**NOTE** If a security manager is present, some **PrintStream** constructors will throw a **SecurityException** if a security violation occurs.

**PrintStream** supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintStream** methods will call the object's **toString( )** method and then display the result.

Somewhat recently (with the release of JDK 5), the **printf( )** method was added to **PrintStream**. It allows you to specify the precise format of the data to be written. The **printf( )** method uses the **Formatter** class (described in Chapter 19) to format data. It then writes this data to the invoking stream. Although formatting can be done manually, by using **Formatter** directly, **printf( )** streamlines the process. It also parallels the C/C++ **printf( )** function, which makes it easy to convert existing C/C++ code into Java. Frankly, **printf( )** was a much welcome addition to the Java API because it greatly simplified the output of formatted data to the console.

The **printf( )** method has the following general forms:

PrintStream printf(String *fmtString*, Object … *args*)
PrintStream printf(Locale *loc*, String *fmtString*, Object … *args*)

The first version writes *args* to standard output in the format specified by *fmtString*, using the default locale. The second lets you specify a locale. Both return the invoking **PrintStream**.

In general, **printf( )** works in a manner similar to the **format( )** method specified by **Formatter**. The *fmtString* consists of two types of items. The first type is composed of characters that are simply copied to the output buffer. The second type contains format specifiers that define the way the subsequent arguments, specified by *args*, are displayed. For complete information on formatting output, including a description of the format specifiers, see the **Formatter** class in Chapter 19.

Because **System.out** is a **PrintStream**, you can call **printf( )** on **System.out**. Thus, **printf( )** can be used in place of **println( )** when writing to the console whenever formatted output is desired. For example, the following program uses **printf( )** to output numeric values in various formats. Prior to JDK 5, such formatting required a bit of work. With the addition of **printf( )**, this is now an easy task.

```
// Demonstrate printf().

class PrintfDemo {
  public static void main(String args[]) {
    System.out.println("Here are some numeric values " +
                       "in different formats.\n");

    System.out.printf("Various integer formats: ");
    System.out.printf("%d %(d %+d %05d\n", 3, -3, 3, 3);

    System.out.println();
    System.out.printf("Default floating-point format: %f\n",
                       1234567.123);
    System.out.printf("Floating-point with commas: %,f\n",
                       1234567.123);
```

```
    System.out.printf("Negative floating-point default: %,f\n",
                      -1234567.123);
    System.out.printf("Negative floating-point option: %,(f\n",
                      -1234567.123);

    System.out.println();

    System.out.printf("Line up positive and negative values:\n");
    System.out.printf("% ,.2f\n% ,.2f\n",
                      1234567.123, -1234567.123);
  }
}
```

The output is shown here:

```
Here are some numeric values in different formats.

Various integer formats: 3 (3) +3 00003

Default floating-point format: 1234567.123000
Floating-point with commas: 1,234,567.123000
Negative floating-point default: -1,234,567.123000
Negative floating-point option: (1,234,567.123000)

Line up positive and negative values:
  1,234,567.12
 -1,234,567.12
```

**PrintStream** also defines the **format( )** method. It has these general forms:

PrintStream format(String *fmtString*, Object … *args*)
PrintStream format(Locale *loc*, String *fmtString*, Object … *args*)

It works exactly like **printf( )**.

## DataOutputStream and DataInputStream

**DataOutputStream** and **DataInputStream** enable you to write or read primitive data to or from a stream. They implement the **DataOutput** and **DataInput** interfaces, respectively. These interfaces define methods that convert primitive values to or from a sequence of bytes. These streams make it easy to store binary data, such as integers or floating-point values, in a file. Each is examined here.

**DataOutputStream** extends **FilterOutputStream**, which extends **OutputStream**. In addition to implementing **DataOutput**, **DataOutputStream** also implements **AutoCloseable**, **Closeable**, and **Flushable**. **DataOutputStream** defines the following constructor:

DataOutputStream(OutputStream *outputStream*)

Here, *outputStream* specifies the output stream to which data will be written. When a **DataOutputStream** is closed (by calling **close( )**), the underlying stream specified by *outputStream* is also closed automatically.

**DataOutputStream** supports all of the methods defined by its superclasses. However, it is the methods defined by the **DataOutput** interface, which it implements, that make it interesting. **DataOutput** defines methods that convert values of a primitive type into a byte sequence and then writes it to the underlying stream. Here is a sampling of these methods:

final void writeDouble(double *value*) throws IOException
final void writeBoolean(boolean *value*) throws IOException
final void writeInt(int *value*) throws IOException

Here, *value* is the value written to the stream.

