| Event Source | Description |
|---|---|
| Button | Generates action events when the button is pressed. |
| Check box | Generates item events when the check box is selected or deselected. |
| Choice | Generates item events when the choice is changed. |
| List | Generates action events when an item is double-clicked; generates item events when an item is selected or deselected. |
| Menu item | Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected. |
| Scroll bar | Generates adjustment events when the scroll bar is manipulated. |
| Text components | Generates text events when the user enters a character. |
| Window | Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**Table 24-2** Event Source Examples

## Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. As it relates to this chapter, listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 24-3 lists

| Interface | Description |
|---|---|
| ActionListener | Defines one method to receive action events. |
| AdjustmentListener | Defines one method to receive adjustment events. |
| ComponentListener | Defines four methods to recognize when a component is hidden, moved, resized, or shown. |
| ContainerListener | Defines two methods to recognize when a component is added to or removed from a container. |
| FocusListener | Defines two methods to recognize when a component gains or loses keyboard focus. |
| ItemListener | Defines one method to recognize when the state of an item changes. |
| KeyListener | Defines three methods to recognize when a key is pressed, released, or typed. |
| MouseListener | Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released. |
| MouseMotionListener | Defines two methods to recognize when the mouse is dragged or moved. |
| MouseWheelListener | Defines one method to recognize when the mouse wheel is moved. |
| TextListener | Defines one method to recognize when a text value changes. |
| WindowFocusListener | Defines two methods to recognize when a window gains or loses input focus. |
| WindowListener | Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit. |

**Table 24-3** Commonly Used Event Listener Interfaces

several commonly used listener interfaces and provides a brief description of the methods that they define. The following sections examine the specific methods that are contained in each interface.

## The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

## The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

## The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

## The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

## The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

## The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

## The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

## The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

## The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

## The MouseWheelListener Interface

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

## The TextListener Interface

This interface defines the **textValueChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textValueChanged(TextEvent *te*)

### The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

### The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent *we*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

## Using the Delegation Event Model

Now that you have learned the theory behind the delegation event model and have had an overview of its various components, it is time to see it in practice. Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it can receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and keyboard.

### Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. (You may also want to implement **MouseWheelListener**, but we won't be doing so, here.) The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is

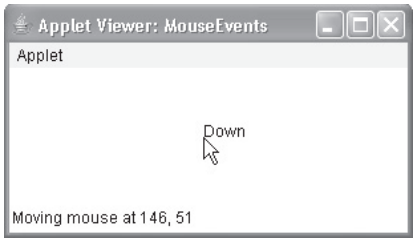
pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area.

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MouseEvents" width=300 height=100>
  </applet>
*/

public class MouseEvents extends Applet
  implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX = 0, mouseY = 0; // coordinates of mouse

  public void init() {
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse clicked.";
    repaint();
  }

  // Handle mouse entered.
  public void mouseEntered(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse entered.";
    repaint();
  }

  // Handle mouse exited.
```

```
  public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse exited.";
    repaint();
  }

  // Handle button pressed.
  public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
  }

  // Handle button released.
  public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "*";
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
    repaint();
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // show status
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
  }

  // Display msg in applet window at current X,Y location.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }
}
```

Sample output from this program is shown here:




Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events. This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets.

Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here:

void addMouseListener(MouseListener *ml*)
void addMouseMotionListener(MouseMotionListener *mml*)

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both.

The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

## Handling Keyboard Events

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface.

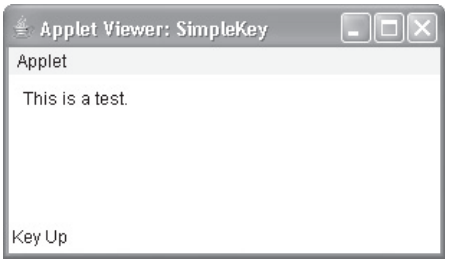
Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed( )** handler.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
/*
  <applet code="SimpleKey" width=300 height=100>
  </applet>
*/

public class SimpleKey extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // output coordinates

  public void init() {
    addKeyListener(this);
  }

  public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");
  }

  public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Display keystrokes.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

Sample output is shown here:




If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed( )** handler. They are not available through **keyTyped( )**.

To identify the keys, you use their virtual key codes. For example, the next applet outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="KeyEvents" width=300 height=100>
  </applet>
*/

public class KeyEvents extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // output coordinates

  public void init() {
    addKeyListener(this);
  }

  public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");

    int key = ke.getKeyCode();
    switch(key) {
      case KeyEvent.VK_F1:
        msg += "<F1>";
        break;
      case KeyEvent.VK_F2:
        msg += "<F2>";
        break;
      case KeyEvent.VK_F3:
        msg += "<F3>";
        break;
      case KeyEvent.VK_PAGE_DOWN:
        msg += "<PgDn>";
        break;
      case KeyEvent.VK_PAGE_UP:
        msg += "<PgUp>";
        break;
      case KeyEvent.VK_LEFT:
        msg += "<Left Arrow>";
        break;
      case KeyEvent.VK_RIGHT:
        msg += "<Right Arrow>";
        break;
    }

    repaint();
  }
```

```
  public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Display keystrokes.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

Sample output is shown here:


The procedures shown in the preceding keyboard and mouse event examples can be generalized to any type of event handling, including those events generated by controls. In later chapters, you will see many examples that handle other types of events, but they will all follow the same basic structure as the programs just described.

## Adapter Classes

Java provides a special feature, called an *adapter class,* that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

Table 24-4 lists several commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

| Adapter Class | Listener Interface |
|---|---|
| ComponentAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| FocusAdapter | FocusListener |
| KeyAdapter | KeyListener |
| MouseAdapter | MouseListener and (as of JDK 6) MouseMotionListener and MouseWheelListener |
| MouseMotionAdapter | MouseMotionListener |
| WindowAdapter | WindowListener, WindowFocusListener, and WindowStateListener |

**Table 24-4** Commonly Used Listener Interfaces Implemented by Adapter Classes

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init( )** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument.

**MyMouseAdapter** extends **MouseAdapter** and overrides the **mouseClicked( )** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** extends **MouseMotionAdapter** and overrides the **mouseDragged( )** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class. (**MouseAdaptor** also provides an empty implementation for **MouseMotionListener**. However, for the sake of illustration, this example handles each separately.)

Note that both of the event listener classes save a reference to the applet. This information is provided as an argument to their constructors and is used later to invoke the **showStatus( )** method.

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AdapterDemo" width=300 height=100>
  </applet>
*/

public class AdapterDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
    addMouseMotionListener(new MyMouseMotionAdapter(this));
  }
}
```

```
class MyMouseAdapter extends MouseAdapter {

  AdapterDemo adapterDemo;
  public MyMouseAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
    adapterDemo.showStatus("Mouse clicked");
  }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
  AdapterDemo adapterDemo;
  public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    adapterDemo.showStatus("Mouse dragged");
  }
}
```



As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener** and **MouseListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods. As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a **KeyAdapter**.

## Inner Classes

In Chapter 7, the basics of inner classes were explained. Here, you will see why they are important. Recall that an *inner class* is a class defined within another class, or even within an expression. This section illustrates how inner classes can be used to simplify the code when using event adapter classes.

To understand the benefit provided by inner classes, consider the applet shown in the following listing. It *does not* use an inner class. Its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed. There are two top-level classes in this program. **MousePressedDemo** extends **Applet**, and **MyMouseAdapter** extends **MouseAdapter**. The **init( )** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener( )** method.

Notice that a reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor. This reference is stored in an instance variable for later use by the **mousePressed( )** method. When the mouse is pressed, it invokes the **showStatus( )** method of the applet

through the stored applet reference. In other words, **showStatus( )** is invoked relative to the applet reference stored by **MyMouseAdapter**.

```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="MousePressedDemo" width=200 height=100>
  </applet>
*/

public class MousePressedDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
  }
}

class MyMouseAdapter extends MouseAdapter {
  MousePressedDemo mousePressedDemo;
  public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
    this.mousePressedDemo = mousePressedDemo;
  }
  public void mousePressed(MouseEvent me) {
    mousePressedDemo.showStatus("Mouse Pressed.");
  }
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**. Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed( )** method can call the **showStatus( )** method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass **MyMouseAdapter( )** a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="InnerClassDemo" width=200 height=100>
  </applet>
*/

public class InnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter());
  }
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      showStatus("Mouse Pressed");
    }
  }
}
```

## Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. This section illustrates how an anonymous inner class can facilitate the writing of event handlers. Consider the applet shown in the following listing. As before, its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="AnonymousInnerClassDemo" width=200 height=100>
  </applet>
*/

public class AnonymousInnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        showStatus("Mouse Pressed");
      }
    });
  }
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init( )** method calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully.

The syntax **new MouseAdapter(){...}** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**. This new class is not named, but it is automatically instantiated when this expression is executed.

Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus( )** method directly.

As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.

This page has been intentionally left blank

CHAPTER

# 25 Introducing the AWT: Working with Windows, Graphics, and Text

The Abstract Window Toolkit (AWT) was Java's first GUI framework, and it has been part of Java since version 1.0. It contains numerous classes and methods that allow you to create windows and simple controls. The AWT was introduced in Chapter 23, where it was used in several short, example applets. This chapter begins a more detailed examination. Here, you will learn how to create and manage windows, manage fonts, output text, and utilize graphics. Chapter 26 describes various AWT controls, such as scroll bars and push buttons. It also explains further aspects of Java's event handling mechanism. Chapter 27 introduces the AWT's imaging subsystem.

It is important to state at the outset that you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. Despite this fact, the AWT remains an important part of Java. To understand why, consider the following.

At the time of this writing, the framework that is most widely used is Swing. Because Swing provides a richer, more flexible GUI framework than does the AWT, it is easy to jump to the conclusion that the AWT is no longer relevant—that it has been fully superseded by Swing. This assumption is, however, false. Instead, an understanding of the AWT is still important because the AWT underpins Swing, with many AWT classes being used either directly or indirectly by Swing. As a result, a solid knowledge of the AWT is still required to use Swing effectively.

Java's newest GUI framework is JavaFX. It is anticipated that, at some point in the future, JavaFX will replace Swing as Java's most popular GUI. Even when this occurs, however, much legacy code that relies on Swing (and thus, the AWT) will still need to be maintained for some time to come. Finally, for some types of small programs (especially small applets) that make only minimal use of a GUI, using the AWT may still be appropriate. Therefore, even though the AWT constitutes Java's oldest GUI framework, a basic working knowledge of its fundamentals is still important today.

Although a common use of the AWT is in applets, it is also used to create stand-alone windows that run in a GUI environment, such as Windows. For the sake of convenience, most of the examples in this chapter are contained in applets. The easiest way to run them

is with the applet viewer. A few examples demonstrate the creation of stand-alone, windowed programs, which can be executed directly.

One last point before beginning: The AWT is quite large and a full description would easily fill an entire book. Therefore, it is not possible to describe in detail every AWT class, method, or instance variable. However, this and the following chapters explain the basic techniques needed to use the AWT. From there, you will be able to explore other parts of the AWT on your own. You will also be ready to move on to Swing.

**NOTE** If you have not yet read Chapter 24, please do so now. It provides an overview of event handling, which is used by many of the examples in this chapter.

## AWT Classes

The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages. Fortunately, because it is logically organized in a top-down, hierarchical fashion, it is easier to understand and use than you might at first believe. Table 25-1 lists some of the many AWT classes.

| Class | Description |
|---|---|
| AWTEvent | Encapsulates AWT events. |
| AWTEventMulticaster | Dispatches events to multiple listeners. |
| BorderLayout | The border layout manager. Border layouts use five components: North, South, East, West, and Center. |
| Button | Creates a push button control. |
| Canvas | A blank, semantics-free window. |
| CardLayout | The card layout manager. Card layouts emulate index cards. Only the one on top is showing. |
| Checkbox | Creates a check box control. |
| CheckboxGroup | Creates a group of check box controls. |
| CheckboxMenuItem | Creates an on/off menu item. |
| Choice | Creates a pop-up list. |
| Color | Manages colors in a portable, platform-independent fashion. |
| Component | An abstract superclass for various AWT components. |
| Container | A subclass of **Component** that can hold other components. |
| Cursor | Encapsulates a bitmapped cursor. |
| Dialog | Creates a dialog window. |
| Dimension | Specifies the dimensions of an object. The width is stored in **width**, and the height is stored in **height**. |
| EventQueue | Queues events. |
| FileDialog | Creates a window from which a file can be selected. |

**Table 25-1** A Sampling of AWT Classes

| Class | Description |
|---|---|
| FlowLayout | The flow layout manager. Flow layout positions components left to right, top to bottom. |
| Font | Encapsulates a type font. |
| FontMetrics | Encapsulates various information related to a font. This information helps you display text in a window. |
| Frame | Creates a standard window that has a title bar, resize corners, and a menu bar. |
| Graphics | Encapsulates the graphics context. This context is used by the various output methods to display output in a window. |
| GraphicsDevice | Describes a graphics device such as a screen or printer. |
| GraphicsEnvironment | Describes the collection of available **Font** and **GraphicsDevice** objects. |
| GridBagConstraints | Defines various constraints relating to the **GridBagLayout** class. |
| GridBagLayout | The grid bag layout manager. Grid bag layout displays components subject to the constraints specified by **GridBagConstraints**. |
| GridLayout | The grid layout manager. Grid layout displays components in a two-dimensional grid. |
| Image | Encapsulates graphical images. |
| Insets | Encapsulates the borders of a container. |
| Label | Creates a label that displays a string. |
| List | Creates a list from which the user can choose. Similar to the standard Windows list box. |
| MediaTracker | Manages media objects. |
| Menu | Creates a pull-down menu. |
| MenuBar | Creates a menu bar. |
| MenuComponent | An abstract class implemented by various menu classes. |
| MenuItem | Creates a menu item. |
| MenuShortcut | Encapsulates a keyboard shortcut for a menu item. |
| Panel | The simplest concrete subclass of **Container**. |
| Point | Encapsulates a Cartesian coordinate pair, stored in **x** and **y**. |
| Polygon | Encapsulates a polygon. |
| PopupMenu | Encapsulates a pop-up menu. |
| PrintJob | An abstract class that represents a print job. |
| Rectangle | Encapsulates a rectangle. |
| Robot | Supports automated testing of AWT-based applications. |
| Scrollbar | Creates a scroll bar control. |
| ScrollPane | A container that provides horizontal and/or vertical scroll bars for another component. |

**Table 25-1** A Sampling of AWT Classes *(continued)*

| Class | Description |
|---|---|
| SystemColor | Contains the colors of GUI widgets such as windows, scroll bars, text, and others. |
| TextArea | Creates a multiline edit control. |
| TextComponent | A superclass for **TextArea** and **TextField**. |
| TextField | Creates a single-line edit control. |
| Toolkit | Abstract class implemented by the AWT. |
| Window | Creates a window with no frame, no menu bar, and no title. |

**Table 25-1** A Sampling of AWT Classes *(continued)*

Although the basic structure of the AWT has been the same since Java 1.0, some of the original methods were deprecated and replaced by new ones. For backward-compatibility, Java still supports all the original 1.0 methods. However, because these methods are not for use with new code, this book does not describe them.

# Window Fundamentals

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from **Panel**, which is used by applets, and those derived from **Frame**, which creates a standard application window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding. Figure 25-1 shows the class hierarchy for **Panel** and **Frame**. Let's look at each of these classes now.

## Component

At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. Except for menus, all user interface elements that are displayed on the screen and that interact with the user are

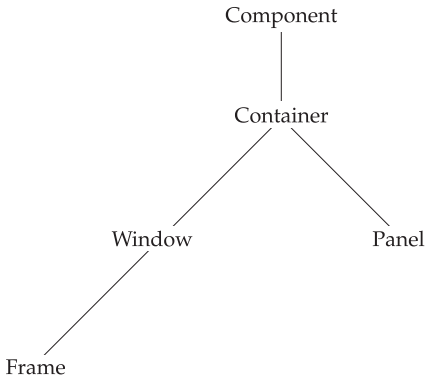



**Figure 25-1** The class hierarchy for **Panel** and **Frame**

subclasses of **Component**. It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting. (You already used many of these methods when you created applets in Chapters 23 and 24.) A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

## Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains. It does this through the use of various layout managers, which you will learn about in Chapter 26.



## Panel

The **Panel** class is a concrete subclass of **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, **setPreferredSize( )**, or **setBounds( )** methods defined by **Component**.

## Window

The **Window** class creates a top-level window. A *top-level window* is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

## Frame

**Frame** encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. The precise look of a **Frame** will differ among environments. A number of environments are reflected in the screen captures shown throughout this book.

## Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. Derived from **Component**, **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

# Working with Frame Windows

In addition to the applet, the type of AWT-based window you will most often create is derived from **Frame**. You will use it to create child windows within applets, and top-level or child windows for stand-alone applications. As mentioned, it creates a standard-style window.

Here are two of **Frame**'s constructors:

Frame( ) throws HeadlessException
Frame(String *title*) throws HeadlessException

The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by *title*. Notice that you cannot specify the dimensions of the window. Instead, you must set the size of the window after it has been created. A **HeadlessException** is thrown if an attempt is made to create a **Frame** instance in an environment that does not support user interaction.

There are several key methods you will use when working with **Frame** windows. They are examined here.

## Setting the Window's Dimensions

The **setSize( )** method is used to set the dimensions of the window. Its signature is shown here:

void setSize(int *newWidth*, int *newHeight*)
void setSize(Dimension *newSize*)

The new size of the window is specified by *newWidth* and *newHeight*, or by the **width** and **height** fields of the **Dimension** object passed in *newSize*. The dimensions are specified in terms of pixels.

The **getSize( )** method is used to obtain the current size of a window. One of its forms is shown here:

Dimension getSize( )

This method returns the current size of the window contained within the **width** and **height** fields of a **Dimension** object.

## Hiding and Showing a Window

After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here:

void setVisible(boolean *visibleFlag*)

The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

## Setting a Window's Title

You can change the title in a frame window using **setTitle( )**, which has this general form:

void setTitle(String *newTitle*)

Here, *newTitle* is the new title for the window.

## Closing a Frame Window

When using a frame window, your program must remove that window from the screen when it is closed, by calling **setVisible(false)**. To intercept a window-close event, you must implement the **windowClosing( )** method of the **WindowListener** interface. Inside **windowClosing( )**, you must remove the window from the screen. The example in the next section illustrates this technique.

# Creating a Frame Window in an AWT-Based Applet



While it is possible to simply create a window by creating an instance of **Frame**, you will seldom do so, because you will not be able to do much with it. For example, you will not be able to receive or process events that occur within it or easily output information to it. Most of the time, you will create a subclass of **Frame**. Doing so lets you override **Frame**'s methods and provide event handling.

Creating a new frame window from within an AWT-based applet is actually quite easy. First, create a subclass of **Frame**. Next, override any of the standard applet methods, such as **init( )**, **start( )**, and **stop( )**, to show or hide the frame as needed. Finally, implement the **windowClosing( )** method of the **WindowListener** interface, calling **setVisible(false)** when the window is closed.

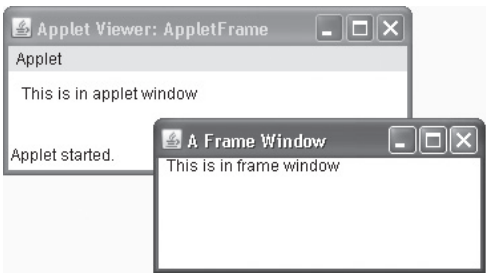
Once you have defined a **Frame** subclass, you can create an object of that class. This causes a frame window to come into existence, but it will not be initially visible. You make it visible by calling **setVisible( )**. When created, the window is given a default height and width. You can set the size of the window explicitly by calling the **setSize( )** method.

The following applet creates a subclass of **Frame** called **SampleFrame**. A window of this subclass is instantiated within the **init( )** method of **AppletFrame**. Notice that **SampleFrame** calls **Frame**'s constructor. This causes a standard frame window to be created with the title passed in **title**. This example overrides the applet's **start( )** and **stop( )** methods so that they show and hide the child window, respectively. This causes the window to be removed automatically when you terminate the applet, when you close the window, or, if using a browser, when you move to another page. It also causes the child window to be shown when the browser returns to the applet.

```
// Create a child frame window from within an applet.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="AppletFrame" width=300 height=50>
  </applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame {
  SampleFrame(String title) {
    super(title);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
```

```
    // register it to receive those events
    addWindowListener(adapter);
  }
  public void paint(Graphics g) {
    g.drawString("This is in frame window", 10, 40);
  }
}

class MyWindowAdapter extends WindowAdapter {
  SampleFrame sampleFrame;

  public MyWindowAdapter(SampleFrame sampleFrame) {
    this.sampleFrame = sampleFrame;
  }

  public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
  }
}

// Create frame window.
public class AppletFrame extends Applet {
  Frame f;
  public void init() {
    f = new SampleFrame("A Frame Window");

    f.setSize(250, 250);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }

  public void paint(Graphics g) {
    g.drawString("This is in applet window", 10, 20);
  }
}
```

Sample output from this program is shown here:



## Handling Events in a Frame Window

Since **Frame** is a subclass of **Component**, it inherits all the capabilities defined by **Component**. This means that you can use and manage a frame window just like you manage an applet's main window, as described earlier in this book. For example, you can override **paint( )** to display output, call **repaint( )** when you need to restore the window, and add event handlers. Whenever an event occurs in a window, the event handlers defined by that window will be called. Each window handles its own events. For example, the following program creates a window that responds to mouse events. The main applet window also responds to mouse events. When you experiment with this program, you will see that mouse events are sent to the window in which the event occurs.



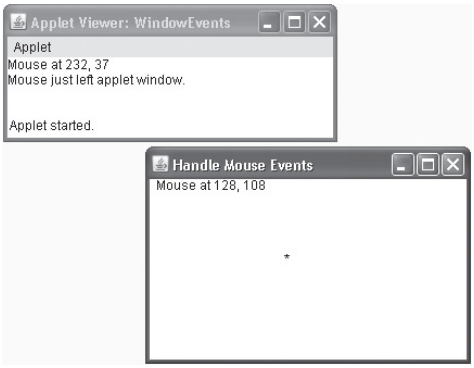
```
// Handle mouse events in both child and applet windows.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="WindowEvents" width=300 height=50>
  </applet>
*/

// Create a subclass of Frame.
class SampleFrame extends Frame
  implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX=10, mouseY=40;
  int movX=0, movY=0;

  SampleFrame(String title) {
    super(title);
    // register this object to receive its own mouse events
    addMouseListener(this);
    addMouseMotionListener(this);
    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // register it to receive those events
    addWindowListener(adapter);
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
  }

  // Handle mouse entered.
  public void mouseEntered(MouseEvent evtObj) {
    // save coordinates
    mouseX = 10;
    mouseY = 54;
    msg = "Mouse just entered child.";
    repaint();
  }
```

```
  // Handle mouse exited.
  public void mouseExited(MouseEvent evtObj) {
    // save coordinates
    mouseX = 10;
    mouseY = 54;
    msg = "Mouse just left child window.";
    repaint();
  }

  // Handle mouse pressed.
  public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
  }

  // Handle mouse released.
  public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "*";
    repaint();
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // save coordinates
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 60);
  }

  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 10, 40);
  }
}

class MyWindowAdapter extends WindowAdapter {
  SampleFrame sampleFrame;
```

```
  public MyWindowAdapter(SampleFrame sampleFrame) {
    this.sampleFrame = sampleFrame;
  }

  public void windowClosing(WindowEvent we) {
    sampleFrame.setVisible(false);
  }
}

// Applet window.
public class WindowEvents extends Applet
  implements MouseListener, MouseMotionListener {

  SampleFrame f;
  String msg = "";
  int mouseX=0, mouseY=10;
  int movX=0, movY=0;

  // Create a frame window.
  public void init() {
    f = new SampleFrame("Handle Mouse Events");
    f.setSize(300, 200);
    f.setVisible(true);

    // register this object to receive its own mouse events
    addMouseListener(this);
    addMouseMotionListener(this);
  }

  // Remove frame window when stopping applet.
  public void stop() {
    f.setVisible(false);
  }

  // Show frame window when starting applet.
  public void start() {
    f.setVisible(true);
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
  }

  // Handle mouse entered.
  public void mouseEntered(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just entered applet window.";
    repaint();
  }
```

```
  // Handle mouse exited.
  public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 24;
    msg = "Mouse just left applet window.";
    repaint();
  }

  // Handle button pressed.
  public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
  }

  // Handle button released.
  public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    movX = me.getX();
    movY = me.getY();
    msg = "*";
    repaint();
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // save coordinates
    movX = me.getX();
    movY = me.getY();
    repaint(0, 0, 100, 20);
  }

  // Display msg in applet window.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
    g.drawString("Mouse at " + movX + ", " + movY, 0, 10);
  }
}
```

Sample output from this program is shown here:





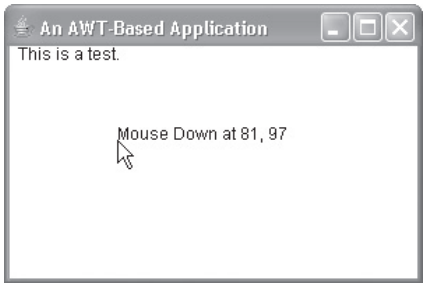

## Creating a Windowed Program

Although creating applets is a common use for Java's AWT, it is also possible to create stand-alone AWT-based applications. To do this, simply create an instance of the window or windows you need inside **main( )**. For example, the following program creates a frame window that responds to mouse clicks and keystrokes:

```
// Create an AWT-based application.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

// Create a frame window.
public class AppWindow extends Frame {
  String keymsg = "This is a test.";
  String mousemsg = "";
  int mouseX=30, mouseY=30;

  public AppWindow() {
    addKeyListener(new MyKeyAdapter(this));
    addMouseListener(new MyMouseAdapter(this));
    addWindowListener(new MyWindowAdapter());
  }

  public void paint(Graphics g) {
    g.drawString(keymsg, 10, 40);
    g.drawString(mousemsg, mouseX, mouseY);
  }

  // Create the window.
  public static void main(String args[]) {
    AppWindow appwin = new AppWindow();

    appwin.setSize(new Dimension(300, 200));
    appwin.setTitle("An AWT-Based Application");
    appwin.setVisible(true);
  }
}
```

```
class MyKeyAdapter extends KeyAdapter {
  AppWindow appWindow;

  public MyKeyAdapter(AppWindow appWindow) {
    this.appWindow = appWindow;
  }

  public void keyTyped(KeyEvent ke) {
    appWindow.keymsg += ke.getKeyChar();
    appWindow.repaint();
  };
}

class MyMouseAdapter extends MouseAdapter {
  AppWindow appWindow;

  public MyMouseAdapter(AppWindow appWindow) {
    this.appWindow = appWindow;
  }

  public void mousePressed(MouseEvent me) {
    appWindow.mouseX = me.getX();
    appWindow.mouseY = me.getY();
    appWindow.mousemsg = "Mouse Down at " + appWindow.mouseX +
                         ", " + appWindow.mouseY;
    appWindow.repaint();
  }
}

class MyWindowAdapter extends WindowAdapter {
  public void windowClosing(WindowEvent we) {
    System.exit(0);
  }
}
```

Sample output from this program is shown here:




Once created, a frame window takes on a life of its own. Notice that **main( )** ends with the call to **appwin.setVisible(true)**. However, the program keeps running until you close the window. In essence, when creating a windowed application, you will use **main( )** to launch its top-level window. After that, your program will function as a GUI-based application, not like the console-based programs used earlier.

# Displaying Information Within a Window

In the most general sense, a window is a container for information. Although we have already output small amounts of text to a window in the preceding examples, we have not begun to take advantage of a window's ability to present high-quality text and graphics. Indeed, much of the power of the AWT comes from its support for these items. For this reason, the remainder of this chapter introduces the AWT's text-, graphics-, and font-handling capabilities. As you will see, they are both powerful and flexible.



# Introducing Graphics

The AWT includes several methods that support graphics. All graphics are drawn relative to a window. This can be the main window of an applet, a child window of an applet, or a stand-alone application window. (These methods are also supported by Swing-based windows.) The origin of each window is at the top-left corner and is 0,0. Coordinates are specified in pixels. All output to a window takes place through a *graphics context*.

A graphics context is encapsulated by the **Graphics** class. Here are two ways in which a graphics context can be obtained:

- It is passed to a method, such as **paint( )** or **update( )**, as an argument.
- It is returned by the **getGraphics( )** method of **Component**.

Among other things, the **Graphics** class defines a number of methods that draw various types of objects, such as lines, rectangles, and arcs. In several cases, objects can be drawn edge-only or filled. Objects are drawn and filled in the currently selected color, which is black by default. When a graphics object is drawn that exceeds the dimensions of the window, output is automatically clipped. A sampling of the drawing methods supported by **Graphics** is presented here.

---

**NOTE** With the release of version 1.2, the graphics capabilities of Java were expanded by the inclusion of several new classes. One of these is **Graphics2D**, which extends **Graphics**. **Graphics2D** supports several powerful enhancements to the basic capabilities provided by **Graphics**. To gain access to this extended functionality, you must cast the graphics context obtained from a method such as **paint( )**, to **Graphics2D**. Although the basic graphics functions supported by **Graphics** are adequate for the purposes of this book, **Graphics2D** is a class that you will want to explore fully on your own if you will be programming sophisticated graphics applications.

## Drawing Lines

Lines are drawn by means of the **drawLine( )** method, shown here:

void drawLine(int *startX*, int *startY*, int *endX*, int *endY*)

**drawLine( )** displays a line in the current drawing color that begins at *startX, startY* and ends at *endX, endY*.

## Drawing Rectangles

The **drawRect( )** and **fillRect( )** methods display an outlined and filled rectangle, respectively. They are shown here:

```
void drawRect(int left, int top, int width, int height)
void fillRect(int left, int top, int width, int height)
```

The upper-left corner of the rectangle is at *left*, *top*. The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use **drawRoundRect( )** or **fillRoundRect( )**, both shown here:

```
void drawRoundRect(int left, int top, int width, int height,
                   int xDiam, int yDiam)

void fillRoundRect(int left, int top, int width, int height,
                   int xDiam, int yDiam)
```

A rounded rectangle has rounded corners. The upper-left corner of the rectangle is at *left*, *top*. The dimensions of the rectangle are specified by *width* and *height*. The diameter of the rounding arc along the X axis is specified by *xDiam*. The diameter of the rounding arc along the Y axis is specified by *yDiam*.

## Drawing Ellipses and Circles

To draw an ellipse, use **drawOval( )**. To fill an ellipse, use **fillOval( )**. These methods are shown here:

```
void drawOval(int left, int top, int width, int height)
void fillOval(int left, int top, int width, int height)
```

The ellipse is drawn within a bounding rectangle whose upper-left corner is specified by *left, top* and whose width and height are specified by *width* and *height*. To draw a circle, specify a square as the bounding rectangle.

## Drawing Arcs

Arcs can be drawn with **drawArc( )** and **fillArc( )**, shown here:

```
void drawArc(int left, int top, int width, int height, int startAngle,
             int sweepAngle)

void fillArc(int left, int top, int width, int height, int startAngle,
             int sweepAngle)
```

The arc is bounded by the rectangle whose upper-left corner is specified by *left, top* and whose width and height are specified by *width* and *height*. The arc is drawn from *startAngle* through the angular distance specified by *sweepAngle*. Angles are specified in degrees. Zero degrees is on the horizontal, at the three o'clock position. The arc is drawn counterclockwise if *sweepAngle* is positive, and clockwise if *sweepAngle* is negative. Therefore, to draw an arc from twelve o'clock to six o'clock, the start angle would be 90 and the sweep angle 180.

## Drawing Polygons

It is possible to draw arbitrarily shaped figures using **drawPolygon( )** and **fillPolygon( )**, shown here:

void drawPolygon(int *x*[ ], int *y*[ ], int *numPoints*)
void fillPolygon(int *x*[ ], int *y*[ ], int *numPoints*)

The polygon's endpoints are specified by the coordinate pairs contained within the *x* and *y* arrays. The number of points defined by these arrays is specified by *numPoints*. There are alternative forms of these methods in which the polygon is specified by a **Polygon** object.



## Demonstrating the Drawing Methods

The following program demonstrates the drawing methods just described.

```
// Draw graphics elements.
import java.awt.*;
import java.applet.*;
/*
<applet code="GraphicsDemo" width=350 height=700>
</applet>
*/
public class GraphicsDemo extends Applet {
  public void paint(Graphics g) {

    // Draw lines.
    g.drawLine(0, 0, 100, 90);
    g.drawLine(0, 90, 100, 10);
    g.drawLine(40, 25, 250, 80);

    // Draw rectangles.
    g.drawRect(10, 150, 60, 50);
    g.fillRect(100, 150, 60, 50);
    g.drawRoundRect(190, 150, 60, 50, 15, 15);
    g.fillRoundRect(280, 150, 60, 50, 30, 40);

    // Draw Ellipses and Circles
    g.drawOval(10, 250, 50, 50);
    g.fillOval(90, 250, 75, 50);
    g.drawOval(190, 260, 100, 40);

    // Draw Arcs
    g.drawArc(10, 350, 70, 70, 0, 180);
    g.fillArc(60, 350, 70, 70, 0, 75);

    // Draw a polygon
    int xpoints[] = {10, 200, 10, 200, 10};
    int ypoints[] = {450, 450, 650, 650, 450};
    int num = 5;

    g.drawPolygon(xpoints, ypoints, num);
  }
}
```

Sample output is shown in Figure 25-2.

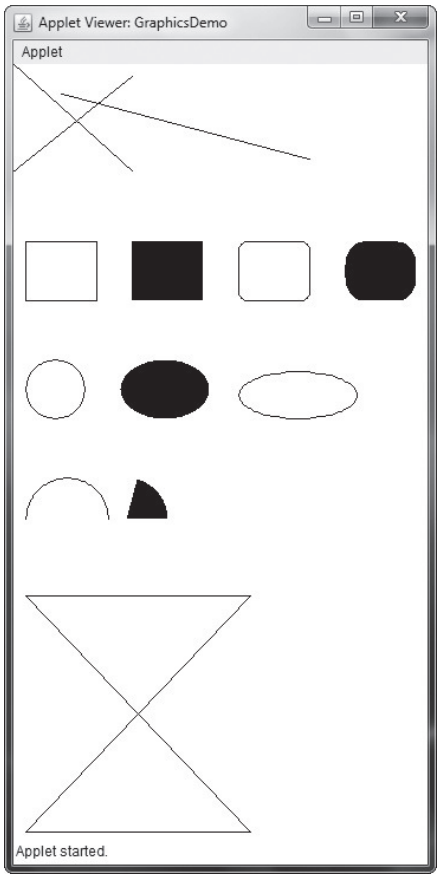



**Figure 25-2** Sample output from the **GraphicsDemo** program

## Sizing Graphics

Often, you will want to size a graphics object to fit the current size of the window in which it is drawn. To do so, first obtain the current dimensions of the window by calling **getSize( )** on the window object. It returns the dimensions of the window encapsulated within a **Dimension** object. Once you have the current size of the window, you can scale your graphical output accordingly.

To demonstrate this technique, here is an applet that will start as a 200×200-pixel square and grow by 25 pixels in width and height with each mouse click until the applet gets larger than 500×500. At that point, the next click will return it to 200×200, and the process starts over.

Within the window, a rectangle is drawn around the inner border of the window; within that rectangle, an *X* is drawn so that it fills the window. This applet works in **appletviewer**, but it may not work in a browser window.

```
// Resizing output to fit the current size of a window.
import java.applet.*;
```

```
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="ResizeMe" width=200 height=200>
  </applet>
*/

public class ResizeMe extends Applet {
  final int inc = 25;
  int max = 500;
  int min = 200;
  Dimension d;

  public ResizeMe() {
    addMouseListener(new MouseAdapter() {
      public void mouseReleased(MouseEvent me) {
        int w = (d.width + inc) > max?min :(d.width + inc);
        int h = (d.height + inc) > max?min :(d.height + inc);
        setSize(new Dimension(w, h));
      }
    });
  }

  public void paint(Graphics g) {
    d = getSize();

    g.drawLine(0, 0, d.width-1, d.height-1);
    g.drawLine(0, d.height-1, d.width-1, 0);
    g.drawRect(0, 0, d.width-1, d.height-1);
  }
}
```



## Working with Color

Java supports color in a portable, device-independent fashion. The AWT color system allows you to specify any color you want. It then finds the best match for that color, given the limits of the display hardware currently executing your program or applet. Thus, your code does not need to be concerned with the differences in the way color is supported by various hardware devices. Color is encapsulated by the **Color** class.

As you saw in Chapter 23, **Color** defines several constants (for example, **Color.black**) to specify a number of common colors. You can also create your own colors, using one of the color constructors. Three commonly used forms are shown here:

Color(int *red*, int *green*, int *blue*)
Color(int *rgbValue*)
Color(float *red*, float *green*, float *blue*)

The first constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255, as in this example:

```
new Color(255, 100, 100); // light red
```

The second color constructor takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Here is an example of this constructor:

```
int newRed = (0xff000000 | (0xc0 << 16) | (0x00 << 8) | 0x00);
Color darkRed = new Color(newRed);
```

The final constructor, **Color(float, float, float)**, takes three **float** values (between 0.0 and 1.0) that specify the relative mix of red, green, and blue.

Once you have created a color, you can use it to set the foreground and/or background color by using the **setForeground( )** and **setBackground( )** methods described in Chapter 23. You can also select it as the current drawing color.

## Color Methods

The **Color** class defines several methods that help manipulate colors. Several are examined here.

### Using Hue, Saturation, and Brightness

The *hue-saturation-brightness (HSB)* color model is an alternative to red-green-blue (RGB) for specifying particular colors. Figuratively, *hue* is a wheel of color. The hue can be specified with a number between 0.0 and 1.0, which is used to obtain an angle into the color wheel. (The principal colors are approximately red, orange, yellow, green, blue, indigo, and violet.) *Saturation* is another scale ranging from 0.0 to 1.0, representing light pastels to intense hues. *Brightness* values also range from 0.0 to 1.0, where 1 is bright white and 0 is black. **Color** supplies two methods that let you convert between RGB and HSB. They are shown here:

static int HSBtoRGB(float *hue*, float *saturation*, float *brightness*)
static float[ ] RGBtoHSB(int *red*, int *green*, int *blue*, float *values*[ ])

**HSBtoRGB( )** returns a packed RGB value compatible with the **Color(int)** constructor. **RGBtoHSB( )** returns a **float** array of HSB values corresponding to RGB integers. If *values* is not **null**, then this array is given the HSB values and returned. Otherwise, a new array is created and the HSB values are returned in it. In either case, the array contains the hue at index 0, saturation at index 1, and brightness at index 2.

### getRed( ), getGreen( ), getBlue( )

You can obtain the red, green, and blue components of a color independently using **getRed( )**, **getGreen( )**, and **getBlue( )**, shown here:

int getRed( )
int getGreen( )
int getBlue( )

Each of these methods returns the RGB color component found in the invoking **Color** object in the lower 8 bits of an integer.

### getRGB( )

To obtain a packed, RGB representation of a color, use **getRGB( )**, shown here:

int getRGB( )

The return value is organized as described earlier.

## Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. You can change this color by calling the **Graphics** method **setColor( )** :

void setColor(Color *newColor*)

Here, *newColor* specifies the new drawing color.

You can obtain the current color by calling **getColor( )**, shown here:

Color getColor( )



## A Color Demonstration Applet

The following applet constructs several colors and draws various objects using these colors:

```
// Demonstrate color.
import java.awt.*;
import java.applet.*;
/*
<applet code="ColorDemo" width=300 height=200>
</applet>
*/

public class ColorDemo extends Applet {
  // draw lines
  public void paint(Graphics g) {
    Color c1 = new Color(255, 100, 100);
    Color c2 = new Color(100, 255, 100);
    Color c3 = new Color(100, 100, 255);

    g.setColor(c1);
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);

    g.setColor(c2);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);

    g.setColor(c3);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    g.setColor(Color.red);
    g.drawOval(10, 10, 50, 50);
    g.fillOval(70, 90, 140, 100);
```

```
    g.setColor(Color.blue);
    g.drawOval(190, 10, 90, 30);
    g.drawRect(10, 10, 60, 50);

    g.setColor(Color.cyan);
    g.fillRect(100, 10, 60, 50);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
  }
}
```

## Setting the Paint Mode

The *paint mode* determines how objects are drawn in a window. By default, new output to a window overwrites any preexisting contents. However, it is possible to have new objects XORed onto the window by using **setXORMode( )**, as follows:

void setXORMode(Color *xorColor*)

Here, *xorColor* specifies the color that will be XORed to the window when an object is drawn. The advantage of XOR mode is that the new object is always guaranteed to be visible no matter what color the object is drawn over.

To return to overwrite mode, call **setPaintMode( )**, shown here:

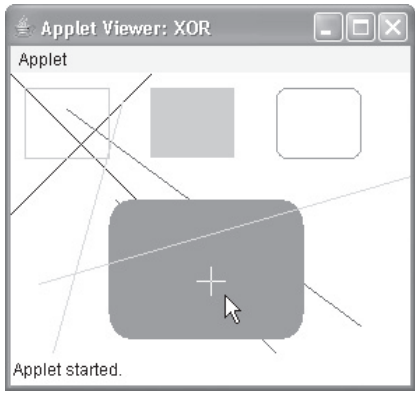
void setPaintMode( )

In general, you will want to use overwrite mode for normal output, and XOR mode for special purposes. For example, the following program displays cross hairs that track the mouse pointer. The cross hairs are XORed onto the window and are always visible, no matter what the underlying color is.

```
// Demonstrate XOR mode.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="XOR" width=400 height=200>
  </applet>
*/

public class XOR extends Applet {
  int chsX=100, chsY=100;

  public XOR() {
    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseMoved(MouseEvent me) {
        int x = me.getX();
        int y = me.getY();
        chsX = x-10;
        chsY = y-10;
        repaint();
      }
    });
  }
```

```
  public void paint(Graphics g) {
    g.drawLine(0, 0, 100, 100);
    g.drawLine(0, 100, 100, 0);
    g.setColor(Color.blue);
    g.drawLine(40, 25, 250, 180);
    g.drawLine(75, 90, 400, 400);
    g.setColor(Color.green);
    g.drawRect(10, 10, 60, 50);
    g.fillRect(100, 10, 60, 50);
    g.setColor(Color.red);
    g.drawRoundRect(190, 10, 60, 50, 15, 15);
    g.fillRoundRect(70, 90, 140, 100, 30, 40);
    g.setColor(Color.cyan);
    g.drawLine(20, 150, 400, 40);
    g.drawLine(5, 290, 80, 19);

    // xor cross hairs
    g.setXORMode(Color.black);
    g.drawLine(chsX-10, chsY, chsX+10, chsY);
    g.drawLine(chsX, chsY-10, chsX, chsY+10);
    g.setPaintMode();
  }
}
```

Sample output from this program is shown here:




## Working with Fonts

The AWT supports multiple type fonts. Years ago, fonts emerged from the domain of traditional typesetting to become an important part of computer-generated documents and displays. The AWT provides flexibility by abstracting font-manipulation operations and allowing for dynamic selection of fonts.

Fonts have a family name, a logical font name, and a face name. The *family name* is the general name of the font, such as Courier. The *logical name* specifies a name, such as Monospaced, that is linked to an actual font at runtime. The *face name* specifies a specific font, such as Courier Italic.

Fonts are encapsulated by the **Font** class. Several of the methods defined by **Font** are listed in Table 25-2.

The **Font** class defines these protected variables:

| Variable | Meaning |
|---|---|
| String name | Name of the font |
| float pointSize | Size of the font in points |
| int size | Size of the font in points |
| int style | Font style |

Several static fields are also defined.

| Method | Description |
|---|---|
| static Font decode(String *str*) | Returns a font given its name. |
| boolean equals(Object *FontObj*) | Returns **true** if the invoking object contains the same font as that specified by *FontObj*. Otherwise, it returns **false**. |
| String getFamily( ) | Returns the name of the font family to which the invoking font belongs. |
| static Font getFont(String *property*) | Returns the font associated with the system property specified by *property*. **null** is returned if *property* does not exist. |
| static Font getFont(String *property*, Font *defaultFont*) | Returns the font associated with the system property specified by *property*. The font specified by *defaultFont* is returned if *property* does not exist. |
| String getFontName() | Returns the face name of the invoking font. |
| String getName( ) | Returns the logical name of the invoking font. |
| int getSize( ) | Returns the size, in points, of the invoking font. |
| int getStyle( ) | Returns the style values of the invoking font. |
| int hashCode( ) | Returns the hash code associated with the invoking object. |
| boolean isBold( ) | Returns **true** if the font includes the **BOLD** style value. Otherwise, **false** is returned. |
| boolean isItalic( ) | Returns **true** if the font includes the **ITALIC** style value. Otherwise, **false** is returned. |
| boolean isPlain( ) | Returns **true** if the font includes the **PLAIN** style value. Otherwise, **false** is returned. |
| String toString( ) | Returns the string equivalent of the invoking font. |

**Table 25-2** A Sampling of Methods Defined by **Font**

## Determining the Available Fonts

When working with fonts, often you need to know which fonts are available on your machine. To obtain this information, you can use the **getAvailableFontFamilyNames( )** method defined by the **GraphicsEnvironment** class. It is shown here:

String[ ] getAvailableFontFamilyNames( )

This method returns an array of strings that contains the names of the available font families.

