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*) throws FileNotFoundException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(File *from*, String *charset*) 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*) throws IOException	Creates a **Scanner** that uses the file specified by *from* as a source for input.
Scanner(Path *from*, String *charset*) 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*, 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*) **Description**: Acts on *tVal* and *uVal*.
BiFunction<T, U, R>	R apply(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal* and returns the result.
BinaryOperator<T>	T apply(T *val1*, T *val2*) **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*) **Description**: Returns **true** if *tVal* and *uVal* satisfy the condition defined by **test( )** and **false** otherwise.
BooleanSupplier	boolean getAsBoolean( ) **Description**: Returns a **boolean** value.
Consumer<T>	void accept(T *val*) **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*) **Description**: Acts on two **double** values and returns a **double** result.
DoubleConsumer	void accept(double *val*) **Description**: Acts on *val*.
DoubleFunction<R>	R apply(double *val*) **Description**: Acts on a **double** value and returns the result.
DoublePredicate	boolean test(double *val*) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.
DoubleSupplier	double getAsDouble( ) **Description**: Returns a **double** result.
DoubleToIntFunction	int applyAsInt(double *val*) **Description**: Acts on a **double** value and returns the result as an **int**.
DoubleToLongFunction	long applyAsLong(double *val*) **Description**: Acts on a **double** value and returns the result as a **long**.
DoubleUnaryOperator	double applyAsDouble(double *val*) **Description**: Acts on a **double** and returns a **double** result.
Function<T, R>	R apply(T *val*) **Description**: Acts on *val* and returns the result.
IntBinaryOperator	int applyAsInt(int *val1*, int *val2*) **Description**: Acts on two **int** values and returns an **int** result.
IntConsumer	int accept(int *val*) **Description**: Acts on *val*.
IntFunction<R>	R apply(int *val*) **Description**: Acts on an **int** value and returns the result.
IntPredicate	boolean test(int *val*) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.
IntSupplier	int getAsInt( ) **Description**: Returns an **int** result.
IntToDoubleFunction	double applyAsDouble(int *val*) **Description**: Acts on an **int** value and returns the result as a **double**.
IntToLongFunction	long applyAsLong(int *val*) **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*) **Description**: Acts on an **int** and returns an **int** result.
LongBinaryOperator	long applyAsLong(long *val1*, long *val2*) **Description**: Acts on two **long** values and returns a **long** result.
LongConsumer	void accept(long *val*) **Description**: Acts on *val*.
LongFunction<R>	R apply(long *val*) **Description**: Acts on a **long** value and returns the result.
LongPredicate	boolean test(long *val*) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.
LongSupplier	long getAsLong( ) **Description**: Returns a **long** result.
LongToDoubleFunction	double applyAsDouble(long *val*) **Description**: Acts on a **long** value and returns the result as a **double**.
LongToIntFunction	int applyAsInt(long *val*) **Description**: Acts on a **long** value and returns the result as an **int**.
LongUnaryOperator	long applyAsLong(long *val*) **Description**: Acts on a **long** and returns a **long** result.
ObjDoubleConsumer<T>	void accept(T *val1*, double *val2*) **Description**: Acts on *val1* and the **double** value *val2*.
ObjIntConsumer<T>	void accept(T *val1*, int *val2*) **Description**: Acts on *val1* and the **int** value *val2*.
ObjLongConsumer<T>	void accept(T *val1*, long *val2*) **Description**: Acts on *val1* and the **long** value *val2*.
Predicate<T>	boolean test(T val) **Description**: Returns **true** if *val* satisfies the condition defined by **test( )** and **false** otherwise.
Supplier<T>	T get( ) **Description**: Returns an object of type **T**.
ToDoubleBiFunction<T, U>	double applyAsDouble(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal* and returns the result as a **double**.
ToDoubleFunction<T>	double applyAsDouble(T *val*) **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*) **Description**: Acts on *tVal* and *uVal* and returns the result as an **int**.
ToIntFunction<T>	int applyAsInt(T *val*) **Description**: Acts on *val* and returns the result as an **int**.
ToLongBiFunction<T, U>	long applyAsLong(T *tVal*, U *uVal*) **Description**: Acts on *tVal* and *uVal* and returns the result as a **long**.
ToLongFunction<T>	long applyAsLong(T *val*) **Description**: Acts on *val* and returns the result as a **long**.
UnaryOperator<T>	T apply(T *val*) **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*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

Table 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*[ ], int *offset*, 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*.