**DataInputStream** is the complement of **DataOuputStream**. It extends **FilterInputStream**, which extends **InputStream**. In addition to implementing the **DataInput** interface, **DataInputStream** also implements **AutoCloseable** and **Closeable**. Here is its only constructor:

DataInputStream(InputStream *inputStream*)

Here, *inputStream* specifies the input stream from which data will be read. When a **DataInputStream** is closed (by calling **close( )**), the underlying stream specified by *inputStream* is also closed automatically.

Like **DataOutputStream**, **DataInputStream** supports all of the methods of its superclasses, but it is the methods defined by the **DataInput** interface that make it unique. These methods read a sequence of bytes and convert them into values of a primitive type. Here is a sampling of these methods:

final double readDouble( ) throws IOException
final boolean readBoolean( ) throws IOException
final int readInt( ) throws IOException

The following program demonstrates the use of **DataOutputStream** and **DataInputStream**:

```
// Demonstrate DataInputStream and DataOutputStream.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class DataIODemo {
  public static void main(String args[]) throws IOException {

    // First, write the data.
    try ( DataOutputStream dout =
            new DataOutputStream(new FileOutputStream("Test.dat")) )
    {
      dout.writeDouble(98.6);
      dout.writeInt(1000);
      dout.writeBoolean(true);
```

```
    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open Output File");
      return;
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    // Now, read the data back.
    try ( DataInputStream din =
            new DataInputStream(new FileInputStream("Test.dat")) )
    {

      double d = din.readDouble();
      int i = din.readInt();
      boolean b = din.readBoolean();

      System.out.println("Here are the values: " +
                         d + " " + i + " " + b);
    } catch(FileNotFoundException e) {
      System.out.println("Cannot Open Input File");
      return;
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

The output is shown here:

```
Here are the values: 98.6 1000 true
```

## RandomAccessFile

**RandomAccessFile** encapsulates a random-access file. It is not derived from **InputStream** or **OutputStream**. Instead, it implements the interfaces **DataInput** and **DataOutput**, which define the basic I/O methods. It also implements the **AutoCloseable** and **Closeable** interfaces. **RandomAccessFile** is special because it supports positioning requests—that is, you can position the file pointer within the file. It has these two constructors:

RandomAccessFile(File *fileObj*, String *access*)
 throws FileNotFoundException

RandomAccessFile(String *filename*, String *access*)
 throws FileNotFoundException

In the first form, *fileObj* specifies the file to open as a **File** object. In the second form, the name of the file is passed in *filename*. In both cases, *access* determines what type of file access is permitted. If it is "r", then the file can be read, but not written. If it is "rw", then the file is opened in read-write mode. If it is "rws", the file is opened for read-write operations and

every change to the file's data or metadata will be immediately written to the physical device. If it is "rwd", the file is opened for read-write operations and every change to the file's data will be immediately written to the physical device.

The method **seek( )**, shown here, is used to set the current position of the file pointer within the file:

void seek(long *newPos*) throws IOException

Here, *newPos* specifies the new position, in bytes, of the file pointer from the beginning of the file. After a call to **seek( )**, the next read or write operation will occur at the new file position.

**RandomAccessFile** implements the standard input and output methods, which you can use to read and write to random access files. It also includes some additional methods. One is **setLength( )**. It has this signature:

void setLength(long *len*) throws IOException

This method sets the length of the invoking file to that specified by *len*. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

# The Character Streams

While the byte stream classes provide sufficient functionality to handle any type of I/O operation, they cannot work directly with Unicode characters. Since one of the main purposes of Java is to support the "write once, run anywhere" philosophy, it was necessary to include direct I/O support for characters. In this section, several of the character I/O classes are discussed. As explained earlier, at the top of the character stream hierarchies are the **Reader** and **Writer** abstract classes. We will begin with them.

## Reader

**Reader** is an abstract class that defines Java's model of streaming character input. It implements the **AutoCloseable**, **Closeable**, and **Readable** interfaces. All of the methods in this class (except for **markSupported( )**) will throw an **IOException** on error conditions. Table 20-3 provides a synopsis of the methods in **Reader**.

## Writer

**Writer** is an abstract class that defines streaming character output. It implements the **AutoCloseable**, **Closeable**, **Flushable**, and **Appendable** interfaces. All of the methods in this class throw an **IOException** in the case of errors. Table 20-4 shows a synopsis of the methods in **Writer**.

| Method | Description |
|---|---|
| abstract void close( ) | Closes the input source. Further read attempts will generate an **IOException**. |
| void mark(int *numChars*) | Places a mark at the current point in the input stream that will remain valid until *numChars* characters are read. |
| boolean markSupported( ) | Returns **true** if **mark( )**/**reset( )** are supported on this stream. |
| int read( ) | Returns an integer representation of the next available character from the invoking input stream. –1 is returned when the end of the file is encountered. |
| int read(char *buffer*[ ]) | Attempts to read up to *buffer.length* characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered. |
| int read(CharBuffer *buffer*) | Attempts to read characters into *buffer* and returns the actual number of characters that were successfully read. –1 is returned when the end of the file is encountered. |
| abstract int read(char *buffer*[ ], int *offset*, int *numChars*) | Attempts to read up to *numChars* characters into *buffer* starting at *buffer*[*offset*], returning the number of characters successfully read. –1 is returned when the end of the file is encountered. |
| boolean ready( ) | Returns **true** if the next input request will not wait. Otherwise, it returns **false**. |
| void reset( ) | Resets the input pointer to the previously set mark. |
| long skip(long *numChars*) | Skips over *numChars* characters of input, returning the number of characters actually skipped. |

**Table 20-3** The Methods Defined by **Reader**

| Method | Description |
|---|---|
| Writer append(char *ch*) | Appends *ch* to the end of the invoking output stream. Returns a reference to the invoking stream. |
| Writer append(CharSequence *chars*) | Appends *chars* to the end of the invoking output stream. Returns a reference to the invoking stream. |
| Writer append(CharSequence *chars*, int *begin*, int *end*) | Appends the subrange of *chars* specified by *begin* and *end*–1 to the end of the invoking output stream. Returns a reference to the invoking stream. |
| abstract void close( ) | Closes the output stream. Further write attempts will generate an **IOException**. |
| abstract void flush( ) | Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers. |

**Table 20-4** The Methods Defined by **Writer**

| Method | Description |
|---|---|
| void write(int *ch*) | Writes a single character to the invoking output stream. Note that the parameter is an **int**, which allows you to call **write** with an expression without having to cast it back to **char**. However, only the low-order 16 bits are written. |
| void write(char *buffer*[ ]) | Writes a complete array of characters to the invoking output stream. |
| abstract void write(char *buffer*[ ], int *offset*, int *numChars*) | Writes a subrange of *numChars* characters from the array *buffer*, beginning at *buffer*[*offset*] to the invoking output stream. |
| void write(String *str*) | Writes *str* to the invoking output stream. |
| void write(String *str*, int *offset*, int *numChars*) | Writes a subrange of *numChars* characters from the string *str*, beginning at the specified *offset*. |

**Table 20-4** The Methods Defined by **Writer** *(continued)*

## FileReader

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Two commonly used constructors are shown here:

FileReader(String *filePath*)
FileReader(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

The following example shows how to read lines from a file and display them on the standard output device. It reads its own source file, which must be in the current directory.

```
// Demonstrate FileReader.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

    try ( FileReader fr = new FileReader("FileReaderDemo.java") )
    {
      int c;

      // Read and display the file.
      while((c = fr.read()) != -1) System.out.print((char) c);

    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

## FileWriter

**FileWriter** creates a **Writer** that you can use to write to a file. Four commonly used constructors are shown here:

FileWriter(String *filePath*)
FileWriter(String *filePath*, boolean *append*)
FileWriter(File *fileObj*)
FileWriter(File *fileObj*, boolean *append*)

They can all throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. *If append* is **true**, then output is appended to the end of the file.

Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

The following example is a character stream version of an example shown earlier when **FileOutputStream** was discussed. This version creates a sample buffer of characters by first making a **String** and then using the **getChars( )** method to extract the character array equivalent. It then creates three files. The first, **file1.txt**, will contain every other character from the sample. The second, **file2.txt**, will contain the entire set of characters. Finally, the third, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileWriter.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class FileWriterDemo {
  public static void main(String args[]) throws IOException {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    char buffer[] = new char[source.length()];
    source.getChars(0, source.length(), buffer, 0);

    try ( FileWriter f0 = new FileWriter("file1.txt");
          FileWriter f1 = new FileWriter("file2.txt");
          FileWriter f2 = new FileWriter("file3.txt") )
    {
      // write to first file
      for (int i=0; i < buffer.length; i += 2) {
        f0.write(buffer[i]);
      }

      // write to second file
      f1.write(buffer);

      // write to third file
      f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
```

```
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

## CharArrayReader

**CharArrayReader** is an implementation of an input stream that uses a character array as the source. This class has two constructors, each of which requires a character array to provide the data source:

CharArrayReader(char *array* [ ])
CharArrayReader(char *array* [ ], int *start*, int *numChars*)

Here, *array* is the input source. The second constructor creates a **Reader** from a subset of your character array that begins with the character at the index specified by *start* and is *numChars* long.