In addition, the **getAllFonts( )** method is defined by the **GraphicsEnvironment** class. It is shown here:

Font[ ] getAllFonts( )

This method returns an array of **Font** objects for all of the available fonts.

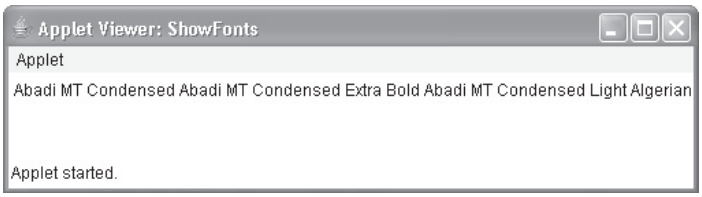
Since these methods are members of **GraphicsEnvironment**, you need a **GraphicsEnvironment** reference to call them. You can obtain this reference by using the **getLocalGraphicsEnvironment( )** static method, which is defined by **GraphicsEnvironment**. It is shown here:

static GraphicsEnvironment getLocalGraphicsEnvironment( )

Here is an applet that shows how to obtain the names of the available font families:

```
// Display Fonts
/*
<applet code="ShowFonts" width=550 height=60>
</applet>
*/
import java.applet.*;
import java.awt.*;

public class ShowFonts extends Applet {
  public void paint(Graphics g) {
    String msg = "";
    String FontList[];

    GraphicsEnvironment ge =
      GraphicsEnvironment.getLocalGraphicsEnvironment();
    FontList = ge.getAvailableFontFamilyNames();
    for(int i = 0; i < FontList.length; i++)
      msg += FontList[i] + " ";

     g.drawString(msg, 4, 16);
  }
}
```

Sample output from this program is shown next. However, when you run this program, you may see a different list of fonts than the one shown in this illustration.




## Creating and Selecting a Font

To create a new font, construct a **Font** object that describes that font. One **Font** constructor has this general form:

Font(String *fontName*, int *fontStyle*, int *pointSize*)

Here, *fontName* specifies the name of the desired font. The name can be specified using either the logical or face name. All Java environments will support the following fonts: Dialog, DialogInput, SansSerif, Serif, and Monospaced. Dialog is the font used by your system's dialog boxes. Dialog is also the default if you don't explicitly set a font. You can also use any other fonts supported by your particular environment, but be careful—these other fonts may not be universally available.

The style of the font is specified by *fontStyle*. It may consist of one or more of these three constants: **Font.PLAIN**, **Font.BOLD**, and **Font.ITALIC**. To combine styles, OR them together. For example, **Font.BOLD | Font.ITALIC** specifies a bold, italics style.

The size, in points, of the font is specified by *pointSize*.

To use a font that you have created, you must select it using **setFont( )**, which is defined by **Component**. It has this general form:

void setFont(Font *fontObj*)

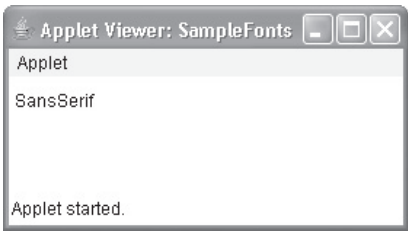
Here, *fontObj* is the object that contains the desired font.

The following program outputs a sample of each standard font. Each time you click the mouse within its window, a new font is selected and its name is displayed.

```
// Show fonts.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
  <applet code="SampleFonts" width=200 height=100>
  </applet>
*/

public class SampleFonts extends Applet {
  int next = 0;
  Font f;
  String msg;
```

```
  public void init() {
    f = new Font("Dialog", Font.PLAIN, 12);
    msg = "Dialog";
    setFont(f);
    addMouseListener(new MyMouseAdapter(this));
  }

  public void paint(Graphics g) {
    g.drawString(msg, 4, 20);
  }
}

class MyMouseAdapter extends MouseAdapter {
  SampleFonts sampleFonts;

  public MyMouseAdapter(SampleFonts sampleFonts) {
    this.sampleFonts = sampleFonts;
  }

  public void mousePressed(MouseEvent me) {
    // Switch fonts with each mouse click.
    sampleFonts.next++;
    switch(sampleFonts.next) {
    case 0:
      sampleFonts.f = new Font("Dialog", Font.PLAIN, 12);
      sampleFonts.msg = "Dialog";
      break;
    case 1:
      sampleFonts.f = new Font("DialogInput", Font.PLAIN, 12);
      sampleFonts.msg = "DialogInput";
      break;
    case 2:
      sampleFonts.f = new Font("SansSerif", Font.PLAIN, 12);
      sampleFonts.msg = "SansSerif";
      break;
    case 3:
      sampleFonts.f = new Font("Serif", Font.PLAIN, 12);
      sampleFonts.msg = "Serif";
      break;
    case 4:
      sampleFonts.f = new Font("Monospaced", Font.PLAIN, 12);
      sampleFonts.msg = "Monospaced";
      break;
    }

    if(sampleFonts.next == 4) sampleFonts.next = -1;

    sampleFonts.setFont(sampleFonts.f);
    sampleFonts.repaint();
  }
}
```

Sample output from this program is shown here:




## Obtaining Font Information

Suppose you want to obtain information about the currently selected font. To do this, you must first get the current font by calling **getFont( )**. This method is defined by the **Graphics** class, as shown here:

```
Font getFont( )
```

Once you have obtained the currently selected font, you can retrieve information about it using various methods defined by **Font**. For example, this applet displays the name, family, size, and style of the currently selected font:

```
// Display font info.
import java.applet.*;
import java.awt.*;
/*
<applet code="FontInfo" width=350 height=60>
</applet>
*/

public class FontInfo extends Applet {
  public void paint(Graphics g) {
    Font f = g.getFont();
    String fontName = f.getName();
    String fontFamily = f.getFamily();
    int fontSize = f.getSize();
    int fontStyle = f.getStyle();

    String msg = "Family: " + fontName;
    msg += ", Font: " + fontFamily;
    msg += ", Size: " + fontSize + ", Style: ";
    if((fontStyle & Font.BOLD) == Font.BOLD)
      msg += "Bold ";
    if((fontStyle & Font.ITALIC) == Font.ITALIC)
      msg += "Italic ";
    if((fontStyle & Font.PLAIN) == Font.PLAIN)
      msg += "Plain ";

    g.drawString(msg, 4, 16);
  }
}
```

## Managing Text Output Using FontMetrics

As just explained, Java supports a number of fonts. For most fonts, characters are not all the same dimension—most fonts are proportional. Also, the height of each character, the length of *descenders* (the hanging parts of letters, such as *y*), and the amount of space between horizontal lines vary from font to font. Further, the point size of a font can be changed. That these (and other) attributes are variable would not be of too much consequence except that Java demands that you, the programmer, manually manage virtually all text output.

Given that the size of each font may differ and that fonts may be changed while your program is executing, there must be some way to determine the dimensions and various other attributes of the currently selected font. For example, to write one line of text after another implies that you have some way of knowing how tall the font is and how many pixels are needed between lines. To fill this need, the AWT includes the **FontMetrics** class, which encapsulates various information about a font. Let's begin by defining the common terminology used when describing fonts:



| | |
|---|---|
| Height | The top-to-bottom size of a line of text |
| Baseline | The line that the bottoms of characters are aligned to (not counting descent) |
| Ascent | The distance from the baseline to the top of a character |
| Descent | The distance from the baseline to the bottom of a character |
| Leading | The distance between the bottom of one line of text and the top of the next |

As you know, we have used the **drawString( )** method in many of the previous examples. It paints a string in the current font and color, beginning at a specified location. However, this location is at the left edge of the baseline of the characters, not at the upper-left corner as is usual with other drawing methods. It is a common error to draw a string at the same coordinate that you would draw a box. For example, if you were to draw a rectangle at coordinate 0,0, you would see a full rectangle. If you were to draw the string "Typesetting" at 0,0, you would only see the tails (or descenders) of the *y, p,* and *g*. As you will see, by using font metrics, you can determine the proper placement of each string that you display.

**FontMetrics** defines several methods that help you manage text output. Several commonly used ones are listed in Table 25-3. These methods help you properly display text in a window. Let's look at some examples.

### Displaying Multiple Lines of Text

Perhaps the most common use of **FontMetrics** is to determine the spacing between lines of text. The second most common use is to determine the length of a string that is being displayed. Here, you will see how to accomplish these tasks.

In general, to display multiple lines of text, your program must manually keep track of the current output position. Each time a newline is desired, the Y coordinate must be advanced to the beginning of the next line. Each time a string is displayed, the X coordinate must be set to the point at which the string ends. This allows the next string to be written so that it begins at the end of the preceding one.

| Method | Description |
|---|---|
| int bytesWidth(byte *b*[ ], int *start*, int *numBytes*) | Returns the width of *numBytes* characters held in array *b*, beginning at *start*. |
| int charWidth(char *c*[ ], int *start*, int *numChars*) | Returns the width of *numChars* characters held in array *c*, beginning at *start*. |
| int charWidth(char *c*) | Returns the width of *c*. |
| int charWidth(int *c*) | Returns the width of *c*. |
| int getAscent( ) | Returns the ascent of the font. |
| int getDescent( ) | Returns the descent of the font. |
| Font getFont( ) | Returns the font. |
| int getHeight( ) | Returns the height of a line of text. This value can be used to output multiple lines of text in a window. |
| int getLeading( ) | Returns the space between lines of text. |
| int getMaxAdvance( ) | Returns the width of the widest character. –1 is returned if this value is not available. |
| int getMaxAscent( ) | Returns the maximum ascent. |
| int getMaxDescent( ) | Returns the maximum descent. |
| int[ ] getWidths( ) | Returns the widths of the first 256 characters. |
| int stringWidth(String *str*) | Returns the width of the string specified by *str*. |
| String toString( ) | Returns the string equivalent of the invoking object. |

**Table 25-3** A Sampling of Methods Defined by **FontMetrics**

To determine the spacing between lines, you can use the value returned by **getLeading( )**. To determine the total height of the font, add the value returned by **getAscent( )** to the value returned by **getDescent( )**. You can then use these values to position each line of text you output. However, in many cases, you will not need to use these individual values. Often, all that you will need to know is the total height of a line, which is the sum of the leading space and the font's ascent and descent values. The easiest way to obtain this value is to call **getHeight( )**. Simply increment the Y coordinate by this value each time you want to advance to the next line when outputting text.

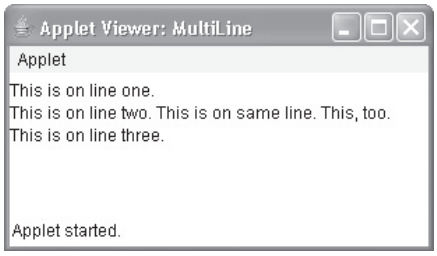
To start output at the end of previous output on the same line, you must know the length, in pixels, of each string that you display. To obtain this value, call **stringWidth( )**. You can use this value to advance the X coordinate each time you display a line.

The following applet shows how to output multiple lines of text in a window. It also displays multiple sentences on the same line. Notice the variables **curX** and **curY**. They keep track of the current text output position.

```
// Demonstrate multiline output.
import java.applet.*;
import java.awt.*;
/*
<applet code="MultiLine" width=300 height=100>
</applet>
*/
```

```
public class MultiLine extends Applet {
  int curX=0, curY=0; // current position

  public void init() {
    Font f = new Font("SansSerif", Font.PLAIN, 12);
    setFont(f);
  }

  public void paint(Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    nextLine("This is on line one.", g);
    nextLine("This is on line two.", g);
    sameLine(" This is on same line.", g);
    sameLine(" This, too.", g);
    nextLine("This is on line three.", g);
    curX = curY = 0; // Reset coordinates for each repaint.
  }

  // Advance to next line.
  void nextLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    curY += fm.getHeight(); // advance to next line
    curX = 0;
    g.drawString(s, curX, curY);
    curX = fm.stringWidth(s); // advance to end of line
  }

  // Display on same line.
  void sameLine(String s, Graphics g) {
    FontMetrics fm = g.getFontMetrics();

    g.drawString(s, curX, curY);
    curX += fm.stringWidth(s); // advance to end of line
  }
}
```

Sample output from this program is shown here:



## Centering Text

Here is an example that centers text, left to right, top to bottom, in a window. It obtains the ascent, descent, and width of the string and computes the position at which it must be displayed to be centered.

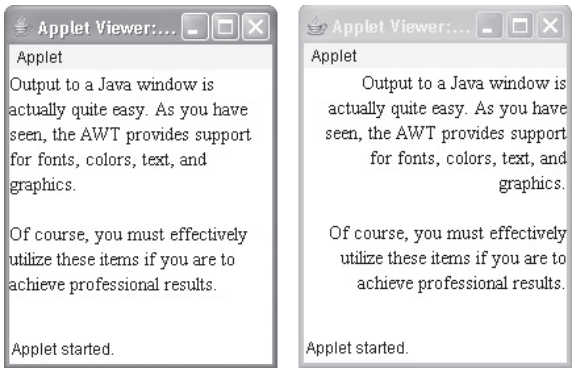
```
// Center text.
import java.applet.*;
import java.awt.*;
/*
  <applet code="CenterText" width=200 height=100>
  </applet>
*/

public class CenterText extends Applet {
  final Font f = new Font("SansSerif", Font.BOLD, 18);

  public void paint(Graphics g) {
    Dimension d = this.getSize();

    g.setColor(Color.white);
    g.fillRect(0, 0, d.width,d.height);
    g.setColor(Color.black);
    g.setFont(f);
    drawCenteredString("This is centered.", d.width, d.height, g);
    g.drawRect(0, 0, d.width-1, d.height-1);
  }

  public void drawCenteredString(String s, int w, int h,
                                 Graphics g) {
    FontMetrics fm = g.getFontMetrics();
    int x = (w - fm.stringWidth(s)) / 2;
    int y = (fm.getAscent() + (h - (fm.getAscent()
             + fm.getDescent()))/2);
    g.drawString(s, x, y);
  }
}
```

Following is a sample output from this program:

## Multiline Text Alignment

When using a word processor, it is common for text to be aligned so that one or more of the edges of the text make a straight line. For example, most word processors can left-justify and/or right-justify text. Most can also center text. In the following program, you will see how to accomplish these actions.

In the program, the string to be justified is broken into individual words. For each word, the program keeps track of its length in the current font and automatically advances to the next line if the word will not fit on the current line. Each completed line is displayed in the window in the currently selected alignment style. Each time you click the mouse in the applet's window, the alignment style is changed. Sample output from this program is shown here:




```
// Demonstrate text alignment.
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
/* <title>Text Layout</title>
   <applet code="TextLayout" width=200 height=200>
   <param name="text" value="Output to a Java window is actually
      quite easy.
      As you have seen, the AWT provides support for
      fonts, colors, text, and graphics. <P> Of course,
      you must effectively utilize these items
      if you are to achieve professional results.">
   <param name="fontname" value="Serif">
   <param name="fontSize" value="14">
  </applet>
*/

public class TextLayout extends Applet {
  final int LEFT = 0;
  final int RIGHT = 1;
  final int CENTER = 2;
  final int LEFTRIGHT =3;
  int align;
```

```
Dimension d;
Font f;
FontMetrics fm;
int fontSize;
int fh, bl;
int space;
String text;

public void init() {
  setBackground(Color.white);
  text = getParameter("text");
  try {
    fontSize = Integer.parseInt(getParameter("fontSize"));}
  catch (NumberFormatException e) {
    fontSize=14;
  }
  align = LEFT;
  addMouseListener(new MyMouseAdapter(this));
}

public void paint(Graphics g) {
  update(g);
}

public void update(Graphics g) {
  d = getSize();
  g.setColor(getBackground());
  g.fillRect(0,0,d.width, d.height);
  if(f==null) f = new Font(getParameter("fontname"),
                           Font.PLAIN, fontSize);
  g.setFont(f);
  if(fm == null) {
      fm = g.getFontMetrics();
      bl = fm.getAscent();
      fh = bl + fm.getDescent();
      space = fm.stringWidth(" ");
  }

  g.setColor(Color.black);
  StringTokenizer st = new StringTokenizer(text);
  int x = 0;
  int nextx;
  int y = 0;
  String word, sp;
  int wordCount = 0;
  String line = "";
  while (st.hasMoreTokens()) {
    word = st.nextToken();
    if(word.equals("<P>")) {
      drawString(g, line, wordCount,
                 fm.stringWidth(line), y+bl);
      line = "";
      wordCount = 0;
```

```
      x = 0;
      y = y + (fh * 2);
    }
    else {
      int w = fm.stringWidth(word);
      if(( nextx = (x+space+w)) > d.width ) {
        drawString(g, line, wordCount,
                   fm.stringWidth(line), y+bl);
        line = "";
        wordCount = 0;
        x = 0;
        y = y + fh;
      }
      if(x!=0) {sp = " ";} else {sp = "";}
      line = line + sp + word;
      x = x + space + w;
      wordCount++;
    }
  }
  drawString(g, line, wordCount, fm.stringWidth(line), y+bl);
}

public void drawString(Graphics g, String line,
                       int wc, int lineW, int y) {
  switch(align) {
    case LEFT: g.drawString(line, 0, y);
      break;
    case RIGHT: g.drawString(line, d.width-lineW,y);
      break;
    case CENTER: g.drawString(line, (d.width-lineW)/2, y);
      break;
    case LEFTRIGHT:
      if(lineW < (int)(d.width*.75)) {
         g.drawString(line, 0, y);
      }
      else {
        int toFill = (d.width - lineW)/wc;
        int nudge = d.width - lineW - (toFill*wc);
        int s = fm.stringWidth(" ");
        StringTokenizer st = new StringTokenizer(line);
        int x = 0;
        while(st.hasMoreTokens()) {
          String word = st.nextToken();
          g.drawString(word, x, y);
          if(nudge>0) {
            x = x + fm.stringWidth(word) + space + toFill + 1;
            nudge--;
          } else {
            x = x + fm.stringWidth(word) + space + toFill;
          }
        }
      }
      break;
```

```
    }
  }
}

class MyMouseAdapter extends MouseAdapter {
  TextLayout tl;

  public MyMouseAdapter(TextLayout tl) {
    this.tl = tl;
  }

  public void mouseClicked(MouseEvent me) {
    tl.align = (tl.align + 1) % 4;
    tl.repaint();
  }
}
```

Let's take a closer look at how this applet works. The applet first creates several constants that will be used to determine the alignment style, and then declares several variables. The **init( )** method obtains the text that will be displayed. It then initializes the font size in a **try-catch** block, which will set the font size to 14 if the **fontSize** parameter is missing from the HTML. The **text** parameter is a long string of text, with the HTML tag **<P>** as a paragraph separator.

The **update( )** method is the engine for this example. It sets the font and gets the baseline and font height from a font metrics object. Next, it creates a **StringTokenizer** and uses it to retrieve the next token (a string separated by whitespace) from the string specified by **text**. If the next token is **<P>**, it advances the vertical spacing. Otherwise, **update( )** checks to see if the length of this token in the current font will go beyond the width of the column. If the line is full of text or if there are no more tokens, the line is output by a custom version of **drawString( )**.

The first three cases in **drawString( )** are simple. Each aligns the string that is passed in **line** to the left or right edge or to the center of the column, depending upon the alignment style. The **LEFTRIGHT** case aligns both the left and right sides of the string. This means that we need to calculate the remaining whitespace (the difference between the width of the string and the width of the column) and distribute that space between each of the words. The last method in this class advances the alignment style each time you click the mouse on the applet's window.

CHAPTER

# 26 Using AWT Controls, Layout Managers, and Menus

This chapter continues our overview of the Abstract Window Toolkit (AWT). It begins with a look at several of the AWT's controls and layout managers. It then discusses menus and the menu bar. The chapter also includes a discussion of two high-level components: the dialog box and the file dialog box. It concludes with another look at event handling.

*Controls* are components that allow a user to interact with your application in various ways—for example, a commonly used control is the push button. A *layout manager* automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

In addition to the controls, a frame window can also include a standard-style *menu bar*. Each entry in a menu bar activates a drop-down menu of options from which the user can choose. This constitutes the *main menu* of an application. As a general rule, a menu bar is positioned at the top of a window. Although different in appearance, menu bars are handled in much the same way as are the other controls.

While it is possible to manually position components within a window, doing so is quite tedious. The layout manager automates this task. For the first part of this chapter, which introduces various controls, the default layout manager will be used. This displays components in a container using left-to-right, top-to-bottom organization. Once the controls have been covered, several layout managers will be examined. There, you will see ways to better manage the positioning of controls.

Before continuing, it is important to emphasize that today you will seldom create GUIs based solely on the AWT because more powerful GUI frameworks (Swing and JavaFX) have been developed for Java. However, the material presented here remains important for the following reasons. First, much of the information and many of the techniques related to controls and event handling are generalizable to the other Java GUI frameworks. (As mentioned in the previous chapter, Swing is built upon the AWT.) Second, the layout managers described here can also be used by Swing. Third, for some small applications, the AWT components might be the appropriate choice. Finally, and perhaps most importantly, you may need to maintain or upgrade legacy code that uses the AWT. Therefore, a basic understanding of the AWT is important for all Java programmers.

# AWT Control Fundamentals

The AWT supports the following types of controls:

- Labels
- Push buttons
- Check boxes
- Choice lists
- Lists
- Scroll bars
- Text Editing

These controls are subclasses of **Component**. Although this is not a particularly rich set of controls, it is sufficient for simple applications. (Note that both Swing and JavaFX provide a substantially larger, more sophisticated set of controls.)

## Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add( )**, which is defined by **Container**. The **add( )** method has several forms. The following form is the one that is used for the first part of this chapter:

Component add(Component *compRef*)

Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.

Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call **remove( )**. This method is also defined by **Container**. Here is one of its forms:

void remove(Component *compRef*)

Here, *compRef* is a reference to the control you want to remove. You can remove all controls by calling **removeAll( )**.

## Responding to Controls

Except for labels, which are passive, all other controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 24, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.

## The HeadlessException

Most of the AWT controls described in this chapter have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). You can use this exception to write code that can adapt to non-interactive environments. (Of course, this is not always possible.) This exception is not handled by the programs in this chapter because an interactive environment is required to demonstrate the AWT controls.



# Labels

The easiest control to use is a label. A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label( ) throws HeadlessException
Label(String *str*) throws HeadlessException
Label(String *str*, int *how*) throws HeadlessException

The first version creates a blank label. The second version creates a label that contains the string specified by *str*. This string is left-justified. The third version creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT**, **Label.RIGHT**, or **Label.CENTER**.

You can set or change the text in a label by using the **setText( )** method. You can obtain the current label by calling **getText( )**. These methods are shown here:

void setText(String *str*)
String getText( )

For **setText( )**, *str* specifies the new label. For **getText( )**, the current label is returned.

You can set the alignment of the string within the label by calling **setAlignment( )**. To obtain the current alignment, call **getAlignment( )**. The methods are as follows:

void setAlignment(int *how*)
int getAlignment( )

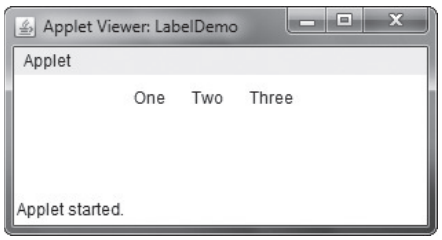
Here, *how* must be one of the alignment constants shown earlier.

The following example creates three labels and adds them to an applet window:

```
// Demonstrate Labels
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/

public class LabelDemo extends Applet {
  public void init() {
    Label one = new Label("One");
    Label two = new Label("Two");
    Label three = new Label("Three");
```

```
    // add labels to applet window
    add(one);
    add(two);
    add(three);
  }
}
```

Here is sample output from the **LabelDemo** applet. Notice that the labels are organized in the window by the default layout manager. Later, you will see how to control more precisely the placement of the labels.




## Using Buttons

Perhaps the most widely used control is the push button. A *push button* is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button**. **Button** defines these two constructors:

Button( ) throws HeadlessException
Button(String *str*) throws HeadlessException

The first version creates an empty button. The second creates a button that contains *str* as a label.

After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String *str*)
String getLabel( )

Here, *str* becomes the new label for the button.

### Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed( )** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button. By default, the action command string is the label of the button. Either the button reference or the action command string can be used to identify the button. (You will soon see examples of each approach.)

Here is an example that creates three buttons labeled "Yes", "No", and "Undecided". Each time one is pressed, a message is displayed that reports which button has been pressed. In this version, the action command of the button (which, by default, is its label) is used to determine which button has been pressed. The label is obtained by calling the **getActionCommand( )** method on the **ActionEvent** object passed to **actionPerformed( )**.

```
// Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ButtonDemo" width=250 height=150>
  </applet>
*/

public class ButtonDemo extends Applet implements ActionListener {
  String msg = "";
  Button yes, no, maybe;

  public void init() {
    yes = new Button("Yes");
    no = new Button("No");
    maybe = new Button("Undecided");

    add(yes);
    add(no);
    add(maybe);

    yes.addActionListener(this);
    no.addActionListener(this);
    maybe.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    String str = ae.getActionCommand();

    if(str.equals("Yes")) {
      msg = "You pressed Yes.";
    }
    else if(str.equals("No")) {
      msg = "You pressed No.";
    }
    else {
      msg = "You pressed Undecided.";
    }

    repaint();
  }

  public void paint(Graphics g) {
     g.drawString(msg, 6, 100);
  }
}
```

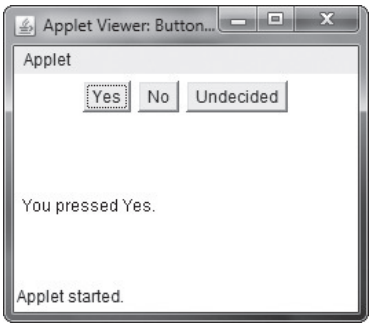



**Figure 26-1** Sample output from the **ButtonDemo** applet

Sample output from the **ButtonDemo** program is shown in Figure 26-1.

As mentioned, in addition to comparing button action command strings, you can also determine which button has been pressed by comparing the object obtained from the **getSource( )** method to the button objects that you added to the window. To do this, you must keep a list of the objects when they are added. The following applet shows this approach:

```
// Recognize Button objects.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ButtonList" width=250 height=150>
  </applet>
*/

public class ButtonList extends Applet implements ActionListener {
  String msg = "";
  Button bList[] = new Button[3];

  public void init() {
    Button yes = new Button("Yes");
    Button no = new Button("No");
    Button maybe = new Button("Undecided");

    // store references to buttons as added
    bList[0] = (Button) add(yes);
    bList[1] = (Button) add(no);
    bList[2] = (Button) add(maybe);

    // register to receive action events
    for(int i = 0; i < 3; i++) {
      bList[i].addActionListener(this);
    }
  }
```

```
  public void actionPerformed(ActionEvent ae) {
    for(int i = 0; i < 3; i++) {
      if(ae.getSource() == bList[i]) {
        msg = "You pressed " + bList[i].getLabel();
      }
    }
    repaint();
  }

  public void paint(Graphics g) {
     g.drawString(msg, 6, 100);
  }
}
```



In this version, the program stores each button reference in an array when the buttons are added to the applet window. (Recall that the **add( )** method returns a reference to the button when it is added.) Inside **actionPerformed( )**, this array is then used to determine which button has been pressed.

For simple programs, it is usually easier to recognize buttons by their labels. However, in situations in which you will be changing the label inside a button during the execution of your program, or using buttons that have the same label, it may be easier to determine which button has been pushed by using its object reference. It is also possible to set the action command string associated with a button to something other than its label by calling **setActionCommand( )**. This method changes the action command string, but does not affect the string used to label the button. Thus, setting the action command enables the action command and the label of a button to differ.

In some cases, you can handle the action events generated by a button (or some other type of control) by use of an anonymous inner class (as described in Chapter 24) or a lambda expression (discussed in Chapter 15). For example, assuming the previous programs, here is a set of action event handlers that use lambda expressions:

```
// Use lambda expressions to handle action events.
yes.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

no.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});

maybe.addActionListener((ae) -> {
  msg = "You pressed " + ae.getActionCommand();
  repaint();
});
```

This code works because **ActionListener** defines a functional interface, which is an interface with exactly one abstract method. Thus, it can be used by a lambda expression. In general, you can use a lambda expression to handle an AWT event when its listener defines a functional interface. For example, **ItemListener** is also a functional interface. Of course,

whether you use the traditional approach, an anonymous inner class, or a lambda expression will be determined by the precise nature of your application. The remaining examples in this chapter use the traditional approach to event handling so that they can be compiled by nearly any version of Java. However, you might find it interesting to try converting the event handlers to lambda expressions or anonymous inner classes, where appropriate.

## Applying Check Boxes

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

**Checkbox** supports these constructors:

```
Checkbox( ) throws HeadlessException
Checkbox(String str) throws HeadlessException
Checkbox(String str, boolean on) throws HeadlessException
Checkbox(String str, boolean on, CheckboxGroup cbGroup) throws HeadlessException
Checkbox(String str, CheckboxGroup cbGroup, boolean on) throws HeadlessException
```

The first form creates a check box whose label is initially blank. The state of the check box is unchecked. The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked. The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared. The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. (Check box groups are described in the next section.) The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. You can obtain the current label associated with a check box by calling **getLabel( )**. To set the label, call **setLabel( )**. These methods are as follows:

```
boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)
```

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

### Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CheckboxDemo" width=240 height=200>
  </applet>
*/

public class CheckboxDemo extends Applet implements ItemListener {
  String msg = "";
  Checkbox windows, android, solaris, mac;

  public void init() {
    windows = new Checkbox("Windows", null, true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac OS: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Sample output is shown in Figure 26-2.

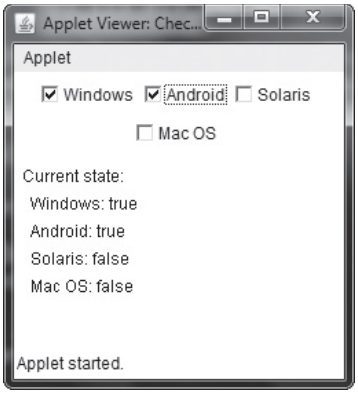



**Figure 26-2** Sample output from the **CheckboxDemo** applet

## CheckboxGroup

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called *radio buttons,* because they act like the station selector on a car radio—only one station can be selected at any one time. To create a set of mutually exclusive check boxes, you must first define the group to which they will belong and then specify that group when you construct the check boxes. Check box groups are objects of type **CheckboxGroup**. Only the default constructor is defined, which creates an empty group.

You can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. You can set a check box by calling **setSelectedCheckbox( )**. These methods are as follows:

Checkbox getSelectedCheckbox( )
void setSelectedCheckbox(Checkbox *which*)

Here, *which* is the check box that you want to be selected. The previously selected check box will be turned off.

Here is a program that uses check boxes that are part of a group:

```
// Demonstrate check box group.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CBGroup" width=240 height=200>
  </applet>
*/

public class CBGroup extends Applet implements ItemListener {
  String msg = "";
  Checkbox windows, android, solaris, mac;
  CheckboxGroup cbg;
```

```
  public void init() {
    cbg = new CheckboxGroup();
    windows = new Checkbox("Windows", cbg, true);
    android = new Checkbox("Android", cbg, false);
    solaris = new Checkbox("Solaris", cbg, false);
    mac = new Checkbox("Mac OS", cbg, false);

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current selection: ";
    msg += cbg.getSelectedCheckbox().getLabel();
    g.drawString(msg, 6, 100);
  }
}
```

Sample output generated by the **CBGroup** applet is shown in Figure 26-3. Notice that the check boxes are now circular in shape.


**Figure 26-3** Sample output from the **CBGroup** applet

## Choice Controls

The **Choice** class is used to create a *pop-up list* of items from which the user may choose. Thus, a **Choice** control is a form of menu. When inactive, a **Choice** component takes up only enough space to show the currently selected item. When the user clicks on it, the whole list of choices pops up, and a new selection can be made. Each item in the list is a string that appears as a left-justified label in the order it is added to the **Choice** object. **Choice** defines only the default constructor, which creates an empty list.

To add a selection to the list, call **add( )**. It has this general form:

void add(String *name*)

Here, *name* is the name of the item being added. Items are added to the list in the order in which calls to **add( )** occur.

To determine which item is currently selected, you may call either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item using the **select( )** method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

int getItemCount( )
void select(int *index*)
void select(String *name*)

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

### Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method.

Here is an example that creates two **Choice** menus. One selects the operating system. The other selects the browser.

```
// Demonstrate Choice lists.
import java.awt.*;
import java.awt.event.*;
```

```
import java.applet.*;
/*
  <applet code="ChoiceDemo" width=300 height=180>
  </applet>
*/

public class ChoiceDemo extends Applet implements ItemListener {
  Choice os, browser;
  String msg = "";

  public void init() {
    os = new Choice();
    browser = new Choice();

    // add items to os list
    os.add("Windows");
    os.add("Android");
    os.add("Solaris");
    os.add("Mac OS");

    // add items to browser list
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Chrome");

    // add choice lists to window
    add(os);
    add(browser);

    // register to receive item events
    os.addItemListener(this);
    browser.addItemListener(this);
  }

  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    msg = "Current OS: ";
    msg += os.getSelectedItem();
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

Sample output is shown in Figure 26-4.

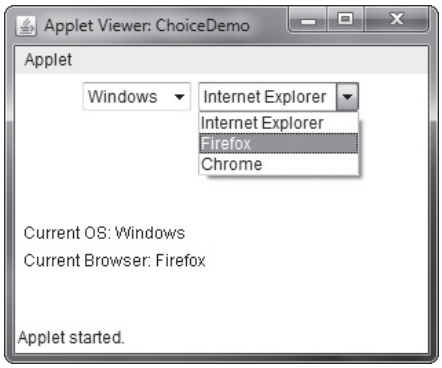



**Figure 26-4** Sample output from the **ChoiceDemo** applet

## Using Lists

The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

List( ) throws HeadlessException
List(int *numRows*) throws HeadlessException
List(int *numRows*, boolean *multipleSelect*) throws HeadlessException

The first version creates a **List** control that allows only one item to be selected at any one time. In the second form, the value of *numRows* specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed). In the third form, if *multipleSelect* is **true**, then the user may select two or more items at a time. If it is **false**, then only one item may be selected.

To add a selection to the list, call **add( )**. It has the following two forms:

void add(String *name*)
void add(String *name*, int *index*)

Here, *name* is the name of the item added to the list. The first form adds items to the end of the list. The second form adds the item at the index specified by *index*. Indexing begins at zero. You can specify –1 to add the item to the end of the list.

For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes( )**, shown here, to determine the current selections:

String[ ] getSelectedItems( )
int[ ] getSelectedIndexes( )

**getSelectedItems( )** returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods are shown here:

int getItemCount( )
void select(int *index*)

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

String getItem(int *index*)

Here, *index* specifies the index of the desired item.

## Handling Lists

To process list events, you will need to implement the **ActionListener** interface. Each time a **List** item is double-clicked, an **ActionEvent** object is generated. Its **getActionCommand( )** method can be used to retrieve the name of the newly selected item. Also, each time an item is selected or deselected with a single click, an **ItemEvent** object is generated. Its **getStateChange( )** method can be used to determine whether a selection or deselection triggered this event. **getItemSelectable( )** returns a reference to the object that triggered this event.

Here is an example that converts the **Choice** controls in the preceding section into **List** components, one multiple choice and the other single choice:

```
// Demonstrate Lists.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="ListDemo" width=300 height=180>
  </applet>
*/

public class ListDemo extends Applet implements ActionListener {
  List os, browser;
  String msg = "";

  public void init() {
    os = new List(4, true);
    browser = new List(4, false);

    // add items to os list
    os.add("Windows");
```

```
    os.add("Android");
    os.add("Solaris");
    os.add("Mac OS");

    // add items to browser list
    browser.add("Internet Explorer");
    browser.add("Firefox");
    browser.add("Chrome");

    browser.select(1);

    // add lists to window
    add(os);
    add(browser);

    // register to receive action events
    os.addActionListener(this);
    browser.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  // Display current selections.
  public void paint(Graphics g) {
    int idx[];

    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
      msg += os.getItem(idx[i]) + "  ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
  }
}
```

Sample output generated by the **ListDemo** applet is shown in Figure 26-5.

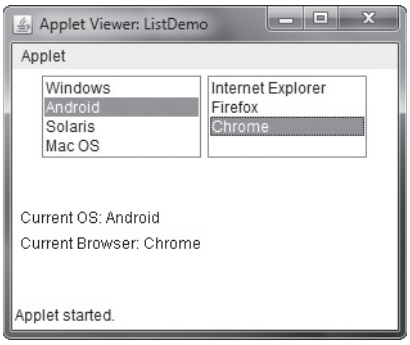



**Figure 26-5** Sample output from the **ListDemo** applet

## Managing Scroll Bars

*Scroll bars* are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box* (or *thumb*) for the scroll bar. The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1. Typically, this action translates into some form of page up and page down. Scroll bars are encapsulated by the **Scrollbar** class.

**Scrollbar** defines the following constructors:

```
Scrollbar( ) throws HeadlessException
Scrollbar(int style) throws HeadlessException
Scrollbar(int style, int initialValue, int thumbSize, int min, int max)
    throws HeadlessException
```

The first form creates a vertical scroll bar. The second and third forms allow you to specify the orientation of the scroll bar. If *style* is **Scrollbar.VERTICAL**, a vertical scroll bar is created. If *style* is **Scrollbar.HORIZONTAL**, the scroll bar is horizontal. In the third form of the constructor, the initial value of the scroll bar is passed in *initialValue*. The number of units represented by the height of the thumb is passed in *thumbSize*. The minimum and maximum values for the scroll bar are specified by *min* and *max*.

If you construct a scroll bar by using one of the first two constructors, then you need to set its parameters by using **setValues( )**, shown here, before it can be used:

```
void setValues(int initialValue, int thumbSize, int min, int max)
```

The parameters have the same meaning as they have in the third constructor just described.

To obtain the current value of the scroll bar, call **getValue( )**. It returns the current setting. To set the current value, call **setValue( )**. These methods are as follows:

```
int getValue( )
void setValue(int newValue)
```

Here, *newValue* specifies the new value for the scroll bar. When you set a value, the slider box inside the scroll bar will be positioned to reflect the new value.

You can also retrieve the minimum and maximum values via **getMinimum( )** and **getMaximum( )**, shown here:

```
int getMinimum( )
int getMaximum( )
```

They return the requested quantity.

By default, 1 is the increment added to or subtracted from the scroll bar each time it is scrolled up or down one line. You can change this increment by calling **setUnitIncrement( )**. By default, page-up and page-down increments are 10. You can change this value by calling **setBlockIncrement( )**. These methods are shown here:

```
void setUnitIncrement(int newIncr)
void setBlockIncrement(int newIncr)
```

## Handling Scroll Bars

To process scroll bar events, you need to implement the **AdjustmentListener** interface. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType( )** method can be used to determine the type of the adjustment. The types of adjustment events are as follows:

| | |
|---|---|
| BLOCK_DECREMENT | A page-down event has been generated. |
| BLOCK_INCREMENT | A page-up event has been generated. |
| TRACK | An absolute tracking event has been generated. |
| UNIT_DECREMENT | The line-down button in a scroll bar has been pressed. |
| UNIT_INCREMENT | The line-up button in a scroll bar has been pressed. |

The following example creates both a vertical and a horizontal scroll bar. The current settings of the scroll bars are displayed. If you drag the mouse while inside the window, the coordinates of each drag event are used to update the scroll bars. An asterisk is displayed at the current drag position. Notice the use of **setPreferredSize( )** to set the size of the scrollbars.