The **close( )** method implemented by **CharArrayReader** does not throw any exceptions. This is because it cannot fail.

The following example uses a pair of **CharArrayReader**s:

```
// Demonstrate CharArrayReader.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

public class CharArrayReaderDemo {
  public static void main(String args[]) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    int length = tmp.length();
    char c[] = new char[length];

    tmp.getChars(0, length, c, 0);
    int i;

    try (CharArrayReader input1 = new CharArrayReader(c) )
    {
      System.out.println("input1 is:");
      while((i = input1.read()) != -1) {
        System.out.print((char)i);
      }
      System.out.println();
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    try ( CharArrayReader input2 = new CharArrayReader(c, 0, 5) )
    {
      System.out.println("input2 is:");
      while((i = input2.read()) != -1) {
        System.out.print((char)i);
      }
```

```
      System.out.println();
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

The **input1** object is constructed using the entire lowercase alphabet, whereas **input2** contains only the first five letters. Here is the output:

```
input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is:
abcde
```



## CharArrayWriter

**CharArrayWriter** is an implementation of an output stream that uses an array as the destination. **CharArrayWriter** has two constructors, shown here:

CharArrayWriter( )
CharArrayWriter(int *numChars*)

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by *numChars*. The buffer is held in the **buf** field of **CharArrayWriter**. The buffer size will be increased automatically, if needed. The number of characters held by the buffer is contained in the **count** field of **CharArrayWriter**. Both **buf** and **count** are protected fields.

The **close( )** method has no effect on a **CharArrayWriter**.

The following example demonstrates **CharArrayWriter** by reworking the sample program shown earlier for **ByteArrayOutputStream**. It produces the same output as the previous version.

```
// Demonstrate CharArrayWriter.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class CharArrayWriterDemo {
  public static void main(String args[]) throws IOException {
    CharArrayWriter f = new CharArrayWriter();
    String s = "This should end up in the array";
    char buf[] = new char[s.length()];

    s.getChars(0, s.length(), buf, 0);

    try {
      f.write(buf);
    } catch(IOException e) {
      System.out.println("Error Writing to Buffer");
      return;
    }
```

```
    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");

    char c[] = f.toCharArray();
    for (int i=0; i<c.length; i++) {
      System.out.print(c[i]);
    }

    System.out.println("\nTo a FileWriter()");

    // Use try-with-resources to manage the file stream.
    try ( FileWriter f2 = new FileWriter("test.txt") )
    {
      f.writeTo(f2);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }

    System.out.println("Doing a reset");
    f.reset();

    for (int i=0; i<3; i++) f.write('X');

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

## BufferedReader

**BufferedReader** improves performance by buffering input. It has two constructors:

BufferedReader(Reader *inputStream*)
BufferedReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered character stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*.

Closing a **BufferedReader** also causes the underlying stream specified by *inputStream* to be closed.

As is the case with the byte-oriented stream, buffering an input character stream also provides the foundation required to support moving backward in the stream within the available buffer. To support this, **BufferedReader** implements the **mark( )** and **reset( )** methods, and **BufferedReader.markSupported( )** returns **true**. JDK 8 adds a new method to **BufferedReader** called **lines( )**. It returns a **Stream** reference to the sequence of lines read by the reader. (**Stream** is part of the new stream API discussed in Chapter 29.)