```
// Demonstrate scroll bars.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SBDemo" width=300 height=200>
  </applet>
*/

public class SBDemo extends Applet
  implements AdjustmentListener, MouseMotionListener {
  String msg = "";
  Scrollbar vertSB, horzSB;

  public void init() {
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    vertSB = new Scrollbar(Scrollbar.VERTICAL,
                           0, 1, 0, height);
    vertSB.setPreferredSize(new Dimension(20, 100));

    horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
                            0, 1, 0, width);
    horzSB.setPreferredSize(new Dimension(100, 20));

    add(vertSB);
    add(horzSB);

    // register to receive adjustment events
```

```
    vertSB.addAdjustmentListener(this);
    horzSB.addAdjustmentListener(this);

    addMouseMotionListener(this);
  }

  public void adjustmentValueChanged(AdjustmentEvent ae) {
    repaint();
  }

  // Update scroll bars to reflect mouse dragging.
  public void mouseDragged(MouseEvent me) {
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
  }

  // Necessary for MouseMotionListener
  public void mouseMoved(MouseEvent me) {
  }

  // Display current value of scroll bars.
  public void paint(Graphics g) {
     msg = "Vertical: " + vertSB.getValue();
     msg += ",  Horizontal: " + horzSB.getValue();
     g.drawString(msg, 6, 160);

     // show current mouse drag position
     g.drawString("*", horzSB.getValue(),
                  vertSB.getValue());
  }
}
```

Sample output from the **SBDemo** applet is shown in Figure 26-6.

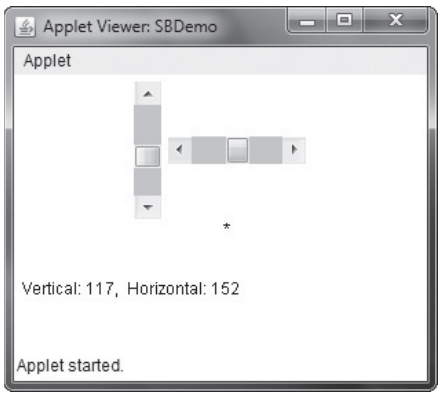



**Figure 26-6** Sample output from the **SBDemo** applet

## Using a TextField

The **TextField** class implements a single-line text-entry area, usually called an *edit control*. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

```
TextField( ) throws HeadlessException
TextField(int numChars) throws HeadlessException
TextField(String str) throws HeadlessException
TextField(String str, int numChars) throws HeadlessException
```

The first version creates a default text field. The second form creates a text field that is *numChars* characters wide. The third form initializes the text field with the string contained in *str*. The fourth form initializes a text field and sets its width.

**TextField** (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText( )**. To set the text, call **setText( )**. These methods are as follows:

```
String getText( )
void setText(String str)
```

Here, *str* is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using **select( )**. Your program can obtain the currently selected text by calling **getSelectedText( )**. These methods are shown here:

```
String getSelectedText( )
void select(int startIndex, int endIndex)
```

**getSelectedText( )** returns the selected text. The **select( )** method selects the characters beginning at *startIndex* and ending at *endIndex* –1.

You can control whether the contents of a text field may be modified by the user by calling **setEditable( )**. You can determine editability by calling **isEditable( )**. These methods are shown here:

```
boolean isEditable( )
void setEditable(boolean canEdit)
```

**isEditable( )** returns **true** if the text may be changed and **false** if not. In **setEditable( )**, if *canEdit* is **true**, the text may be changed. If it is **false**, the text cannot be altered.

There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling **setEchoChar( )**. This method specifies a single character that the **TextField** will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the **echoCharIsSet( )** method. You can retrieve the echo character by calling the **getEchoChar( )** method. These methods are as follows:

```
void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )
```

Here, *ch* specifies the character to be echoed. If *ch* is zero, then normal echoing is restored.

## Handling a TextField

Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Here is an example that creates the classic user name and password screen:

```
// Demonstrate text field.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="TextFieldDemo" width=380 height=150>
  </applet>
*/

public class TextFieldDemo extends Applet
  implements ActionListener {

  TextField name, pass;

  public void init() {
    Label namep = new Label("Name: ", Label.RIGHT);
    Label passp = new Label("Password: ", Label.RIGHT);
    name = new TextField(12);
    pass = new TextField(8);
    pass.setEchoChar('?');

    add(namep);
    add(name);
    add(passp);
    add(pass);

    // register to receive action events
    name.addActionListener(this);
    pass.addActionListener(this);
  }

  // User pressed Enter.
  public void actionPerformed(ActionEvent ae) {
    repaint();
  }

  public void paint(Graphics g) {
    g.drawString("Name: " + name.getText(), 6, 60);
    g.drawString("Selected text in name: "
                 + name.getSelectedText(), 6, 80);
    g.drawString("Password: " + pass.getText(), 6, 100);
  }
}
```

Sample output from the **TextFieldDemo** applet is shown in Figure 26-7.

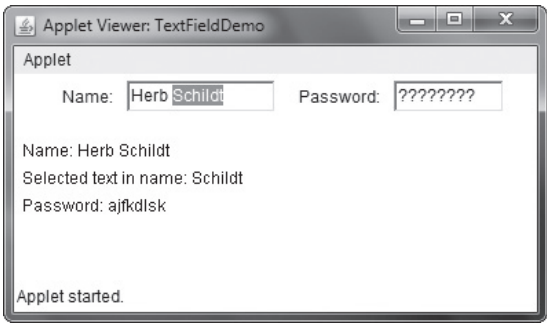



**Figure 26-7** Sample output from the **TextFieldDemo** applet

## Using a TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called **TextArea**. Following are the constructors for **TextArea**:

TextArea( ) throws HeadlessException
TextArea(int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*) throws HeadlessException
TextArea(String *str*, int *numLines*, int *numChars*, int *sBars*) throws HeadlessException

Here, *numLines* specifies the height, in lines, of the text area, and *numChars* specifies its width, in characters. Initial text can be specified by *str*. In the fifth form, you can specify the scroll bars that you want the control to have. *sBars* must be one of these values:

| | |
|---|---|
| SCROLLBARS_BOTH | SCROLLBARS_NONE |
| SCROLLBARS_HORIZONTAL_ONLY | SCROLLBARS_VERTICAL_ONLY |

**TextArea** is a subclass of **TextComponent**. Therefore, it supports the **getText( )**, **setText( )**, **getSelectedText( )**, **select( )**, **isEditable( )**, and **setEditable( )** methods described in the preceding section.

**TextArea** adds the following editing methods:

void append(String *str*)
void insert(String *str*, int *index*)
void replaceRange(String *str*, int *startIndex*, int *endIndex*)

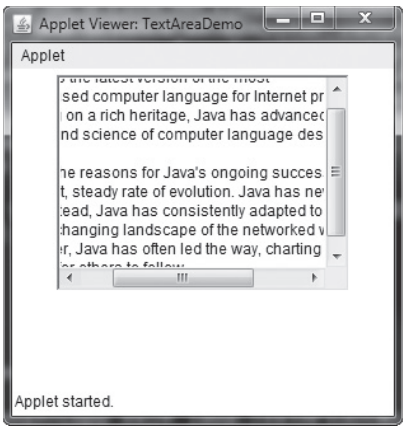
The **append( )** method appends the string specified by *str* to the end of the current text. **insert( )** inserts the string passed in *str* at the specified index. To replace text, call **replaceRange( )**. It replaces the characters from *startIndex* to *endIndex*–1, with the replacement text passed in *str*.

Text areas are almost self-contained controls. Your program incurs virtually no management overhead. Normally, your program simply obtains the current text when it is needed. You can, however, listen for **TextEvent**s, if you choose.

The following program creates a **TextArea** control:

```
// Demonstrate TextArea.
import java.awt.*;
import java.applet.*;
/*
<applet code="TextAreaDemo" width=300 height=250>
</applet>
*/

public class TextAreaDemo extends Applet {
  public void init() {
    String val =
      "Java 8 is the latest version of the most\n" +
      "widely-used computer language for Internet programming.\n" +
      "Building on a rich heritage, Java has advanced both\n" +
      "the art and science of computer language design.\n\n" +
      "One of the reasons for Java's ongoing success is its\n" +
      "constant, steady rate of evolution. Java has never stood\n" +
      "still. Instead, Java has consistently adapted to the\n" +
      "rapidly changing landscape of the networked world.\n" +
      "Moreover, Java has often led the way, charting the\n" +
      "course for others to follow.";

    TextArea text = new TextArea(val, 10, 30);
    add(text);
  }
}
```

Here is sample output from the **TextAreaDemo** applet:




## Understanding Layout Managers

All of the components that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter, a layout manager automatically arranges your controls within a window by using some type of algorithm. If you have

programmed for other GUI environments, such as Windows, then you may have laid out your controls by hand. While it is possible to lay out Java controls by hand, too, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized. This is a chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no call to **setLayout( )** is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The **setLayout( )** method has the following general form:

void setLayout(LayoutManager *layoutObj*)

Here, *layoutObj* is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass **null** for *layoutObj*. If you do this, you will need to determine the shape and position of each component manually, using the **setBounds( )** method defined by **Component**. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its **minimumLayoutSize( )** and **preferredLayoutSize( )** methods. Each component that is being managed by a layout manager contains the **getPreferredSize( )** and **getMinimumSize( )** methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default values are provided otherwise.

Java has several predefined **LayoutManager** classes, several of which are described next. You can use the layout manager that best fits your application.

## FlowLayout

**FlowLayout** is the default layout manager. This is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

FlowLayout( )
FlowLayout(int *how*)
FlowLayout(int *how*, int *horz*, int *vert*)

The first form creates the default layout, which centers components and leaves five pixels of space between each component. The second form lets you specify how each line is aligned. Valid values for *how* are as follows:

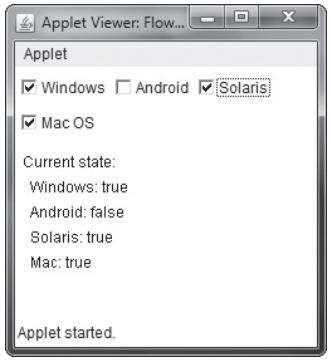
FlowLayout.LEFT
FlowLayout.CENTER
FlowLayout.RIGHT
FlowLayout.LEADING
FlowLayout.TRAILING

These values specify left, center, right, leading edge, and trailing edge alignment, respectively. The third constructor allows you to specify the horizontal and vertical space left between components in *horz* and *vert,* respectively.

Here is a version of the **CheckboxDemo** applet shown earlier in this chapter, modified so that it uses left-aligned flow layout:

```
// Use left-aligned flow layout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="FlowLayoutDemo" width=240 height=200>
  </applet>
*/

public class FlowLayoutDemo extends Applet
  implements ItemListener {

  String msg = "";
  Checkbox windows, android, solaris, mac;

  public void init() {
    // set left-aligned flow layout
    setLayout(new FlowLayout(FlowLayout.LEFT));

    windows = new Checkbox("Windows", null, true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    add(windows);
    add(android);
    add(solaris);
    add(mac);

    // register to receive item events
    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }
```

```
  // Repaint when status of a check box changes.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {

    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```




Here is sample output generated by the **FlowLayoutDemo** applet. Compare this with the output from the **CheckboxDemo** applet, shown earlier in Figure 26-2.

## BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center. Here are the constructors defined by **BorderLayout**:

BorderLayout( )
BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

**BorderLayout** defines the following constants that specify the regions:

| | |
|---|---|
| BorderLayout.CENTER | BorderLayout.SOUTH |
| BorderLayout.EAST | BorderLayout.WEST |
| BorderLayout.NORTH | |

When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

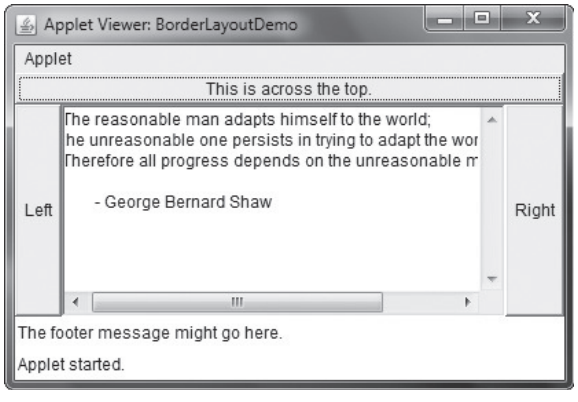
void add(Component *compRef*, Object *region*)

Here, *compRef* is a reference to the component to be added, and *region* specifies where the component will be added.

Here is an example of a **BorderLayout** with a component in each layout area:

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/

public class BorderLayoutDemo extends Applet {
  public void init() {
    setLayout(new BorderLayout());

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    String msg = "The reasonable man adapts " +
      "himself to the world;\n" +
      "the unreasonable one persists in " +
      "trying to adapt the world to himself.\n" +
      "Therefore all progress depends " +
      "on the unreasonable man.\n\n" +
      "         - George Bernard Shaw\n\n";

    add(new TextArea(msg), BorderLayout.CENTER);
  }
}
```

Sample output from the **BorderLayoutDemo** applet is shown here:



## Using Insets

Sometimes you will want to leave a small amount of space between the container that holds your components and the window that contains it. To do this, override the **getInsets( )** method that is defined by **Container**. This method returns an **Insets** object that contains the top, bottom, left, and right inset to be used when the container is displayed. These values are used by the layout manager to inset the components when it lays out the window. The constructor for **Insets** is shown here:

Insets(int *top*, int *left*, int *bottom*, int *right*)

The values passed in *top, left, bottom,* and *right* specify the amount of space between the container and its enclosing window.

The **getInsets( )** method has this general form:

Insets getInsets( )

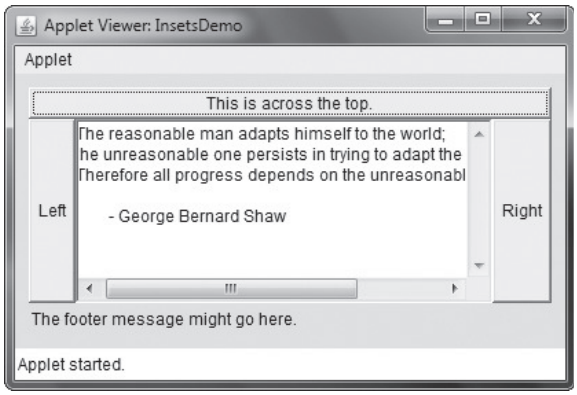
When overriding this method, you must return a new **Insets** object that contains the inset spacing you desire.

Here is the preceding **BorderLayout** example modified so that it insets its components ten pixels from each border. The background color has been set to cyan to help make the insets more visible.

```
// Demonstrate BorderLayout with insets.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="InsetsDemo" width=400 height=200>
</applet>
*/

public class InsetsDemo extends Applet {
  public void init() {
    // set background color so insets can be easily seen
    setBackground(Color.cyan);

    setLayout(new BorderLayout());

    add(new Button("This is across the top."),
        BorderLayout.NORTH);
    add(new Label("The footer message might go here."),
        BorderLayout.SOUTH);
    add(new Button("Right"), BorderLayout.EAST);
    add(new Button("Left"), BorderLayout.WEST);

    String msg = "The reasonable man adapts " +
      "himself to the world;\n" +
      "the unreasonable one persists in " +
      "trying to adapt the world to himself.\n" +
      "Therefore all progress depends " +
      "on the unreasonable man.\n\n" +
      "        - George Bernard Shaw\n\n";
```

```
    add(new TextArea(msg), BorderLayout.CENTER);
  }

  // add insets
  public Insets getInsets() {
    return new Insets(10, 10, 10, 10);
  }
}
```

Sample output from the **InsetsDemo** applet is shown here:




## GridLayout

**GridLayout** lays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns. The constructors supported by **GridLayout** are shown here:

GridLayout( )
GridLayout(int *numRows*, int *numColumns*)
GridLayout(int *numRows*, int *numColumns*, int *horz*, int *vert*)

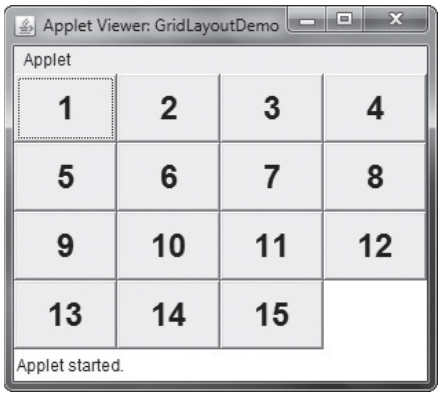
The first form creates a single-column grid layout. The second form creates a grid layout with the specified number of rows and columns. The third form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. Either *numRows* or *numColumns* can be zero. Specifying *numRows* as zero allows for unlimited-length columns. Specifying *numColumns* as zero allows for unlimited-length rows.

Here is a sample program that creates a 4×4 grid and fills it in with 15 buttons, each labeled with its index:

```
// Demonstrate GridLayout
import java.awt.*;
import java.applet.*;
/*
<applet code="GridLayoutDemo" width=300 height=200>
</applet>
*/
```

```
public class GridLayoutDemo extends Applet {
  static final int n = 4;
  public void init() {
    setLayout(new GridLayout(n, n));

    setFont(new Font("SansSerif", Font.BOLD, 24));

    for(int i = 0; i < n; i++) {
      for(int j = 0; j < n; j++) {
        int k = i * n + j;
        if(k > 0)
          add(new Button("" + k));
      }
    }
  }
}
```

Following is sample output generated by the **GridLayoutDemo** applet:




**TIP** You might try using this example as the starting point for a 15-square puzzle.

## CardLayout

The **CardLayout** class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other layouts and have them hidden, ready to be activated when needed.

**CardLayout** provides these two constructors:

CardLayout( )
CardLayout(int *horz*, int *vert*)

The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

Use of a card layout requires a bit more work than the other layouts. The cards are typically held in an object of type **Panel**. This panel must have **CardLayout** selected as its layout manager. The cards that form the deck are also typically objects of type **Panel**. Thus, you must create a panel that contains the deck and a panel for each card in the deck. Next, you add to the appropriate panel the components that form each card. You then add these panels to the panel for which **CardLayout** is the layout manager. Finally, you add this panel to the window. Once these steps are complete, you must provide some way for the user to select between cards. One common approach is to include one push button for each card in the deck.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, you will use this form of **add( )** when adding cards to a panel:

void add(Component *panelRef*, Object *name*)

Here, *name* is a string that specifies the name of the card whose panel is specified by *panelRef*.

After you have created a deck, your program activates a card by calling one of the following methods defined by **CardLayout**:

void first(Container *deck*)
void last(Container *deck*)
void next(Container *deck*)
void previous(Container *deck*)
void show(Container *deck*, String *cardName*)

Here, *deck* is a reference to the container (usually a panel) that holds the cards, and *cardName* is the name of a card. Calling **first( )** causes the first card in the deck to be shown. To show the last card, call **last( )**. To show the next card, call **next( )**. To show the previous card, call **previous( )**. Both **next( )** and **previous( )** automatically cycle back to the top or bottom of the deck, respectively. The **show( )** method displays the card whose name is passed in *cardName*.

The following example creates a two-level card deck that allows the user to select an operating system. Windows-based operating systems are displayed in one card. Mac OS and Solaris are displayed in the other card.

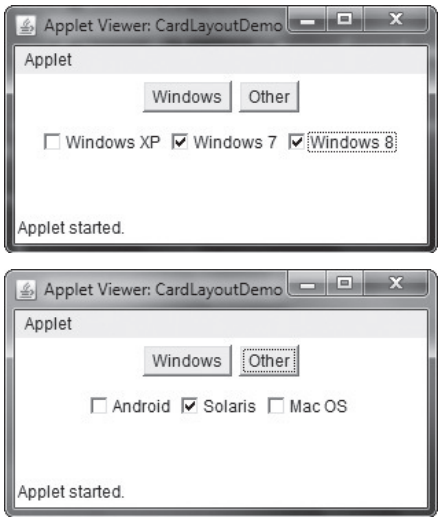
```
// Demonstrate CardLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="CardLayoutDemo" width=300 height=100>
  </applet>
*/

public class CardLayoutDemo extends Applet
  implements ActionListener, MouseListener {
```

```
Checkbox windowsXP, windows7, windows8, android, solaris, mac;
Panel osCards;
CardLayout cardLO;
Button Win, Other;

public void init() {
  Win = new Button("Windows");
  Other = new Button("Other");
  add(Win);
  add(Other);

  cardLO = new CardLayout();
  osCards = new Panel();
  osCards.setLayout(cardLO); // set panel layout to card layout

  windowsXP = new Checkbox("Windows XP", null, true);
  windows7 = new Checkbox("Windows 7", null, false);
  windows8 = new Checkbox("Windows 8", null, false);
  android = new Checkbox("Android");
  solaris = new Checkbox("Solaris");
  mac = new Checkbox("Mac OS");

  // add Windows check boxes to a panel
  Panel winPan = new Panel();
  winPan.add(windowsXP);
  winPan.add(windows7);
  winPan.add(windows8);

  // Add other OS check boxes to a panel
  Panel otherPan = new Panel();
  otherPan.add(android);
  otherPan.add(solaris);
  otherPan.add(mac);

  // add panels to card deck panel
  osCards.add(winPan, "Windows");
  osCards.add(otherPan, "Other");

  // add cards to main applet panel
  add(osCards);

  // register to receive action events
  Win.addActionListener(this);
  Other.addActionListener(this);

  // register mouse events
  addMouseListener(this);
}

// Cycle through panels.
public void mousePressed(MouseEvent me) {
  cardLO.next(osCards);
}
```

```
  // Provide empty implementations for the other MouseListener methods.
  public void mouseClicked(MouseEvent me) {
  }
  public void mouseEntered(MouseEvent me) {
  }
  public void mouseExited(MouseEvent me) {
  }
  public void mouseReleased(MouseEvent me) {
  }

  public void actionPerformed(ActionEvent ae) {
    if(ae.getSource() == Win) {

      cardLO.show(osCards, "Windows");
    }
    else {
      cardLO.show(osCards, "Other");
    }
  }
}
```



Here is sample output generated by the **CardLayoutDemo** applet. Each card is activated by pushing its button. You can also cycle through the cards by clicking the mouse.




## GridBagLayout

Although the preceding layouts are perfectly acceptable for many uses, some situations will require that you take a bit more control over how the components are arranged. A good way to do this is to use a grid bag layout, which is specified by the **GridBagLayout** class. What makes the grid bag useful is that you can specify the relative placement of components by specifying their positions within cells inside a grid. The key to the grid bag is that each component can be a different size, and each row in the grid can have a different number

of columns. This is why the layout is called a *grid bag*. It's a collection of small grids joined together.

The location and size of each component in a grid bag are determined by a set of constraints linked to it. The constraints are contained in an object of type **GridBagConstraints**. Constraints include the height and width of a cell, and the placement of a component, its alignment, and its anchor point within the cell.

The general procedure for using a grid bag is to first create a new **GridBagLayout** object and to make it the current layout manager. Then, set the constraints that apply to each component that will be added to the grid bag. Finally, add the components to the layout manager. Although **GridBagLayout** is a bit more complicated than the other layout managers, it is still quite easy to use once you understand how it works.

**GridBagLayout** defines only one constructor, which is shown here:

```
GridBagLayout( )
```

**GridBagLayout** defines several methods, of which many are protected and not for general use. There is one method, however, that you must use: **setConstraints( )**. It is shown here:

```
void setConstraints(Component comp, GridBagConstraints cons)
```

Here, *comp* is the component for which the constraints specified by *cons* apply. This method sets the constraints that apply to each component in the grid bag.

The key to successfully using **GridBagLayout** is the proper setting of the constraints, which are stored in a **GridBagConstraints** object. **GridBagConstraints** defines several fields that you can set to govern the size, placement, and spacing of a component. These are shown in Table 26-1. Several are described in greater detail in the following discussion.

| Field | Purpose |
|---|---|
| int anchor | Specifies the location of a component within a cell. The default is **GridBagConstraints.CENTER**. |
| int fill | Specifies how a component is resized if the component is smaller than its cell. Valid values are **GridBagConstraints.NONE** (the default), **GridBagConstraints.HORIZONTAL**, **GridBagConstraints.VERTICAL**, **GridBagConstraints.BOTH**. |
| int gridheight | Specifies the height of component in terms of cells. The default is 1. |
| int gridwidth | Specifies the width of component in terms of cells. The default is 1. |
| int gridx | Specifies the X coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**. |
| int gridy | Specifies the Y coordinate of the cell to which the component will be added. The default value is **GridBagConstraints.RELATIVE**. |
| Insets insets | Specifies the insets. Default insets are all zero. |
| int ipadx | Specifies extra horizontal space that surrounds a component within a cell. The default is 0. |
| int ipady | Specifies extra vertical space that surrounds a component within a cell. The default is 0. |

**Table 26-1** Constraint Fields Defined by **GridBagConstraints**

| Field | Purpose |
|---|---|
| double weightx | Specifies a weight value that determines the horizontal spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window. |
| double weighty | Specifies a weight value that determines the vertical spacing between cells and the edges of the container that holds them. The default value is 0.0. The greater the weight, the more space that is allocated. If all values are 0.0, extra space is distributed evenly between the edges of the window. |

**Table 26-1** Constraint Fields Defined by **GridBagConstraints** *(continued)*



**GridBagConstraints** also defines several static fields that contain standard constraint values, such as **GridBagConstraints.CENTER** and **GridBagConstraints.VERTICAL**.

When a component is smaller than its cell, you can use the **anchor** field to specify where within the cell the component's top-left corner will be located. There are three types of values that you can give to **anchor**. The first are absolute:

| | |
|---|---|
| GridBagConstraints.CENTER | GridBagConstraints.SOUTH |
| GridBagConstraints.EAST | GridBagConstraints.SOUTHEAST |
| GridBagConstraints.NORTH | GridBagConstraints.SOUTHWEST |
| GridBagConstraints.NORTHEAST | GridBagConstraints.WEST |
| GridBagConstraints.NORTHWEST | |

As their names imply, these values cause the component to be placed at the specific locations.

The second type of values that can be given to **anchor** is relative, which means the values are relative to the container's orientation, which might differ for non-Western languages. The relative values are shown here:

| | |
|---|---|
| GridBagConstraints.FIRST_LINE_END | GridBagConstraints.LINE_END |
| GridBagConstraints.FIRST_LINE_START | GridBagConstraints.LINE_START |
| GridBagConstraints.LAST_LINE_END | GridBagConstraints.PAGE_END |
| GridBagConstraints.LAST_LINE_START | GridBagConstraints.PAGE_START |

Their names describe the placement.

The third type of values that can be given to **anchor** allows you to position components relative to the baseline of the row. These values are shown here:

| | |
|---|---|
| GridBagConstraints.BASELINE | GridBagConstraints.BASELINE_LEADING |
| GridBagConstraints.BASELINE_TRAILING | GridBagConstraints.ABOVE_BASELINE |
| GridBagConstraints.ABOVE_BASELINE_LEADING | GridBagConstraints.ABOVE_BASELINE_ TRAILING |
| GridBagConstraints.BELOW_BASELINE | GridBagConstraints.BELOW_BASELINE_ LEADING |
| GridBagConstraints. BELOW_BASELINE_TRAILING | |

The horizontal position can be either centered, against the leading edge (LEADING), or against the trailing edge (TRAILING).

The **weightx** and **weighty** fields are both quite important and quite confusing at first glance. In general, their values determine how much of the extra space within a container is allocated to each row and column. By default, both these values are zero. When all values within a row or a column are zero, extra space is distributed evenly between the edges of the window. By increasing the weight, you increase that row or column's allocation of space proportional to the other rows or columns. The best way to understand how these values work is to experiment with them a bit.

The **gridwidth** variable lets you specify the width of a cell in terms of cell units. The default is 1. To specify that a component use the remaining space in a row, use **GridBagConstraints.REMAINDER**. To specify that a component use the next-to-last cell in a row, use **GridBagConstraints.RELATIVE**. The **gridheight** constraint works the same way, but in the vertical direction.

You can specify a padding value that will be used to increase the minimum size of a cell. To pad horizontally, assign a value to **ipadx**. To pad vertically, assign a value to **ipady**.

Here is an example that uses **GridBagLayout** to demonstrate several of the points just discussed:

```
// Use GridBagLayout.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="GridBagDemo" width=250 height=200>
  </applet>
*/

public class GridBagDemo extends Applet
  implements ItemListener {

  String msg = "";
  Checkbox windows, android, solaris, mac;

  public void init() {
    GridBagLayout gbag = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbag);

    // Define check boxes.
    windows = new Checkbox("Windows ", null, true);
    android = new Checkbox("Android");
    solaris = new Checkbox("Solaris");
    mac = new Checkbox("Mac OS");

    // Define the grid bag.

    // Use default row weight of 0 for first row.
    gbc.weightx = 1.0; // use a column weight of 1
```

```
    gbc.ipadx = 200; // pad by 200 units
    gbc.insets = new Insets(4, 4, 0, 0); // inset slightly from top left

    gbc.anchor = GridBagConstraints.NORTHEAST;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(windows, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(android, gbc);

    // Give second row a weight of 1.
    gbc.weighty = 1.0;

    gbc.gridwidth = GridBagConstraints.RELATIVE;
    gbag.setConstraints(solaris, gbc);

    gbc.gridwidth = GridBagConstraints.REMAINDER;
    gbag.setConstraints(mac, gbc);

    // Add the components.
    add(windows);
    add(android);
    add(solaris);
    add(mac);

    // Register to receive item events.
    windows.addItemListener(this);
    android.addItemListener(this);
    solaris.addItemListener(this);
    mac.addItemListener(this);
  }

  // Repaint when status of a check box changes.
  public void itemStateChanged(ItemEvent ie) {
    repaint();
  }

  // Display current state of the check boxes.
  public void paint(Graphics g) {
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = "  Windows: " + windows.getState();
    g.drawString(msg, 6, 100);
    msg = "  Android: " + android.getState();
    g.drawString(msg, 6, 120);
    msg = "  Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = "  Mac: " + mac.getState();
    g.drawString(msg, 6, 160);
  }
}
```

Sample output produced by the program is shown here.


In this layout, the operating system check boxes are positioned in a 2×2 grid. Each cell has a horizontal padding of 200. Each component is inset slightly (by 4 units) from the top left. The column weight is set to 1, which causes any extra horizontal space to be distributed evenly between the columns. The first row uses a default weight of 0; the second has a weight of 1. This means that any extra vertical space is added to the second row.

**GridBagLayout** is a powerful layout manager. It is worth taking some time to experiment with and explore. Once you understand what the various settings do, you can use **GridBagLayout** to position components with a high degree of precision.

## Menu Bars and Menus

A top-level window can have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**. In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItem** objects. Each **MenuItem** object represents something that can be selected by the user. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem** and will have a check mark next to them when they are selected.

To create a menu bar, first create an instance of **MenuBar**. This class defines only the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar. Following are the constructors for **Menu**:

Menu( ) throws HeadlessException
Menu(String *optionName*) throws HeadlessException
Menu(String *optionName*, boolean *removable*) throws HeadlessException

Here, *optionName* specifies the name of the menu selection. If *removable* is **true**, the menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar. (Removable menus are implementation-dependent.) The first form creates an empty menu.

Individual menu items are of type **MenuItem**. It defines these constructors:

MenuItem( ) throws HeadlessException
MenuItem(String *itemName*) throws HeadlessException
MenuItem(String *itemName*, MenuShortcut *keyAccel*) throws HeadlessException

Here, *itemName* is the name shown in the menu, and *keyAccel* is the menu shortcut for this item.

You can disable or enable a menu item by using the **setEnabled( )** method. Its form is shown here:

```
void setEnabled(boolean enabledFlag)
```

If the argument *enabledFlag* is **true**, the menu item is enabled. If **false**, the menu item is disabled.

You can determine an item's status by calling **isEnabled( )**. This method is shown here:

```
boolean isEnabled( )
```

**isEnabled( )** returns **true** if the menu item on which it is called is enabled. Otherwise, it returns **false**.

You can change the name of a menu item by calling **setLabel( )**. You can retrieve the current name by using **getLabel( )**. These methods are as follows:

```
void setLabel(String newName)
String getLabel( )
```

Here, *newName* becomes the new name of the invoking menu item. **getLabel( )** returns the current name.

You can create a checkable menu item by using a subclass of **MenuItem** called **CheckboxMenuItem**. It has these constructors:

```
CheckboxMenuItem( ) throws HeadlessException
CheckboxMenuItem(String itemName) throws HeadlessException
CheckboxMenuItem(String itemName, boolean on) throws HeadlessException
```

Here, *itemName* is the name shown in the menu. Checkable items operate as toggles. Each time one is selected, its state changes. In the first two forms, the checkable entry is unchecked. In the third form, if *on* is **true**, the checkable entry is initially checked. Otherwise, it is cleared.

You can obtain the status of a checkable item by calling **getState( )**. You can set it to a known state by using **setState( )**. These methods are shown here:

```
boolean getState( )
void setState(boolean checked)
```

If the item is checked, **getState( )** returns **true**. Otherwise, it returns **false**. To check an item, pass **true** to **setState( )**. To clear an item, pass **false**.

Once you have created a menu item, you must add the item to a **Menu** object by using **add( )**, which has the following general form:

```
MenuItem add(MenuItem item)
```

Here, *item* is the item being added. Items are added to a menu in the order in which the calls to **add( )** take place. The *item* is returned.

Once you have added all items to a **Menu** object, you can add that object to the menu bar by using this version of **add( )** defined by **MenuBar**:

```
Menu add(Menu menu)
```

Here, *menu* is the menu being added. The *menu* is returned.

Menus generate events only when an item of type **MenuItem** or **CheckboxMenuItem** is selected. They do not generate events when a menu bar is accessed to display a drop-down menu, for example. Each time a menu item is selected, an **ActionEvent** object is generated. By default, the action command string is the name of the menu item. However, you can specify a different action command string by calling **setActionCommand( )** on the menu item. Each time a check box menu item is checked or unchecked, an **ItemEvent** object is generated. Thus, you must implement the **ActionListener** and/or **ItemListener** interfaces in order to handle these menu events.

The **getItem( )** method of **ItemEvent** returns a reference to the item that generated this event. The general form of this method is shown here:

Object getItem( )

Following is an example that adds a series of nested menus to a pop-up window. The item selected is displayed in the window. The state of the two check box menu items is also displayed.

```
// Illustrate menus.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MenuDemo" width=250 height=250>
  </applet>
*/

// Create a subclass of Frame.
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // create menu bar and add it to frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // create the menu items
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4, item5;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(item4 = new MenuItem("-"));
    file.add(item5 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item6, item7, item8, item9;
    edit.add(item6 = new MenuItem("Cut"));
    edit.add(item7 = new MenuItem("Copy"));
```

```
    edit.add(item8 = new MenuItem("Paste"));
    edit.add(item9 = new MenuItem("-"));

    Menu sub = new Menu("Special");
    MenuItem item10, item11, item12;
    sub.add(item10 = new MenuItem("First"));
    sub.add(item11 = new MenuItem("Second"));
    sub.add(item12 = new MenuItem("Third"));
    edit.add(sub);

    // these are checkable menu items
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);

    // create an object to handle action and item events
    MyMenuHandler handler = new MyMenuHandler(this);
    // register it to receive those events
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    item11.addActionListener(handler);
    item12.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);
    // register it to receive those events
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 220);
     else
      g.drawString("Debug is off.", 10, 220);

     if(test.getState())
       g.drawString("Testing is on.", 10, 240);
     else
```

```
        g.drawString("Testing is off.", 10, 240);
    }
}

class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;

    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;

    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }

    // Handle action events.
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = ae.getActionCommand();
        if(arg.equals("New..."))
            msg += "New.";
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
        else if(arg.equals("Edit"))
            msg += "Edit.";
        else if(arg.equals("Cut"))
            msg += "Cut.";
        else if(arg.equals("Copy"))
            msg += "Copy.";
        else if(arg.equals("Paste"))
            msg += "Paste.";
        else if(arg.equals("First"))
            msg += "First.";
        else if(arg.equals("Second"))
            msg += "Second.";
        else if(arg.equals("Third"))
            msg += "Third.";
        else if(arg.equals("Debug"))
            msg += "Debug.";
        else if(arg.equals("Testing"))
            msg += "Testing.";
```

```
    menuFrame.msg = msg;
    menuFrame.repaint();
  }

  // Handle item events.
  public void itemStateChanged(ItemEvent ie) {
    menuFrame.repaint();
  }
}

// Create frame window.
public class MenuDemo extends Applet {
  Frame f;

  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(new Dimension(width, height));

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

Sample output from the **MenuDemo** applet is shown in Figure 26-8.

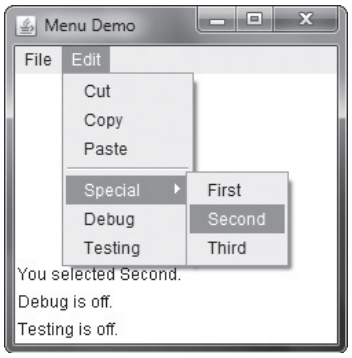



**Figure 26-8** Sample output from the **MenuDemo** applet

There is one other menu-related class that you might find interesting: **PopupMenu**. It works just like **Menu**, but produces a menu that can be displayed at a specific location. **PopupMenu** provides a flexible, useful alternative for some types of menuing situations.

## Dialog Boxes

Often, you will want to use a *dialog box* to hold a set of related controls. Dialog boxes are primarily used to obtain user input and are often child windows of a top-level window. Dialog boxes don't have menu bars, but in other respects, they function like frame windows. (You can add controls to them, for example, in the same way that you add controls to a frame window.) Dialog boxes may be modal or modeless. When a *modal* dialog box is active, all input is directed to it until it is closed. This means that you cannot access other parts of your program until you have closed the dialog box. When a *modeless* dialog box is active, input focus can be directed to another window in your program. Thus, other parts of your program remain active and accessible. In the AWT, dialog boxes are of type **Dialog**. Two commonly used constructors are shown here:

Dialog(Frame *parentWindow*, boolean *mode*)
Dialog(Frame *parentWindow*, String *title*, boolean *mode*)

Here, *parentWindow* is the owner of the dialog box. If *mode* is **true**, the dialog box is modal. Otherwise, it is modeless. The title of the dialog box can be passed in *title*. Generally, you will subclass **Dialog**, adding the functionality required by your application.

Following is a modified version of the preceding menu program that displays a modeless dialog box when the New option is chosen. Notice that when the dialog box is closed, **dispose( )** is called. This method is defined by **Window**, and it frees all system resources associated with the dialog box window.

```
// Demonstrate Dialog box.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="DialogDemo" width=250 height=250>
  </applet>
*/

// Create a subclass of Dialog.
class SampleDialog extends Dialog implements ActionListener {
  SampleDialog(Frame parent, String title) {
    super(parent, title, false);
    setLayout(new FlowLayout());
    setSize(300, 200);

    add(new Label("Press this button:"));
    Button b;
    add(b = new Button("Cancel"));
    b.addActionListener(this);
  }
```

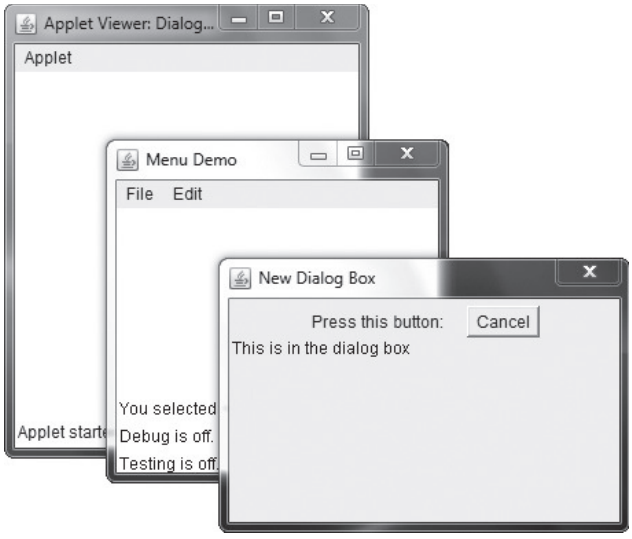
```
  public void actionPerformed(ActionEvent ae) {
    dispose();
  }

  public void paint(Graphics g) {
    g.drawString("This is in the dialog box", 10, 70);
  }
}

// Create a subclass of Frame.
class MenuFrame extends Frame {
  String msg = "";
  CheckboxMenuItem debug, test;

  MenuFrame(String title) {
    super(title);

    // create menu bar and add it to frame
    MenuBar mbar = new MenuBar();
    setMenuBar(mbar);

    // create the menu items
    Menu file = new Menu("File");
    MenuItem item1, item2, item3, item4;
    file.add(item1 = new MenuItem("New..."));
    file.add(item2 = new MenuItem("Open..."));
    file.add(item3 = new MenuItem("Close"));
    file.add(new MenuItem("-"));
    file.add(item4 = new MenuItem("Quit..."));
    mbar.add(file);

    Menu edit = new Menu("Edit");
    MenuItem item5, item6, item7;
    edit.add(item5 = new MenuItem("Cut"));
    edit.add(item6 = new MenuItem("Copy"));
    edit.add(item7 = new MenuItem("Paste"));
    edit.add(new MenuItem("-"));

    Menu sub = new Menu("Special", true);
    MenuItem item8, item9, item10;
    sub.add(item8 = new MenuItem("First"));
    sub.add(item9 = new MenuItem("Second"));
    sub.add(item10 = new MenuItem("Third"));
    edit.add(sub);

    // these are checkable menu items
    debug = new CheckboxMenuItem("Debug");
    edit.add(debug);
    test = new CheckboxMenuItem("Testing");
    edit.add(test);

    mbar.add(edit);
```

```
    // create an object to handle action and item events
    MyMenuHandler handler = new MyMenuHandler(this);
    // register it to receive those events
    item1.addActionListener(handler);
    item2.addActionListener(handler);
    item3.addActionListener(handler);
    item4.addActionListener(handler);
    item5.addActionListener(handler);
    item6.addActionListener(handler);
    item7.addActionListener(handler);
    item8.addActionListener(handler);
    item9.addActionListener(handler);
    item10.addActionListener(handler);
    debug.addItemListener(handler);
    test.addItemListener(handler);

    // create an object to handle window events
    MyWindowAdapter adapter = new MyWindowAdapter(this);

    // register it to receive those events
    addWindowListener(adapter);
  }

  public void paint(Graphics g) {
    g.drawString(msg, 10, 200);

    if(debug.getState())
      g.drawString("Debug is on.", 10, 220);
    else
      g.drawString("Debug is off.", 10, 220);

    if(test.getState())
      g.drawString("Testing is on.", 10, 240);
    else
    g.drawString("Testing is off.", 10, 240);
  }
}

class MyWindowAdapter extends WindowAdapter {
  MenuFrame menuFrame;

  public MyWindowAdapter(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }

  public void windowClosing(WindowEvent we) {
    menuFrame.dispose();
  }
}

class MyMenuHandler implements ActionListener, ItemListener {
  MenuFrame menuFrame;
```

```
  public MyMenuHandler(MenuFrame menuFrame) {
    this.menuFrame = menuFrame;
  }

  // Handle action events.
  public void actionPerformed(ActionEvent ae) {
    String msg = "You selected ";
    String arg = ae.getActionCommand();
    // Activate a dialog box when New is selected.
    if(arg.equals("New...")) {
       msg += "New.";
       SampleDialog d = new
         SampleDialog(menuFrame, "New Dialog Box");
       d.setVisible(true);
    }
    // Try defining other dialog boxes for these options.
    else if(arg.equals("Open..."))
      msg += "Open.";
    else if(arg.equals("Close"))
      msg += "Close.";
    else if(arg.equals("Quit..."))
      msg += "Quit.";
    else if(arg.equals("Edit"))
      msg += "Edit.";
    else if(arg.equals("Cut"))
      msg += "Cut.";
    else if(arg.equals("Copy"))
      msg += "Copy.";
    else if(arg.equals("Paste"))
      msg += "Paste.";
    else if(arg.equals("First"))
      msg += "First.";
    else if(arg.equals("Second"))
      msg += "Second.";
    else if(arg.equals("Third"))
      msg += "Third.";
    else if(arg.equals("Debug"))
      msg += "Debug.";
    else if(arg.equals("Testing"))
      msg += "Testing.";
    menuFrame.msg = msg;
    menuFrame.repaint();
  }

  public void itemStateChanged(ItemEvent ie) {
      menuFrame.repaint();
  }
}

// Create frame window.
public class DialogDemo extends Applet {
  Frame f;
```

```
  public void init() {
    f = new MenuFrame("Menu Demo");
    int width = Integer.parseInt(getParameter("width"));
    int height = Integer.parseInt(getParameter("height"));

    setSize(width, height);

    f.setSize(width, height);
    f.setVisible(true);
  }

  public void start() {
    f.setVisible(true);
  }

  public void stop() {
    f.setVisible(false);
  }
}
```

Here is sample output from the **DialogDemo** applet:




**TIP** On your own, try defining dialog boxes for the other options presented by the menus.

## FileDialog

Java provides a built-in dialog box that lets the user specify a file. To create a file dialog box, instantiate an object of type **FileDialog**. This causes a file dialog box to be displayed.

Usually, this is the standard file dialog box provided by the operating system. Here are three **FileDialog** constructors:

FileDialog(Frame *parent*)
FileDialog(Frame *parent*, String *boxName*)
FileDialog(Frame *parent*, String *boxName*, int *how*)

Here, *parent* is the owner of the dialog box. The *boxName* parameter specifies the name displayed in the box's title bar. If *boxName* is omitted, the title of the dialog box is empty. If *how* is **FileDialog.LOAD**, then the box is selecting a file for reading. If *how* is **FileDialog.SAVE**, the box is selecting a file for writing. If *how* is omitted, the box is selecting a file for reading.

**FileDialog** provides methods that allow you to determine the name of the file and its path as selected by the user. Here are two examples:

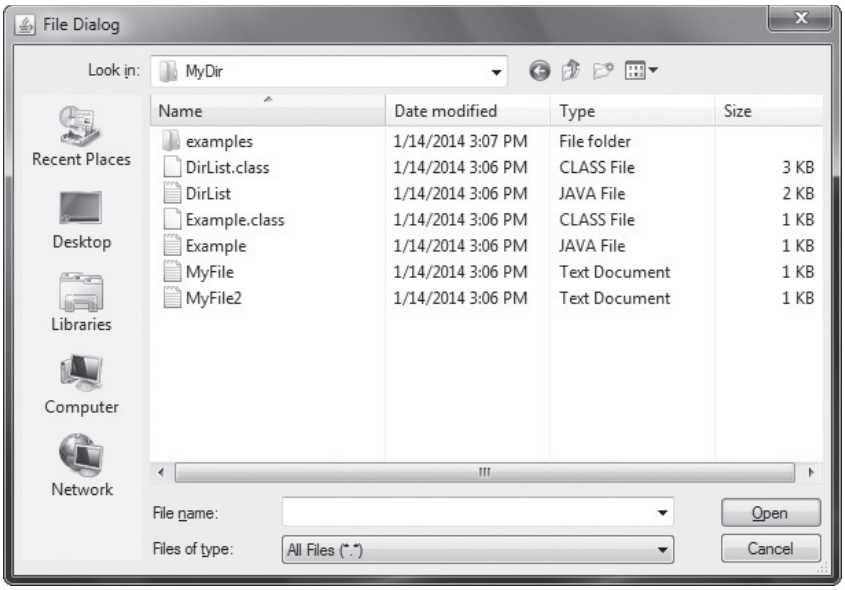
String getDirectory( )
String getFile( )

These methods return the directory and the filename, respectively.

The following program activates the standard file dialog box:

```
/* Demonstrate File Dialog box.

   This is an application, not an applet.
*/
import java.awt.*;
import java.awt.event.*;

//  Create a subclass of Frame.
class SampleFrame extends Frame {
  SampleFrame(String title) {
    super(title);

    // remove the window when closed
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent we) {
        System.exit(0);
      }
    });
  }
}

// Demonstrate FileDialog.
class FileDialogDemo {
  public static void main(String args[]) {
    // create a frame that owns the dialog
    Frame f = new SampleFrame("File Dialog Demo");
    f.setVisible(true);
    f.setSize(100, 100);

    FileDialog fd = new FileDialog(f, "File Dialog");

    fd.setVisible(true);
  }
}
```

The output generated by this program is shown here. (The precise configuration of the dialog box may vary.)




One last point: Beginning with JDK 7, you can use **FileDialog** to select a list of files. This functionality is supported by the **setMultipleMode( )**, **isMultipleMode( )**, and **getFiles( )** methods.

## A Word About Overriding paint()

Before concluding our examination of AWT controls, a short word about overriding **paint( )** is in order. Although not relevant to the simple AWT examples shown in this book, when overriding **paint( )**, there are times when it is necessary to call the superclass implementation of **paint( )**. Therefore, for some programs, you will need to use this **paint( )** skeleton:

```
public void paint(Graphics g) {

  // code to repaint this window

  // Call superclass paint()
  super.paint(g);
}
```

In Java, there are two general types of components: heavyweight and lightweight. A heavyweight component has its own native window, which is called its *peer*. A lightweight component is implemented completely in Java code and uses the window provided by an ancestor. The AWT controls described and used in this chapter are all heavyweight. However, if a container holds any lightweight components (that is, has lightweight child components), your override of **paint( )** for that container must call **super.paint( )**. By calling **super.paint( )**, you ensure that any lightweight child components, such as lightweight controls, get properly painted. If you are unsure of a child component's type, you can call **isLightweight( )**, defined by **Component**, to find out. It returns **true** if the component is lightweight, and **false** otherwise.

This page has been intentionally left blank

CHAPTER

# 27 Images

This chapter examines the **Image** class and the **java.awt.image** package. Together, they provide support for *imaging* (the display and manipulation of graphical images). An *image* is simply a rectangular graphical object. Images are a key component of web design. In fact, the inclusion of the **<img>** tag in the Mosaic browser at NCSA (National Center for Supercomputer Applications) is what caused the Web to begin to grow explosively in 1993. This tag was used to include an image *inline* with the flow of hypertext. Java expands upon this basic concept, allowing images to be managed under program control. Because of its importance, Java provides extensive support for imaging.

Images are objects of the **Image** class, which is part of the **java.awt** package. Images are manipulated using the classes found in the **java.awt.image** package. There are a large number of imaging classes and interfaces defined by **java.awt.image**, and it is not possible to examine them all. Instead, we will focus on those that form the foundation of imaging. Here are the **java.awt.image** classes discussed in this chapter:

| | |
|---|---|
| CropImageFilter | MemoryImageSource |
| FilteredImageSource | PixelGrabber |
| ImageFilter | RGBImageFilter |

These are the interfaces that we will use:

| | | |
|---|---|---|
| ImageConsumer | ImageObserver | ImageProducer |

Also examined is the **MediaTracker** class, which is part of **java.awt**.

## File Formats

Originally, web images could only be in GIF format. The GIF image format was created by CompuServe in 1987 to make it possible for images to be viewed while online, so it was well suited to the Internet. GIF images can have only up to 256 colors each. This limitation

caused the major browser vendors to add support for JPEG images in 1995. The JPEG format was created by a group of photographic experts to store full-color-spectrum, continuous-tone images. These images, when properly created, can be of much higher fidelity as well as more highly compressed than a GIF encoding of the same source image. Another file format is PNG. It too is an alternative to GIF. In almost all cases, you will never care or notice which format is being used in your programs. The Java image classes abstract the differences behind a clean interface.

# Image Fundamentals: Creating, Loading, and Displaying

There are three common operations that occur when you work with images: creating an image, loading an image, and displaying an image. In Java, the **Image** class is used to refer to images in memory and to images that must be loaded from external sources. Thus, Java provides ways for you to create a new image object and ways to load one. It also provides a means by which an image can be displayed. Let's look at each.

## Creating an Image Object

You might expect that you create a memory image using something like the following:

```
Image test = new Image(200, 100); // Error -- won't work
```

Not so. Because images must eventually be painted on a window to be seen, the **Image** class doesn't have enough information about its environment to create the proper data format for the screen. Therefore, the **Component** class in **java.awt** has a factory method called **createImage( )** that is used to create **Image** objects. (Remember that all of the AWT components are subclasses of **Component**, so all support this method.)

The **createImage( )** method has the following two forms:

Image createImage(ImageProducer *imgProd*)
Image createImage(int *width*, int *height*)

The first form returns an image produced by *imgProd*, which is an object of a class that implements the **ImageProducer** interface. (We will look at image producers later.) The second form returns a blank (that is, empty) image that has the specified width and height. Here is an example:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
```

This creates an instance of **Canvas** and then calls the **createImage( )** method to actually make an **Image** object. At this point, the image is blank. Later, you will see how to write data to it.

## Loading an Image

The other way to obtain an image is to load one. One way to do this is to use the **getImage( )** method defined by the **Applet** class. It has the following forms:

Image getImage(URL *url*)
Image getImage(URL *url*, String *imageName*)

The first version returns an **Image** object that encapsulates the image found at the location specified by *url*. The second version returns an **Image** object that encapsulates the image found at the location specified by *url* and having the name specified by *imageName*.

## Displaying an Image

Once you have an image, you can display it by using **drawImage( )**, which is a member of the **Graphics** class. It has several forms. The one we will be using is shown here:

boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

This displays the image passed in *imgObj* with its upper-left corner specified by *left* and *top*. *imgOb* is a reference to a class that implements the **ImageObserver** interface. This interface is implemented by all AWT (and Swing) components. An *image observer* is an object that can monitor an image while it loads. **ImageObserver** is described in the next section.

With **getImage( )** and **drawImage( )**, it is actually quite easy to load and display an image. Here is a sample applet that loads and displays a single image. The file **Lilies.jpg** is loaded, but you can substitute any GIF, JPG, or PNG file you like (just make sure it is available in the same directory with the HTML file that contains the applet).

```
/*
 * <applet code="SimpleImageLoad" width=400 height=345>
 *  <param name="img" value="Lilies.jpg">
 * </applet>
 */
import java.awt.*;
import java.applet.*;

public class SimpleImageLoad extends Applet
{
  Image img;

  public void init() {
    img = getImage(getDocumentBase(), getParameter("img"));
  }

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

In the **init( )** method, the **img** variable is assigned to the image returned by **getImage( )**. The **getImage( )** method uses the string returned by **getParameter("img")** as the filename for the image. This image is loaded from a **URL** that is relative to the result of **getDocumentBase( )**, which is the **URL** of the HTML page this applet tag was in. The filename returned by **getParameter("img")** comes from the applet tag **<param name= "img" value="Lilies.jpg">**. This is the equivalent, if a little slower, of using the HTML tag **<img src="Lilies.jpg" width=400 height=345>**. Figure 27-1 shows what it looks like when you run the program.

When this applet runs, it starts loading **img** in the **init( )** method. Onscreen you can see the image as it loads from the network, because **Applet**'s implementation of the **ImageObserver** interface calls **paint( )** every time more image data arrives.


**Figure 27-1** Sample output from **SimpleImageLoad**

Seeing the image load is somewhat informative, but it might be better if you use the time it takes to load the image to do other things in parallel. That way, the fully formed image can simply appear on the screen in an instant, once it is fully loaded. You can use **ImageObserver**, described next, to monitor loading an image while you paint the screen with other information.

## ImageObserver

**ImageObserver** is an interface used to receive notification as an image is being generated, and it defines only one method: **imageUpdate( )**. Using an image observer allows you to perform other actions, such as show a progress indicator or an attract screen, as you are informed of the progress of the download. This kind of notification is very useful when an image is being loaded over a slow network.

The **imageUpdate( )** method has this general form:

```
boolean imageUpdate(Image imgObj, int flags, int left, int top,
                    int width, int height)
```

Here, *imgObj* is the image being loaded, and *flags* is an integer that communicates the status of the update report. The four integers *left, top, width,* and *height* represent a rectangle that contains different values depending on the values passed in *flags*. **imageUpdate( )** should return **false** if it has completed loading, and **true** if there is more image to process.

The *flags* parameter contains one or more bit flags defined as static variables inside the **ImageObserver** interface. These flags and the information they provide are listed in Table 27-1.

| Flag | Meaning |
|---|---|
| WIDTH | The *width* parameter is valid and contains the width of the image. |
| HEIGHT | The *height* parameter is valid and contains the height of the image. |
| PROPERTIES | The properties associated with the image can now be obtained using **imgObj.getProperty( )**. |
| SOMEBITS | More pixels needed to draw the image have been received. The parameters *left, top, width*, and *height* define the rectangle containing the new pixels. |
| FRAMEBITS | A complete frame that is part of a multiframe image, which was previously drawn, has been received. This frame can be displayed. The *left, top, width*, and *height* parameters are not used. |
| ALLBITS | The image is now complete. The *left, top, width,* and *height* parameters are not used. |
| ERROR | An error has occurred to an image that was being tracked asynchronously. The image is incomplete and cannot be displayed. No further image information will be received. The ABORT flag will also be set to indicate that the image production was aborted. |
| ABORT | An image that was being tracked asynchronously was aborted before it was complete. However, if an error has not occurred, accessing any part of the image's data will restart the production of the image. |

**Table 27-1** Bit Flags of the **imageUpdate( )** *flags* Parameter



The **Applet** class has an implementation of the **imageUpdate( )** method for the **ImageObserver** interface that is used to repaint images as they are loaded. You can override this method in your class to change that behavior.

Here is a simple example of an **imageUpdate( )** method:

```
public boolean imageUpdate(Image img, int flags,
                           int x, int y, int w, int h) {
  if ((flags & ALLBITS) == 0) {
    System.out.println("Still processing the image.");
    return true;
  } else {
    System.out.println("Done processing the image.");
    return false;
  }
}
```

## Double Buffering

Not only are images useful for storing pictures, as we've just shown, but you can also use them as offscreen drawing surfaces. This allows you to render any image, including text and graphics, to an offscreen buffer that you can display at a later time. The advantage to doing this is that the image is seen only when it is complete. Drawing a complicated image could take several milliseconds or more, which can be seen by the user as flashing or flickering. This flashing is distracting and causes the user to perceive your rendering as slower than it actually is. Use of an offscreen image to reduce flicker is called *double buffering*, because the

screen is considered a buffer for pixels, and the offscreen image is the second buffer, where you can prepare pixels for display.

Earlier in this chapter, you saw how to create a blank **Image** object. Now you will see how to draw on that image rather than the screen. As you recall from earlier chapters, you need a **Graphics** object in order to use any of Java's rendering methods. Conveniently, the **Graphics** object that you can use to draw on an **Image** is available via the **getGraphics( )** method. Here is a code fragment that creates a new image, obtains its graphics context, and fills the entire image with red pixels:

```
Canvas c = new Canvas();
Image test = c.createImage(200, 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200, 100);
```

Once you have constructed and filled an offscreen image, it will still not be visible. To actually display the image, call **drawImage( )**. Here is an example that draws a time-consuming image to demonstrate the difference that double buffering can make in perceived drawing time:

```
/*
 * <applet code=DoubleBuffer width=250 height=250>
 * </applet>
 */
 import java.awt.*;
 import java.awt.event.*;
 import java.applet.*;

 public class DoubleBuffer extends Applet {
   int gap = 3;
   int mx, my;
   boolean flicker = true;
   Image buffer = null;
   int w, h;

   public void init() {
     Dimension d = getSize();
     w = d.width;
     h = d.height;
     buffer = createImage(w, h);
     addMouseMotionListener(new MouseMotionAdapter() {
       public void mouseDragged(MouseEvent me) {
         mx = me.getX();
         my = me.getY();
         flicker = false;
         repaint();
       }
       public void mouseMoved(MouseEvent me) {
         mx = me.getX();
         my = me.getY();
         flicker = true;
```

```
        repaint();
      }
    });
  }

  public void paint(Graphics g) {
    Graphics screengc = null;

    if (!flicker) {
      screengc = g;
      g = buffer.getGraphics();
    }

    g.setColor(Color.blue);
    g.fillRect(0, 0, w, h);

    g.setColor(Color.red);
    for (int i=0; i<w; i+=gap)
      g.drawLine(i, 0, w-i, h);
    for (int i=0; i<h; i+=gap)
      g.drawLine(0, i, w, h-i);

    g.setColor(Color.black);
    g.drawString("Press mouse button to double buffer", 10, h/2);

    g.setColor(Color.yellow);
    g.fillOval(mx - gap, my - gap, gap*2+1, gap*2+1);

    if (!flicker) {
      screengc.drawImage(buffer, 0, 0, null);
    }
  }

  public void update(Graphics g) {
    paint(g);
  }
}
```

This simple applet has a complicated **paint( )** method. It fills the background with blue and then draws a red moiré pattern on top of that. It paints some black text on top of that and then paints a yellow circle centered at the coordinates **mx**, **my**. The **mouseMoved( )** and **mouseDragged( )** methods are overridden to track the mouse position. These methods are identical, except for the setting of the **flicker** Boolean variable. **mouseMoved( )** sets **flicker** to **true**, and **mouseDragged( )** sets it to **false**. This has the effect of calling **repaint( )** with **flicker** set to **true** when the mouse is moved (but no button is pressed) and set to **false** when the mouse is dragged with any button pressed.

When **paint( )** gets called with **flicker** set to **true**, we see each drawing operation as it is executed on the screen. In the case where a mouse button is pressed and **paint( )** is called with **flicker** set to **false**, we see quite a different picture. The **paint( )** method swaps the **Graphics** reference **g** with the graphics context that refers to the offscreen canvas, **buffer**, which we created in **init( )**. Then all of the drawing operations are invisible. At the end of **paint( )**, we simply call **drawImage( )** to show the results of these drawing methods all at once.

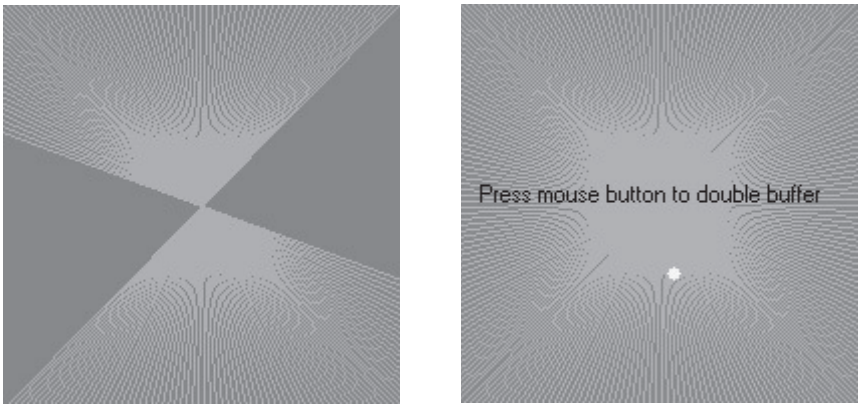



**Figure 27-2** Output from **DoubleBuffer** without (left) and with (right) double buffering

Notice that it is okay to pass in a **null** as the fourth parameter to **drawImage( )**. This is the parameter used to pass an **ImageObserver** object that receives notification of image events. Since this is an image that is not being produced from a network stream, we have no need for notification. The left snapshot in Figure 27-2 is what the applet looks like with the mouse button not pressed. As you can see, the image was in the middle of repainting when this snapshot was taken. The right snapshot shows how, when a mouse button is pressed, the image is always complete and clean due to double buffering.

## MediaTracker

A **MediaTracker** is an object that will check the status of an arbitrary number of images in parallel. To use **MediaTracker**, you create a new instance and use its **addImage( )** method to track the loading status of an image. **addImage( )** has the following general forms:

void addImage(Image *imgObj*, int *imgID*)
void addImage(Image *imgObj*, int *imgID*, int *width*, int *height*)

Here, *imgObj* is the image being tracked. Its identification number is passed in *imgID*. ID numbers do not need to be unique. You can use the same number with several images as a means of identifying them as part of a group. Furthermore, images with lower IDs are given priority over those with higher IDs when loading. In the second form, *width* and *height* specify the dimensions of the object when it is displayed.

Once you've registered an image, you can check whether it's loaded, or you can wait for it to completely load. To check the status of an image, call **checkID( )**. The version used in this chapter is shown here:

boolean checkID(int *imgID*)

Here, *imgID* specifies the ID of the image you want to check. The method returns **true** if all images that have the specified ID have been loaded (or if an error or user-abort has terminated loading). Otherwise, it returns **false**. You can use the **checkAll( )** method to see if all images being tracked have been loaded.

You should use **MediaTracker** when loading a group of images. If all of the images that you're interested in aren't downloaded, you can display something else to entertain the user until they all arrive.

---

**CAUTION** If you use **MediaTracker** once you've called **addImage( )** on an image, a reference in **MediaTracker** will prevent the system from garbage collecting it. If you want the system to be able to garbage collect images that were being tracked, make sure it can collect the **MediaTracker** instance as well.

Here's an example that loads a three-image slide show and displays a nice bar chart of the loading progress:

```
/*
 * <applet code="TrackedImageLoad" width=400 height=345>
 * <param name="img"
 * value="Lilies+SunFlower+ConeFlowers">
 * </applet>
 */
import java.util.*;
import java.applet.*;
import java.awt.*;

public class TrackedImageLoad extends Applet implements Runnable {
  MediaTracker tracker;
  int tracked;
  int frame_rate = 5;
  int current_img = 0;
  Thread motor;
  static final int MAXIMAGES = 10;
  Image img[] = new Image[MAXIMAGES];
  String name[] = new String[MAXIMAGES];
  volatile boolean stopFlag;

  public void init() {
    tracker = new MediaTracker(this);
    StringTokenizer st = new StringTokenizer(getParameter("img"),
                                             "+");

    while(st.hasMoreTokens() && tracked <= MAXIMAGES) {
      name[tracked] = st.nextToken();
      img[tracked] = getImage(getDocumentBase(),
                              name[tracked] + ".jpg");
      tracker.addImage(img[tracked], tracked);
      tracked++;
    }
  }

  public void paint(Graphics g) {
    String loaded = "";
    int donecount = 0;

    for(int i=0; i<tracked; i++) {
      if (tracker.checkID(i, true)) {
```

```
        donecount++;
        loaded += name[i] + " ";
      }
    }

    Dimension d = getSize();
    int w = d.width;
    int h = d.height;

    if (donecount == tracked) {
      frame_rate = 1;
      Image i = img[current_img++];
      int iw = i.getWidth(null);
      int ih = i.getHeight(null);
      g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
      if (current_img >= tracked)
        current_img = 0;
    } else {
      int x = w * donecount / tracked;
      g.setColor(Color.black);
      g.fillRect(0, h/3, x, 16);
      g.setColor(Color.white);
      g.fillRect(x, h/3, w-x, 16);
      g.setColor(Color.black);
      g.drawString(loaded, 10, h/2);
    }
  }

  public void start() {
    motor = new Thread(this);
    stopFlag = false;
    motor.start();
  }

  public void stop() {
    stopFlag = true;
  }

  public void run() {
    motor.setPriority(Thread.MIN_PRIORITY);
    while (true) {
      repaint();
      try {
        Thread.sleep(1000/frame_rate);
      } catch (InterruptedException e) {
        System.out.println("Interrupted");
        return;
      }
      if(stopFlag)
        return;
     }
   }
}
```

This example creates a new **MediaTracker** in the **init( )** method and then adds each of the named images as a tracked image with **addImage( )**. In the **paint( )** method, it calls **checkID( )** on each of the images that we're tracking. If all of the images are loaded, they are displayed. If not, a simple bar chart of the number of images loaded is shown, with the names of the fully loaded images displayed underneath the bar.

# ImageProducer

**ImageProducer** is an interface for objects that want to produce data for images. An object that implements the **ImageProducer** interface will supply integer or byte arrays that represent image data and produce **Image** objects. As you saw earlier, one form of the **createImage( )** method takes an **ImageProducer** object as its argument. There are two image producers contained in **java.awt.image**: **MemoryImageSource** and **FilteredImageSource**. Here, we will examine **MemoryImageSource** and create a new **Image** object from data generated in an applet.



## MemoryImageSource

**MemoryImageSource** is a class that creates a new **Image** from an array of data. It defines several constructors. Here is the one we will be using:

MemoryImageSource(int *width*, int *height*, int *pixel*[ ], int *offset*,
int *scanLineWidth*)

The **MemoryImageSource** object is constructed out of the array of integers specified by *pixel*, in the default RGB color model to produce data for an **Image** object. In the default color model, a pixel is an integer with Alpha, Red, Green, and Blue (0xAARRGGBB). The Alpha value represents a degree of transparency for the pixel. Fully transparent is 0 and fully opaque is 255. The width and height of the resulting image are passed in *width* and *height*. The starting point in the pixel array to begin reading data is passed in *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

The following short example generates a **MemoryImageSource** object using a variation on a simple algorithm (a bitwise-exclusive-OR of the x and y address of each pixel) from the book *Beyond Photography, The Digital Darkroom* by Gerard J. Holzmann (Prentice Hall, 1988).

```
/*
 * <applet code="MemoryImageGenerator" width=256 height=256>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class MemoryImageGenerator extends Applet {
  Image img;
  public void init() {
    Dimension d = getSize();
    int w = d.width;
```

```
    int h = d.height;
    int pixels[] = new int[w * h];
    int i = 0;

    for(int y=0; y<h; y++) {
      for(int x=0; x<w; x++) {
        int r = (x^y)&0xff;
        int g = (x*2^y*2)&0xff;
        int b = (x*4^y*4)&0xff;
        pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
      }
    }
    img = createImage(new MemoryImageSource(w, h, pixels, 0, w));
  }

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, this);
  }
}
```

The data for the new **MemoryImageSource** is created in the **init( )** method. An array of integers is created to hold the pixel values; the data is generated in the nested **for** loops where the **r**, **g**, and **b** values get shifted into a pixel in the **pixels** array. Finally, **createImage( )** is called with a new instance of a **MemoryImageSource** created from the raw pixel data as its parameter. Figure 27-3 shows the image when we run the applet. (It looks much nicer in color.)

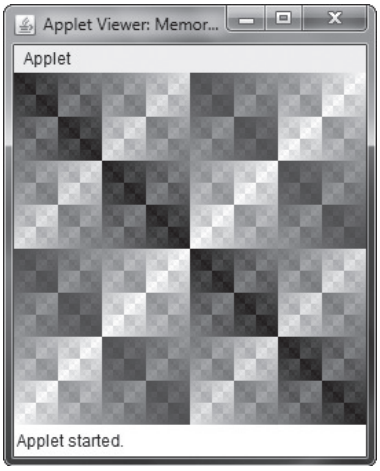



**Figure 27-3** Sample output from **MemoryImageGenerator**

# ImageConsumer

**ImageConsumer** is an interface for objects that want to take pixel data from images and supply it as another kind of data. This, obviously, is the opposite of **ImageProducer**, described earlier. An object that implements the **ImageConsumer** interface is going to create **int** or **byte** arrays that represent pixels from an **Image** object. We will examine the **PixelGrabber** class, which is a simple implementation of the **ImageConsumer** interface.

## PixelGrabber

The **PixelGrabber** class is defined within **java.lang.image**. It is the inverse of the **MemoryImageSource** class. Rather than constructing an image from an array of pixel values, it takes an existing image and *grabs* the pixel array from it. To use **PixelGrabber**, you first create an array of **int**s big enough to hold the pixel data, and then you create a **PixelGrabber** instance passing in the rectangle that you want to grab. Finally, you call **grabPixels( )** on that instance.

The **PixelGrabber** constructor that is used in this chapter is shown here:

PixelGrabber(Image *imgObj*, int *left*, int *top*, int *width*, int *height*, int *pixel* [ ],
int *offset*, int *scanLineWidth*)

Here, *imgObj* is the object whose pixels are being grabbed. The values of *left* and *top* specify the upper-left corner of the rectangle, and *width* and *height* specify the dimensions of the rectangle from which the pixels will be obtained. The pixels will be stored in *pixel* beginning at *offset*. The width of a scan line (which is often the same as the width of the image) is passed in *scanLineWidth*.

**grabPixels( )** is defined like this:

boolean grabPixels( )
throws InterruptedException

boolean grabPixels(long *milliseconds*)
throws InterruptedException

Both methods return **true** if successful and **false** otherwise. In the second form, *milliseconds* specifies how long the method will wait for the pixels. Both throw **InterruptedException** if execution is interrupted by another thread.

Here is an example that grabs the pixels from an image and then creates a histogram of pixel brightness. The *histogram* is simply a count of pixels that are a certain brightness for all brightness settings between 0 and 255. After the applet paints the image, it draws the histogram over the top.

```
/*
 * <applet code=HistoGrab width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * </applet> */
import java.applet.*;
import java.awt.* ;
import java.awt.image.* ;
```

```
public class HistoGrab extends Applet {
  Dimension d;
  Image img;
  int iw, ih;
  int pixels[];
  int w, h;
  int hist[] = new int[256];
  int max_hist = 0;

  public void init() {
    d = getSize();
    w = d.width;
    h = d.height;

    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);

      t.addImage(img, 0);
      t.waitForID(0);
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      pixels = new int[iw * ih];
      PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                         pixels, 0, iw);
      pg.grabPixels();
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
      return;
    }

    for (int i=0; i<iw*ih; i++) {
      int p = pixels[i];
      int r = 0xff & (p >> 16);
      int g = 0xff & (p >> 8);
      int b = 0xff & (p);
      int y = (int) (.33 * r + .56 * g + .11 * b);
      hist[y]++;
    }
    for (int i=0; i<256; i++) {
      if (hist[i] > max_hist)
        max_hist = hist[i];
    }
  }

  public void update() {}

  public void paint(Graphics g) {
    g.drawImage(img, 0, 0, null);
    int x = (w - 256) / 2;
    int lasty = h - h * hist[0] / max_hist;
```

```
    for (int i=0; i<256; i++, x++) {
      int y = h - h * hist[i] / max_hist;
      g.setColor(new Color(i, i, i));
      g.fillRect(x, y, 1, h);
      g.setColor(Color.red);
      g.drawLine(x-1,lasty,x,y);
      lasty = y;
    }
  }
}
```

Figure 27-4 shows an example image and its histogram.



## ImageFilter

Given the **ImageProducer** and **ImageConsumer** interface pair—and their concrete classes **MemoryImageSource** and **PixelGrabber**—you can create an arbitrary set of translation filters that takes a source of pixels, modifies them, and passes them on to an arbitrary consumer. This mechanism is analogous to the way concrete classes are created from the abstract I/O classes **InputStream**, **OutputStream**, **Reader**, and **Writer** (described in Chapter 20). This stream model for images is completed by the introduction of the **ImageFilter** class. Some subclasses of **ImageFilter** in the **java.awt.image** package are **AreaAveragingScaleFilter**, **CropImageFilter**, **ReplicateScaleFilter**, and **RGBImageFilter**. There is also an implementation of **ImageProducer** called **FilteredImageSource**, which takes an arbitrary **ImageFilter** and wraps it around an **ImageProducer** to filter the pixels it produces. An instance of **FilteredImageSource**

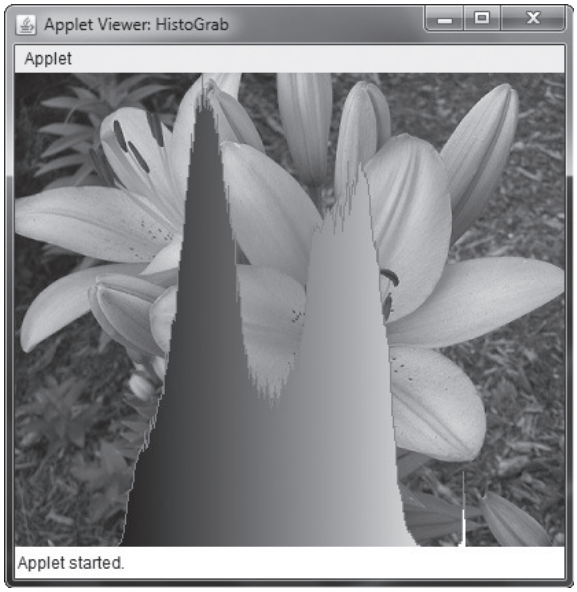



**Figure 27-4** Sample output from **HistoGrab**

can be used as an **ImageProducer** in calls to **createImage( )**, in much the same way that **BufferedInputStream**s can be passed off as **InputStream**s.

In this chapter, we examine two filters: **CropImageFilter** and **RGBImageFilter**.

## CropImageFilter

**CropImageFilter** filters an image source to extract a rectangular region. One situation in which this filter is valuable is where you want to use several small images from a single, larger source image. Loading twenty 2K images takes much longer than loading a single 40K image that has many frames of an animation tiled into it. If every subimage is the same size, then you can easily extract these images by using **CropImageFilter** to disassemble the block once your program starts. Here is an example that creates 16 images taken from a single image. The tiles are then scrambled by swapping a random pair from the 16 images 32 times.

```
/*
 * <applet code=TileImage width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.image.*;

public class TileImage extends Applet {
  Image img;
  Image cell[] = new Image[4*4];
  int iw, ih;
  int tw, th;

  public void init() {
    try {
      img = getImage(getDocumentBase(), getParameter("img"));
      MediaTracker t = new MediaTracker(this);
      t.addImage(img, 0);
      t.waitForID(0);
      iw = img.getWidth(null);
      ih = img.getHeight(null);
      tw = iw / 4;
      th = ih / 4;
      CropImageFilter f;
      FilteredImageSource fis;
      t = new MediaTracker(this);
      for (int y=0; y<4; y++) {
        for (int x=0; x<4; x++) {
          f = new CropImageFilter(tw*x, th*y, tw, th);
          fis = new FilteredImageSource(img.getSource(), f);
          int i = y*4+x;
          cell[i] = createImage(fis);
          t.addImage(cell[i], i);
        }
```

```
      }
      t.waitForAll();
      for (int i=0; i<32; i++) {
        int si = (int)(Math.random() * 16);
        int di = (int)(Math.random() * 16);
        Image tmp = cell[si];
        cell[si] = cell[di];
        cell[di] = tmp;
      }
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
    }
  }

  public void update(Graphics g) {
    paint(g);
  }

  public void paint(Graphics g) {
    for (int y=0; y<4; y++) {
      for (int x=0; x<4; x++) {
        g.drawImage(cell[y*4+x], x * tw, y * th, null);
      }
    }
  }
}
```

Figure 27-5 shows the flowers image scrambled by the **TileImage** applet.


**Figure 27-5** Sample output from **TileImage**

## RGBImageFilter

The **RGBImageFilter** is used to convert one image to another, pixel by pixel, transforming the colors along the way. This filter could be used to brighten an image, to increase its contrast, or even to convert it to grayscale.

To demonstrate **RGBImageFilter**, we have developed a somewhat complicated example that employs a dynamic plug-in strategy for image-processing filters. We've created an interface for generalized image filtering so that an applet can simply load these filters based on **<param>** tags without having to know about all of the **ImageFilter**s in advance. This example consists of the main applet class called **ImageFilterDemo**, the interface called **PlugInFilter**, and a utility class called **LoadedImage**, which encapsulates some of the **MediaTracker** methods we've been using in this chapter. Also included are three filters—**Grayscale**, **Invert**, and **Contrast**—which simply manipulate the color space of the source image using **RGBImageFilter**s, and two more classes—**Blur** and **Sharpen**—which do more complicated "convolution" filters that change pixel data based on the pixels surrounding each pixel of source data. **Blur** and **Sharpen** are subclasses of an abstract helper class called **Convolver**. Let's look at each part of our example.

### ImageFilterDemo.java

The **ImageFilterDemo** class is the applet framework for our sample image filters. It employs a simple **BorderLayout**, with a **Panel** at the *South* position to hold the buttons that will represent each filter. A **Label** object occupies the *North* slot for informational messages about filter progress. The *Center* is where the image (which is encapsulated in the **LoadedImage Canvas** subclass, described later) is put. We parse the buttons/filters out of the **filters <param>** tag, separating them with +'s using a **StringTokenizer**.

The **actionPerformed( )** method is interesting because it uses the label from a button as the name of a filter class that it tries to load with **(PlugInFilter) Class.forName(a).newInstance( )**. This method is robust and takes appropriate action if the button does not correspond to a proper class that implements **PlugInFilter**.

```
/*
 * <applet code=ImageFilterDemo width=400 height=345>
 * <param name=img value=Lilies.jpg>
 * <param name=filters value="Grayscale+Invert+Contrast+Blur+Sharpen">
 * </applet>
 */
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class ImageFilterDemo extends Applet implements ActionListener {
  Image img;
  PlugInFilter pif;
  Image fimg;
  Image curImg;
  LoadedImage lim;
  Label lab;
  Button reset;
```

```
  public void init() {
    setLayout(new BorderLayout());
    Panel p = new Panel();

    add(p, BorderLayout.SOUTH);
    reset = new Button("Reset");
    reset.addActionListener(this);
    p.add(reset);
    StringTokenizer st = new StringTokenizer(getParameter("filters"), "+");

    while(st.hasMoreTokens()) {
      Button b = new Button(st.nextToken());
      b.addActionListener(this);
      p.add(b);
    }

    lab = new Label("");
    add(lab, BorderLayout.NORTH);

    img = getImage(getDocumentBase(), getParameter("img"));
    lim = new LoadedImage(img);
    add(lim, BorderLayout.CENTER);
  }

  public void actionPerformed(ActionEvent ae) {
    String a = "";

    try {
      a = ae.getActionCommand();
      if (a.equals("Reset")) {
        lim.set(img);
        lab.setText("Normal");
      }
      else {
        pif = (PlugInFilter) Class.forName(a).newInstance();
        fimg = pif.filter(this, img);
        lim.set(fimg);
        lab.setText("Filtered: " + a);
      }
      repaint();
    } catch (ClassNotFoundException e) {
      lab.setText(a + " not found");
      lim.set(img);
      repaint();
    } catch (InstantiationException e) {
      lab.setText("couldn't new " + a);
    } catch (IllegalAccessException e) {
      lab.setText("no access: " + a);
    }
  }
}
```

Figure 27-6 shows what the applet looks like when it is first loaded using the applet tag shown at the top of this source file.

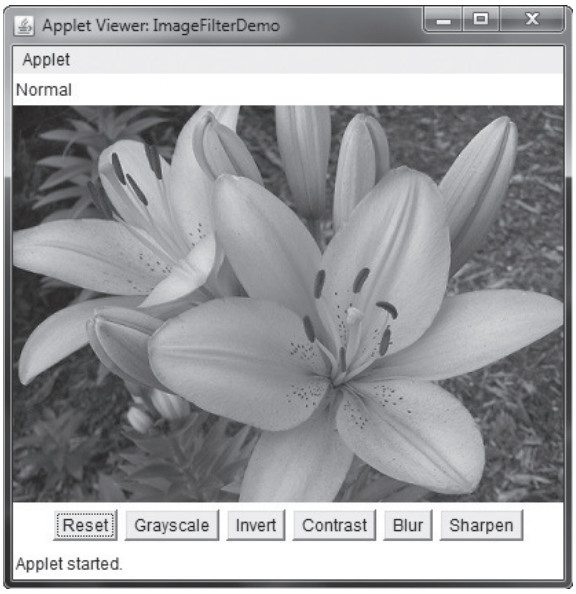



**Figure 27-6** Sample normal output from **ImageFilterDemo**

## PlugInFilter.java

**PlugInFilter** is a simple interface used to abstract image filtering. It has only one method, **filter( )**, which takes the applet and the source image and returns a new image that has been filtered in some way.

```
interface PlugInFilter {
  java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

## LoadedImage.java

**LoadedImage** is a convenient subclass of **Canvas**, which takes an image at construction time and synchronously loads it using **MediaTracker**. **LoadedImage** then behaves properly inside of **LayoutManager** control, because it overrides the **getPreferredSize( )** and **getMinimumSize( )** methods. Also, it has a method called **set( )** that can be used to set a new **Image** to be displayed in this **Canvas**. That is how the filtered image is displayed after the plug-in is finished.

```
import java.awt.*;

public class LoadedImage extends Canvas {
  Image img;

  public LoadedImage(Image i) {
    set(i);
  }

  void set(Image i) {
    MediaTracker mt = new MediaTracker(this);
```

```
    mt.addImage(i, 0);
    try {
      mt.waitForAll();
    } catch (InterruptedException e) {
      System.out.println("Interrupted");
      return;
    }
    img = i;
    repaint();
  }

  public void paint(Graphics g) {
    if (img == null) {
      g.drawString("no image", 10, 30);
    } else {
      g.drawImage(img, 0, 0, this);
    }
  }

  public Dimension getPreferredSize() {
    return new Dimension(img.getWidth(this), img.getHeight(this));
  }

  public Dimension getMinimumSize() {
    return getPreferredSize();
  }
}
```



## Grayscale.java

The **Grayscale** filter is a subclass of **RGBImageFilter**, which means that **Grayscale** can use itself as the **ImageFilter** parameter to **FilteredImageSource**'s constructor. Then all it needs to do is override **filterRGB( )** to change the incoming color values. It takes the red, green, and blue values and computes the brightness of the pixel, using the NTSC (National Television Standards Committee) color-to-brightness conversion factor. It then simply returns a gray pixel that is the same brightness as the color source.

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

class Grayscale extends RGBImageFilter implements PlugInFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = (rgb >> 16) & 0xff;
    int g = (rgb >> 8) & 0xff;
    int b = rgb & 0xff;
    int k = (int) (.56 * g + .33 * r + .11 * b);
    return (0xff000000 | k << 16 | k << 8 | k);
  }
}
```

## Invert.java

The **Invert** filter is also quite simple. It takes apart the red, green, and blue channels and then inverts them by subtracting them from 255. These inverted values are packed back into a pixel value and returned.

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

class Invert extends RGBImageFilter implements PlugInFilter {
  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = 0xff - (rgb >> 16) & 0xff;
    int g = 0xff - (rgb >> 8) & 0xff;
    int b = 0xff - rgb & 0xff;
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

Figure 27-7 shows the image after it has been run through the **Invert** filter.

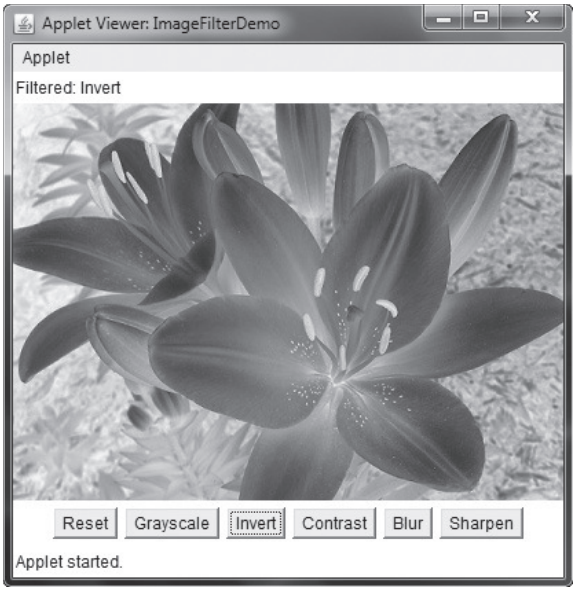



**Figure 27-7** Using the **Invert** filter with **ImageFilterDemo**

## Contrast.java

The **Contrast** filter is very similar to **Grayscale**, except its override of **filterRGB( )** is slightly more complicated. The algorithm it uses for contrast enhancement takes the red, green, and blue values separately and boosts them by 1.2 times if they are already brighter than 128. If they are below 128, then they are divided by 1.2. The boosted values are properly clamped at 255 by the **multclamp( )** method.

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

public class Contrast extends RGBImageFilter implements PlugInFilter {

  public Image filter(Applet a, Image in) {
    return a.createImage(new FilteredImageSource(in.getSource(), this));
  }

  private int multclamp(int in, double factor) {
    in = (int) (in * factor);
    return in > 255 ? 255 : in;
  }

  double gain = 1.2;
  private int cont(int in) {
    return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
  }

  public int filterRGB(int x, int y, int rgb) {
    int r = cont((rgb >> 16) & 0xff);
    int g = cont((rgb >> 8) & 0xff);
    int b = cont(rgb & 0xff);
    return (0xff000000 | r << 16 | g << 8 | b);
  }
}
```

Figure 27-8 shows the image after **Contrast** is pressed.

## Convolver.java

The abstract class **Convolver** handles the basics of a convolution filter by implementing the **ImageConsumer** interface to move the source pixels into an array called **imgpixels**. It also creates a second array called **newimgpixels** for the filtered data. Convolution filters sample a small rectangle of pixels around each pixel in an image, called the *convolution kernel*. This area, 3 x 3 pixels in this demo, is used to decide how to change the center pixel in the area.

**NOTE** The reason that the filter can't modify the **imgpixels** array in place is that the next pixel on a scan line would try to use the original value for the previous pixel, which would have just been filtered away.


**Figure 27-8** Using the **Contrast** filter with **ImageFilterDemo**

The two concrete subclasses, shown in the next section, simply implement the **convolve( )** method, using **imgpixels** for source data and **newimgpixels** to store the result.

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

abstract class Convolver implements ImageConsumer, PlugInFilter {
  int width, height;
  int imgpixels[], newimgpixels[];
  boolean imageReady = false;

  abstract void convolve(); // filter goes here...

  public Image filter(Applet a, Image in) {
    imageReady = false;
    in.getSource().startProduction(this);

    waitForImage();
    newimgpixels = new int[width*height];

    try {
      convolve();
    } catch (Exception e) {
      System.out.println("Convolver failed: " + e);
      e.printStackTrace();
    }
```

```
  return a.createImage(
    new MemoryImageSource(width, height, newimgpixels, 0, width));
}

synchronized void waitForImage() {
  try {
    while(!imageReady) wait();
  } catch (Exception e) {
    System.out.println("Interrupted");
  }
}

public void setProperties(java.util.Hashtable<?,?> dummy) { }
public void setColorModel(ColorModel dummy) { }
public void setHints(int dummy) { }

public synchronized void imageComplete(int dummy) {
imageReady = true;
  notifyAll();
}

public void setDimensions(int x, int y) {
  width = x;
  height = y;
  imgpixels = new int[x*y];
}

public void setPixels(int x1, int y1, int w, int h,
  ColorModel model, byte pixels[], int off, int scansize) {
  int pix, x, y, x2, y2, sx, sy;

  x2 = x1+w;
  y2 = y1+h;
  sy = off;
  for(y=y1; y<y2; y++) {
    sx = sy;
    for(x=x1; x<x2; x++) {
      pix = model.getRGB(pixels[sx++]);
      if((pix & 0xff000000) == 0)
        pix = 0x00ffffff;
      imgpixels[y*width+x] = pix;
    }
    sy += scansize;
  }
}

public void setPixels(int x1, int y1, int w, int h,
  ColorModel model, int pixels[], int off, int scansize) {
    int pix, x, y, x2, y2, sx, sy;

    x2 = x1+w;
    y2 = y1+h;
    sy = off;
```

```
      for(y=y1; y<y2; y++) {
        sx = sy;
        for(x=x1; x<x2; x++) {
          pix = model.getRGB(pixels[sx++]);
        if((pix & 0xff000000) == 0)
          pix = 0x00ffffff;
        imgpixels[y*width+x] = pix;
      }
      sy += scansize;
    }
  }
}
```

**NOTE** A built-in convolution filter called **ConvolveOp** is provided by **java.awt.image**. You may want to explore its capabilities on your own.

## Blur.java

The **Blur** filter is a subclass of **Convolver** and simply runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3 x 3 box surrounding it. The corresponding output pixel in **newimgpixels** is that average value.

```
public class Blur extends Convolver {
  public void convolve() {
    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            rs += r;
            gs += g;
            bs += b;
          }
        }

        rs /= 9;
        gs /= 9;
        bs /= 9;

       newimgpixels[y*width+x] = (0xff000000 |
                                  rs << 16 | gs << 8 | bs);
      }
    }
  }
}
```

Figure 27-9 shows the applet after **Blur**.


**Figure 27-9** Using the **Blur** filter with **ImageFilterDemo**



## Sharpen.java

The **Sharpen** filter is also a subclass of **Convolver** and is (more or less) the inverse of **Blur**. It runs through every pixel in the source image array, **imgpixels**, and computes the average of the 3 x 3 box surrounding it, not counting the center. The corresponding output pixel in **newimgpixels** has the difference between the center pixel and the surrounding average added to it. This basically says that if a pixel is 30 brighter than its surroundings, make it another 30 brighter. If, however, it is 10 darker, then make it another 10 darker. This tends to accentuate edges while leaving smooth areas unchanged.

```
public class Sharpen extends Convolver {

  private final int clamp(int c) {
    return (c > 255 ? 255 : (c < 0 ? 0 : c));
  }

  public void convolve() {
    int r0=0, g0=0, b0=0;

    for(int y=1; y<height-1; y++) {
      for(int x=1; x<width-1; x++) {
        int rs = 0;
        int gs = 0;
        int bs = 0;

        for(int k=-1; k<=1; k++) {
          for(int j=-1; j<=1; j++) {
```

```
            int rgb = imgpixels[(y+k)*width+x+j];
            int r = (rgb >> 16) & 0xff;
            int g = (rgb >> 8) & 0xff;
            int b = rgb & 0xff;
            if (j == 0 && k == 0) {
              r0 = r;
              g0 = g;
              b0 = b;
            } else {
              rs += r;
              gs += g;
              bs += b;
            }
          }
        }

        rs >>= 3;
        gs >>= 3;
        bs >>= 3;
        newimgpixels[y*width+x] = (0xff000000 |
                                   clamp(r0+r0-rs) << 16 |
                                   clamp(g0+g0-gs) << 8 |
                                   clamp(b0+b0-bs));
      }
    }
  }
}
```

Figure 27-10 shows the applet after **Sharpen**.

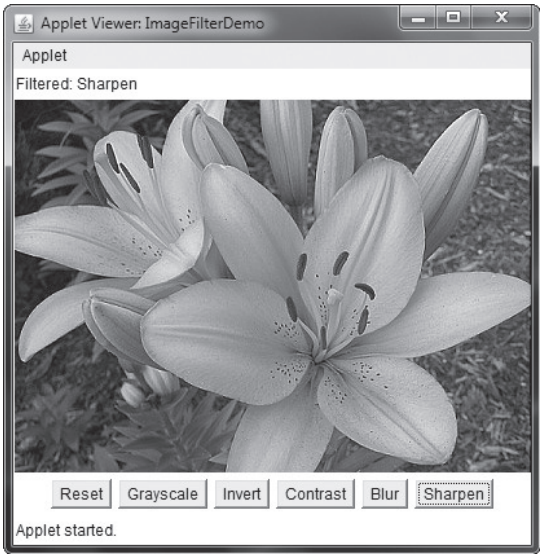



**Figure 27-10** Using the **Sharpen** filter with **ImageFilterDemo**

## Additional Imaging Classes

In addition to the imaging classes described in this chapter, **java.awt.image** supplies several others that offer enhanced control over the imaging process and that support advanced imaging techniques. Also available is the imaging package called **javax.imageio**. This package supports plug-ins that handle various image formats. If sophisticated graphical output is of special interest to you, then you will want to explore the additional classes found in **java.awt.image** and **javax.imageio**.