The following example reworks the **BufferedInputStream** example, shown earlier, so that it uses a **BufferedReader** character stream rather than a buffered byte stream. As before, it uses the **mark( )** and **reset( )** methods to parse a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset( )** happens and where it does not. Output is the same as that shown earlier.

```
// Use buffered input.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class BufferedReaderDemo {
  public static void main(String args[]) throws IOException {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);

    CharArrayReader in = new CharArrayReader(buf);
    int c;
    boolean marked = false;

    try ( BufferedReader f = new BufferedReader(in) )
    {

      while ((c = f.read()) != -1) {
        switch(c) {
        case '&':
          if (!marked) {
            f.mark(32);
            marked = true;
          } else {
            marked = false;
          }
          break;
        case ';':
          if (marked) {
            marked = false;
            System.out.print("(c)");
          } else
            System.out.print((char) c);
          break;
        case ' ':
          if (marked) {
            marked = false;
            f.reset();
            System.out.print("&");
          } else
            System.out.print((char) c);
          break;
        default:
          if (!marked)
            System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

## BufferedWriter

A **BufferedWriter** is a **Writer** that buffers output. Using a **BufferedWriter** can improve performance by reducing the number of times data is actually physically written to the output device.

A **BufferedWriter** has these two constructors:

BufferedWriter(Writer *outputStream*)
BufferedWriter(Writer *outputStream*, int *bufSize*)

The first form creates a buffered stream using a buffer with a default size. In the second, the size of the buffer is passed in *bufSize*.

## PushbackReader

The **PushbackReader** class allows one or more characters to be returned to the input stream. This allows you to look ahead in the input stream. Here are its two constructors:

PushbackReader(Reader *inputStream*)
PushbackReader(Reader *inputStream*, int *bufSize*)

The first form creates a buffered stream that allows one character to be pushed back. In the second, the size of the pushback buffer is passed in *bufSize*.

Closing a **PushbackReader** also closes the underlying stream specified by *inputStream*.

**PushbackReader** provides **unread( )**, which returns one or more characters to the invoking input stream. It has the three forms shown here:

void unread(int *ch*) throws IOException
void unread(char *buffer* [ ]) throws IOException
void unread(char *buffer* [ ], int *offset*, int *numChars*) throws IOException

The first form pushes back the character passed in *ch*. This will be the next character returned by a subsequent call to **read( )**. The second form returns the characters in *buffer*. The third form pushes back *numChars* characters beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to return a character when the pushback buffer is full.

The following program reworks the earlier **PushbackInputStream** example by replacing **PushbackInputStream** with **PushbackReader**. As before, it shows how a programming language parser can use a pushback stream to deal with the difference between the == operator for comparison and the = operator for assignment.

```
// Demonstrate unread().
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

class PushbackReaderDemo {
  public static void main(String args[]) {
    String s = "if (a == 4) a = 0;\n";
    char buf[] = new char[s.length()];
    s.getChars(0, s.length(), buf, 0);
    CharArrayReader in = new CharArrayReader(buf);

    int c;
```

```
    try ( PushbackReader f = new PushbackReader(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '=':
          if ((c = f.read()) == '=')
            System.out.print(".eq.");
          else {
            System.out.print("<-");
            f.unread(c);
          }
          break;
        default:
          System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```



## PrintWriter

**PrintWriter** is essentially a character-oriented version of **PrintStream**. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces. **PrintWriter** has several constructors. The following have been supplied by **PrintWriter** from the start:

PrintWriter(OutputStream *outputStream*)
PrintWriter(OutputStream *outputStream*, boolean *autoFlushingOn*)
PrintWriter(Writer *outputStream*)
PrintWriter(Writer *outputStream*, boolean *autoFlushingOn*)

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *autoFlushingOn* parameter controls whether the output buffer is automatically flushed every time **println( )**, **printf( )**, or **format( )** is called. If *autoFlushingOn* is **true**, flushing automatically takes place. If **false**, flushing is not automatic. Constructors that do not specify the *autoFlushingOn* parameter do not automatically flush.

The next set of constructors gives you an easy way to construct a **PrintWriter** that writes its output to a file.

PrintWriter(File *outputFile*) throws FileNotFoundException
PrintWriter(File *outputFile*, String *charSet*)
 throws FileNotFoundException, UnsupportedEncodingException
PrintWriter(String *outputFileName*) throws FileNotFoundException
PrintWriter(String *outputFileName*, String *charSet*)
 throws FileNotFoundException, UnsupportedEncodingException

These allow a **PrintWriter** to be created from a **File** object or by specifying the name of a file. In either case, the file is automatically created. Any preexisting file by the same name is destroyed. Once created, the **PrintWriter** object directs all output to the specified file. You can specify a character encoding by passing its name in *charSet*.

**PrintWriter** supports the **print( )** and **println( )** methods for all types, including **Object**. If an argument is not a primitive type, the **PrintWriter** methods will call the object's **toString( )** method and then output the result.