This page has been intentionally left blank

CHAPTER

# 28 The Concurrency Utilities

From the start, Java has provided built-in support for multithreading and synchronization. For example, new threads can be created by implementing **Runnable** or by extending **Thread**; synchronization is available by use of the **synchronized** keyword; and interthread communication is supported by the **wait( )** and **notify( )** methods that are defined by **Object**. In general, this built-in support for multithreading was one of Java's most important innovations and is still one of its major strengths.

However, as conceptually pure as Java's original support for multithreading is, it is not ideal for all applications—especially those that make intensive use of multiple threads. For example, the original multithreading support does not provide several high-level features, such as semaphores, thread pools, and execution managers, that facilitate the creation of intensively concurrent programs.

It is important to explain at the outset that many Java programs make use of multithreading and are, therefore, "concurrent." For example, many applets and servlets use multithreading. However, as it is used in this chapter, the term *concurrent program* refers to a program that makes *extensive, integral* use of concurrently executing threads. An example of such a program is one that uses separate threads to simultaneously compute the partial results of a larger computation. Another example is a program that coordinates the activities of several threads, each of which seeks access to information in a database. In this case, read-only accesses might be handled differently from those that require read/write capabilities.

To begin to handle the needs of a concurrent program, JDK 5 added the *concurrency utilities,* also commonly referred to as the *concurrent API.* The original set of concurrency utilities supplied many features that had long been wanted by programmers who develop concurrent applications. For example, it offered synchronizers (such as the semaphore), thread pools, execution managers, locks, several concurrent collections, and a streamlined way to use threads to obtain computational results.

Although the original concurrent API was impressive in its own right, it was significantly expanded by JDK 7. The most important addition was the *Fork/Join Framework.* The Fork/Join Framework facilitates the creation of programs that make use of multiple processors (such as those found in multicore systems). Thus, it streamlines the development of programs in

which two or more pieces execute with true simultaneity (that is, true parallel execution), not just time-slicing. As you can easily imagine, parallel execution can dramatically increase the speed of certain operations. Because multicore systems are now commonplace, the inclusion of the Fork/Join Framework was as timely as it was powerful. With the release of JDK 8, the Fork/Join Framework was further enhanced.

In addition, JDK 8 included some new features related to other parts of the concurrent API. Thus, the concurrent API continues to evolve and expand to meet the needs of the contemporary computing environment.

The original concurrent API was quite large, and the additions made by JDK 7 and JDK 8 have increased its size substantially. As you might expect, many of the issues surrounding the concurrency utilities are quite complex. It is beyond the scope of this book to discuss all of its facets. The preceding notwithstanding, it is important for all programmers to have a general, working knowledge of key aspects of the concurrent API. Even in programs that are not intensively parallel, features such as synchronizers, callable threads, and executors, are applicable to a wide variety of situations. Perhaps most importantly, because of the rise of multicore computers, solutions involving the Fork/Join Framework are becomming more common. For these reasons, this chapter presents an overview of several core features defined by the concurrency utilities and shows a number of examples that demonstrate their use. It concludes with an introduction to the Fork/Join Framework.

# The Concurrent API Packages

The concurrency utilities are contained in the **java.util.concurrent** package and in its two subpackages: **java.util.concurrent.atomic** and **java.util.concurrent.locks**. A brief overview of their contents is given here.

## java.util.concurrent

**java.util.concurrent** defines the core features that support alternatives to the built-in approaches to synchronization and interthread communication. It defines the following key features:

- Synchronizers
- Executors
- Concurrent collections
- The Fork/Join Framework

*Synchronizers* offer high-level ways of synchronizing the interactions between multiple threads. The synchronizer classes defined by **java.util.concurrent** are

| | |
|---|---|
| Semaphore | Implements the classic semaphore. |
| CountDownLatch | Waits until a specified number of events have occurred. |
| CyclicBarrier | Enables a group of threads to wait at a predefined execution point. |
| Exchanger | Exchanges data between two threads. |
| Phaser | Synchronizes threads that advance through multiple phases of an operation. |

Notice that each synchronizer provides a solution to a specific type of synchronization problem. This enables each synchronizer to be optimized for its intended use. In the past, these types of synchronization objects had to be crafted by hand. The concurrent API standardizes them and makes them available to all Java programmers.

*Executors* manage thread execution. At the top of the executor hierarchy is the **Executor** interface, which is used to initiate a thread. **ExecutorService** extends **Executor** and provides methods that manage execution. There are three implementations of **ExecutorService**: **ThreadPoolExecutor**, **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. **java.util.concurrent** also defines the **Executors** utility class, which includes a number of static methods that simplify the creation of various executors.

Related to executors are the **Future** and **Callable** interfaces. A **Future** contains a value that is returned by a thread after it executes. Thus, its value becomes defined "in the future," when the thread terminates. **Callable** defines a thread that returns a value.

**java.util.concurrent** defines several concurrent collection classes, including **ConcurrentHashMap**, **ConcurrentLinkedQueue**, and **CopyOnWriteArrayList**. These offer concurrent alternatives to their related classes defined by the Collections Framework.

The *Fork/Join Framework* supports parallel programming. Its main classes are **ForkJoinTask**, **ForkJoinPool**, **RecursiveTask**, and **RecursiveAction**.

Finally, to better handle thread timing, **java.util.concurrent** defines the **TimeUnit** enumeration.

### java.util.concurrent.atomic

**java.util.concurrent.atomic** facilitates the use of variables in a concurrent environment. It provides a means of efficiently updating the value of a variable without the use of locks. This is accomplished through the use of classes, such as **AtomicInteger** and **AtomicLong**, and methods, such as **compareAndSet( )**, **decrementAndGet( )**, and **getAndSet( )**. These methods execute as a single, non-interruptible operation.

### java.util.concurrent.locks

**java.util.concurrent.locks** provides an alternative to the use of synchronized methods. At the core of this alternative is the **Lock** interface, which defines the basic mechanism used to acquire and relinquish access to an object. The key methods are **lock( )**, **tryLock( )**, and **unlock( )**. The advantage to using these methods is greater control over synchronization.

The remainder of this chapter takes a closer look at the constituents of the concurrent API.

## Using Synchronization Objects

Synchronization objects are supported by the **Semaphore**, **CountDownLatch**, **CyclicBarrier**, **Exchanger**, and **Phaser** classes. Collectively, they enable you to handle several formerly difficult synchronization situations with ease. They are also applicable to a wide range of programs—even those that contain only limited concurrency. Because the synchronization objects will be of interest to nearly all Java programs, each is examined here in some detail.

## Semaphore

The synchronization object that many readers will immediately recognize is **Semaphore**, which implements a classic semaphore. A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are *permits* that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

In general, to use a semaphore, the thread that wants access to the shared resource tries to acquire a permit. If the semaphore's count is greater than zero, then the thread acquires a permit, which causes the semaphore's count to be decremented. Otherwise, the thread will be blocked until a permit can be acquired. When the thread no longer needs access to the shared resource, it releases the permit, which causes the semaphore's count to be incremented. If there is another thread waiting for a permit, then that thread will acquire a permit at that time. Java's **Semaphore** class implements this mechanism.

**Semaphore** has the two constructors shown here:

Semaphore(int *num*)
Semaphore(int *num*, boolean *how*)

Here, *num* specifies the initial permit count. Thus, *num* specifies the number of threads that can access a shared resource at any one time. If *num* is one, then only one thread can access the resource at any one time. By default, waiting threads are granted a permit in an undefined order. By setting *how* to **true**, you can ensure that waiting threads are granted a permit in the order in which they requested access.

To acquire a permit, call the **acquire( )** method, which has these two forms:

void acquire( ) throws InterruptedException
void acquire(int *num*) throws InterruptedException

The first form acquires one permit. The second form acquires *num* permits. Most often, the first form is used. If the permit cannot be granted at the time of the call, then the invoking thread suspends until the permit is available.

To release a permit, call **release( )**, which has these two forms:

void release( )
void release(int *num*)

The first form releases one permit. The second form releases the number of permits specified by *num*.

To use a semaphore to control access to a resource, each thread that wants to use that resource must first call **acquire( )** before accessing the resource. When the thread is done with the resource, it must call **release( )**. Here is an example that illustrates the use of a semaphore:

```
// A simple semaphore example.

import java.util.concurrent.*;

class SemDemo {
```

```
  public static void main(String args[]) {
    Semaphore sem = new Semaphore(1);

    new IncThread(sem, "A");
    new DecThread(sem, "B");

  }
}

// A shared resource.
class Shared {
  static int count = 0;
}

// A thread of execution that increments count.
class IncThread implements Runnable {
  String name;
  Semaphore sem;

  IncThread(Semaphore s, String n) {
    sem = s;
    name = n;
    new Thread(this).start();
  }

  public void run() {

    System.out.println("Starting " + name);

    try {
      // First, get a permit.
      System.out.println(name + " is waiting for a permit.");
      sem.acquire();
      System.out.println(name + " gets a permit.");

      // Now, access shared resource.
      for(int i=0; i < 5; i++) {
        Shared.count++;
        System.out.println(name + ": " + Shared.count);

        // Now, allow a context switch -- if possible.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    // Release the permit.
    System.out.println(name + " releases the permit.");
    sem.release();
  }
}
```

```
// A thread of execution that decrements count.
class DecThread implements Runnable {
  String name;
  Semaphore sem;

  DecThread(Semaphore s, String n) {
    sem = s;
    name = n;
    new Thread(this).start();
  }

  public void run() {

    System.out.println("Starting " + name);

    try {
      // First, get a permit.
      System.out.println(name + " is waiting for a permit.");
      sem.acquire();
      System.out.println(name + " gets a permit.");

      // Now, access shared resource.
      for(int i=0; i < 5; i++) {
        Shared.count--;
        System.out.println(name + ": " + Shared.count);

        // Now, allow a context switch -- if possible.
        Thread.sleep(10);
      }
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    // Release the permit.
    System.out.println(name + " releases the permit.");
    sem.release();
  }
}
```

The output from the program is shown here. (The precise order in which the threads execute may vary.)

```
Starting A
A is waiting for a permit.
A gets a permit.
A: 1
Starting B
B is waiting for a permit.
A: 2
A: 3
A: 4
A: 5
A releases the permit.
B gets a permit.
```

```
B: 4
B: 3
B: 2
B: 1
B: 0
B releases the permit.
```

The program uses a semaphore to control access to the **count** variable, which is a static variable within the **Shared** class. **Shared.count** is incremented five times by the **run( )** method of **IncThread** and decremented five times by **DecThread**. To prevent these two threads from accessing **Shared.count** at the same time, access is allowed only after a permit is acquired from the controlling semaphore. After access is complete, the permit is released. In this way, only one thread at a time will access **Shared.count**, as the output shows.

In both **IncThread** and **DecThread**, notice the call to **sleep( )** within **run( )**. It is used to "prove" that accesses to **Shared.count** are synchronized by the semaphore. In **run( )**, the call to **sleep( )** causes the invoking thread to pause between each access to **Shared.count**. This would normally enable the second thread to run. However, because of the semaphore, the second thread must wait until the first has released the permit, which happens only after all accesses by the first thread are complete. Thus, **Shared.count** is first incremented five times by **IncThread** and then decremented five times by **DecThread**. The increments and decrements are *not* intermixed.

Without the use of the semaphore, accesses to **Shared.count** by both threads would have occurred simultaneously, and the increments and decrements would be intermixed. To confirm this, try commenting out the calls to **acquire( )** and **release( )**. When you run the program, you will see that access to **Shared.count** is no longer synchronized, and each thread accesses it as soon as it gets a timeslice.

Although many uses of a semaphore are as straightforward as that shown in the preceding program, more intriguing uses are also possible. Here is an example. The following program reworks the producer/consumer program shown in Chapter 11 so that it uses two semaphores to regulate the producer and consumer threads, ensuring that each call to **put( )** is followed by a corresponding call to **get( )**:

```
// An implementation of a producer and consumer
// that use semaphores to control synchronization.

import java.util.concurrent.Semaphore;

class Q {
  int n;

  // Start with consumer semaphore unavailable.
  static Semaphore semCon = new Semaphore(0);
  static Semaphore semProd = new Semaphore(1);

  void get() {
    try {
      semCon.acquire();
    } catch(InterruptedException e) {
      System.out.println("InterruptedException caught");
    }
```

```
    System.out.println("Got: " + n);
    semProd.release();
  }

  void put(int n) {
    try {
      semProd.acquire();
    } catch(InterruptedException e) {
      System.out.println("InterruptedException caught");
    }

    this.n = n;
    System.out.println("Put: " + n);
    semCon.release();
  }
}

class Producer implements Runnable {
  Q q;

  Producer(Q q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }

  public void run() {
    for(int i=0; i < 20; i++) q.put(i);
  }
}

class Consumer implements Runnable {
  Q q;

  Consumer(Q q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }

  public void run() {
    for(int i=0; i < 20; i++) q.get();
  }
}

class ProdCon {
  public static void main(String args[]) {
    Q q = new Q();
    new Consumer(q);
    new Producer(q);
  }
}
```

A portion of the output is shown here:

```
Put: 0
Got: 0
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
.
.
.
```



As you can see, the calls to **put( )** and **get( )** are synchronized. That is, each call to **put( )** is followed by a call to **get( )** and no values are missed. Without the semaphores, multiple calls to **put( )** would have occurred without matching calls to **get( )**, resulting in values being missed. (To prove this, remove the semaphore code and observe the results.)

The sequencing of **put( )** and **get( )** calls is handled by two semaphores: **semProd** and **semCon**. Before **put( )** can produce a value, it must acquire a permit from **semProd**. After it has set the value, it releases **semCon**. Before **get( )** can consume a value, it must acquire a permit from **semCon**. After it consumes the value, it releases **semProd**. This "give and take" mechanism ensures that each call to **put( )** must be followed by a call to **get( )**.

Notice that **semCon** is initialized with no available permits. This ensures that **put( )** executes first. The ability to set the initial synchronization state is one of the more powerful aspects of a semaphore.

## CountDownLatch

Sometimes you will want a thread to wait until one or more events have occurred. To handle such a situation, the concurrent API supplies **CountDownLatch**. A **CountDownLatch** is initially created with a count of the number of events that must occur before the latch is released. Each time an event happens, the count is decremented. When the count reaches zero, the latch opens.

**CountDownLatch** has the following constructor:

CountDownLatch(int *num*)

Here, *num* specifies the number of events that must occur in order for the latch to open.

To wait on the latch, a thread calls **await( )**, which has the forms shown here:

void await( ) throws InterruptedException
boolean await(long *wait*, TimeUnit *tu*) throws InterruptedException

The first form waits until the count associated with the invoking **CountDownLatch** reaches zero. The second form waits only for the period of time specified by *wait*. The units represented by *wait* are specified by *tu*, which is an object the **TimeUnit** enumeration.

(**TimeUnit** is described later in this chapter.) It returns **false** if the time limit is reached and **true** if the countdown reaches zero

To signal an event, call the **countDown( )** method, shown next:

```
void countDown( )
```

Each call to **countDown( )** decrements the count associated with the invoking object.

The following program demonstrates **CountDownLatch**. It creates a latch that requires five events to occur before it opens.

```
// An example of CountDownLatch.

import java.util.concurrent.CountDownLatch;

class CDLDemo {
  public static void main(String args[]) {
    CountDownLatch cdl = new CountDownLatch(5);

    System.out.println("Starting");

    new MyThread(cdl);

    try {
      cdl.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    System.out.println("Done");
  }
}

class MyThread implements Runnable {
  CountDownLatch latch;

  MyThread(CountDownLatch c) {
    latch = c;
    new Thread(this).start();
  }

  public void run() {
    for(int i = 0; i<5; i++) {
      System.out.println(i);
      latch.countDown(); // decrement count
    }
  }
}
```

The output produced by the program is shown here:

```
Starting
0
1
```

```
2
3
4
Done
```

Inside **main( )**, a **CountDownLatch** called **cdl** is created with an initial count of five. Next, an instance of **MyThread** is created, which begins execution of a new thread. Notice that **cdl** is passed as a parameter to **MyThread**'s constructor and stored in the **latch** instance variable. Then, the main thread calls **await( )** on **cdl**, which causes execution of the main thread to pause until **cdl**'s count has been decremented five times.

Inside the **run( )** method of **MyThread**, a loop is created that iterates five times. With each iteration, the **countDown( )** method is called on **latch**, which refers to **cdl** in **main( )**. After the fifth iteration, the latch opens, which allows the main thread to resume.

**CountDownLatch** is a powerful yet easy-to-use synchronization object that is appropriate whenever a thread must wait for one or more events to occur.



## CyclicBarrier

A situation not uncommon in concurrent programming occurs when a set of two or more threads must wait at a predetermined execution point until all threads in the set have reached that point. To handle such a situation, the concurrent API supplies the **CyclicBarrier** class. It enables you to define a synchronization object that suspends until the specified number of threads has reached the barrier point.

**CyclicBarrier** has the following two constructors:

CyclicBarrier(int *numThreads*)
CyclicBarrier(int *numThreads*, Runnable *action*)

Here, *numThreads* specifies the number of threads that must reach the barrier before execution continues. In the second form, *action* specifies a thread that will be executed when the barrier is reached.

Here is the general procedure that you will follow to use **CyclicBarrier**. First, create a **CyclicBarrier** object, specifying the number of threads that you will be waiting for. Next, when each thread reaches the barrier, have it call **await( )** on that object. This will pause execution of the thread until all of the other threads also call **await( )**. Once the specified number of threads has reached the barrier, **await( )** will return and execution will resume. Also, if you have specified an action, then that thread is executed.

The **await( )** method has the following two forms:

int await( ) throws InterruptedException, BrokenBarrierException

int await(long *wait*, TimeUnit *tu*)
  throws InterruptedException, BrokenBarrierException, TimeoutException

The first form waits until all the threads have reached the barrier point. The second form waits only for the period of time specified by *wait*. The units represented by *wait* are specified by *tu*. Both forms return a value that indicates the order that the threads arrive at the barrier point. The first thread returns a value equal to the number of threads waited upon minus one. The last thread returns zero.

Here is an example that illustrates **CyclicBarrier**. It waits until a set of three threads has reached the barrier. When that occurs, the thread specified by **BarAction** executes.

```
// An example of CyclicBarrier.

import java.util.concurrent.*;

class BarDemo {
  public static void main(String args[]) {
    CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

    System.out.println("Starting");

    new MyThread(cb, "A");
    new MyThread(cb, "B");
    new MyThread(cb, "C");

  }
}

// A thread of execution that uses a CyclicBarrier.

class MyThread implements Runnable {
  CyclicBarrier cbar;
  String name;

  MyThread(CyclicBarrier c, String n) {
    cbar = c;
    name = n;
    new Thread(this).start();
  }

  public void run() {

    System.out.println(name);

    try {
      cbar.await();
    } catch (BrokenBarrierException exc) {
      System.out.println(exc);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
  }
}

// An object of this class is called when the
// CyclicBarrier ends.
class BarAction implements Runnable {
  public void run() {
    System.out.println("Barrier Reached!");
  }
}
```

The output is shown here. (The precise order in which the threads execute may vary.)

```
Starting
A
B
C
Barrier Reached!
```

A **CyclicBarrier** can be reused because it will release waiting threads each time the specified number of threads calls **await( )**. For example, if you change **main( )** in the preceding program so that it looks like this:

```
public static void main(String args[]) {
CyclicBarrier cb = new CyclicBarrier(3, new BarAction() );

  System.out.println("Starting");

  new MyThread(cb, "A");
  new MyThread(cb, "B");
  new MyThread(cb, "C");
  new MyThread(cb, "X");
  new MyThread(cb, "Y");
  new MyThread(cb, "Z");

}
```

the following output will be produced. (The precise order in which the threads execute may vary.)

```
Starting
A
B
C
Barrier Reached!
X
Y
Z
Barrier Reached!
```

As the preceding example shows, the **CyclicBarrier** offers a streamlined solution to what was previously a complicated problem.

## Exchanger

Perhaps the most interesting of the synchronization classes is **Exchanger**. It is designed to simplify the exchange of data between two threads. The operation of an **Exchanger** is astoundingly simple: it simply waits until two separate threads call its **exchange( )** method. When that occurs, it exchanges the data supplied by the threads. This mechanism is both elegant and easy to use. Uses for **Exchanger** are easy to imagine. For example, one thread might prepare a buffer for receiving information over a network connection. Another thread might fill that buffer with the information from the connection. The two threads work together so that each time a new buffer is needed, an exchange is made.

**Exchanger** is a generic class that is declared as shown here:

Exchanger<V>

Here, **V** specifies the type of the data being exchanged.

The only method defined by **Exchanger** is **exchange( )**, which has the two forms shown here:

V exchange(V *objRef*) throws InterruptedException

V exchange(V *objRef*, long *wait*, TimeUnit *tu*)
  throws InterruptedException, TimeoutException

Here, *objRef* is a reference to the data to exchange. The data received from the other thread is returned. The second form of **exchange( )** allows a time-out period to be specified. The key point about **exchange( )** is that it won't succeed until it has been called on the same **Exchanger** object by two separate threads. Thus, **exchange( )** synchronizes the exchange of the data.

Here is an example that demonstrates **Exchanger**. It creates two threads. One thread creates an empty buffer that will receive the data put into it by the second thread. In this case, the data is a string. Thus, the first thread exchanges an empty string for a full one.

```
// An example of Exchanger.

import java.util.concurrent.Exchanger;

class ExgrDemo {
  public static void main(String args[]) {
    Exchanger<String> exgr = new Exchanger<String>();

    new UseString(exgr);
    new MakeString(exgr);
  }
}

// A Thread that constructs a string.
class MakeString implements Runnable {
  Exchanger<String> ex;
  String str;

  MakeString(Exchanger<String> c) {
    ex = c;
    str = new String();

    new Thread(this).start();
  }

  public void run() {
    char ch = 'A';

    for(int i = 0; i < 3; i++) {

      // Fill Buffer
      for(int j = 0; j < 5; j++)
```

```
        str += ch++;

      try {
        // Exchange a full buffer for an empty one.
        str = ex.exchange(str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}

// A Thread that uses a string.
class UseString implements Runnable {
  Exchanger<String> ex;
  String str;
  UseString(Exchanger<String> c) {
    ex = c;
    new Thread(this).start();
  }

  public void run() {

    for(int i=0; i < 3; i++) {
      try {
        // Exchange an empty buffer for a full one.
        str = ex.exchange(new String());
        System.out.println("Got: " + str);
      } catch(InterruptedException exc) {
        System.out.println(exc);
      }
    }
  }
}
```

Here is the output produced by the program:

```
Got: ABCDE
Got: FGHIJ
Got: KLMNO
```

In the program, the **main( )** method creates an **Exchanger** for strings. This object is then used to synchronize the exchange of strings between the **MakeString** and **UseString** classes. The **MakeString** class fills a string with data. The **UseString** exchanges an empty string for a full one. It then displays the contents of the newly constructed string. The exchange of empty and full buffers is synchronized by the **exchange( )** method, which is called by both classes' **run( )** method.

## Phaser

Another synchronization class is called **Phaser**. Its primary purpose is to enable the synchronization of threads that represent one or more phases of activity. For example, you might have a set of threads that implement three phases of an order-processing application. In the first phase, separate threads are used to validate customer information, check inventory, and confirm pricing. When that phase is complete, the second phase has two threads that compute shipping costs and all applicable tax. After that, a final phase confirms payment and determines estimated shipping time. In the past, to synchronize the multiple threads that comprise this scenario would require a bit of work on your part. With the inclusion of **Phaser**, the process is now much easier.

To begin, it helps to know that a **Phaser** works a bit like a **CyclicBarrier**, described earlier, except that it supports multiple phases. As a result, **Phaser** lets you define a synchronization object that waits until a specific phase has completed. It then advances to the next phase, again waiting until that phase concludes. It is important to understand that **Phaser** can also be used to synchronize only a single phase. In this regard, it acts much like a **CyclicBarrier**. However, its primary use is to synchronize multiple phases.

**Phaser** defines four constructors. Here are the two used in this section:

```
Phaser( )
Phaser(int numParties)
```

The first creates a phaser that has a registration count of zero. The second sets the registration count to *numParties*. The term *party* is often applied to the objects that register with a phaser. Although often there is a one-to-correspondence between the number of registrants and the number of threads being synchronized, this is not required. In both cases, the current phase is zero. That is, when a **Phaser** is created, it is initially at phase zero.

In general, here is how you use **Phaser**. First, create a new instance of **Phaser**. Next, register one or more parties with the phaser, either by calling **register( )** or by specifying the number of parties in the constructor. For each registered party, have the phaser wait until all registered parties complete a phase. A party signals this by calling one of a variety of methods supplied by **Phaser**, such as **arrive( )** or **arriveAndAwaitAdvance( )**. After all parties have arrived, the phase is complete, and the phaser can move on to the next phase (if there is one), or terminate. The following sections explain the process in detail.

To register parties after a **Phaser** has been constructed, call **register( )**. It is shown here:

```
int register()
```

It returns the phase number of the phase to which it is registered.

To signal that a party has completed a phase, it must call **arrive( )** or some variation of **arrive( )**. When the number of arrivals equals the number of registered parties, the phase is completed and the **Phaser** moves on to the next phase (if there is one). The **arrive( )** method has this general form:

```
int arrive( )
```

This method signals that a party (normally a thread of execution) has completed some task (or portion of a task). It returns the current phase number. If the phaser has been terminated, then it returns a negative value. The **arrive( )** method does not suspend

execution of the calling thread. This means that it does not wait for the phase to be completed. This method should be called only by a registered party.

If you want to indicate the completion of a phase and then wait until all other registrants have also completed that phase, use **arriveAndAwaitAdvance( )**. It is shown here:

int arriveAndAwaitAdvance( )

It waits until all parties have arrived. It returns the next phase number or a negative value if the phaser has been terminated. This method should be called only by a registered party.

A thread can arrive and then deregister itself by calling **arriveAndDeregister( )**. It is shown here:

int arriveAndDeregister( )

It returns the current phase number or a negative value if the phaser has been terminated. It does not wait until the phase is complete. This method should be called only by a registered party.

To obtain the current phase number, call **getPhase( )**, which is shown here:

final int getPhase( )

When a **Phaser** is created, the first phase will be 0, the second phase 1, the third phase 2, and so on. A negative value is returned if the invoking **Phaser** has been terminated.

Here is an example that shows **Phaser** in action. It creates three threads, each of which have three phases. It uses a **Phaser** to synchronize each phase.

```
// An example of Phaser.

import java.util.concurrent.*;

class PhaserDemo {
  public static void main(String args[]) {
    Phaser phsr = new Phaser(1);
    int curPhase;

    System.out.println("Starting");

    new MyThread(phsr, "A");
    new MyThread(phsr, "B");
    new MyThread(phsr, "C");

    // Wait for all threads to complete phase one.
    curPhase = phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Phase " + curPhase + " Complete");

    // Wait for all threads to complete phase two.
    curPhase =  phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Phase " + curPhase + " Complete");

    curPhase =  phsr.getPhase();
    phsr.arriveAndAwaitAdvance();
    System.out.println("Phase " + curPhase + " Complete");
```

```
    // Deregister the main thread.
    phsr.arriveAndDeregister();

    if(phsr.isTerminated())
      System.out.println("The Phaser is terminated");
  }
}

// A thread of execution that uses a Phaser.
class MyThread implements Runnable {
  Phaser phsr;
  String name;

  MyThread(Phaser p, String n) {
    phsr = p;
    name = n;
    phsr.register();
    new Thread(this).start();
  }

  public void run() {

    System.out.println("Thread " + name + " Beginning Phase One");
    phsr.arriveAndAwaitAdvance(); // Signal arrival.

    // Pause a bit to prevent jumbled output. This is for illustration
    // only. It is not required for the proper operation of the phaser.
    try {
      Thread.sleep(10);
    } catch(InterruptedException e) {
      System.out.println(e);
    }

    System.out.println("Thread " + name + " Beginning Phase Two");
    phsr.arriveAndAwaitAdvance(); // Signal arrival.

    // Pause a bit to prevent jumbled output. This is for illustration
    // only. It is not required for the proper operation of the phaser.
    try {
      Thread.sleep(10);
    } catch(InterruptedException e) {
      System.out.println(e);
    }

    System.out.println("Thread " + name + " Beginning Phase Three");
    phsr.arriveAndDeregister(); // Signal arrival and deregister.
  }
}
```

The output is shown here:

```
Starting
Thread A Beginning Phase One
Thread C Beginning Phase One
```

```
Thread B Beginning Phase One
Phase 0 Complete
Thread B Beginning Phase Two
Thread C Beginning Phase Two
Thread A Beginning Phase Two
Phase 1 Complete
Thread C Beginning Phase Three
Thread B Beginning Phase Three
Thread A Beginning Phase Three
Phase 2 Complete
The Phaser is terminated
```



Let's look closely at the key sections of the program. First, in **main( )**, a **Phaser** called **phsr** is created with an initial party count of 1 (which corresponds to the main thread). Then three threads are started by creating three **MyThread** objects. Notice that **MyThread** is passed a reference to **phsr** (the phaser). The **MyThread** objects use this phaser to synchronize their activities. Next, **main( )** calls **getPhase( )** to obtain the current phase number (which is initially zero) and then calls **arriveAndAwaitAdvance( )**. This causes **main( )** to suspend until phase zero has completed. This won't happen until all **MyThread**s also call **arriveAndAwaitAdvance( )**. When this occurs, **main( )** will resume execution, at which point it displays that phase zero has completed, and it moves on to the next phase. This process repeats until all three phases have finished. Then, **main( )** calls **arriveAndDeregister( )**. At that point, all three **MyThread**s have also deregistered. Since this results in there being no registered parties when the phaser advances to the next phase, the phaser is terminated.

Now look at **MyThread**. First, notice that the constructor is passed a reference to the phaser that it will use and then registers with the new thread as a party on that phaser. Thus, each new **MyThread** becomes a party registered with the passed-in phaser. Also notice that each thread has three phases. In this example, each phase consists of a placeholder that simply displays the name of the thread and what it is doing. Obviously, in real-world code, the thread would be performing more meaningful actions. Between the first two phases, the thread calls **arriveAndAwaitAdvance( )**. Thus, each thread waits until all threads have completed the phase (and the main thread is ready). After all threads have arrived (including the main thread), the phaser moves on to the next phase. After the third phase, each thread deregisters itself with a call to **arriveAndDeregister( )**. As the comments in **MyThread** explain, the calls to **sleep( )** are used for the purposes of illustration to ensure that the output is not jumbled because of the multithreading. They are not needed to make the phaser work properly. If you remove them, the output may look a bit jumbled, but the phases will still be synchronized correctly.

One other point: Although the preceding example used three threads that were all of the same type, this is not a requirement. Each party that uses a phaser can be unique, with each performing some separate task.

It is possible to take control of precisely what happens when a phase advance occurs. To do this, you must override the **onAdvance( )** method. This method is called by the run time when a **Phaser** advances from one phase to the next. It is shown here:

protected boolean onAdvance(int *phase*, int *numParties*)

Here, *phase* will contain the current phase number prior to being incremented and *numParties* will contain the number of registered parties. To terminate the phaser, **onAdvance( )** must return **true**. To keep the phaser alive, **onAdvance( )** must return **false**.

The default version of **onAdvance( )** returns **true** (thus terminating the phaser) when there are no registered parties. As a general rule, your override should also follow this practice.

One reason to override **onAdvance( )** is to enable a phaser to execute a specific number of phases and then stop. The following example gives you the flavor of this usage. It creates a class called **MyPhaser** that extends **Phaser** so that it will run a specified number of phases. It does this by overriding the **onAdvance( )** method. The **MyPhaser** constructor accepts one argument, which specifies the number of phases to execute. Notice that **MyPhaser** automatically registers one party. This behavior is useful in this example, but the needs of your own applications may differ.

```
// Extend Phaser and override onAdvance() so that only a specific
// number of phases are executed.

import java.util.concurrent.*;

// Extend MyPhaser to allow only a specific number of phases
// to be executed.
class MyPhaser extends Phaser {
  int numPhases;

  MyPhaser(int parties, int phaseCount) {
    super(parties);
    numPhases = phaseCount - 1;
  }

  // Override onAdvance() to execute the specified
  // number of phases.
  protected boolean onAdvance(int p, int regParties) {
    // This println() statement is for illustration only.
    // Normally, onAdvance() will not display output.
    System.out.println("Phase " + p + " completed.\n");

    // If all phases have completed, return true
    if(p == numPhases || regParties == 0) return true;

    // Otherwise, return false.
    return false;
  }
}

class PhaserDemo2 {
  public static void main(String args[]) {

    MyPhaser phsr = new MyPhaser(1, 4);

    System.out.println("Starting\n");

    new MyThread(phsr, "A");
    new MyThread(phsr, "B");
    new MyThread(phsr, "C");

    // Wait for the specified number of phases to complete.
    while(!phsr.isTerminated()) {
```

```
      phsr.arriveAndAwaitAdvance();
    }

    System.out.println("The Phaser is terminated");
  }
}

// A thread of execution that uses a Phaser.
class MyThread implements Runnable {
  Phaser phsr;
  String name;

  MyThread(Phaser p, String n) {
    phsr = p;
    name = n;
    phsr.register();
    new Thread(this).start();
  }

  public void run() {

    while(!phsr.isTerminated()) {
      System.out.println("Thread " + name + " Beginning Phase " +
                         phsr.getPhase());

      phsr.arriveAndAwaitAdvance();

      // Pause a bit to prevent jumbled output. This is for illustration
      // only. It is not required for the proper operation of the phaser.
      try {
        Thread.sleep(10);
      } catch(InterruptedException e) {
        System.out.println(e);
      }
    }
  }
}
```

The output from the program is shown here:

```
Starting

Thread B Beginning Phase 0
Thread A Beginning Phase 0
Thread C Beginning Phase 0
Phase 0 completed.

Thread A Beginning Phase 1
Thread B Beginning Phase 1
Thread C Beginning Phase 1
Phase 1 completed.

Thread C Beginning Phase 2
Thread B Beginning Phase 2
Thread A Beginning Phase 2
Phase 2 completed.
```

```
Thread C Beginning Phase 3
Thread B Beginning Phase 3
Thread A Beginning Phase 3
Phase 3 completed.

The Phaser is terminated
```

Inside **main( )**, one instance of **Phaser** is created. It is passed 4 as an argument, which means that it will execute four phases and then stop. Next, three threads are created and then the following loop is entered:

```
// Wait for the specified number of phases to complete.
while(!phsr.isTerminated()) {
  phsr.arriveAndAwaitAdvance();
}
```

This loop simply calls **arriveAndAwaitAdvance( )** until the phaser is terminated. The phaser won't terminate until the specified number of phases have been executed. In this case, the loop continues to execute until four phases have run. Next, notice that the threads also call **arriveAndAwaitAdvance( )** within a loop that runs until the phaser is terminated. This means that they will execute until the specified number of phases has been completed.

Now, look closely at the code for **onAdvance( )**. Each time **onAdvance( )** is called, it is passed the current phase and the number of registered parties. If the current phase equals the specified phase, or if the number of registered parties is zero, **onAdvance( )** returns **true**, thus stopping the phaser. This is accomplished with this line of code:

```
// If all phases have completed, return true
if(p == numPhases || regParties == 0) return true;
```

As you can see, very little code is needed to accommodate the desired outcome.

Before moving on, it is useful to point out that you don't necessarily need to explicitly extend **Phaser** as the previous example does to simply override **onAdvance( )**. In some cases, more compact code can be created by using an anonymous inner class to override **onAdvance( )**.

**Phaser** has additional capabilities that may be of use in your applications. You can wait for a specific phase by calling **awaitAdvance( )**, which is shown here:

int awaitAdvance(int *phase*)

Here, *phase* indicates the phase number on which **awaitAdvance( )** will wait until a transition to the next phase takes place. It will return immediately if the argument passed to *phase* is not equal to the current phase. It will also return immediately if the phaser is terminated. However, if *phase* is passed the current phase, then it will wait until the phase increments. This method should be called only by a registered party. There is also an interruptible version of this method called **awaitAdvanceInterruptibly( )**.

To register more than one party, call **bulkRegister( )**. To obtain the number of registered parties, call **getRegisteredParties( )**. You can also obtain the number of arrived parties and unarrived parties by calling **getArrivedParties( )** and **getUnarrivedParties( )**, respectively. To force the phaser to enter a terminated state, call **forceTermination( )**.

**Phaser** also lets you create a tree of phasers. This is supported by two additional constructors, which let you specify the parent, and the **getParent( )** method.

## Using an Executor

The concurrent API supplies a feature called an *executor* that initiates and controls the execution of threads. As such, an executor offers an alternative to managing threads through the **Thread** class.

At the core of an executor is the **Executor** interface. It defines the following method:

```
void execute(Runnable thread)
```

The thread specified by *thread* is executed. Thus, **execute( )** starts the specified thread.

The **ExecutorService** interface extends **Executor** by adding methods that help manage and control the execution of threads. For example, **ExecutorService** defines **shutdown( )**, shown here, which stops the invoking **ExecutorService**.

```
void shutdown( )
```

**ExecutorService** also defines methods that execute threads that return results, that execute a set of threads, and that determine the shutdown status. We will look at several of these methods a little later.

Also defined is the interface **ScheduledExecutorService**, which extends **ExecutorService** to support the scheduling of threads.

The concurrent API defines three predefined executor classes: **ThreadPoolExecutor** and **ScheduledThreadPoolExecutor**, and **ForkJoinPool**. **ThreadPoolExecutor** implements the **Executor** and **ExecutorService** interfaces and provides support for a managed pool of threads. **ScheduledThreadPoolExecutor** also implements the **ScheduledExecutorService** interface to allow a pool of threads to be scheduled. **ForkJoinPool** implements the **Executor** and **ExecutorService** interfaces and is used by the Fork/Join Framework. It is described later in this chapter.

A thread pool provides a set of threads that is used to execute various tasks. Instead of each task using its own thread, the threads in the pool are used. This reduces the overhead associated with creating many separate threads. Although you can use **ThreadPoolExecutor** and **ScheduledThreadPoolExecutor** directly, most often you will want to obtain an executor by calling one of the following static factory methods defined by the **Executors** utility class. Here are some examples:

```
static ExecutorService newCachedThreadPool( )
static ExecutorService newFixedThreadPool(int numThreads)
static ScheduledExecutorService newScheduledThreadPool(int numThreads)
```

**newCachedThreadPool( )** creates a thread pool that adds threads as needed but reuses threads if possible. **newFixedThreadPool( )** creates a thread pool that consists of a specified number of threads. **newScheduledThreadPool( )** creates a thread pool that supports thread scheduling. Each returns a reference to an **ExecutorService** that can be used to manage the pool.

### A Simple Executor Example

Before going any further, a simple example that uses an executor will be of value. The following program creates a fixed thread pool that contains two threads. It then uses that

pool to execute four tasks. Thus, four tasks share the two threads that are in the pool. After the tasks finish, the pool is shut down and the program ends.

```
// A simple example that uses an Executor.

import java.util.concurrent.*;

class SimpExec {
  public static void main(String args[]) {
    CountDownLatch cdl = new CountDownLatch(5);
    CountDownLatch cdl2 = new CountDownLatch(5);
    CountDownLatch cdl3 = new CountDownLatch(5);
    CountDownLatch cdl4 = new CountDownLatch(5);
    ExecutorService es = Executors.newFixedThreadPool(2);

    System.out.println("Starting");

    // Start the threads.
    es.execute(new MyThread(cdl, "A"));
    es.execute(new MyThread(cdl2, "B"));
    es.execute(new MyThread(cdl3, "C"));
    es.execute(new MyThread(cdl4, "D"));

    try {
      cdl.await();
      cdl2.await();
      cdl3.await();
      cdl4.await();
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Done");
  }
}

class MyThread implements Runnable {
  String name;
  CountDownLatch latch;

  MyThread(CountDownLatch c, String n) {
    latch = c;
    name = n;

    new Thread(this);
  }

  public void run() {

    for(int i = 0; i < 5; i++) {
      System.out.println(name + ": " + i);
      latch.countDown();
    }
  }
}
```

The output from the program is shown here. (The precise order in which the threads execute may vary.)

```
Starting
A: 0
A: 1
A: 2
A: 3
A: 4
C: 0
C: 1
C: 2
C: 3
C: 4
D: 0
D: 1
D: 2
D: 3
D: 4
B: 0
B: 1
B: 2
B: 3
B: 4
Done
```

As the output shows, even though the thread pool contains only two threads, all four tasks are still executed. However, only two can run at the same time. The others must wait until one of the pooled threads is available for use.

The call to **shutdown( )** is important. If it were not present in the program, then the program would not terminate because the executor would remain active. To try this for yourself, simply comment out the call to **shutdown( )** and observe the result.

## Using Callable and Future

One of the most interesting features of the concurrent API is the **Callable** interface. This interface represents a thread that returns a value. An application can use **Callable** objects to compute results that are then returned to the invoking thread. This is a powerful mechanism because it facilitates the coding of many types of numerical computations in which partial results are computed simultaneously. It can also be used to run a thread that returns a status code that indicates the successful completion of the thread.

**Callable** is a generic interface that is defined like this:

```
interface Callable<V>
```

Here, **V** indicates the type of data returned by the task. **Callable** defines only one method, **call( )**, which is shown here:

```
V call( ) throws Exception
```

Inside **call( )**, you define the task that you want performed. After that task completes, you return the result. If the result cannot be computed, **call( )** must throw an exception.

A **Callable** task is executed by an **ExecutorService**, by calling its **submit( )** method. There are three forms of **submit( )**, but only one is used to execute a **Callable**. It is shown here:

<T> Future<T> submit(Callable<T> *task*)

Here, *task* is the **Callable** object that will be executed in its own thread. The result is returned through an object of type **Future**.

**Future** is a generic interface that represents the value that will be returned by a **Callable** object. Because this value is obtained at some future time, the name **Future** is appropriate. **Future** is defined like this:

interface Future<V>

Here, **V** specifies the type of the result.

To obtain the returned value, you will call **Future**'s **get( )** method, which has these two forms:

V get( )
  throws InterruptedException, ExecutionException

V get(long *wait*, TimeUnit *tu*)
  throws InterruptedException, ExecutionException, TimeoutException

The first form waits for the result indefinitely. The second form allows you to specify a timeout period in *wait*. The units of *wait* are passed in *tu*, which is an object of the **TimeUnit** enumeration, described later in this chapter.

The following program illustrates **Callable** and **Future** by creating three tasks that perform three different computations. The first returns the summation of a value, the second computes the length of the hypotenuse of a right triangle given the length of its sides, and the third computes the factorial of a value. All three computations occur simultaneously.

```
// An example that uses a Callable.

import java.util.concurrent.*;

class CallableDemo {
  public static void main(String args[]) {
    ExecutorService es = Executors.newFixedThreadPool(3);
    Future<Integer> f;
    Future<Double> f2;
    Future<Integer> f3;

    System.out.println("Starting");

    f = es.submit(new Sum(10));
    f2 = es.submit(new Hypot(3, 4));
    f3 = es.submit(new Factorial(5));

    try {
      System.out.println(f.get());
      System.out.println(f2.get());
      System.out.println(f3.get());
```

```
    } catch (InterruptedException exc) {
      System.out.println(exc);
    }
    catch (ExecutionException exc) {
      System.out.println(exc);
    }

    es.shutdown();
    System.out.println("Done");
  }
}

// Following are three computational threads.

class Sum implements Callable<Integer> {
  int stop;

  Sum(int v) { stop = v; }

  public Integer call() {
    int sum = 0;
    for(int i = 1; i <= stop; i++) {
      sum += i;
    }
    return sum;
  }
}

class Hypot implements Callable<Double> {
  double side1, side2;

  Hypot(double s1, double s2) {
    side1 = s1;
    side2 = s2;
  }

  public Double call() {
    return Math.sqrt((side1*side1) + (side2*side2));
  }
}

class Factorial implements Callable<Integer> {
  int stop;

  Factorial(int v) { stop = v; }

  public Integer call() {
    int fact = 1;
    for(int i = 2; i <= stop; i++) {
      fact *= i;
    }
    return fact;
  }
}
```

The output is shown here:

```
Starting
55
5.0
120
Done
```

## The TimeUnit Enumeration

The concurrent API defines several methods that take an argument of type **TimeUnit**, which indicates a time-out period. **TimeUnit** is an enumeration that is used to specify the *granularity* (or resolution) of the timing. **TimeUnit** is defined within **java.util.concurrent**. It can be one of the following values:

DAYS
HOURS
MINUTES
SECONDS
MICROSECONDS
MILLISECONDS
NANOSECONDS

Although **TimeUnit** lets you specify any of these values in calls to methods that take a timing argument, there is no guarantee that the system is capable of the specified resolution.

Here is an example that uses **TimeUnit**. The **CallableDemo** class, shown in the previous section, is modified as shown next to use the second form of **get( )** that takes a **TimeUnit** argument.

```
try {
  System.out.println(f.get(10, TimeUnit.MILLISECONDS));
  System.out.println(f2.get(10, TimeUnit.MILLISECONDS));
  System.out.println(f3.get(10, TimeUnit.MILLISECONDS));
} catch (InterruptedException exc) {
  System.out.println(exc);
}
catch (ExecutionException exc) {
  System.out.println(exc);
} catch (TimeoutException exc) {
  System.out.println(exc);
}
```

In this version, no call to **get( )** will wait more than 10 milliseconds.

The **TimeUnit** enumeration defines various methods that convert between units. These are shown here:

long convert(long *tval*, TimeUnit *tu*)
long toMicros(long *tval*)
long toMillis(long *tval*)
long toNanos(long *tval*)

long toSeconds(long *tval*)
long toDays(long *tval*)
long toHours(long *tval*)
long toMinutes(long *tval*)

The **convert( )** method converts *tval* into the specified unit and returns the result. The **to** methods perform the indicated conversion and return the result.

**TimeUnit** also defines the following timing methods:

void sleep(long *delay*) throws InterruptedExecution
void timedJoin(Thread *thrd*, long *delay*) throws InterruptedExecution
void timedWait(Object *obj*, long *delay*) throws InterruptedExecution

Here, **sleep( )** pauses execution for the specified delay period, which is specified in terms of the invoking enumeration constant. It translates into a call to **Thread.sleep( )**. The **timedJoin( )** method is a specialized version of **Thread.join( )** in which *thrd* pauses for the time period specified by *delay,* which is described in terms of the invoking time unit. The **timedWait( )** method is a specialized version of **Object.wait( )** in which *obj* is waited on for the period of time specified by *delay,* which is described in terms of the invoking time unit.



## The Concurrent Collections

As explained, the concurrent API defines several collection classes that have been engineered for concurrent operation. They include:

ArrayBlockingQueue
ConcurrentHashMap
ConcurrentLinkedDeque
ConcurrentLinkedQueue
ConcurrentSkipListMap
ConcurrentSkipListSet
CopyOnWriteArrayList
CopyOnWriteArraySet
DelayQueue
LinkedBlockingDeque
LinkedBlockingQueue
LinkedTransferQueue
PriorityBlockingQueue
SynchronousQueue

These offer concurrent alternatives to their related classes defined by the Collections Framework. These collections work much like the other collections except that they provide concurrency support. Programmers familiar with the Collections Framework will have no trouble using these concurrent collections.

## Locks

The **java.util.concurrent.locks** package provides support for *locks,* which are objects that offer an alternative to using **synchronized** to control access to a shared resource. In general, here is how a lock works. Before accessing a shared resource, the lock that protects that

resource is acquired. When access to the resource is complete, the lock is released. If a second thread attempts to acquire the lock when it is in use by another thread, the second thread will suspend until the lock is released. In this way, conflicting access to a shared resource is prevented.

Locks are particularly useful when multiple threads need to access the value of shared data. For example, an inventory application might have a thread that first confirms that an item is in stock and then decreases the number of items on hand as each sale occurs. If two or more of these threads are running, then without some form of synchronization, it would be possible for one thread to be in the middle of a transaction when the second thread begins its transaction. The result could be that both threads would assume that adequate inventory exists, even if there is only sufficient inventory on hand to satisfy one sale. In this type of situation, a lock offers a convenient means of handling the needed synchronization.

The **Lock** interface defines a lock. The methods defined by **Lock** are shown in Table 28-1. In general, to acquire a lock, call **lock( )**. If the lock is unavailable, **lock( )** will wait. To release a lock, call **unlock( )**. To see if a lock is available, and to acquire it if it is, call **tryLock( )**. This method will not wait for the lock if it is unavailable. Instead, it returns **true** if the lock is acquired and **false** otherwise. The **newCondition( )** method returns a **Condition** object associated with the lock. Using a **Condition**, you gain detailed control of the lock through methods such as **await( )** and **signal( )**, which provide functionality similar to **Object.wait( )** and **Object.notify( )**.

**java.util.concurrent.locks** supplies an implementation of **Lock** called **ReentrantLock**. **ReentrantLock** implements a *reentrant lock,* which is a lock that can be repeatedly entered by the thread that currently holds the lock. Of course, in the case of a thread reentering a lock, all calls to **lock( )** must be offset by an equal number of calls to **unlock( )**. Otherwise, a thread seeking to acquire the lock will suspend until the lock is not in use.

| Method | Description |
|---|---|
| void lock( ) | Waits until the invoking lock can be acquired. |
| void lockInterruptibly( ) throws InterruptedException | Waits until the invoking lock can be acquired, unless interrupted. |
| Condition newCondition( ) | Returns a **Condition** object that is associated with the invoking lock. |
| boolean tryLock( ) | Attempts to acquire the lock. This method will not wait if the lock is unavailable. Instead, it returns **true** if the lock has been acquired and **false** if the lock is currently in use by another thread. |
| boolean tryLock(long *wait*, TimeUnit *tu*) throws InterruptedException | Attempts to acquire the lock. If the lock is unavailable, this method will wait no longer than the period specified by *wait*, which is in *tu* units. It returns **true** if the lock has been acquired and **false** if the lock cannot be acquired within the specified period. |
| void unlock( ) | Releases the lock. |

**Table 28-1** The **Lock** Methods

The following program demonstrates the use of a lock. It creates two threads that access a shared resource called **Shared.count**. Before a thread can access **Shared.count**, it must obtain a lock. After obtaining the lock, **Shared.count** is incremented and then, before releasing the lock, the thread sleeps. This causes the second thread to attempt to obtain the lock. However, because the lock is still held by the first thread, the second thread must wait until the first thread stops sleeping and releases the lock. The output shows that access to **Shared.count** is, indeed, synchronized by the lock.



```
// A simple lock example.

import java.util.concurrent.locks.*;

class LockDemo {

  public static void main(String args[]) {
    ReentrantLock lock = new ReentrantLock();

    new LockThread(lock, "A");
    new LockThread(lock, "B");
  }
}

// A shared resource.
class Shared {
  static int count = 0;
}

// A thread of execution that increments count.
class LockThread implements Runnable {
  String name;
  ReentrantLock lock;

  LockThread(ReentrantLock lk, String n) {
    lock = lk;
    name = n;
    new Thread(this).start();
  }

  public void run() {

    System.out.println("Starting " + name);

    try {
      // First, lock count.
      System.out.println(name + " is waiting to lock count.");
      lock.lock();
      System.out.println(name + " is locking count.");

      Shared.count++;
      System.out.println(name + ": " + Shared.count);

      // Now, allow a context switch -- if possible.
      System.out.println(name + " is sleeping.");
```

```
      Thread.sleep(1000);
    } catch (InterruptedException exc) {
      System.out.println(exc);
    } finally {
      // Unlock
      System.out.println(name + " is unlocking count.");
      lock.unlock();
    }
  }
}
```

The output is shown here. (The precise order in which the threads execute may vary.)

```
Starting A
A is waiting to lock count.
A is locking count.
A: 1
A is sleeping.
Starting B
B is waiting to lock count.
A is unlocking count.
B is locking count.
B: 2
B is sleeping.
B is unlocking count.
```

**java.util.concurrent.locks** also defines the **ReadWriteLock** interface. This interface specifies a lock that maintains separate locks for read and write access. This enables multiple locks to be granted for readers of a resource as long as the resource is not being written. **ReentrantReadWriteLock** provides an implementation of **ReadWriteLock**.

---

**NOTE** JDK 8 adds a specialized lock called **StampedLock**. It does not implement the **Lock** or **ReadWriteLock** interfaces. It does, however, provide a mechanism that enables aspects of it to be used like a **Lock** or **ReadWriteLock**.

## Atomic Operations

**java.util.concurrent.atomic** offers an alternative to the other synchronization features when reading or writing the value of some types of variables. This package offers methods that get, set, or compare the value of a variable in one uninterruptible (that is, atomic) operation. This means that no lock or other synchronization mechanism is required.

Atomic operations are accomplished through the use of classes, such as **AtomicInteger** and **AtomicLong**, and methods such as **get( )**, **set( )**, **compareAndSet( )**, **decrementAndGet( )**, and **getAndSet( )**, which perform the action indicated by their names.

Here is an example that demonstrates how access to a shared integer can be synchronized by the use of **AtomicInteger**:

```
// A simple example of Atomic.

import java.util.concurrent.atomic.*;
```

```
class AtomicDemo {

  public static void main(String args[]) {
    new AtomThread("A");
    new AtomThread("B");
    new AtomThread("C");
  }
}

class Shared {
  static AtomicInteger ai = new AtomicInteger(0);
}

// A thread of execution that increments count.
class AtomThread implements Runnable {
  String name;

  AtomThread(String n) {
    name = n;
    new Thread(this).start();
  }

public void run() {

  System.out.println("Starting " + name);

  for(int i=1; i <= 3; i++)
    System.out.println(name + " got: " +
           Shared.ai.getAndSet(i));
  }
}
```



In the program, a static **AtomicInteger** named **ai** is created by **Shared**. Then, three threads of type **AtomThread** are created. Inside **run( )**, **Shared.ai** is modified by calling **getAndSet( )**. This method returns the previous value and then sets the value to the one passed as an argument. The use of **AtomicInteger** prevents two threads from writing to **ai** at the same time.

In general, the atomic operations offer a convenient (and possibly more efficient) alternative to the other synchronization mechanisms when only a single variable is involved. Beginning with JDK 8, **java.util.concurrent.atomic** also provides four classes that support lock-free cumulative operations. These are **DoubleAccumulator**, **DoubleAdder**, **LongAccumulator**, and **LongAdder**. The accumulator classes support a series of user-specified operations. The adder classes maintain a cumulative sum.

## Parallel Programming via the Fork/Join Framework

In recent years, an important new trend has emerged in software development: *parallel programming*. Parallel programming is the name commonly given to the techniques that take advantage of computers that contain two or more processors (multicore). As most readers will know, multicore computers are becoming commonplace. The advantage that multi-processor environments offer is the ability to significantly increase program performance. As a result, there has been a growing need for a mechanism that gives Java

programmers a simple, yet effective way to make use of multiple processors in a clean, scalable manner. To answer this need, JDK 7 added several new classes and interfaces that support parallel programming. They are commonly referred to as the *Fork/Join Framework*. It is one of the more important additions that has recently been made to the Java class library. The Fork/Join Framework is defined in the **java.util.concurrent** package.

The Fork/Join Framework enhances multithreaded programming in two important ways. First, it simplifies the creation and use of multiple threads. Second, it automatically makes use of multiple processors. In other words, by using the Fork/Join Framework you enable your applications to automatically scale to make use of the number of available processors. These two features make the Fork/Join Framework the recommended approach to multithreading when parallel processing is desired.

Before continuing, it is important to point out the distinction between traditional multithreading and parallel programming. In the past, most computers had a single CPU and multithreading was primarily used to take advantage of idle time, such as when a program is waiting for user input. Using this approach, one thread can execute while another is waiting. In other words, on a single-CPU system, multithreading is used to allow two or more tasks to share the CPU. This type of multithreading is typically supported by an object of type **Thread** (as described in Chapter 11). Although this type of multithreading will always remain quite useful, it was not optimized for situations in which two or more CPUs are available (multicore computers).

When multiple CPUs are present, a second type of multithreading capability that supports true parallel execution is required. With two or more CPUs, it is possible to execute portions of a program simultaneously, with each part executing on its own CPU. This can be used to significantly speed up the execution of some types of operations, such as sorting, transforming, or searching a large array. In many cases, these types of operations can be broken down into smaller pieces (each acting on a portion of the array), and each piece can be run on its own CPU. As you can imagine, the gain in efficiency can be enormous. Simply put: Parallel programming will be part of nearly every programmer's future because it offers a way to dramatically improve program performance.

## The Main Fork/Join Classes

The Fork/Join Framework is packaged in **java.util.concurrent**. At the core of the Fork/Join Framework are the following four classes:

| | |
|---|---|
| ForkJoinTask<V> | An abstract class that defines a task |
| ForkJoinPool | Manages the execution of **ForkJoinTask**s |
| RecursiveAction | A subclass of **ForkJoinTask<V>** for tasks that do not return values |
| RecursiveTask<V> | A subclass of **ForkJoinTask<V>** for tasks that return values |

Here is how they relate. A **ForkJoinPool** manages the execution of **ForkJoinTask**s. **ForkJoinTask** is an abstract class that is extended by the abstract classes **RecursiveAction** and **RecursiveTask**. Typically, your code will extend these classes to create a task. Before looking at the process in detail, an overview of the key aspects of each class will be helpful.

**NOTE** The class **CountedCompleter** (added by JDK 8) also extends **ForkJoinTask**. However, a discussion of **CountedCompleter** is beyond the scope of this book.

## ForkJoinTask<V>

**ForkJoinTask<V>** is an abstract class that defines a task that can be managed by a **ForkJoinPool**. The type parameter **V** specifies the result type of the task. **ForkJoinTask** differs from **Thread** in that **ForkJoinTask** represents lightweight abstraction of a task, rather than a thread of execution. **ForkJoinTask**s are executed by threads managed by a thread pool of type **ForkJoinPool**. This mechanism allows a large number of tasks to be managed by a small number of actual threads. Thus, **ForkJoinTask**s are very efficient when compared to threads.

**ForkJoinTask** defines many methods. At the core are **fork( )** and **join( )**, shown here:

```
final ForkJoinTask<V> fork( )
final V join( )
```

The **fork( )** method submits the invoking task for asynchronous execution of the invoking task. This means that the thread that calls **fork( )** continues to run. The **fork( )** method returns **this** after the task is scheduled for execution. Prior to JDK 8, **fork( )** could be executed only from within the computational portion of another **ForkJoinTask**, which is running within a **ForkJoinPool**. (You will see how to create the computational portion of a task shortly.) However, with the advent of JDK 8, if **fork( )** is not called while executing within a **ForkJoinPool**, then a common pool is automatically used. The **join( )** method waits until the task on which it is called terminates. The result of the task is returned. Thus, through the use of **fork( )** and **join( )**, you can start one or more new tasks and then wait for them to finish.

Another important **ForkJoinTask** method is **invoke( )**. It combines the fork and join operations into a single call because it begins a task and then waits for it to end. It is shown here:

```
final V invoke( )
```

The result of the invoking task is returned.

You can invoke more than one task at a time by using **invokeAll( )**. Two of its forms are shown here:

```
static void invokeAll(ForkJoinTask<?> taskA, ForkJoinTask<?> taskB)
static void invokeAll(ForkJoinTask<?> ... taskList)
```

In the first case, *taskA* and *taskB* are executed. In the second case, all specified tasks are executed. In both cases, the calling thread waits until all of the specified tasks have terminated. Prior to JDK 8, the **invokeAll( )** method could be executed only from within the computational portion of another **ForkJoinTask**, which is running within a **ForkJoinPool**. JDK 8's inclusion of the common pool relaxed this requirement.

## RecursiveAction

A subclass of **ForkJoinTask** is **RecursiveAction**. This class encapsulates a task that does not return a result. Typically, your code will extend **RecursiveAction** to create a task that has a **void** return type. **RecursiveAction** specifies four methods, but only one is usually of interest: the abstract method called **compute( )**. When you extend **RecursiveAction** to create a concrete class, you will put the code that defines the task inside **compute( )**. The **compute( )** method represents the *computational* portion of the task.

The **compute( )** method is defined by **RecursiveAction** like this:

```
protected abstract void compute( )
```

Notice that **compute( )** is **protected** and **abstract**. This means that it must be implemented by a subclass (unless that subclass is also abstract).

In general, **RecursiveAction** is used to implement a recursive, divide-and-conquer strategy for tasks that don't return results. (See "The Divide-and-Conquer Strategy" later in this chapter.)

## RecursiveTask<V>

Another subclass of **ForkJoinTask** is **RecursiveTask<V>**. This class encapsulates a task that returns a result. The result type is specified by **V**. Typically, your code will extend **RecursiveTask<V>** to create a task that returns a value. Like **RecursiveAction**, it too specifies four methods, but often only the abstract **compute( )** method is used, which represents the computational portion of the task. When you extend **RecursiveTask<V>** to create a concrete class, put the code that represents the task inside **compute( )**. This code must also return the result of the task.

The **compute( )** method is defined by **RecursiveTask<V>** like this:

```
protected abstract V compute( )
```

Notice that **compute( )** is **protected** and **abstract**. This means that it must be implemented by a subclass. When implemented, it must return the result of the task.

In general, **RecursiveTask** is used to implement a recursive, divide-and-conquer strategy for tasks that return results. (See "The Divide-and-Conquer Strategy" later in this chapter.)

## ForkJoinPool

The execution of **ForkJoinTask**s takes place within a **ForkJoinPool**, which also manages the execution of the tasks. Therefore, in order to execute a **ForkJoinTask**, you must first have a **ForkJoinPool**. Beginning with JDK 8, there are two ways to acquire a **ForkJoinPool**. First, you can explicitly create one by using a **ForkJoinPool** constructor. Second, you can use what is referred to as the *common pool*. The common pool (which was added by JDK 8) is a static **ForkJoinPool** that is automatically available for your use. Each method is introduced here, beginning with manually constructing a pool.

**ForkJoinPool** defines several constructors. Here are two commonly used ones:

```
ForkJoinPool( )
ForkJoinPool(int pLevel)
```

The first creates a default pool that supports a level of parallelism equal to the number of processors available in the system. The second lets you specify the level of parallelism. Its value must be greater than zero and not more than the limits of the implementation. The level of parallelism determines the number of threads that can execute concurrently. As a result, the level of parallelism effectively determines the number of tasks that can be executed simultaneously. (Of course, the number of tasks that can execute simultaneously cannot exceed the number of processors.) It is important to understand that the level of parallelism *does not*, however, limit the number of tasks that can be managed by the pool. A **ForkJoinPool** can manage many more tasks than its level of parallelism. Also, the level of parallelism is only a target. It is not a guarantee.

After you have created an instance of **ForkJoinPool**, you can start a task in a number of different ways. The first task started is often thought of as the main task. Frequently, the main task begins subtasks that are also managed by the pool. One common way to begin a main task is to call **invoke( )** on the **ForkJoinPool**. It is shown here:

<T> T invoke(ForkJoinTask<T> *task*)

This method begins the task specified by *task*, and it returns the result of the task. This means that the calling code waits until **invoke( )** returns.

To start a task without waiting for its completion, you can use **execute( )**. Here is one of its forms:

void execute(ForkJoinTask<?> *task*)

In this case, *task* is started, but the calling code does not wait for its completion. Rather, the calling code continues execution asynchronously.

Beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because a common pool is available for your use. In general, if you are not using a pool that you explicitly created, then the common pool will automatically be used. Although it won't always be necessary, you can obtain a reference to the common pool by calling **commonPool( )**, which is defined by **ForkJoinPool**. It is shown here:

static ForkJoinPool commonPool( )

A reference to the common pool is returned. The common pool provides a default level of parallelism. It can be set by use of a system property. (See the API documentation for details.) Typically, the default common pool is a good choice for many applications. Of course, you can always construct your own pool.

There are two basic ways to start a task using the common pool. First, you can obtain a reference to the pool by calling **commonPool( )** and then use that reference to call **invoke( )** or **execute( )**, as just described. Second, you can call **ForkJoinTask** methods such as **fork( )** or **invoke( )** on the task from outside its computational portion. In this case, the common pool will automatically be used. In other words, **fork( )** and **invoke( )** will start a task using the common pool if the task is not already running within a **ForkJoinPool**.

**ForkJoinPool** manages the execution of its threads using an approach called *work-stealing*. Each worker thread maintains a queue of tasks. If one worker thread's queue is empty, it will take a task from another worker thread. This adds to overall efficiency and helps maintain a balanced load. (Because of demands on CPU time by other processes in the system, even two worker threads with identical tasks in their respective queues may not complete at the same time.)

One other point: **ForkJoinPool** uses daemon threads. A daemon thread is automatically terminated when all user threads have terminated. Thus, there is no need to explicitly shut down a **ForkJoinPool**. However, with the exception of the common pool, it is possible to do so by calling **shutdown( )**. The **shutdown( )** method has no effect on the common pool.

## The Divide-and-Conquer Strategy

As a general rule, users of the Fork/Join Framework will employ a *divide-and-conquer* strategy that is based on recursion. This is why the two subclasses of **ForkJoinTask** are called **RecursiveAction** and **RecursiveTask**. It is anticipated that you will extend one of these classes when creating your own fork/join task.

The divide-and-conquer strategy is based on recursively dividing a task into smaller subtasks until the size of a subtask is small enough to be handled sequentially. For example, a task that applies a transform to each element in an array of *N* integers can be broken down into two subtasks in which each transforms half the elements in the array. That is, one subtask transforms the elements 0 to *N*/2, and the other transforms the elements *N*/2 to *N*. In turn, each subtask can be reduced to another set of subtasks, each transforming half of the remaining elements. This process of dividing the array will continue until a threshold is reached in which a sequential solution is faster than creating another division.

The advantage of the divide-and-conquer strategy is that the processing can occur in parallel. Therefore, instead of cycling through an entire array using a single thread, pieces of the array can be processed simultaneously. Of course, the divide-and-conquer approach works in many cases in which an array (or collection) is not present, but the most common uses involve some type of array, collection, or grouping of data.

One of the keys to best employing the divide-and-conquer strategy is correctly selecting the threshold at which sequential processing (rather than further division) is used. Typically, an optimal threshold is obtained through profiling the execution characteristics. However, very significant speed-ups will still occur even when a less-than-optimal threshold is used. It is, however, best to avoid overly large or overly small thresholds. At the time of this writing, the Java API documentation for **ForkJoinTask<T>** states that, as a rule-of-thumb, a task should perform somewhere between 100 and 10,000 computational steps.

It is also important to understand that the optimal threshold value is also affected by how much time the computation takes. If each computational step is fairly long, then smaller thresholds might be better. Conversely, if each computational step is quite short, then larger thresholds could yield better results. For applications that are to be run on a known system, with a known number of processors, you can use the number of processors to make informed decisions about the threshold value. However, for applications that will be running on a variety of systems, the capabilities of which are not known in advance, you can make no assumptions about the execution environment.

One other point: Although multiple processors may be available on a system, other tasks (and the operating system, itself) will be competing with your application for CPU time. Thus, it is important not to assume that your program will have unrestricted access to all CPUs. Furthermore, different runs of the same program may display different run time characteristics because of varying task loads.

## A Simple First Fork/Join Example

At this point, a simple example that demonstrates the Fork/Join Framework and the divide-and-conquer strategy will be helpful. Following is a program that transforms the elements in an array of **double** into their square roots. It does so via a subclass of **RecursiveAction**. Notice that it creates its own **ForkJoinPool**.

```
// A simple example of the basic divide-and-conquer strategy.
// In this case, RecursiveAction is used.
import java.util.concurrent.*;
import java.util.*;

// A ForkJoinTask (via RecursiveAction) that transforms
// the elements in an array of doubles into their square roots.
```

```
class SqrtTransform extends RecursiveAction {
  // The threshold value is arbitrarily set at 1,000 in this example.
  // In real-world code, its optimal value can be determined by
  // profiling and experimentation.
  final int seqThreshold = 1000;

  // Array to be accessed.
  double[] data;

  // Determines what part of data to process.
  int start, end;

  SqrtTransform(double[] vals, int s, int e ) {
    data = vals;
    start = s;
    end = e;
  }

  // This is the method in which parallel computation will occur.
  protected void compute() {

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThreshold) {
      // Transform each element into its square root.
      for(int i = start; i < end; i++) {
         data[i] = Math.sqrt(data[i]);
      }
    }
    else {
      // Otherwise, continue to break the data into smaller pieces.

      // Find the midpoint.
      int middle = (start + end) / 2;

      // Invoke new tasks, using the subdivided data.
      invokeAll(new SqrtTransform(data, start, middle),
                new SqrtTransform(data, middle, end));
    }
  }
}

// Demonstrate parallel execution.
class ForkJoinDemo {
  public static void main(String args[]) {
    // Create a task pool.
    ForkJoinPool fjp = new ForkJoinPool();

    double[] nums = new double[100000];

    // Give nums some values.
    for(int i = 0; i < nums.length; i++)
      nums[i] = (double) i;
```

```
    System.out.println("A portion of the original sequence:");

    for(int i=0; i < 10; i++)
      System.out.print(nums[i] + " ");
    System.out.println("\n");

    SqrtTransform task = new SqrtTransform(nums, 0, nums.length);

    // Start the main ForkJoinTask.
    fjp.invoke(task);

    System.out.println("A portion of the transformed sequence" +
                       " (to four decimal places):");
    for(int i=0; i < 10; i++)
      System.out.format("%.4f ", nums[i]);
    System.out.println();
  }
}
```

The output from the program is shown here:

```
A portion of the original sequence:
0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

A portion of the transformed sequence (to four decimal places):
0.0000 1.0000 1.4142 1.7321 2.0000 2.2361 2.4495 2.6458 2.8284 3.0000
```

As you can see, the values of the array elements have been transformed into their square roots.

Let's look closely at how this program works. First, notice that **SqrtTransform** is a class that extends **RecursiveAction**. As explained, **RecursiveAction** extends **ForkJoinTask** for tasks that do not return results. Next, notice the **final** variable **seqThreshold**. This is the value that determines when sequential processing will take place. This value is set (somewhat arbitrarily) to 1,000. Next, notice that a reference to the array to be processed is stored in **data** and that the fields **start** and **end** are used to indicate the boundaries of the elements to be accessed.

The main action of the program takes place in **compute( )**. It begins by checking if the number of elements to be processed is below the sequential processing threshold. If it is, then those elements are processed (by computing their square root in this example). If the sequential processing threshold has not been reached, then two new tasks are started by calling **invokeAll( )**. In this case, each subtask processes half the elements. As explained earlier, **invokeAll( )** waits until both tasks return. After all of the recursive calls unwind, each element in the array will have been modified, with much of the action taking place in parallel (if multiple processors are available).

As mentioned, beginning with JDK 8, it is not necessary to explicitly construct a **ForkJoinPool** because a common pool is available for your use. Furthermore, using the common pool is a simple matter. For example, you can obtain a reference to the common pool by calling the static **commonPool( )** method defined by **ForkJoinPool**. Therefore, the preceding program could be rewritten to use the common pool by replacing the call to the **ForkJoinPool** constructor with a call to **commonPool( )**, as shown here:

```
ForkJoinPool fjp = ForkJoinPool.commonPool();
```

Alternatively, there is no need to explicitly obtain a reference to the common pool because calling the **ForkJoinTask** methods **invoke( )** or **fork( )** on a task that is not already part of a pool will cause it to execute within the common pool automatically. For example, in the preceding program, you can eliminate the **fjp** variable entirely and start the task using this line:

```
task.invoke();
```

As this discussion shows, the common pool is one of the enhancements JDK 8 made to the Fork/Join Framework that improves its ease-of-use. Furthermore, in many cases, the common pool is the preferable approach, assuming that JDK 7 compatibility is not required.



## Understanding the Impact of the Level of Parallelism

Before moving on, it is important to understand the impact that the level of parallelism has on the performance of a fork/join task and how the parallelism and the threshold interact. The program shown in this section lets you experiment with different degrees of parallelism and threshold values. Assuming that you are using a multicore computer, you can interactively observe the effect of these values.

In the preceding example, the default level of parallelism was used. However, you can specify the level of parallelism that you want. One way is to specify it when you create a **ForkJoinPool** using this constructor:

ForkJoinPool(int *pLevel*)

Here, *pLevel* specifies the level of parallelism, which must be greater than zero and less than the implementation defined limit.

The following program creates a fork/join task that transforms an array of **double**s. The transformation is arbitrary, but it is designed to consume several CPU cycles. This was done to ensure that the effects of changing the threshold or the level of parallelism would be more clearly displayed. To use the program, specify the threshold value and the level of parallelism on the command line. The program then runs the tasks. It also displays the amount of time it takes the tasks to run. To do this, it uses **System.nanoTime( )**, which returns the value of the JVM's high-resolution timer.

```
// A simple program that lets you experiment with the effects of
// changing the threshold and parallelism of a ForkJoinTask.
import java.util.concurrent.*;

// A ForkJoinTask (via RecursiveAction) that performs a
// a transform on the elements of an array of doubles.
class Transform extends RecursiveAction {

  // Sequential threshold, which is set by the constructor.
  int seqThreshold;

  // Array to be accessed.
  double[] data;

  // Determines what part of data to process.
  int start, end;
```

```
  Transform(double[] vals, int s, int e, int t ) {
    data = vals;
    start = s;
    end = e;
    seqThreshold = t;
  }

  // This is the method in which parallel computation will occur.
  protected void compute() {

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThreshold) {
      // The following code assigns an element at an even index the
      // square root of its original value. An element at an odd
      // index is assigned its cube root. This code is designed
      // to simply consume CPU time so that the effects of concurrent
      // execution are more readily observable.
      for(int i = start; i < end; i++) {
        if((data[i] % 2) == 0)
          data[i] = Math.sqrt(data[i]);
        else
          data[i] = Math.cbrt(data[i]);
      }
    }
    else {
      // Otherwise, continue to break the data into smaller pieces.

      // Find the midpoint.
      int middle = (start + end) / 2;

      // Invoke new tasks, using the subdivided data.
      invokeAll(new Transform(data, start, middle, seqThreshold),
                new Transform(data, middle, end, seqThreshold));
    }
  }
}

// Demonstrate parallel execution.
class FJExperiment {

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

    if(args.length !=  2) {
      System.out.println("Usage: FJExperiment parallelism threshold ");
      return;
    }

    pLevel = Integer.parseInt(args[0]);
    threshold = Integer.parseInt(args[1]);

    // These variables are used to time the task.
    long beginT, endT;
```

```
    // Create a task pool. Notice that the parallelism level is set.
    ForkJoinPool fjp = new ForkJoinPool(pLevel);

    double[] nums = new double[1000000];

    for(int i = 0; i < nums.length; i++)
      nums[i] = (double) i;

    Transform task = new Transform(nums, 0, nums.length, threshold);

    // Starting timing.
    beginT = System.nanoTime();

    // Start the main ForkJoinTask.
    fjp.invoke(task);

    // End timing.
    endT = System.nanoTime();

    System.out.println("Level of parallelism: " + pLevel);
    System.out.println("Sequential threshold: " + threshold);
    System.out.println("Elapsed time: " + (endT - beginT) + " ns");
    System.out.println();
  }
}
```



To use the program, specify the level of parallelism followed by the threshold limit. You should try experimenting with different values for each, observing the results. Remember, to be effective, you must run the code on a computer with at least two processors. Also, understand that two different runs may (almost certainly will) produce different results because of the effect of other processes in the system consuming CPU time.

To give you an idea of the difference that parallelism makes, try this experiment. First, execute the program like this:

```
java FJExperiment 1 1000
```

This requests 1 level of parallelism (essentially sequential execution) with a threshold of 1,000. Here is a sample run produced on a dual-core computer:

```
Level of parallelism: 1
Sequential threshold: 1000
Elapsed time: 259677487 ns
```

Now, specify 2 levels of parallelism like this:

```
java FJExperiment 2 1000
```

Here is sample output from this run produced by the same dual-core computer:

```
Level of parallelism: 2
Sequential threshold: 1000
Elapsed time: 169254472 ns
```

As is evident, adding parallelism substantially decreases execution time, thus increasing the speed of the program. You should experiment with varying the threshold and parallelism on your own computer. The results may surprise you.

Here are two other methods that you might find useful when experimenting with the execution characteristics of a fork/join program. First, you can obtain the level of parallelism by calling **getParallelism( )**, which is defined by **ForkJoinPool**. It is shown here:

```
int getParallelism( )
```

It returns the parallelism level currently in effect. Recall that for pools that you create, by default, this value will equal the number of available processors. (To obtain the parallelism level for the common pool, you can also use **getCommonPoolParallelism( )**, which was added by JDK 8.) Second, you can obtain the number of processors available in the system by calling **availableProcessors( )**, which is defined by the **Runtime** class. It is shown here:

```
int availableProcessors( )
```

The value returned may change from one call to the next because of other system demands.

## An Example that Uses RecursiveTask<V>

The two preceding examples are based on **RecursiveAction**, which means that they concurrently execute tasks that do not return results. To create a task that returns a result, use **RecursiveTask**. In general, solutions are designed in the same manner as just shown. The key difference is that the **compute( )** method returns a result. Thus, you must aggregate the results, so that when the first invocation finishes, it returns the overall result. Another difference is that you will typically start a subtask by calling **fork( )** and **join( )** explicitly (rather than implicitly by calling **invokeAll( )**, for example).

The following program demonstrates **RecursiveTask**. It creates a task called **Sum** that returns the summation of the values in an array of **double**. In this example, the array consists of 5,000 elements. However, every other value is negative. Thus, the first values in the array are 0, –1, 2, –3, 4, and so on. (So that this example will work with both JDK 7 and JDK 8, it creates its own pool. You might try changing it to use the common pool as an exercise.)

```
// A simple example that uses RecursiveTask<V>.
import java.util.concurrent.*;

// A RecursiveTask that computes the summation of an array of doubles.
class Sum extends RecursiveTask<Double> {

  // The sequential threshold value.
  final int seqThresHold = 500;

  // Array to be accessed.
  double[] data;

  // Determines what part of data to process.
  int start, end;

  Sum(double[] vals, int s, int e ) {
    data = vals;
```

```
    start = s;
    end = e;
  }

  // Find the summation of an array of doubles.
  protected Double compute() {
    double sum = 0;

    // If number of elements is below the sequential threshold,
    // then process sequentially.
    if((end - start) < seqThresHold) {
      // Sum the elements.
      for(int i = start; i < end; i++) sum += data[i];
    }
    else {
      // Otherwise, continue to break the data into smaller pieces.

      // Find the midpoint.
      int middle = (start + end) / 2;

      // Invoke new tasks, using the subdivided data.
      Sum subTaskA = new Sum(data, start, middle);
      Sum subTaskB = new Sum(data, middle, end);

      // Start each subtask by forking.
      subTaskA.fork();
      subTaskB.fork();

      // Wait for the subtasks to return, and aggregate the results.
      sum = subTaskA.join() + subTaskB.join();
    }
      // Return the final sum.
      return sum;
  }
}

// Demonstrate parallel execution.
class RecurTaskDemo {
  public static void main(String args[]) {
    // Create a task pool.
    ForkJoinPool fjp = new ForkJoinPool();

    double[] nums = new double[5000];

    // Initialize nums with values that alternate between
    // positive and negative.
    for(int i=0; i < nums.length; i++)
      nums[i] = (double) (((i%2) == 0) ? i : -i) ;

    Sum task = new Sum(nums, 0, nums.length);

    // Start the ForkJoinTasks.  Notice that, in this case,
    // invoke() returns a result.
    double summation = fjp.invoke(task);
```

```
    System.out.println("Summation " + summation);
  }
}
```

Here's the output from the program:

```
Summation -2500.0
```

There are a couple of interesting items in this program. First, notice that the two subtasks are executed by calling **fork( )**, as shown here:

```
subTaskA.fork();
subTaskB.fork();
```

In this case, **fork( )** is used because it starts a task but does not wait for it to finish. (Thus, it asynchronously runs the task.) The result of each task is obtained by calling **join( )**, as shown here:

```
sum = subTaskA.join() + subTaskB.join();
```

This statement waits until each task ends. It then adds the results of each and assigns the total to **sum**. Thus, the summation of each subtask is added to the running total. Finally, **compute( )** ends by returning **sum**, which will be the final total when the first invocation returns.

There are other ways to approach the handling of the asynchronous execution of the subtasks. For example, the following sequence uses **fork( )** to start **subTaskA** and uses **invoke( )** to start and wait for **subTaskB**:

```
subTaskA.fork();
sum = subTaskB.invoke() + subTaskA.join();
```

Another alternative is to have **subTaskB** call **compute( )** directly, as shown here:

```
subTaskA.fork();
sum = subTaskB.compute() + subTaskA.join();
```

## Executing a Task Asynchronously

The preceding programs have called **invoke( )** on a **ForkJoinPool** to initiate a task. This approach is commonly used when the calling thread must wait until the task has completed (which is often the case) because **invoke( )** does not return until the task has terminated. However, you can start a task asynchronously. In this approach, the calling thread continues to execute. Thus, both the calling thread and the task execute simultaneously. To start a task asynchronously, use **execute( )**, which is also defined by **ForkJoinPool**. It has the two forms shown here:

void execute(ForkJoinTask<?> *task*)

void execute(Runnable *task*)

In both forms, *task* specifies the task to run. Notice that the second form lets you specify a **Runnable** rather than a **ForkJoinTask** task. Thus, it forms a bridge between Java's traditional approach to multithreading and the new Fork/Join Framework. It is important to remember that the threads used by a **ForkJoinPool** are daemon. Thus, they will end when the main thread ends. As a result, you may need to keep the main thread alive until the tasks have finished.

## Cancelling a Task

A task can be cancelled by calling **cancel( )**, which is defined by **ForkJoinTask**. It has this general form:

```
boolean cancel(boolean interuptOK)
```

It returns **true** if the task on which it was called is cancelled. It returns **false** if the task has ended or can't be cancelled. At this time, the *interruptOK* parameter is not used by the default implementation. In general, **cancel( )** is intended to be called from code outside the task because a task can easily cancel itself by returning.

You can determine if a task has been cancelled by calling **isCancelled( )**, as shown here:

```
final boolean isCancelled( )
```

It returns **true** if the invoking task has been cancelled prior to completion and **false** otherwise.

## Determining a Task's Completion Status

In addition to **isCancelled( )**, which was just described, **ForkJoinTask** includes two other methods that you can use to determine a task's completion status. The first is **isCompletedNormally( )**, which is shown here:

```
final boolean isCompletedNormally( )
```

It returns **true** if the invoking task completed normally, that is, if it did not throw an exception and it was not cancelled via a call to **cancel( )**. It returns **false** otherwise.

The second is **isCompletedAbnormally( )**, which is shown here:

```
final boolean isCompletedAbnormally( )
```

It returns **true** if the invoking task completed because it was cancelled or because it threw an exception. It returns **false** otherwise.

## Restarting a Task

Normally, you cannot rerun a task. In other words, once a task completes, it cannot be restarted. However, you can reinitialize the state of the task (after it has completed) so it can be run again. This is done by calling **reinitialize( )**, as shown here:

```
void reinitialize( )
```

This method resets the state of the invoking task. However, any modification made to any persistent data that is operated upon by the task will not be undone. For example, if the task modifies an array, then those modifications are not undone by calling **reinitialize( )**.

## Things to Explore

The preceding discussion presented the fundamentals of the Fork/Join Framework and described several commonly used methods. However, Fork/Join is a rich framework that includes additional capabilities that give you extended control over concurrency. Although it is far beyond the scope of this book to examine all of the issues and nuances surrounding parallel programming and the Fork/Join Framework, a sampling of the other features are mentioned here.

### A Sampling of Other ForkJoinTask Features

In some cases, you will want to ensure that methods such as **invokeAll( )** and **fork( )** are called only from within a **ForkJoinTask**. (This may be especially important when using JDK 7, which does not support the common pool.) This is usually a simple matter, but occasionally, you may have code that can be executed from either inside or outside a task. You can determine if your code is executing inside a task by calling **inForkJoinPool( )**.

You can convert a **Runnable** or **Callable** object into a **ForkJoinTask** by using the **adapt( )** method defined by **ForkJoinTask**. It has three forms, one for converting a **Callable**, one for a **Runnable** that does not return a result, and one for a **Runnable** that does return a result. In the case of a **Callable**, the **call( )** method is run. In the case of **Runnable**, the **run( )** method is run.

You can obtain an approximate count of the number of tasks that are in the queue of the invoking thread by calling **getQueuedTaskCount( )**. You can obtain an approximate count of how many tasks the invoking thread has in its queue that are in excess of the number of other threads in the pool that might "steal" them, by calling **getSurplusQueuedTaskCount( )**. Remember, in the Fork/Join Framework, work-stealing is one way in which a high level of efficiency is obtained. Although this process is automatic, in some cases, the information may prove helpful in optimizing through-put.

**ForkJoinTask** defines the following variants of **join( )** and **invoke( )** that begin with the prefix **quietly**. They are shown here:

| | |
|---|---|
| final void quietlyJoin( ) | Joins a task, but does not return a result or throw an exception |
| final void quietlyInvoke( ) | Invokes a task, but does not return a result or throw an exception. |

In essence, these methods are similar to their non-quiet counterparts except they don't return values or throw exceptions.

You can attempt to "un-invoke" (in other words, unschedule) a task by calling **tryUnfork( )**.

JDK 8 adds several methods, such as **getForkJoinTaskTag( )** and **setForkJoinTaskTag( )**, that support tags. Tags are short integer values that are linked with a task. They may be useful in specialized applications.

**ForkJoinTask** implements **Serializable**. Thus, it can be serialized. However, serialization is not used during execution.

### A Sampling of Other ForkJoinPool Features

One method that is quite useful when tuning fork/join applications is **ForkJoinPool**'s override of **toString( )**. It displays a "user-friendly" synopsis of the state of the pool. To see it in action, use this sequence to start and then wait for the task in the **FJExperiment** class of the task experimenter program shown earlier:

```
// Asynchronously start the main ForkJoinTask.
fjp.execute(task);

// Display the state of the pool while waiting.
while(!task.isDone()) {
  System.out.println(fjp);
}
```

When you run the program, you will see a series of messages on the screen that describe the state of the pool. Here is an example of one. Of course, your output may vary, based on the number of processors, threshold values, task load, and so on.

```
java.util.concurrent.ForkJoinPool@141d683[Running, parallelism = 2,
size = 2, active = 0, running = 2, steals = 0, tasks = 0, submissions = 1]
```

You can determine if a pool is currently idle by calling **isQuiescent( )**. It returns **true** if the pool has no active threads and **false** otherwise.

You can obtain the number of worker threads currently in the pool by calling **getPoolSize( )**. You can obtain an approximate count of the active threads in the pool by calling **getActiveThreadCount( )**.

To shut down a pool, call **shutdown( )**. Currently active tasks will still be executed, but no new tasks can be started. To stop a pool immediately, call **shutdownNow( )**. In this case, an attempt is made to cancel currently active tasks. (It is important to point out, however, that neither of these methods affects the common pool.) You can determine if a pool is shut down by calling **isShutdown( )**. It returns **true** if the pool has been shut down and **false** otherwise. To determine if the pool has been shut down and all tasks have been completed, call **isTerminated( )**.

## Some Fork/Join Tips

Here are a few tips to help you avoid some of the more troublesome pitfalls associated with using the Fork/Join Framework. First, avoid using a sequential threshold that is too low. In general, erring on the high side is better than erring on the low side. If the threshold is too low, more time can be consumed generating and switching tasks than in processing the tasks. Second, usually it is best to use the default level of parallelism. If you specify a smaller number, it may significantly reduce the benefits of using the Fork/Join Framework.

In general, a **ForkJoinTask** should not use synchronized methods or synchronized blocks of code. Also, you will not normally want to have the **compute( )** method use other types of synchronization, such as a semaphore. (The new **Phaser** can, however, be used when appropriate because it is compatible with the fork/join mechanism.) Remember, the main idea behind a **ForkJoinTask** is the divide-and-conquer strategy. Such an approach does not normally lend itself to situations in which outside synchronization is needed. Also, avoid situations in which substantial blocking will occur through I/O. Therefore, in general,

a **ForkJoinTask** will not perform I/O. Simply put, to best utilize the Fork/Join Framework, a task should perform a computation that can run without outside blocking or synchronization.

One last point: Except under unusual circumstances, do not make assumptions about the execution environment that your code will run in. This means you should not assume that some specific number of processors will be available, or that the execution characteristics of your program won't be affected by other processes running at the same time.

## The Concurrency Utilities Versus Java's Traditional Approach

Given the power and flexibility found in the concurrency utilities, it is natural to ask the following question: Do they replace Java's traditional approach to multithreading and synchronization? The answer is a resounding no! The original support for multithreading and the built-in synchronization features are still the mechanism that should be employed for many, many Java programs, applets, and servlets. For example, **synchronized**, **wait( )**, and **notify( )** offer elegant solutions to a wide range of problems. However, when extra control is needed, the concurrency utilities are available to handle the chore. Furthermore, the Fork/Join Framework offers a powerful way to integrate parallel programming techniques into your more sophisticated applications.

CHAPTER

# 29 The Stream API

Of the many new features added by JDK 8, the two that are, arguably, the most important are lambda expressions and the stream API. Lambda expressions were described in Chapter 15. The stream API is described here. As you will see, the stream API is designed with lambda expressions in mind. Moreover, the stream API provides some of the most significant demonstrations of the power that lambdas bring to Java.

Although its design compatibility with lambda expressions is impressive, the key aspect of the stream API is its ability to perform very sophisticated operations that search, filter, map, or otherwise manipulate data. For example, using the stream API, you can construct sequences of actions that resemble, in concept, the type of database queries for which you might use SQL. Furthermore, in many cases, such actions can be performed in parallel, thus providing a high level of efficiency, especially when large data sets are involved. Put simply, the stream API provides a powerful means of handling data in an efficient, yet easy to use way.

Before continuing, an important point needs to be made: The stream API uses some of Java's most advanced features. To fully understand and utilize it requires a solid understanding of generics and lambda expressions. The basic concepts of parallel execution and a working knowledge of the Collections Framework are also needed. (See Chapters 14, 15, 18, and 28.)

## Stream Basics

Let's begin by defining the term *stream* as it applies to the stream API: a stream is a conduit for data. Thus, a stream represents a sequence of objects. A stream operates on a data source, such as an array or a collection. A stream, itself, never provides storage for the data. It simply moves data, possibly filtering, sorting, or otherwise operating on that data in the process. As a general rule, however, a stream operation by itself does not modify the data source. For example, sorting a stream does not change the order of the source. Rather, sorting a stream results in the creation of a new stream that produces the sorted result.

**NOTE** It is necessary to state that the term *stream* as used here differs from the use of *stream* when the I/O classes were described earlier in this book. Although an I/O stream can act conceptually much like one of the streams defined by **java.util.stream**, they are not the same. Thus, throughout this chapter, when the term *stream* is used, it refers to objects based on one of the stream types described here.

## Stream Interfaces

The stream API defines several stream interfaces, which are packaged in **java.util.stream**. At the foundation is **BaseStream**, which defines the basic functionality available in all streams. **BaseStream** is a generic interface declared like this:

```
interface BaseStream<T, S extends BaseStream<T, S>>
```

Here, **T** specifies the type of the elements in the stream, and **S** specifies the type of stream that extends **BaseStream**. **BaseStream** extends the **AutoCloseable** interface; thus, a stream can be managed in a **try**-with-resources statement. In general, however, only those streams whose data source requires closing (such as those connected to a file) will need to be closed. In most cases, such as those in which the data source is a collection, there is no need to close the stream. The methods declared by **BaseStream** are shown in Table 29-1.

| Method | Description |
|---|---|
| void close( ) | Closes the invoking stream, calling any registered close handlers. (As explained in the text, few streams need to be closed.) |
| boolean isParallel( ) | Returns **true** if the invoking stream is parallel. Returns **false** if the stream is sequential. |
| Iterator<T> iterator( ) | Obtains an iterator to the stream and returns a reference to it. (Terminal operation.) |
| S onClose(Runnable *handler*) | Returns a new stream with the close handler specified by *handler*. This handler will be called when the stream is closed. (Intermediate operation.) |
| S parallel( ) | Returns a parallel stream based on the invoking stream. If the invoking stream is already parallel, then that stream is returned. (Intermediate operation.) |
| S sequential( ) | Returns a sequential stream based on the invoking stream. If the invoking stream is already sequential, then that stream is returned. (Intermediate operation.) |
| Spliterator<T> spliterator( ) | Obtains a spliterator to the stream and returns a reference to it. (Terminal operation.) |
| S unordered( ) | Returns an unordered stream based on the invoking stream. If the invoking stream is already unordered, then that stream is returned. (Intermediate operation.) |

**Table 29-1** The Methods Declared by **BaseStream**

From **BaseStream** are derived several types of stream interfaces. The most general of these is **Stream**. It is declared as shown here:

interface Stream<T>

Here, **T** specifies the type of the elements in the stream. Because it is generic, **Stream** is used for all reference types. In addition to the methods that it inherits from **BaseStream**, the **Stream** interface adds several of its own, a sampling of which is shown in Table 29-2.



| Method | Description |
|---|---|
| <R, A> R collect(Collector<? super T, A, R> *collectorFunc*) | Collects elements into a container, which is changeable, and returns the container. This is called a mutable reduction operation. Here, **R** specifies the type of the resulting container and **T** specifies the element type of the invoking stream. **A** specifies the internal accumulated type. The *collectorFunc* specifies how the collection process works. (Terminal operation.) |
| long count( ) | Counts the number of elements in the stream and returns the result. (Terminal operation.) |
| Stream<T> filter(Predicate<? super T> *pred*) | Produces a stream that contains those elements from the invoking stream that satisfy the predicate specified by *pred*. (Intermediate operation.) |
| void forEach(Consumer<? super T> *action*) | For each element in the invoking stream, the code specified by *action* is executed. (Terminal operation.) |
| <R> Stream<R> map(Function<? super T, ? extends R> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new stream that contains those elements. (Intermediate operation.) |
| DoubleStream mapToDouble( ToDoubleFunction<? super T> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new **DoubleStream** that contains those elements. (Intermediate operation.) |
| IntStream mapToInt( ToIntFunction<? super T> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new **IntStream** that contains those elements. (Intermediate operation.) |
| LongStream mapToLong( ToLongFunction<? super T> *mapFunc*) | Applies *mapFunc* to the elements from the invoking stream, yielding a new **LongStream** that contains those elements. (Intermediate operation.) |
| Optional<T> max( Comparator<? super T> *comp*) | Using the ordering specified by *comp*, finds and returns the maximum element in the invoking stream. (Terminal operation.) |

**Table 29-2** A Sampling of Methods Declared by **Stream**

| Method | Description |
|---|---|
| Optional<T> min(Comparator<? super T> *comp*) | Using the ordering specified by *comp*, finds and returns the minimum element in the invoking stream. (Terminal operation.) |
| T reduce(T *identityVal*,<br>BinaryOperator<T> *accumulator*) | Returns a result based on the elements in the invoking stream. This is called a reduction operation. (Terminal operation.) |
| Stream<T> sorted( ) | Produces a new stream that contains the elements of the invoking stream sorted in natural order. (Intermediate operation.) |
| Object[] toArray( ) | Creates an array from the elements in the invoking stream. (Terminal operation.) |

**Table 29-2** A Sampling of Methods Declared by **Stream** *(continued)*

In both tables, notice that many of the methods are notated as being either *terminal* or *intermediate*. The difference between the two is very important. A *terminal* operation consumes the stream. It is used to produce a result, such as finding the minimum value in the stream, or to execute some action, as is the case with the **forEach( )** method. Once a stream has been consumed, it cannot be reused. *Intermediate* operations produce another stream. Thus, intermediate operations can be used to create a *pipeline* that performs a sequence of actions. One other point: intermediate operations do not take place immediately. Instead, the specified action is performed when a terminal operation is executed on the new stream created by an intermediate operation. This mechanism is referred to as *lazy behavior,* and the intermediate operations are referred to as *lazy*. The use of lazy behavior enables the stream API to perform more efficiently.

Another key aspect of streams is that some intermediate operations are *stateless* and some are *stateful*. In a stateless operation, each element is processed independently of the others. In a stateful operation, the processing of an element may depend on aspects of the other elements. For example, sorting is a stateful operation because an element's order depends on the values of the other elements. Thus, the **sorted( )** method is stateful. However, filtering elements based on a stateless predicate is stateless because each element is handled individually. Thus, **filter( )** can (and should be) stateless. The difference between stateless and stateful operations is especially important when parallel processing of a stream is desired because a stateful operation may require more than one pass to complete.

Because **Stream** operates on object references, it can't operate directly on primitive types. To handle primitive type streams, the stream API defines the following interfaces:

DoubleStream

IntStream

LongStream

These streams all extend **BaseStream** and have capabilities similar to **Stream** except that they operate on primitive types rather than reference types. They also provide some convenience methods, such as **boxed( )**, that facilitate their use. Because streams of objects are the most common, **Stream** is the primary focus of this chapter, but the primitive type streams can be used in much the same way.

## How to Obtain a Stream

You can obtain a stream in a number of ways. Perhaps the most common is when a stream is obtained for a collection. Beginning with JDK 8, the **Collection** interface has been expanded to include two methods that obtain a stream from a collection. The first is **stream( )**, shown here:

default Stream<E> stream( )

Its default implementation returns a sequential stream. The second method is **parallelStream( )**, shown next:

default Stream<E> parallelStream( )

Its default implementation returns a parallel stream, if possible. (If a parallel stream can not be obtained, a sequential stream may be returned instead.) Parallel streams support parallel execution of stream operations. Because **Collection** is implemented by every collection, these methods can be used to obtain a stream from any collection class, such as **ArrayList** or **HashSet**.

A stream can also be obtained from an array by use of the static **stream( )** method, which was added to the **Arrays** class by JDK 8. One of its forms is shown here:

static <T> Stream<T> stream(T[ ] *array*)

This method returns a sequential stream to the elements in *array*. For example, given an array called **addresses** of type **Address**, the following obtains a stream to it:

```
Stream<Address> addrStrm = Arrays.stream(addresses);
```

Several overloads of the **stream( )** method are also defined, such as those that handle arrays of the primitive types. They return a stream of type **IntStream**, **DoubleStream**, or **LongStream**.

Streams can be obtained in a variety of other ways. For example, many stream operations return a new stream, and a stream to an I/O source can be obtained by calling **lines( )** on a **BufferedReader**. However a stream is obtained, it can be used in the same way as any other stream.

## A Simple Stream Example

Before going any further, let's work through an example that uses streams. The following program creates an **ArrayList** called **myList** that holds a collection of integers (which are automatically boxed into the **Integer** reference type). Next, it obtains a stream that uses **myList** as a source. It then demonstrates various stream operations.

```
// Demonstrate several stream operations.

import java.util.*;
import java.util.stream.*;

class StreamDemo {

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

```
    // Create a list of Integer values.
    ArrayList<Integer> myList = new ArrayList<>( );
    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    System.out.println("Original list: " + myList);

    // Obtain a Stream to the array list.
    Stream<Integer> myStream = myList.stream();

    // Obtain the minimum and maximum value by use of min(),
    // max(), isPresent(), and get().
    Optional<Integer> minVal = myStream.min(Integer::compare);
    if(minVal.isPresent()) System.out.println("Minimum value: " +
                                              minVal.get());

    // Must obtain a new stream because previous call to min()
    // is a terminal operation that consumed the stream.
    myStream = myList.stream();
    Optional<Integer> maxVal = myStream.max(Integer::compare);
    if(maxVal.isPresent()) System.out.println("Maximum value: " +
                                              maxVal.get());

    // Sort the stream by use of sorted().
    Stream<Integer> sortedStream = myList.stream().sorted();

    // Display the sorted stream by use of forEach().
    System.out.print("Sorted stream: ");
    sortedStream.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Display only the odd values by use of filter().
    Stream<Integer> oddVals =
           myList.stream().sorted().filter((n) -> (n % 2) == 1);
    System.out.print("Odd values: ");
    oddVals.forEach((n) -> System.out.print(n + " "));
    System.out.println();

    // Display only the odd values that are greater than 5. Notice that
    // two filter operations are pipelined.
    oddVals = myList.stream().filter( (n) -> (n % 2) == 1)
                            .filter((n) -> n > 5);
    System.out.print("Odd values greater than 5: ");
    oddVals.forEach((n) -> System.out.print(n + " ") );
    System.out.println();
  }
}
```

The output is shown here:

```
Original list: [7, 18, 10, 24, 17, 5]
Minimum value: 5
Maximum value: 24
Sorted stream: 5 7 10 17 18 24
Odd values: 5 7 17
Odd values greater than 5: 7 17
```

Let's look closely at each stream operation. After creating an **ArrayList**, the program obtains a stream for the list by calling **stream( )**, as shown here:

```
Stream<Integer> myStream = myList.stream();
```

As explained, the **Collection** interface now defines the **stream( )** method, which obtains a stream from the invoking collection. Because **Collection** is implemented by every collection class, **stream( )** can be used to obtain stream for any type of collection, including the **ArrayList** used here. In this case, a reference to the stream is assigned to **myStream**.

Next, the program obtains the minimum value in the stream (which is, of course, also the minimum value in the data source) and displays it, as shown here:

```
Optional<Integer> minVal = myStream.min(Integer::compare);
if(minVal.isPresent()) System.out.println("Minimum value: " +
                                          minVal.get());
```

Recall from Table 29-2 that **min( )** is declared like this:

Optional<T> min(Comparator<? super T> *comp*)

First, notice that the type of **min( )'s** parameter is a **Comparator**. This comparator is used to compare two elements in the stream. In the example, **min( )** is passed a method reference to **Integer**'s **compare( )** method, which is used to implement a **Comparator** capable of comparing two **Integer**s. Next, notice that the return type of **min( )** is **Optional**. The **Optional** class is described in Chapter 19, but briefly, here is how it works. **Optional** is a generic class packaged in **java.util** and declared like this:

class Optional<T>

Here, **T** specifies the element type. An **Optional** instance can either contain a value of type **T** or be empty. You can use **isPresent( )** to determine if a value is present. Assuming that a value is available, it can be obtained by calling **get( )**. In this example, the object returned will hold the minimum value of the stream as an **Integer** object.

One other point about the preceding line: **min( )** is a terminal operation that consumes the stream. Thus, **myStream** cannot be used again after **min( )** executes.

The next lines obtain and display the maximum value in the stream:

```
myStream = myList.stream();
Optional<Integer> maxVal = myStream.max(Integer::compare);
if(maxVal.isPresent()) System.out.println("Maximum value: " +
                                          maxVal.get());
```

First, **myStream** is once again assigned the stream returned by **myList.stream( )**. As just explained, this is necessary because the previous call to **min( )** consumed the previous stream. Thus, a new one is needed. Next, the **max( )** method is called to obtain the maximum value. Like **min( )**, **max( )** returns an **Optional** object. Its value is obtained by calling **get( )**.

The program then obtains a sorted stream through the use of this line:

```
Stream<Integer> sortedStream = myList.stream().sorted();
```

Here, the **sorted( )** method is called on the stream returned by **myList.stream( )**. Because **sorted( )** is an intermediate operation, its result is a new stream, and this is the stream assigned to **sortedStream**. The contents of the sorted stream are displayed by use of **forEach( )**:

```
sortedStream.forEach((n) -> System.out.print(n + " "));
```

Here, the **forEach( )** method executes an operation on each element in the stream. In this case, it simply calls **System.out.print( )** for each element in **sortedStream**. This is accomplished by use of a lambda expression. The **forEach( )** method has this general form:

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

**Consumer** is a generic functional interface declared in **java.util.function**. Its abstract method is **accept( )**, shown here:

void accept(T *objRef*)

The lambda expression in the call to **forEach( )** provides the implementation of **accept( )**. The **forEach( )** method is a terminal operation. Thus, after it completes, the stream has been consumed.

Next, a sorted stream is filtered by **filter( )** so that it contains only odd values:

```
Stream<Integer> oddVals =
       myList.stream().sorted().filter((n) -> (n % 2) == 1);
```

The **filter( )** method filters a stream based on a predicate. It returns a new stream that contains only those elements that satisfy the predicate. It is shown here:

Stream<T> filter(Predicate<? super T> *pred*)

**Predicate** is a generic functional interface defined in **java.util.function**. Its abstract method is **test( )**, which is shown here:

boolean test(T *objRef*)

It returns **true** if the object referred to by *objRef* satisfies the predicate, and **false** otherwise. The lambda expression passed to **filter( )** implements this method. Because **filter( )** is an intermediate operation, it returns a new stream that contains filtered values, which, in this case, are the odd numbers. These elements are then displayed via **forEach( )** as before.

Because **filter( )**, or any other intermediate operation, returns a new stream, it is possible to filter a filtered stream a second time. This is demonstrated by the following line, which produces a stream that contains only those odd values greater than 5:

```
oddVals = myList.stream().filter((n) -> (n % 2) == 1)
                         .filter((n) -> n > 5);
```

Notice that lambda expressions are passed to both filters.



## Reduction Operations

Consider the **min( )** and **max( )** methods in the preceding example program. Both are terminal operations that return a result based on the elements in the stream. In the language of the stream API, they represent *reduction operations* because each reduces a stream to a single value—in this case, the minimum and maximum. The stream API refers to these as *special case* reductions because they perform a specific function. In addition to **min( )** and **max( )**, other special case reductions are also available, such as **count( )**, which counts the number of elements in a stream. However, the stream API generalizes this concept by providing the **reduce( )** method. By using **reduce( )**, you can return a value from a stream based on any arbitrary criteria. By definition, all reduction operations are terminal operations.

**Stream** defines three versions of **reduce( )**. The two we will use first are shown here:

Optional<T> reduce(BinaryOperator<T> *accumulator*)

T reduce(T *identityVal*, BinaryOperator<T> *accumulator*)

The first form returns an object of type **Optional**, which contains the result. The second form returns an object of type **T** (which is the element type of the stream). In both forms, *accumulator* is a function that operates on two values and produces a result. In the second form, *identityVal* is a value such that an accumulator operation involving *identityVal* and any element of the stream yields that element, unchanged. For example, if the operation is addition, then the identity value will be 0 because 0 + x is x. For multiplication, the value will be 1, because 1 * x is x.

**BinaryOperator** is a functional interface declared in **java.util.function** that extends the **BiFunction** functional interface. **BiFunction** defines this abstract method:

R apply(T *val*, U *val2*)

Here, **R** specifies the result type, **T** is the type of the first operand, and **U** is the type of second operand. Thus, **apply( )** applies a function to its two operands (*val* and *val2*) and returns the result. When **BinaryOperator** extends **BiFunction**, it specifies the same type for all the type parameters. Thus, as it relates to **BinaryOperator**, **apply( )** looks like this:

T apply(T *val*, T *val2*)

Furthermore, as it relates to **reduce( )**, *val* will contain the previous result and *val2* will contain the next element. In its first invocation, *val* will contain either the identity value or the first element, depending on which version of **reduce( )** is used.

It is important to understand that the accumulator operation must satisfy three constraints. It must be

- Stateless
- Non-interfering
- Associative

As explained earlier, *stateless* means that the operation does not rely on any state information. Thus, each element is processed independently. *Non-interfering* means that the data source is not modified by the operation. Finally, the operation must be *associative*. Here, the term *associative* is used in its normal, arithmetic sense, which means that, given an associative operator used in a sequence of operations, it does not matter which pair of operands are processed first. For example,

(10 * 2) * 7

yields the same result as

10 * (2 * 7)

Associativity is of particular importance to the use of reduction operations on parallel streams, discussed in the next section.

The following program demonstrates the versions of **reduce( )** just described:

```
// Demonstrate the reduce() method.

import java.util.*;
import java.util.stream.*;

class StreamDemo2 {

  public static void main(String[] args) {

    // Create a list of Integer values.
    ArrayList<Integer> myList = new ArrayList<>( );

    myList.add(7);
    myList.add(18);
    myList.add(10);
    myList.add(24);
    myList.add(17);
    myList.add(5);

    // Two ways to obtain the integer product of the elements
    // in myList by use of reduce().
    Optional<Integer> productObj = myList.stream().reduce((a,b) -> a*b);
    if(productObj.isPresent())
      System.out.println("Product as Optional: " + productObj.get());

    int product = myList.stream().reduce(1, (a,b) -> a*b);
    System.out.println("Product as int: " + product);
  }
}
```

As the output here shows, both uses of **reduce( )** produce the same result:

```
Product as Optional: 2570400
Product as int: 2570400
```

In the program, the first version of **reduce( )** uses the lambda expression to produce a product of two values. In this case, because the stream contains **Integer** values, the **Integer** objects are automatically unboxed for the multiplication and reboxed to return the result. The two values represent the current value of the running result and the next element in the stream. The final result is returned in an object of type **Optional**. The value is obtained by calling **get( )** on the returned object.

In the second version, the identity value is explicitly specified, which for multiplication is 1. Notice that the result is returned as an object of the element type, which is **Integer** in this case.

Although simple reduction operations such as multiplication are useful for examples, reductions are not limited in this regard. For example, assuming the preceding program, the following obtains the product of only the even values:

```
int evenProduct = myList.stream().reduce(1, (a,b) -> {
                    if(b%2 == 0) return a*b; else return a;
                  });
```

Pay special attention to the lambda expression. If **b** is even, then **a** * **b** is returned. Otherwise, **a** is returned. This works because **a** holds the current result and **b** holds the next element, as explained earlier.

## Using Parallel Streams

Before exploring any more of the stream API, it will be helpful to discuss parallel streams. As has been pointed out previously in this book, the parallel execution of code via multicore processors can result in a substantial increase in performance. Because of this, parallel programming has become an important part of the modern programmer's job. However, parallel programming can be complex and error-prone. One of the benefits that the stream library offers is the ability to easily—and reliably—parallel process certain operations.

Parallel processing of a stream is quite simple to request: just use a parallel stream. As mentioned earlier, one way to obtain a parallel stream is to use the **parallelStream( )** method defined by **Collection**. Another way to obtain a parallel stream is to call the **parallel( )** method on a sequential stream. The **parallel( )** method is defined by **BaseStream**, as shown here:

S parallel()

It returns a parallel stream based on the sequential stream that invokes it. (If it is called on a stream that is already parallel, then the invoking stream is returned.) Understand, of course, that even with a parallel stream, parallelism will be achieved only if the environment supports it.

Once a parallel stream has been obtained, operations on the stream can occur in parallel, assuming that parallelism is supported by the environment. For example, the first **reduce( )**

operation in the preceding program can be parallelized by substituting **parallelStream( )** for the call to **stream( )**:

```
Optional<Integer> productObj = myList.parallelStream().reduce((a,b) -> a*b);
```

The results will be the same, but the multiplications can occur in different threads.

As a general rule, any operation applied to a parallel stream must be stateless. It should also be non-interfering and associative. This ensures that the results obtained by executing operations on a parallel stream are the same as those obtained from executing the same operations on a sequential stream.

When using parallel streams, you might find the following version of **reduce( )** especially helpful. It gives you a way to specify how partial results are combined:

<U> U reduce(U *identityVal*, BiFunction<U, ? super T, U> *accumulator*
BinaryOperator<U> *combiner*)

In this version, *combiner* defines the function that combines two values that have been produced by the *accumulator* function. Assuming the preceding program, the following statement computes the product of the elements in **myList** by use of a parallel stream:

```
int parallelProduct = myList.parallelStream().reduce(1, (a,b) -> a*b,
                                                     (a,b) -> a*b);
```

As you can see, in this example, both the accumulator and combiner perform the same function. However, there are cases in which the actions of the accumulator must differ from those of the combiner. For example, consider the following program. Here, **myList** contains a list of **double** values. It then uses the combiner version of **reduce( )** to compute the product of the *square roots* of each element in the list.

```
// Demonstrate the use of a combiner with reduce()

import java.util.*;
import java.util.stream.*;

class StreamDemo3 {

  public static void main(String[] args) {

    // This is now a list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);
```

```
    double productOfSqrRoots = myList.parallelStream().reduce(
                                        1.0,
                                        (a,b) -> a * Math.sqrt(b),
                                        (a,b) -> a * b
                                    );

    System.out.println("Product of square roots: " + productOfSqrRoots);
  }
}
```

Notice that the accumulator function multiplies the square roots of two elements, but the combiner multiplies the partial results. Thus, the two functions differ. Moreover, for this computation to work correctly, they *must* differ. For example, if you tried to obtain the product of the square roots of the elements by using the following statement, an error would result:

```
// This won't work.
double productOfSqrRoots2 = myList.parallelStream().reduce(
                                    1.0,
                                    (a,b) -> a * Math.sqrt(b));
```

In this version of **reduce( )**, the accumulator and the combiner function are one and the same. This results in an error because when two partial results are combined, their square roots are multiplied together rather than the partial results, themselves.

As a point of interest, if the stream in the preceding call to **reduce( )** had been changed to a sequential stream, then the operation would yield the correct answer because there would have been no need to combine two partial results. The problem occurs when a parallel stream is used.

You can switch a parallel stream to sequential by calling the **sequential( )** method, which is specified by **BaseStream**. It is shown here:

S sequential( )

In general, a stream can be switched between parallel and sequential on an as-needed basis.

There is one other aspect of a stream to keep in mind when using parallel execution: the order of the elements. Streams can be either ordered or unordered. In general, if the data source is ordered, then the stream will also be ordered. However, when using a parallel stream, a performance boost can sometimes be obtained by allowing a stream to be unordered. When a parallel stream is unordered, each partition of the stream can be operated on independently, without having to coordinate with the others. In cases in which the order of the operations does not matter, it is possible to specify unordered behavior by calling the **unordered( )** method, shown here:

S unordered( )

One other point: the **forEach( )** method may not preserve the ordering of a parallel stream. If you want to perform an operation on each element in a parallel stream while preserving the order, consider using **forEachOrdered( )**. It is used just like **forEach( )**.

## Mapping

Often it is useful to map the elements of one stream to another. For example, a stream that contains a database of name, telephone, and e-mail address information might map only the name and e-mail address portions to another stream. As another example, you might want to apply some transformation to the elements in a stream. To do this, you could map the transformed elements to a new stream. Because mapping operations are quite common, the stream API provides built-in support for them. The most general mapping method is **map( )**. It is shown here:

<R> Stream<R> map(Function<? super T, ? extends R> *mapFunc*)

Here, **R** specifies the type of elements of the new stream; **T** is the type of elements of the invoking stream; and *mapFunc* is an instance of **Function**, which does the mapping. The map function must be stateless and non-interfering. Since a new stream is returned, **map( )** is an intermediate method.

**Function** is a functional interface declared in **java.util.function**. It is declared as shown here:

Function<T, R>

As it relates to **map( )**, **T** is the element type and **R** is the result of the mapping. **Function** has the abstract method shown here:

R apply(T *val*)

Here, *val* is a reference to the object being mapped. The mapped result is returned.

The following is a simple example of **map( )**. It provides a variation on the previous example program. As before, the program computes the product of the square roots of the values in an **ArrayList**. In this version, however, the square roots of the elements are first mapped to a new stream. Then, **reduce( )** is employed to compute the product.

```
// Map one stream to another.

import java.util.*;
import java.util.stream.*;

class StreamDemo4 {

  public static void main(String[] args) {

    // A list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(7.0);
    myList.add(18.0);
    myList.add(10.0);
    myList.add(24.0);
    myList.add(17.0);
    myList.add(5.0);

    // Map the square root of the elements in myList to a new stream.
    Stream<Double> sqrtRootStrm = myList.stream().map((a) -> Math.sqrt(a));
```

```
    // Find the product of the square roots.
    double productOfSqrRoots = sqrtRootStrm.reduce(1.0, (a,b) -> a*b);

    System.out.println("Product of square roots is " + productOfSqrRoots);
  }
}
```

The output is the same as before. The difference between this version and the previous is simply that the transformation (i.e., the computation of the square roots) occurs during mapping, rather than during the reduction. Because of this, it is possible to use the two-parameter form of **reduce( )** to compute the product because it is no longer necessary to provide a separate combiner function.

Here is an example that uses **map( )** to create a new stream that contains only selected fields from the original stream. In this case, the original stream contains objects of type **NamePhoneEmail**, which contains names, phone numbers, and e-mail addresses. The program then maps only the names and phone numbers to a new stream of **NamePhone** objects. The e-mail addresses are discarded.

```
// Use map() to create a new stream that contains only
// selected aspects of the original stream.

import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}

class StreamDemo5 {

  public static void main(String[] args) {

    // A list of names, phone numbers, and e-mail addresses.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>( );
```

```
    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));

    System.out.println("Original values in myList: ");
    myList.stream().forEach( (a) -> {
      System.out.println(a.name + " " + a.phonenum + " " + a.email);
    });
    System.out.println();

    // Map just the names and phone numbers to a new stream.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                     (a) -> new NamePhone(a.name,a.phonenum)
                                    );

    System.out.println("List of names and phone numbers: ");
    nameAndPhone.forEach( (a) -> {
      System.out.println(a.name + " " + a.phonenum);
    });
  }
}
```

The output, shown here, verifies the mapping:

```
Original values in myList:
Larry 555-5555 Larry@HerbSchildt.com
James 555-4444 James@HerbSchildt.com
Mary 555-3333 Mary@HerbSchildt.com

List of names and phone numbers:
Larry 555-5555
James 555-4444
Mary 555-3333
```

Because you can pipeline more than one intermediate operation together, you can easily create very powerful actions. For example, the following statement uses **filter( )** and then **map( )** to produce a new stream that contains only the name and phone number of the elements with the name "James":

```
Stream<NamePhone> nameAndPhone = myList.stream().
                                 filter((a) -> a.name.equals("James")).
                                 map((a) -> new NamePhone(a.name,a.phonenum));
```

This type of filter operation is very common when creating database-style queries. As you gain experience with the stream API, you will find that such chains of operations can be used to create very sophisticated queries, merges, and selections on a data stream.

In addition to the version just described, three other versions of **map( )** are provided. They return a primitive stream, as shown here:

IntStream mapToInt(ToIntFunction<? super T> *mapFunc*)

LongStream mapToLong(ToLongFunction<? super T> *mapFunc*)

DoubleStream mapToDouble(ToDoubleFunction<? super T> *mapFunc*)

Each *mapFunc* must implement the abstract method defined by the specified interface, returning a value of the indicated type. For example, **ToDoubleFunction** specifies the **applyAsDouble(T *val* )** method, which must return the value of its parameter as a **double**.

Here is an example that uses a primitive stream. It first creates an **ArrayList** of **Double** values. It then uses **stream( )** followed by **mapToInt( )** to create an **IntStream** that contains the ceiling of each value.

```
// Map a Stream to an IntStream.

import java.util.*;
import java.util.stream.*;

class StreamDemo6 {

  public static void main(String[] args) {

    // A list of double values.
    ArrayList<Double> myList = new ArrayList<>( );

    myList.add(1.1);
    myList.add(3.6);
    myList.add(9.2);
    myList.add(4.7);
    myList.add(12.1);
    myList.add(5.0);

    System.out.print("Original values in myList: ");
    myList.stream().forEach( (a) -> {
      System.out.print(a + " ");
    });
    System.out.println();

    // Map the ceiling of the elements in myList to an IntStream.
    IntStream cStrm = myList.stream().mapToInt((a) -> (int) Math.ceil(a));

    System.out.print("The ceilings of the values in myList: ");
    cStrm.forEach( (a) -> {
      System.out.print(a + " ");
    });

  }
}
```

The output is shown here:

```
Original values in myList: 1.1 3.6 9.2 4.7 12.1 5.0
The ceilings of the values in myList: 2 4 10 5 13 5
```

The stream produced by **mapToInt( )** contains the ceiling values of the original elements in **myList**.

Before leaving the topic of mapping, it is necessary to point out that the stream API also provides methods that support *flat maps*. These are **flatMap( )**, **flatMapToInt( )**, **flatMapToLong( )**, and **flatMapToDouble( )**. The flat map methods are designed to handle situations in which each element in the original stream is mapped to more than one element in the resulting stream.

## Collecting

As the preceding examples have shown, it is possible (indeed, common) to obtain a stream from a collection. Sometimes it is desirable to obtain the opposite: to obtain a collection from a stream. To perform such an action, the stream API provides the **collect( )** method. It has two forms. The one we will use first is shown here:

<R, A> R collect(Collector<? super T, A, R> *collectorFunc*)

Here, **R** specifies the type of the result, and **T** specifies the element type of the invoking stream. The internal accumulated type is specified by **A**. The *collectorFunc* specifies how the collection process works. The **collect( )** method is a terminal operation.

The **Collector** interface is declared in **java.util.stream**, as shown here:

interface Collector<T, A, R>

**T**, **A**, and **R** have the same meanings as just described. **Collector** specifies several methods, but for the purposes of this chapter, we won't need to implement them. Instead, we will use two of the predefined collectors that are provided by the **Collectors** class, which is packaged in **java.util.stream**.

The **Collectors** class defines a number of static collector methods that you can use as-is. The two we will use are **toList( )** and **toSet( )**, shown here:

static <T> Collector<T, ?, List<T>> toList( )

static <T> Collector<T, ?, Set<T>> toSet( )

The **toList( )** method returns a collector that can be used to collect elements into a **List**. The **toSet( )** method returns a collector that can be used to collect elements into a **Set**. For example, to collect elements into a **List**, you can call **collect( )** like this:

```
collect(Collectors.toList())
```

The following program puts the preceding discussion into action. It reworks the example in the previous section so that it collects the names and phone numbers into a **List** and a **Set**.

```
// Use collect() to create a List and a Set from a stream.

import java.util.*;
import java.util.stream.*;

class NamePhoneEmail {
  String name;
  String phonenum;
  String email;

  NamePhoneEmail(String n, String p, String e) {
    name = n;
    phonenum = p;
    email = e;
  }
}

class NamePhone {
  String name;
  String phonenum;

  NamePhone(String n, String p) {
    name = n;
    phonenum = p;
  }
}

class StreamDemo7 {

  public static void main(String[] args) {

    // A list of names, phone numbers, and e-mail addresses.
    ArrayList<NamePhoneEmail> myList = new ArrayList<>( );

    myList.add(new NamePhoneEmail("Larry", "555-5555",
                                  "Larry@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("James", "555-4444",
                                  "James@HerbSchildt.com"));
    myList.add(new NamePhoneEmail("Mary", "555-3333",
                                  "Mary@HerbSchildt.com"));

    // Map just the names and phone numbers to a new stream.
    Stream<NamePhone> nameAndPhone = myList.stream().map(
                                       (a) -> new NamePhone(a.name,a.phonenum)
                                     );

    // Use collect to create a List of the names and phone numbers.
    List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

```
    System.out.println("Names and phone numbers in a List:");
    for(NamePhone e : npList)
      System.out.println(e.name + ": " + e.phonenum);

    // Obtain another mapping of the names and phone numbers.
    nameAndPhone = myList.stream().map(
                                    (a) -> new NamePhone(a.name,a.phonenum)
                                   );

    // Now, create a Set by use of collect().
    Set<NamePhone> npSet = nameAndPhone.collect(Collectors.toSet());

    System.out.println("\nNames and phone numbers in a Set:");
    for(NamePhone e : npSet)
      System.out.println(e.name + ": " + e.phonenum);
  }
}
```

The output is shown here:

```
Names and phone numbers in a List:
Larry: 555-5555
James: 555-4444
Mary: 555-3333

Names and phone numbers in a Set:
James: 555-4444
Larry: 555-5555
Mary: 555-3333
```

In the program, the following line collects the name and phone numbers into a **List** by using **toList( )**:

```
List<NamePhone> npList = nameAndPhone.collect(Collectors.toList());
```

After this line executes, the collection referred to by **npList** can be used like any other **List** collection. For example, it can be cycled through by using a for-each **for** loop, as shown in the next line:

```
for(NamePhone e : npList)
  System.out.println(e.name + ": " + e.phonenum);
```

The creation of a **Set** via **collect(Collectors.toSet( ))** works in the same way. The ability to move data from a collection to a stream, and then back to a collection again is a very powerful attribute of the stream API. It gives you the ability to operate on a collection through a stream, but then repackage it as a collection. Furthermore, the stream operations can, if appropriate, occur in parallel.

The version of **collect( )** used by the previous example is quite convenient, and often the one you want, but there is a second version that gives you more control over the collection process. It is shown here:

<R> R collect(Supplier<R> *target*, BiConsumer<R, ? super T> *accumulator*,
BiConsumer <R, R> *combiner*)

Here, *target* specifies how the object that holds the result is created. For example, to use a **LinkedList** as the result collection, you would specify its constructor. The *accumulator* function adds an element to the result and *combiner* combines two partial results. Thus, these functions work similarly to the way they do in **reduce( )**. For both, they must be stateless and non-interfering. They must also be associative.

Note that the *target* parameter is of type **Supplier**. It is a functional interface declared in **java.util.function**. It specifies only the **get( )** method, which has no parameters and, in this case, returns an object of type **R**. Thus, as it relates to **collect( )**, **get( )** returns a reference to a mutable storage object, such as a collection.

Note also that the types of *accumulator* and *combiner* are **BiConsumer**. This is a functional interface defined in **java.util.function**. It specifies the abstract method **accept( )** that is shown here:

void accept(T *obj*, U *obj2*)

This method performs some type of operation on *obj* and *obj2*. As it relates to *accumulator*, *obj* specifies the target collection, and *obj2* specifies the element to add to that collection. As it relates to *combiner*, *obj* and *obj2* specify two collections that will be combined.

Using the version of **collect( )** just described, you could use a **LinkedList** as the target in the preceding program, as shown here:

```
LinkedList<NamePhone> npList = nameAndPhone.collect(
                                () -> new LinkedList<>(),
                                (list, element) -> list.add(element),
                                (listA,listB ) -> listA.addAll(listB));
```

Notice that the first argument to **collect( )** is a lambda expression that returns a new **LinkedList**. The second argument uses the standard collection method **add( )** to add an element to the list. The third element uses **addAll( )** to combine two linked lists. As a point of interest, you can use any method defined by **LinkedList** to add an element to the list. For example, you could use **addFirst( )** to add elements to the start of the list, as shown here:

```
(list, element) -> list.addFirst(element)
```

As you may have guessed, it is not always necessary to specify a lambda expression for the arguments to **collect( )**. Often, method and/or constructor references will suffice. For example, again assuming the preceding program, this statement creates a **HashSet** that contains all of the elements in the **nameAndPhone** stream:

```
HashSet<NamePhone> npSet = nameAndPhone.collect(HashSet::new,
                                                HashSet::add,
                                                HashSet::addAll);
```

Notice that the first argument specifies the **HashSet** constructor reference. The second and third specify method references to **HashSet**'s **add( )** and **addAll( )** methods.

One last point: In the language of the stream API, the **collect( )** method performs what is called a *mutable reduction*. This is because the result of the reduction is a mutable (i.e., changeable) storage object, such as a collection.

# Iterators and Streams

Although a stream is not a data storage object, you can still use an iterator to cycle through its elements in much the same way as you would use an iterator to cycle through the elements of a collection. The stream API supports two types of iterators. The first is the traditional **Iterator**. The second is **Spliterator**, which was added by JDK 8. It provides significant advantages in certain situations when used with parallel streams.

## Use an Iterator with a Stream

As just mentioned, you can use an iterator with a stream in just the same way that you do with a collection. Iterators are discussed in Chapter 18, but a brief review will be useful here. Iterators are objects that implement the **Iterator** interface declared in **java.util**. Its two key methods are **hasNext( )** and **next( )**. If there is another element to iterate, **hasNext( )** returns **true**, and **false** otherwise. The **next( )** method returns the next element in the iteration.

---

**NOTE** JDK 8 adds additional iterator types that handle the primitive streams: **PrimitiveIterator**, **PrimitiveIterator.OfDouble**, **PrimitiveIterator.OfLong**, and **PrimitiveIterator.OfInt**. These iterators all extend the **Iterator** interface and work in the same general way as those based directly on **Iterator**.

---

To obtain an iterator to a stream, call **iterator( )** on the stream. The version used by **Stream** is shown here.

```
Iterator<T> iterator( )
```

Here, **T** specifies the element type. (The primitive streams return iterators of the appropriate primitive type.)

The following program shows how to iterate through the elements of a stream. In this case, the strings in an **ArrayList** are iterated, but the process is the same for any type of stream.

```
// Use an iterator with a stream.

import java.util.*;
import java.util.stream.*;

class StreamDemo8 {

  public static void main(String[] args) {

    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();
```

```
    // Obtain an iterator to the stream.
    Iterator<String> itr = myStream.iterator();

    // Iterate the elements in the stream.
    while(itr.hasNext())
      System.out.println(itr.next());
  }
}
```

The output is shown here:

```
Alpha
Beta
Gamma
Delta
Phi
Omega
```



## Use Spliterator

**Spliterator** offers an alternative to **Iterator**, especially when parallel processing is involved. In general, **Spliterator** is more sophisticated than **Iterator**, and a discussion of **Spliterator** is found in Chapter 18. However, it will be useful to review its key features here. **Spliterator** defines several methods, but we only need to use three. The first is **tryAdvance( )**. It performs an action on the next element and then advances the iterator. It is shown here:

boolean tryAdvance(Consumer<? super T> *action*)

Here, *action* specifies the action that is executed on the next element in the iteration. **tryAdvance( )** returns **true** if there is a next element. It returns **false** if no elements remain. As discussed earlier in this chapter, **Consumer** declares one method called **accept( )** that receives an element of type **T** as an argument and returns **void**.

Because **tryAdvance( )** returns **false** when there are no more elements to process, it makes the iteration loop construct very simple, for example:

```
while(splitItr.tryAdvance( // perform action here );
```

As long as **tryAdvance( )** returns **true**, the action is applied to the next element. When **tryAdvance( )** returns **false**, the iteration is complete. Notice how **tryAdvance( )** consolidates the purposes of **hasNext( )** and **next( )** provided by **Iterator** into a single method. This improves the efficiency of the iteration process.

The following version of the preceding program substitutes a **Spliterator** for the **Iterator**:

```
// Use a Spliterator.

import java.util.*;
import java.util.stream.*;

class StreamDemo9 {

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

```
    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();

    // Obtain a Spliterator.
    Spliterator<String> splitItr = myStream.spliterator();

    // Iterate the elements of the stream.
    while(splitItr.tryAdvance((n) -> System.out.println(n)));
  }
}
```

The output is the same as before.

In some cases, you might want to perform some action on each element collectively, rather than one at a time. To handle this type of situation, **Spliterator** provides the **forEachRemaining( )** method, shown here:

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

This method applies *action* to each unprocessed element and then returns. For example, assuming the preceding program, the following displays the strings remaining in the stream:

```
splitItr.forEachRemaining((n) -> System.out.println(n));
```

Notice how this method eliminates the need to provide a loop to cycle through the elements one at a time. This is another advantage of **Spliterator**.

One other **Spliterator** method of particular interest is **trySplit( )**. It splits the elements being iterated in two, returning a new **Spliterator** to one of the partitions. The other partition remains accessible by the original **Spliterator**. It is shown here:

Spliterator<T> trySplit( )

If it is not possible to split the invoking **Spliterator**, **null** is returned. Otherwise, a reference to the partition is returned. For example, here is another version of the preceding program that demonstrates **trySplit( )**:

```
// Demonstrate trySplit().

import java.util.*;
import java.util.stream.*;

class StreamDemo10 {

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

```
    // Create a list of Strings.
    ArrayList<String> myList = new ArrayList<>( );
    myList.add("Alpha");
    myList.add("Beta");
    myList.add("Gamma");
    myList.add("Delta");
    myList.add("Phi");
    myList.add("Omega");

    // Obtain a Stream to the array list.
    Stream<String> myStream = myList.stream();

    // Obtain a Spliterator.
    Spliterator<String> splitItr = myStream.spliterator();

    // Now, split the first iterator.
    Spliterator<String> splitItr2 = splitItr.trySplit();

     // If splitItr could be split, use splitItr2 first.
    if(splitItr2 != null) {
      System.out.println("Output from splitItr2: ");
      splitItr2.forEachRemaining((n) -> System.out.println(n));
    }

    // Now, use the splitItr.
    System.out.println("\nOutput from splitItr: ");
    splitItr.forEachRemaining((n) -> System.out.println(n));
  }
}
```

The output is shown here:

```
Output from splitItr2:
Alpha
Beta
Gamma

Output from splitItr:
Delta
Phi
Omega
```

Although splitting the **Spliterator** in this simple illustration is of no practical value, splitting can be of *great value* when parallel processing over large data sets. However, in many cases, it is better to use one of the other **Stream** methods in conjunction with a parallel stream, rather than manually handling these details with **Spliterator**. **Spliterator** is primarily for the cases in which none of the predefined methods seems appropriate.

## More to Explore in the Stream API

This chapter has discussed several key aspects of the stream API and introduced the techniques required to use them, but the stream API has much more to offer. To begin, here are a few of the other methods provided by **Stream** that you will find helpful:

- To determine if one or more elements in a stream satisfy a specified predicate, use **allMatch( )**, **anyMatch( )**, or **noneMatch( )**.
- To obtain the number of elements in the stream, call **count( )**.
- To obtain a stream that contains only unique elements, use **distinct( )**.
- To create a stream that contains a specified set of elements, use **of( )**.

One last point: the stream API is a powerful addition to Java. It is likely that it will be enhanced over time to include even more functionality. Therefore, a periodic perusal of its API documentation is advised.

CHAPTER

# 30 Regular Expressions and Other Packages

When Java was originally released, it included a set of eight packages, called the *core API*. Each subsequent release added to the API. Today, the Java API contains a very large number of packages. Many of the packages support areas of specialization that are beyond the scope of this book. However, several packages warrant an examination here. Four are **java.util.regex**, **java.lang.reflect**, **java.rmi**, and **java.text**. They support regular expression processing, reflection, Remote Method Invocation (RMI), and text formatting, respectively. The chapter ends by introducing the new date and time API in **java.time** and its subpackages.

The *regular expression* package lets you perform sophisticated pattern matching operations. This chapter provides an in-depth introduction to this package along with extensive examples. *Reflection* is the ability of software to analyze itself. It is an essential part of the Java Beans technology that is covered in Chapter 37. *Remote Method Invocation (RMI)* allows you to build Java applications that are distributed among several machines. This chapter provides a simple client/server example that uses RMI. The *text formatting* capabilities of **java.text** have many uses. The one examined here formats date and time strings. The new date and time API supplies an up-to-date approach to handling date and time.

## The Core Java API Packages

At the time of this writing, Table 30-1 lists all of the core API packages defined by Java (those in the **java** namespace) and summarizes their functions.

| Package | Primary Function |
|---|---|
| java.applet | Supports construction of applets. |
| java.awt | Provides capabilities for graphical user interfaces. |
| java.awt.color | Supports color spaces and profiles. |
| java.awt.datatransfer | Transfers data to and from the system clipboard. |

**Table 30-1** The Core Java API Packages

| Package | Primary Function |
|---|---|
| java.awt.dnd | Supports drag-and-drop operations. |
| java.awt.event | Handles events. |
| java.awt.font | Represents various types of fonts. |
| java.awt.geom | Allows you to work with geometric shapes. |
| java.awt.im | Allows input of Japanese, Chinese, and Korean characters to text editing components. |
| java.awt.im.spi | Supports alternative input devices. |
| java.awt.image | Processes images. |
| java.awt.image.renderable | Supports rendering-independent images. |
| java.awt.print | Supports general print capabilities. |
| java.beans | Allows you to build software components. |
| java.beans.beancontext | Provides an execution environment for Beans. |
| java.io | Inputs and outputs data. |
| java.lang | Provides core functionality. |
| java.lang.annotation | Supports annotations (metadata). |
| java.lang.instrument | Supports program instrumentation. |
| java.lang.invoke | Supports dynamic languages. |
| java.lang.management | Supports management of the execution environment. |
| java.lang.ref | Enables some interaction with the garbage collector. |
| java.lang.reflect | Analyzes code at run time. |
| java.math | Handles large integers and decimal numbers. |
| java.net | Supports networking. |
| java.nio | Top-level package for the NIO classes. Encapsulates buffers. |
| java.nio.channels | Encapsulates channels, which are used by the NIO system. |
| java.nio.channels.spi | Supports service providers for channels. |
| java.nio.charset | Encapsulates character sets. |
| java.nio.charset.spi | Supports service providers for character sets. |
| java.nio.file | Provides NIO support for files. |
| java.nio.file.attribute | Supports NIO file attributes. |
| java.nio.file.spi | Supports NIO service providers for files. |
| java.rmi | Provides remote method invocation. |
| java.rmi.activation | Activates persistent objects. |
| java.rmi.dgc | Manages distributed garbage collection. |
| java.rmi.registry | Maps names to remote object references. |
| java.rmi.server | Supports remote method invocation. |
| java.security | Handles certificates, keys, digests, signatures, and other security functions. |

**Table 30-1** The Core Java API Packages *(continued)*

| Package | Primary Function |
|---|---|
| java.security.acl | Manages access control lists. |
| java.security.cert | Parses and manages certificates. |
| java.security.interfaces | Defines interfaces for DSA (Digital Signature Algorithm) keys. |
| java.security.spec | Specifies keys and algorithm parameters. |
| java.sql | Communicates with a SQL (Structured Query Language) database. |
| java.text | Formats, searches, and manipulates text. |
| java.text.spi | Supports service providers for text formatting classes in **java.text**. |
| java.time | Primary support for the new date and time API. (Added by JDK 8.) |
| java.time.chrono | Supports alternative, non-Gregorian calendars. (Added by JDK 8.) |
| java.time.format | Supports date and time formatting. (Added by JDK 8.) |
| java.time.temporal | Supports extended date and time functionality. (Added by JDK 8.) |
| java.time.zone | Supports time zones. (Added by JDK 8.) |
| java.util | Contains common utilities. |
| java.util.concurrent | Supports the concurrent utilities. |
| java.util.concurrent.atomic | Supports atomic (that is, indivisible) operations on variables without the use of locks. |
| java.util.concurrent.locks | Supports synchronization locks. |
| java.util.function | Provides several functional interfaces. (Added by JDK 8.) |
| java.util.jar | Creates and reads JAR files. |
| java.util.logging | Supports logging of information related to a program's execution. |
| java.util.prefs | Encapsulates information relating to user preference. |
| java.util.regex | Supports regular expression processing. |
| java.util.spi | Supports service providers for the utility classes in **java.util**. |
| java.util.stream | Supports the new stream API. (Added by JDK 8.) |
| java.util.zip | Reads and writes compressed and uncompressed ZIP files. |

**Table 30-1** The Core Java API Packages *(continued)*



## Regular Expression Processing

The **java.util.regex** package supports regular expression processing. As the term is used here, a *regular expression* is a string of characters that describes a character sequence. This general description, called a *pattern*, can then be used to find matches in other character sequences. Regular expressions can specify wildcard characters, sets of characters, and various quantifiers. Thus, you can specify a regular expression that represents a general form that can match several different specific character sequences.

There are two classes that support regular expression processing: **Pattern** and **Matcher**. These classes work together. Use **Pattern** to define a regular expression. Match the pattern against another sequence using **Matcher**.

## Pattern

The **Pattern** class defines no constructors. Instead, a pattern is created by calling the **compile( )** factory method. One of its forms is shown here:

static Pattern compile(String *pattern*)

Here, *pattern* is the regular expression that you want to use. The **compile( )** method transforms the string in *pattern* into a pattern that can be used for pattern matching by the **Matcher** class. It returns a **Pattern** object that contains the pattern.

Once you have created a **Pattern** object, you will use it to create a **Matcher**. This is done by calling the **matcher( )** factory method defined by **Pattern**. It is shown here:

Matcher matcher(CharSequence *str*)

Here *str* is the character sequence that the pattern will be matched against. This is called the *input sequence*. **CharSequence** is an interface that defines a read-only set of characters. It is implemented by the **String** class, among others. Thus, you can pass a string to **matcher( )**.

## Matcher

The **Matcher** class has no constructors. Instead, you create a **Matcher** by calling the **matcher( )** factory method defined by **Pattern**, as just explained. Once you have created a **Matcher**, you will use its methods to perform various pattern matching operations.

The simplest pattern matching method is **matches( )**, which simply determines whether the character sequence matches the pattern. It is shown here:

boolean matches( )

It returns **true** if the sequence and the pattern match, and **false** otherwise. Understand that the entire sequence must match the pattern, not just a subsequence of it.

To determine if a subsequence of the input sequence matches the pattern, use **find( )**. One version is shown here:

boolean find( )

It returns **true** if there is a matching subsequence and **false** otherwise. This method can be called repeatedly, allowing it to find all matching subsequences. Each call to **find( )** begins where the previous one left off.

You can obtain a string containing the last matching sequence by calling **group( )**. One of its forms is shown here:

String group( )

The matching string is returned. If no match exists, then an **IllegalStateException** is thrown.

You can obtain the index within the input sequence of the current match by calling **start( )**. The index one past the end of the current match is obtained by calling **end( )**. The forms used in this chapter are shown here:

int start( )
int end( )

Both throw **IllegalStateException** if no match exists.

You can replace all occurrences of a matching sequence with another sequence by calling **replaceAll( )**, shown here:

String replaceAll(String *newStr*)

Here, *newStr* specifies the new character sequence that will replace the ones that match the pattern. The updated input sequence is returned as a string.

## Regular Expression Syntax

Before demonstrating **Pattern** and **Matcher**, it is necessary to explain how to construct a regular expression. Although no rule is complicated by itself, there are a large number of them, and a complete discussion is beyond the scope of this chapter. However, a few of the more commonly used constructs are described here.

In general, a regular expression is comprised of normal characters, character classes (sets of characters), wildcard characters, and quantifiers. A normal character is matched as-is. Thus, if a pattern consists of "xy", then the only input sequence that will match it is "xy". Characters such as newline and tab are specified using the standard escape sequences, which begin with a \. For example, a newline is specified by **\n**. In the language of regular expressions, a normal character is also called a *literal*.

A character class is a set of characters. A character class is specified by putting the characters in the class between brackets. For example, the class [wxyz] matches w, x, y, or z. To specify an inverted set, precede the characters with a **^**. For example, [^wxyz] matches any character except w, x, y, or z. You can specify a range of characters using a hyphen. For example, to specify a character class that will match the digits 1 through 9, use [1-9].

The wildcard character is the **.** (dot) and it matches any character. Thus, a pattern that consists of "." will match these (and other) input sequences: "A", "a", "x", and so on.

A quantifier determines how many times an expression is matched. The quantifiers are shown here:

| | |
|---|---|
| + | Match one or more. |
| * | Match zero or more. |
| ? | Match zero or one. |

For example, the pattern "x+" will match "x", "xx", and "xxx", among others.

One other point: In general, if you specify an invalid expression, a **PatternSyntaxException** will be thrown.

## Demonstrating Pattern Matching

The best way to understand how regular expression pattern matching operates is to work through some examples. The first, shown here, looks for a match with a literal pattern:

```
// A simple pattern matching demo.
import java.util.regex.*;

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

```
    Pattern pat;
    Matcher mat;
    boolean found;

    pat = Pattern.compile("Java");
    mat = pat.matcher("Java");
    found = mat.matches(); // check for a match

    System.out.println("Testing Java against Java.");
    if(found) System.out.println("Matches");
    else System.out.println("No Match");

    System.out.println();

    System.out.println("Testing Java against Java 8.");
    mat = pat.matcher("Java 8"); // create a new matcher

    found = mat.matches(); // check for a match

    if(found) System.out.println("Matches");
    else System.out.println("No Match");
  }
}
```

The output from the program is shown here:

```
Testing Java against Java.
Matches

Testing Java against Java 8.
No Match
```

Let's look closely at this program. The program begins by creating the pattern that contains the sequence "Java". Next, a **Matcher** is created for that pattern that has the input sequence "Java". Then, the **matches( )** method is called to determine if the input sequence matches the pattern. Because the sequence and the pattern are the same, **matches( )** returns **true**. Next, a new **Matcher** is created with the input sequence "Java 8" and **matches( )** is called again. In this case, the pattern and the input sequence differ, and no match is found. Remember, the **matches( )** function returns **true** only when the input sequence precisely matches the pattern. It will not return **true** just because a subsequence matches.

You can use **find( )** to determine if the input sequence contains a subsequence that matches the pattern. Consider the following program:

```
// Use find() to find a subsequence.
import java.util.regex.*;

class RegExpr2 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("Java");
    Matcher mat = pat.matcher("Java 8");

    System.out.println("Looking for Java in Java 8.");
```

```
    if(mat.find()) System.out.println("subsequence found");
    else System.out.println("No Match");
  }
}
```

The output is shown here:

```
Looking for Java in Java 8.
subsequence found
```

In this case, **find( )** finds the subsequence "Java".

The **find( )** method can be used to search the input sequence for repeated occurrences of the pattern because each call to **find( )** picks up where the previous one left off. For example, the following program finds two occurrences of the pattern "test":

```
// Use find() to find multiple subsequences.
import java.util.regex.*;

class RegExpr3 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("test");
    Matcher mat = pat.matcher("test 1 2 3 test");

    while(mat.find()) {
      System.out.println("test found at index " +
                         mat.start());
    }
  }
}
```

The output is shown here:

```
test found at index 0
test found at index 11
```

As the output shows, two matches were found. The program uses the **start( )** method to obtain the index of each match.

## Using Wildcards and Quantifiers

Although the preceding programs show the general technique for using **Pattern** and **Matcher**, they don't show their power. The real benefit of regular expression processing is not seen until wildcards and quantifiers are used. To begin, consider the following example that uses the + quantifier to match any arbitrarily long sequence of Ws:

```
// Use a quantifier.
import java.util.regex.*;

class RegExpr4 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("W+");
    Matcher mat = pat.matcher("W WW WWW");
```

```
    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output from the program is shown here:

```
Match: W
Match: WW
Match: WWW
```

As the output shows, the regular expression pattern "W+" matches any arbitrarily long sequence of Ws.

The next program uses a wildcard to create a pattern that will match any sequence that begins with *e* and ends with *d*. To do this, it uses the dot wildcard character along with the + quantifier.

```
// Use wildcard and quantifier.
import java.util.regex.*;

class RegExpr5 {
  public static void main(String args[]) {
    Pattern pat = Pattern.compile("e.+d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

You might be surprised by the output produced by the program, which is shown here:

```
Match: extend cup end
```

Only one match is found, and it is the longest sequence that begins with *e* and ends with *d*. You might have expected two matches: "extend" and "end". The reason that the longer sequence is found is that, by default, **find( )** matches the longest sequence that fits the pattern. This is called *greedy behavior*. You can specify *reluctant behavior* by adding the **?** quantifier to the pattern, as shown in this version of the program. It causes the shortest matching pattern to be obtained.

```
// Use the ? quantifier.
import java.util.regex.*;

class RegExpr6 {
  public static void main(String args[]) {
    // Use reluctant matching behavior.
    Pattern pat = Pattern.compile("e.+?d");
    Matcher mat = pat.matcher("extend cup end table");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output from the program is shown here:

```
Match: extend
Match: end
```

As the output shows, the pattern "e.+?d" will match the shortest sequence that begins with *e* and ends with *d*. Thus, two matches are found.

## Working with Classes of Characters

Sometimes you will want to match any sequence that contains one or more characters, in any order, that are part of a set of characters. For example, to match whole words, you want to match any sequence of the letters of the alphabet. One of the easiest ways to do this is to use a character class, which defines a set of characters. Recall that a character class is created by putting the characters you want to match between brackets. For example, to match the lowercase characters a through z, use **[a-z]**. The following program demonstrates this technique:

```
// Use a character class.
import java.util.regex.*;

class RegExpr7 {
  public static void main(String args[]) {
    // Match lowercase words.
    Pattern pat = Pattern.compile("[a-z]+");
    Matcher mat = pat.matcher("this is a test.");

    while(mat.find())
      System.out.println("Match: " + mat.group());
  }
}
```

The output is shown here:

```
Match: this
Match: is
Match: a
Match: test
```

## Using replaceAll( )

The **replaceAll( )** method supplied by **Matcher** lets you perform powerful search and replace operations that use regular expressions. For example, the following program replaces all occurrences of sequences that begin with "Jon" with "Eric":

```
// Use replaceAll().
import java.util.regex.*;

class RegExpr8 {
  public static void main(String args[]) {
    String str = "Jon Jonathan Frank Ken Todd";

    Pattern pat = Pattern.compile("Jon.*? ");
    Matcher mat = pat.matcher(str);
```

```
    System.out.println("Original sequence: " + str);

    str = mat.replaceAll("Eric ");

    System.out.println("Modified sequence: " + str);

  }
}
```

The output is shown here:

```
Original sequence: Jon Jonathan Frank Ken Todd
Modified sequence: Eric Eric Frank Ken Todd
```

Because the regular expression "Jon.*? " matches any string that begins with Jon followed by zero or more characters, ending in a space, it can be used to match and replace both Jon and Jonathan with the name Eric. Such a substitution is not easily accomplished without pattern matching capabilities.

## Using split( )

You can reduce an input sequence into its individual tokens by using the **split( )** method defined by **Pattern**. One form of the **split( )** method is shown here:

String[ ] split(CharSequence *str*)

It processes the input sequence passed in *str*, reducing it into tokens based on the delimiters specified by the pattern.

For example, the following program finds tokens that are separated by spaces, commas, periods, and exclamation points:

```
// Use split().
import java.util.regex.*;

class RegExpr9 {
  public static void main(String args[]) {

    // Match lowercase words.
    Pattern pat = Pattern.compile("[ ,.!]");

    String strs[] = pat.split("one two,alpha9 12!done.");

    for(int i=0; i < strs.length; i++)
      System.out.println("Next token: " + strs[i]);

  }
}
```

The output is shown here:

```
Next token: one
Next token: two
Next token: alpha9
```

```
Next token: 12
Next token: done
```

As the output shows, the input sequence is reduced to its individual tokens. Notice that the delimiters are not included.

### Two Pattern-Matching Options

Although the pattern-matching techniques described in the foregoing offer the greatest flexibility and power, there are two alternatives which you might find useful in some circumstances. If you only need to perform a one-time pattern match, you can use the **matches( )** method defined by **Pattern**. It is shown here:

static boolean matches(String *pattern*, CharSequence *str*)

It returns **true** if *pattern* matches *str* and **false** otherwise. This method automatically compiles *pattern* and then looks for a match. If you will be using the same pattern repeatedly, then using **matches( )** is less efficient than compiling the pattern and using the pattern-matching methods defined by **Matcher**, as described previously.

You can also perform a pattern match by using the **matches( )** method implemented by **String**. It is shown here:

boolean matches(String *pattern*)

If the invoking string matches the regular expression in *pattern*, then **matches( )** returns **true**. Otherwise, it returns **false**.

### Exploring Regular Expressions

The overview of regular expressions presented in this section only hints at their power. Since text parsing, manipulation, and tokenization are a large part of programming, you will likely find Java's regular expression subsystem a powerful tool that you can use to your advantage. It is, therefore, wise to explore the capabilities of regular expressions. Experiment with several different types of patterns and input sequences. Once you understand how regular expression pattern matching works, you will find it useful in many of your programming endeavors.

## Reflection

Reflection is the ability of software to analyze itself. This is provided by the **java.lang.reflect** package and elements in **Class**. Reflection is an important capability, especially when using components called Java Beans. It allows you to analyze a software component and describe its capabilities dynamically, at run time rather than at compile time. For example, by using reflection, you can determine what methods, constructors, and fields a class supports. Reflection was introduced in Chapter 12. It is examined further here.

The package **java.lang.reflect** includes several interfaces. Of special interest is **Member**, which defines methods that allow you to get information about a field, constructor, or method of a class. There are also ten classes in this package. These are listed in Table 30-2.

| Class | Primary Function |
|---|---|
| AccessibleObject | Allows you to bypass the default access control checks. |
| Array | Allows you to dynamically create and manipulate arrays. |
| Constructor | Provides information about a constructor. |
| Executable | An abstract superclass extended by **Method** and **Constructor**. (Added by JDK 8.) |
| Field | Provides information about a field. |
| Method | Provides information about a method. |
| Modifier | Provides information about class and member access modifiers. |
| Parameter | Provides information about parameters. (Added by JDK 8.) |
| Proxy | Supports dynamic proxy classes. |
| ReflectPermission | Allows reflection of private or protected members of a class. |

**Table 30-2** Classes Defined in **java.lang.reflect**

The following application illustrates a simple use of the Java reflection capabilities. It prints the constructors, fields, and methods of the class **java.awt.Dimension**. The program begins by using the **forName( )** method of **Class** to get a class object for **java.awt.Dimension**. Once this is obtained, **getConstructors( )**, **getFields( )**, and **getMethods( )** are used to analyze this class object. They return arrays of **Constructor**, **Field**, and **Method** objects that provide the information about the object. The **Constructor**, **Field**, and **Method** classes define several methods that can be used to obtain information about an object. You will want to explore these on your own. However, each supports the **toString( )** method. Therefore, using **Constructor**, **Field**, and **Method** objects as arguments to the **println( )** method is straightforward, as shown in the program.

```
// Demonstrate reflection.
import java.lang.reflect.*;
public class ReflectionDemo1 {
  public static void main(String args[]) {
    try {
      Class<?> c = Class.forName("java.awt.Dimension");
      System.out.println("Constructors:");
      Constructor<?> constructors[] = c.getConstructors();
      for(int i = 0; i < constructors.length; i++) {
        System.out.println(" " + constructors[i]);
      }

      System.out.println("Fields:");
      Field fields[] = c.getFields();
      for(int i = 0; i < fields.length; i++) {
        System.out.println(" " + fields[i]);
      }

      System.out.println("Methods:");
      Method methods[] = c.getMethods();
```

```
      for(int i = 0; i < methods.length; i++) {
        System.out.println(" " + methods[i]);
      }
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

Here is the output from this program. (The precise order may differ slightly from that shown.)

```
Constructors:
 public java.awt.Dimension(int,int)
 public java.awt.Dimension()
 public java.awt.Dimension(java.awt.Dimension)
Fields:
 public int java.awt.Dimension.width
 public int java.awt.Dimension.height
Methods:
 public int java.awt.Dimension.hashCode()
 public boolean java.awt.Dimension.equals(java.lang.Object)
 public java.lang.String java.awt.Dimension.toString()
 public java.awt.Dimension java.awt.Dimension.getSize()
 public void java.awt.Dimension.setSize(double,double)
 public void java.awt.Dimension.setSize(java.awt.Dimension)
 public void java.awt.Dimension.setSize(int,int)
 public double java.awt.Dimension.getHeight()
 public double java.awt.Dimension.getWidth()
 public java.lang.Object java.awt.geom.Dimension2D.clone()
 public void java.awt.geom.
             Dimension2D.setSize(java.awt.geom.Dimension2D)
 public final native java.lang.Class java.lang.Object.getClass()
 public final native void java.lang.Object.wait(long)
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait()
   throws java.lang.InterruptedException
 public final void java.lang.Object.wait(long,int)
   throws java.lang.InterruptedException
 public final native void java.lang.Object.notify()
 public final native void java.lang.Object.notifyAll()
```

The next example uses Java's reflection capabilities to obtain the public methods of a class. The program begins by instantiating class **A**. The **getClass( )** method is applied to this object reference, and it returns the **Class** object for class **A**. The **getDeclaredMethods( )** method returns an array of **Method** objects that describe only the methods declared by this class. Methods inherited from superclasses such as **Object** are not included.

Each element of the **methods** array is then processed. The **getModifiers( )** method returns an **int** containing flags that describe which modifiers apply for this element. The **Modifier** class provides a set of **is*X*** methods, shown in Table 30-3, that can be used to examine this value. For example, the static method **isPublic( )** returns **true** if its argument

| Method | Description |
|---|---|
| static boolean isAbstract(int *val*) | Returns **true** if *val* has the **abstract** flag set and **false** otherwise. |
| static boolean isFinal(int *val*) | Returns **true** if *val* has the **final** flag set and **false** otherwise. |
| static boolean isInterface(int *val*) | Returns **true** if *val* has the **interface** flag set and **false** otherwise. |
| static boolean isNative(int *val*) | Returns **true** if *val* has the **native** flag set and **false** otherwise. |
| static boolean isPrivate(int *val*) | Returns **true** if *val* has the **private** flag set and **false** otherwise. |
| static boolean isProtected(int *val*) | Returns **true** if *val* has the **protected** flag set and **false** otherwise. |
| static boolean isPublic(int *val*) | Returns **true** if *val* has the **public** flag set and **false** otherwise. |
| static boolean isStatic(int *val*) | Returns **true** if *val* has the **static** flag set and **false** otherwise. |
| static boolean isStrict(int *val*) | Returns **true** if *val* has the **strict** flag set and **false** otherwise. |
| static boolean isSynchronized(int *val*) | Returns **true** if *val* has the **synchronized** flag set and **false** otherwise. |
| static boolean isTransient(int *val*) | Returns **true** if *val* has the **transient** flag set and **false** otherwise. |
| static boolean isVolatile(int *val*) | Returns **true** if *val* has the **volatile** flag set and **false** otherwise. |

**Table 30-3** The "is" Methods Defined by **Modifier** That Determine Modifiers

includes the **public** modifier. Otherwise, it returns **false**. In the following program, if the method supports public access, its name is obtained by the **getName( )** method and is then printed.

```
// Show public methods.
import java.lang.reflect.*;
public class ReflectionDemo2 {
  public static void main(String args[]) {

    try {
      A a = new A();
      Class<?> c = a.getClass();
      System.out.println("Public Methods:");
      Method methods[] = c.getDeclaredMethods();
      for(int i = 0; i < methods.length; i++) {
        int modifiers = methods[i].getModifiers();
        if(Modifier.isPublic(modifiers)) {
```

```
          System.out.println(" " + methods[i].getName());
        }
      }
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}

class A {
  public void a1() {
  }
  public void a2() {
  }
  protected void a3() {
  }
  private void a4() {
  }
}
```

Here is the output from this program:

```
Public Methods:
 a1
 a2
```

**Modifier** also includes a set of static methods that return the type of modifiers that can be applied to a specific type of program element. These methods are

static int classModifiers( )

static int constructorModifiers( )

static int fieldModifiers( )

static int interfaceModifiers( )

static int methodModifiers( )

static int parameterModifiers( ) (Added by JDK 8.)

For example, **methodModifiers( )** returns the modifiers that can be applied to a method. Each method returns flags, packed into an **int**, that indicate which modifiers are legal. The modifier values are defined by constants in **Modifier**, which include **PROTECTED**, **PUBLIC**, **PRIVATE**, **STATIC**, **FINAL**, and so on.

## Remote Method Invocation (RMI)

Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows you to build distributed applications. While a complete discussion of RMI is outside the scope of this book, the following simplified example describes the basic principles involved.

## A Simple Client/Server Application Using RMI

This section provides step-by-step directions for building a simple client/server application by using RMI. The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

### Step One: Enter and Compile the Source Code

This application uses four source files. The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members. Its purpose is simply to indicate that an interface uses remote methods. All remote methods can throw a **RemoteException**.

```
import java.rmi.*;

public interface AddServerIntf extends Remote {
  double add(double d1, double d2) throws RemoteException;
}
```

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add( )** method is straightforward. Remote objects typically extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

```
import java.rmi.*;
import java.rmi.server.*;

public class AddServerImpl extends UnicastRemoteObject
  implements AddServerIntf {

  public AddServerImpl() throws RemoteException {
  }
  public double add(double d1, double d2) throws RemoteException {
    return d1 + d2;
  }
}
```

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the **rebind( )** method of the **Naming** class (found in **java.rmi**). That method associates a name with an object reference. The first argument to the **rebind( )** method is a string that names the server as "AddServer". Its second argument is a reference to an instance of **AddServerImpl**.

```
import java.net.*;
import java.rmi.*;

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

```
    try {
      AddServerImpl addServerImpl = new AddServerImpl();
      Naming.rebind("AddServer", addServerImpl);
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command-line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

The application begins by forming a string that follows the URL syntax. This URL uses the **rmi** protocol. The string includes the IP address or name of the server and the string "AddServer". The program then invokes the **lookup( )** method of the **Naming** class. This method accepts one argument, the **rmi** URL, and returns a reference to an object of type **AddServerIntf**. All remote method invocations can then be directed to this object.

The program continues by displaying its arguments and then invokes the remote **add( )** method. The sum is returned from this method and is then printed.

```
import java.rmi.*;

public class AddClient {
  public static void main(String args[]) {
    try {
      String addServerURL = "rmi://" + args[0] + "/AddServer";
      AddServerIntf addServerIntf =
                    (AddServerIntf)Naming.lookup(addServerURL);
      System.out.println("The first number is: " + args[1]);
      double d1 = Double.valueOf(args[1]).doubleValue();
      System.out.println("The second number is: " + args[2]);

      double d2 = Double.valueOf(args[2]).doubleValue();
      System.out.println("The sum is: " + addServerIntf.add(d1, d2));
    }
    catch(Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

After you enter all the code, use **javac** to compile the four source files that you created.

## Step Two: Manually Generate a Stub if Required

In the context of RMI, a *stub* is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub. The stub works with the other parts of the RMI system to formulate a request that is sent to the remote machine.

A remote method may accept arguments that are simple types or objects. In the latter case, the object may have references to other objects. All of this information must be sent to

the remote machine. That is, an object passed as an argument to a remote method call must be serialized and sent to the remote machine. Recall from Chapter 20 that the serialization facilities also recursively process all referenced objects.

If a response must be returned to the client, the process works in reverse. Note that the serialization and deserialization facilities are also used if objects are returned to a client.

Prior to Java 5, stubs needed to be built manually by using **rmic**. This step is not required for modern versions of Java. However, if you are working in a legacy environment, then you can use the **rmic** compiler, as shown here, to build a stub:

```
rmic AddServerImpl
```

This command generates the file **AddServerImpl_Stub.class**. When using **rmic**, be sure that **CLASSPATH** is set to include the current directory.

### Step Three: Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl_Stub.class** (if needed), and **AddServerIntf.class** to a directory on the client machine. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Stub.class** (if needed), and **AddServer.class** to a directory on the server machine.

---

**NOTE** RMI has techniques for dynamic class loading, but they are not used by the example at hand. Instead, all of the files that are used by the client and server applications must be installed manually on those machines.

### Step Four: Start the RMI Registry on the Server Machine

The JDK provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. First, check that the **CLASSPATH** environment variable includes the directory in which your files are located. Then, start the RMI Registry from the command line, as shown here:

```
start rmiregistry
```

When this command returns, you should see that a new window has been created. You need to leave this window open until you are done experimenting with the RMI example.

### Step Five: Start the Server

The server code is started from the command line, as shown here:

```
java AddServer
```

Recall that the **AddServer** code instantiates **AddServerImpl** and registers that object with the name "AddServer".

### Step Six: Start the Client

The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

In the first line, the name of the server is provided. The second line uses its IP address (11.12.13.14).

You can try this example without actually having a remote server. To do so, simply install all of the programs on the same machine, start **rmiregistry**, start **AddServer**, and then execute **AddClient** using this command line:

```
java AddClient 127.0.0.1 8 9
```

Here, the address 127.0.0.1 is the "loop back" address for the local machine. Using this address allows you to exercise the entire RMI mechanism without actually having to install the server on a remote computer. (If you are using a firewall, then this approach may not work.)

In either case, sample output from this program is shown here:

```
The first number is: 8
The second number is: 9
The sum is: 17.0
```

**NOTE** When working with RMI in the real world, it may be necessary for the server to install a security manager.



# Formatting Date and Time with java.text

The package **java.text** allows you to format, parse, search, and manipulate text. This section examines two of its most commonly used classes: those that format date and time information. However, it is important to state at the outset that the new date and time API described later in this chapter offers a modern approach to handling date and time that also supports formatting. Of course, legacy code will continue to use the classes shown here for some time.

## DateFormat Class

**DateFormat** is an abstract class that provides the ability to format and parse dates and times. The **getDateInstance( )** method returns an instance of **DateFormat** that can format date information. It is available in these forms:

```
static final DateFormat getDateInstance( )
static final DateFormat getDateInstance(int style)
static final DateFormat getDateInstance(int style, Locale locale)
```

The argument *style* is one of the following values: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG**, or **FULL**. These are **int** constants defined by **DateFormat**. They cause different details about the date to be presented. The argument *locale* is one of the static references defined by **Locale** (refer to Chapter 19 for details). If the *style* and/or *locale* is not specified, defaults are used.

One of the most commonly used methods in this class is **format( )**. It has several overloaded forms, one of which is shown here:

```
final String format(Date d)
```

The argument is a **Date** object that is to be displayed. The method returns a string containing the formatted information.

The following listing illustrates how to format date information. It begins by creating a **Date** object. This captures the current date and time information. Then it outputs the date information by using different styles and locales.

```
// Demonstrate date formats.
import java.text.*;
import java.util.*;

public class DateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getDateInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japan: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.KOREA);
    System.out.println("Korea: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.LONG, Locale.UK);
    System.out.println("United Kingdom: " + df.format(date));

    df = DateFormat.getDateInstance(DateFormat.FULL, Locale.US);
    System.out.println("United States: " + df.format(date));
  }
}
```

Sample output from this program is shown here:

```
Japan: 14/01/01
Korea: 2014. 1. 1
United Kingdom: 01 January 2014
United States: Wednesday, January 1, 2014
```

The **getTimeInstance( )** method returns an instance of **DateFormat** that can format time information. It is available in these versions:

```
static final DateFormat getTimeInstance( )
static final DateFormat getTimeInstance(int style)
static final DateFormat getTimeInstance(int style, Locale locale)
```

The argument *style* is one of the following values: **DEFAULT**, **SHORT**, **MEDIUM**, **LONG**, or **FULL**. These are **int** constants defined by **DateFormat**. They cause different details about the time to be presented. The argument *locale* is one of the static references defined by **Locale**. If the *style* and/or *locale* is not specified, defaults are used.

The following listing illustrates how to format time information. It begins by creating a **Date** object. This captures the current date and time information. Then it outputs the time information by using different styles and locales.

```
// Demonstrate time formats.
import java.text.*;
import java.util.*;
```

```
public class TimeFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    DateFormat df;

    df = DateFormat.getTimeInstance(DateFormat.SHORT, Locale.JAPAN);
    System.out.println("Japan: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.LONG, Locale.UK);
    System.out.println("United Kingdom: " + df.format(date));

    df = DateFormat.getTimeInstance(DateFormat.FULL, Locale.CANADA);
    System.out.println("Canada: " + df.format(date));
  }
}
```



Sample output from this program is shown here:

```
Japan: 13:06
United Kingdom: 13:06:53 CST
Canada: 1:06:53 o'clock PM CST
```

The **DateFormat** class also has a **getDateTimeInstance( )** method that can format both date and time information. You may wish to experiment with it on your own.

## SimpleDateFormat Class

**SimpleDateFormat** is a concrete subclass of **DateFormat**. It allows you to define your own formatting patterns that are used to display date and time information.

One of its constructors is shown here:

SimpleDateFormat(String *formatString*)

The argument *formatString* describes how date and time information is displayed. An example of its use is given here:

```
SimpleDateFormat sdf = SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
```

The symbols used in the formatting string determine the information that is displayed. Table 30-4 lists these symbols and gives a description of each.

In most cases, the number of times a symbol is repeated determines how that data is presented. Text information is displayed in an abbreviated form if the pattern letter is repeated less than four times. Otherwise, the unabbreviated form is used. For example, a zzzz pattern can display Pacific Daylight Time, and a zzz pattern can display PDT.

For numbers, the number of times a pattern letter is repeated determines how many digits are presented. For example, hh:mm:ss can present 01:51:15, but h:m:s displays the same time value as 1:51:15.

Finally, M or MM causes the month to be displayed as one or two digits. However, three or more repetitions of M cause the month to be displayed as a text string.

| Symbol | Description |
|---|---|
| a | AM or PM |
| d | Day of month (1–31) |
| h | Hour in AM/PM (1–12) |
| k | Hour in day (1–24) |
| m | Minute in hour (0–59) |
| s | Second in minute (0–59) |
| u | Day of week, with Monday being 1 |
| w | Week of year (1–52) |
| y | Year |
| z | Time zone |
| D | Day of year (1–366) |
| E | Day of week (for example, Thursday) |
| F | Day of week in month |
| G | Era (for example, AD or BC) |
| H | Hour in day (0–23) |
| K | Hour in AM/PM (0–11) |
| L | Month |
| M | Month |
| S | Millisecond in second |
| W | Week of month (1–5) |
| X | Time zone in ISO 8601 format |
| Y | Week year |
| Z | Time zone in RFC 822 format |

**Table 30-4** Formatting String Symbols for **SimpleDateFormat**

The following program shows how this class is used:

```
// Demonstrate SimpleDateFormat.
import java.text.*;
import java.util.*;

public class SimpleDateFormatDemo {
  public static void main(String args[]) {
    Date date = new Date();
    SimpleDateFormat sdf;
    sdf = new SimpleDateFormat("hh:mm:ss");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("dd MMM yyyy hh:mm:ss zzz");
    System.out.println(sdf.format(date));
    sdf = new SimpleDateFormat("E MMM dd yyyy");
```

```
    System.out.println(sdf.format(date));
  }
}
```

Sample output from this program is shown here:

```
01:30:51
01 Jan 2014 01:30:51 CST
Wed Jan 01 2014
```



# The Time and Date API Added by JDK 8

In Chapter 19, Java's long-standing approach to handling date and time through the use of classes such as **Calendar** and **GregorianCalendar** was discussed. At the time of this writing, this traditional approach is still in widespread use and is something that all Java programmers need to be familiar with. However, with the release of JDK 8, Java now includes another approach to handling time and date. This new approach is defined in the following packages:

| Package | Description |
|---|---|
| java.time | Provides top-level classes that support time and date. |
| java.time.chrono | Supports alternative, non-Gregorian calendars. |
| java.time.format | Supports time and date formatting. |
| java.time.temporal | Supports extended date and time functionality. |
| java.time.zone | Supports time zones. |

These new packages define a large number of classes, interfaces, and enumerations that provide extensive, finely grained support for time and date operations. Because of the number of elements that comprise the new time and date API, it can seem fairly intimidating at first. However, it is well organized and logically structured. Its size reflects the detail of control and flexibility that it provides. Although it is far beyond the scope of this book to examine each element in this extensive API, we will look at several of its main classes. As you will see, these classes are sufficient for many uses.

## Time and Date Fundamentals

In **java.time** are defined several top-level classes that give you easy access to the time and date. Three of these are **LocalDate**, **LocalTime**, and **LocalDateTime**. As their names suggest, they encapsulate the local date, time, and date and time. Using these classes, it is easy to obtain the current date and time, format the date and time, and compare dates and times, among other operations.

**LocalDate** encapsulates a date that uses the default Gregorian calendar as specified by ISO 8601. **LocalTime** encapsulates a time, as specified by ISO 8601. **LocalDateTime** encapsulates both date and time. These classes contain a large number of methods that give you access to the date and time components, allow you to compare dates and times, add or subtract date or time components, and so on. Because a common naming convention for methods is employed, once you know how to use one of these classes, the others are easy to master.

**LocalDate**, **LocalTime**, and **LocalDateTime** do not define public constructors. Rather, to obtain an instance, you will use a factory method. One very convenient method is **now( )**, which is defined for all three classes. It returns the current date and/or time of the system. Each class defines several versions, but we will use its simplest form. Here is the version we will use as defined by **LocalDate**:

```
static LocalDate now( )
```

The version for **LocalTime** is shown here:

```
static LocalTime now( )
```

The version for **LocalDateTime** is shown here:

```
static LocalDateTime now( )
```

As you can see, in each case, an appropriate object is returned. The object returned by **now( )** can be displayed in its default, human-readable form by use of a **println( )** statement, for example. However, it is also possible to take full control over the formatting of date and time.

The following program uses **LocalDate** and **LocalTime** to obtain the current date and time and then displays them. Notice how **now( )** is called to retrieve the current date and time.

```
// A simple example of LocalDate and LocalTime.
import java.time.*;

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

    LocalDate curDate = LocalDate.now();
    System.out.println(curDate);

    LocalTime curTime = LocalTime.now();
    System.out.println(curTime);
  }
}
```

Sample output is shown here:

```
2014-01-01
14:03:41.436
```

The output reflects the default format that is given to the date and time. (The next section shows how to specify a different format.)

Because the preceding program displays both the current date and the current time, it could have been more easily written using the **LocalDateTime** class. In this approach, only a single instance needs to be created and only a single call to **now( )** is required, as shown here:

```
LocalDateTime curDateTime = LocalDateTime.now();
System.out.println(curDateTime);
```

Using this approach, the default output includes both date and time. Here is a sample:

```
2014-01-01T14:04:56.799
```

One other point: from a **LocalDateTime** instance, it is possible to obtain a reference to the date or time component by using the **toLocalDate( )** and **toLocalTime( )** methods, shown here:

LocalDate toLocalDate( )

LocalTime toLocalTime( )

Each returns a reference to the indicated element.



## Formatting Date and Time

Although the default formats shown in the preceding examples will be adequate for some uses, often you will want to specify a different format. Fortunately, this is easy to do because **LocalDate**, **LocalTime**, and **LocalDateTime** all provide the **format( )** method, shown here:

String format(DateTimeFormatter *fmtr*)

Here, *fmtr* specifies the instance of **DateTimeFormatter** that will provide the format.

**DateTimeFormatter** is packaged in **java.time.format**. To obtain a **DateTimeFormatter** instance, you will typically use one of its factory methods. Three are shown here:

static DateTimeFormatter ofLocalizedDate(FormatStyle *fmtDate*)

static DateTimeFormatter ofLocalizedTime(FormatStyle *fmtTime*)

static DateTimeFormatter ofLocalizedDateTime(FormatStyle *fmtDate*,
FormatStyle *fmtTime*)

Of course, the type of **DateTimeFormatter** that you create will be based on the type of object it will be operating on. For example, if you want to format the date in a **LocalDate** instance, then use **ofLocalizedDate( )**. The specific format is specified by the **FormatStyle** parameter.

**FormatStyle** is an enumeration that is packaged in **java.time.format**. It defines the following constants:

FULL

LONG

MEDIUM

SHORT

These specify the level of detail that will be displayed. (Thus, this form of **DateTimeFormatter** works similarly to **java.text.DateFormat**, described earlier in this chapter.)

Here is an example that uses **DateTimeFormatter** to display the current date and time:

```
// Demonstrate DateTimeFormatter.
import java.time.*;
import java.time.format.*;

class DateTimeDemo2 {
  public static void main(String args[]) {

    LocalDate curDate = LocalDate.now();
    System.out.println(curDate.format(
           DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL)));

    LocalTime curTime = LocalTime.now();
    System.out.println(curTime.format(
           DateTimeFormatter.ofLocalizedTime(FormatStyle.SHORT)));
  }
}
```

Sample output is shown here:

```
Wednesday, January 1, 2014
2:16 PM
```

In some situations, you may want a format different from the ones you can specify by use of **FormatStyle**. One way to accomplish this is to use a predefined formatter, such as **ISO_DATE** or **ISO_TIME**, provided by **DateTimeFormatter**. Another way is to create a custom format by specifying a pattern. To do this, you can use the **ofPattern( )** factory method of **DateTimeFormatter**. One version is shown here:

static DateTimeFormatter ofPattern(String *fmtPattern*)

Here, *fmtPattern* specifies a string that contains the date and time pattern that you want. It returns a **DateTimeFormatter** that will format according to that pattern. The default locale is used.

In general, a pattern consists of format specifiers, called *pattern letters*. A pattern letter will be replaced by the date or time component that it specifies. The full list of pattern letters is shown in the API documentation for **ofPattern( )**. Here is a sampling. Note that the pattern letters are case-sensitive.

| | |
|---|---|
| a | AM/PM indicator |
| d | Day in month |
| E | Day in week |
| h | Hour, 12-hour clock |
| H | Hour, 24-hour clock |
| M | Month |
| m | Minutes |
| s | Seconds |
| y | Year |

In general, the precise output that you see will be determined by how many times a pattern letter is repeated. (Thus, **DateTimeFormatter** works a bit like **java.text.SimpleDateFormat**, described earlier in this chapter.) For example, assuming that the month is April, the patterns:

```
M MM MMM MMMM
```

produce the following formatted output:

```
4 04 Apr April
```

Frankly, experimentation is the best way to understand what each pattern letter does and how various repetitions affect the output.

When you want to output a pattern letter as text, enclose the text between single quotation marks. In general, it is a good idea to enclose all non-pattern characters within single quotation marks to avoid problems if the set of pattern letters changes in subsequent versions of Java.

The following program demonstrates the use of a date and time pattern:

```
// Create a custom date and time format.
import java.time.*;
import java.time.format.*;

class DateTimeDemo3 {
  public static void main(String args[]) {

    LocalDateTime curDateTime = LocalDateTime.now();
    System.out.println(curDateTime.format(
               DateTimeFormatter.ofPattern("MMMM d',' yyyy h':'mm a")));
  }
}
```

Sample output is shown here:

```
January 1, 2014 2:22 PM
```

One other point about creating custom date and time output: **LocalDate**, **LocalTime**, and **LocalDateTime** define methods that let you obtain various date and time components. For example, **getHour( )** returns the hour as an **int**; **getMonth( )** returns the month in the form of a **Month** enumeration value; and **getYear( )** returns the year as an **int**. Using these, and other methods, you can manually construct output. You can also use these values for other purposes, such as when creating specialized timers.

## Parsing Date and Time Strings

**LocalDate**, **LocalTime**, and **LocalDateTime** provide the ability to parse date and/or time strings. To do this, call **parse( )** on an instance of one of those classes. It has two forms. The first uses the default formatter that parses the date and/or time formatted in the standard ISO fashion, such as 03:31 for time and 2014-08-02 for date. The form of this version of

**parse( )** for **LocalDateTime** is shown here. (Its form for the other classes is similar except for the type of object returned.)

```
static LocalDateTime parse(CharSequence dateTimeStr)
```

Here, *dateTimeStr* is a string that contains the date and time in the proper format. If the format is invalid, an error will result.

If you want to parse a date and/or time string that is in a format other than ISO format, you can use a second form of **parse( )** that lets you specify your own formatter. The version specified by **LocalDateTime** is shown next. (The other classes provide a similar form except for the return type.)

```
static LocalDateTime parse(CharSequence dateTimeStr,
                           DateTimeFormatter dateTimeFmtr)
```

Here, *dateTimeFmtr* specifies the formatter that you want to use.

Here is a simple example that parses a date and time string by use of a custom formatter:

```
// Parse a date and time.
import java.time.*;
import java.time.format.*;

class DateTimeDemo4 {
  public static void main(String args[]) {

    // Obtain a LocalDateTime object by parsing a date and time string.
    LocalDateTime curDateTime =
         LocalDateTime.parse("June 21, 2014 12:01 AM",
                  DateTimeFormatter.ofPattern("MMMM d',' yyyy hh':'mm a"));

     // Now, display the parsed date and time.
    System.out.println(curDateTime.format(
               DateTimeFormatter.ofPattern("MMMM d',' yyyy h':'mm a")));
  }
}
```

Sample output is shown here:

```
June 21, 2014 12:01 AM
```

## Other Things to Explore in java.time

Although you will want to explore all of the date and time packages, a good place to start is with **java.time**. It contains a great deal of functionality that you may find useful. Begin by examining the methods defined by **LocalDate**, **LocalTime**, and **LocalDateTime**. Each has methods that let you add or subtract dates and/or times, adjust dates and/or times by a given component, compare dates and/or times, and create instances based on date and/or time components, among others. Other classes in **java.time** that you may find of particular interest include **Instant**, **Duration**, and **Period**. **Instant** encapsulates an instant in time. **Duration** encapsulates a length of time. **Period** encapsulates a length of date.

PART

# III

# Introducing GUI Programming with Swing

This page has been intentionally left blank