**PrintWriter** also supports the **printf( )** method. It works the same way it does in the **PrintStream** class described earlier: It allows you to specify the precise format of the data. Here is how **printf( )** is declared in **PrintWriter**:

PrintWriter printf(String *fmtString*, Object … *args*)
PrintWriter printf(Locale *loc*, String *fmtString*, Object …*args*)

The first version writes *args* to standard output in the format specified by *fmtString*, using the default locale. The second lets you specify a locale. Both return the invoking **PrintWriter**.

The **format( )** method is also supported. It has these general forms:

PrintWriter format(String *fmtString*, Object … *args*)
PrintWriter format(Locale *loc*, String *fmtString*, Object … *args*)

It works exactly like **printf( )**.

## The Console Class

The **Console** class was added to **java.io** by JDK 6. It is used to read from and write to the console, if one exists. It implements the **Flushable** interface. **Console** is primarily a convenience class because most of its functionality is available through **System.in** and **System.out**. However, its use can simplify some types of console interactions, especially when reading strings from the console.

**Console** supplies no constructors. Instead, a **Console** object is obtained by calling **System.console( )**, which is shown here:

static Console console( )

If a console is available, then a reference to it is returned. Otherwise, **null** is returned. A console will not be available in all cases. Thus, if **null** is returned, no console I/O is possible.

**Console** defines the methods shown in Table 20-5. Notice that the input methods, such as **readLine( )**, throw **IOError** if an input error occurs. **IOError** is a subclass of **Error**. It indicates an I/O failure that is beyond the control of your program. Thus, you will not normally catch an **IOError**. Frankly, if an **IOError** is thrown while accessing the console, it usually means there has been a catastrophic system failure.

Also notice the **readPassword( )** methods. These methods let your application read a password without echoing what is typed. When reading passwords, you should "zero-out" both the array that holds the string entered by the user and the array that holds the password that the string is tested against. This reduces the chance that a malicious program will be able to obtain a password by scanning memory.

| Method | Description |
|---|---|
| void flush( ) | Causes buffered output to be written physically to the console. |
| Console format(String *fmtString*, Object...*args*) | Writes *args* to the console using the format specified by *fmtString*. |
| Console printf(String *fmtString*, Object...*args*) | Writes *args* to the console using the format specified by *fmtString*. |
| Reader reader( ) | Returns a reference to a **Reader** connected to the console. |
| String readLine( ) | Reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| String readLine(String *fmtString*, Object…*args*) | Displays a prompting string formatted as specified by *fmtString* and *args*, and then reads and returns a string entered at the keyboard. Input stops when the user presses ENTER. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| char[ ] readPassword( ) | Reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| char[ ] readPassword(String *fmtString*, Object… *args*) | Displays a prompting string formatted as specified by *fmtString* and *args*, and then reads a string entered at the keyboard. Input stops when the user presses ENTER. The string is not displayed. If the end of the console input stream has been reached, **null** is returned. An **IOError** is thrown on failure. |
| PrintWriter writer( ) | Returns a reference to a **Writer** connected to the console. |

**Table 20-5** The Methods Defined by **Console**

Here is an example that demonstrates the **Console** class:

```
// Demonstrate Console.
import java.io.*;

class ConsoleDemo {
  public static void main(String args[]) {
    String str;
    Console con;
```

```
    // Obtain a reference to the console.
    con = System.console();
    // If no console available, exit.
    if(con == null) return;

    // Read a string and then display it.
    str = con.readLine("Enter a string: ");
    con.printf("Here is your string: %s\n", str);
  }
}
```

Here is sample output:

```
Enter a string: This is a test.
Here is your string: This is a test.
```

# Serialization

*Serialization* is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement *Remote Method Invocation (RMI)*. RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it. (More information about RMI appears in Chapter 30.)

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

An overview of the interfaces and classes that support serialization follows.

## Serializable

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable, all of its subclasses are also serializable.

Variables that are declared as **transient** are not saved by the serialization facilities. Also, **static** variables are not saved.

## Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The **Externalizable** interface is designed for these situations.

The **Externalizable** interface defines these two methods:

```
void readExternal(ObjectInput inStream)
  throws IOException, ClassNotFoundException
void writeExternal(ObjectOutput outStream)
  throws IOException
```

In these methods, *inStream* is the byte stream from which the object is to be read, and *outStream* is the byte stream to which the object is to be written.

## ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** and **AutoCloseable** interfaces and supports object serialization. It defines the methods shown in Table 20-6. Note especially the **writeObject( )** method. This is called to serialize an object. All of these methods will throw an **IOException** on error conditions.

| Method | Description |
|---|---|
| void close( ) | Closes the invoking stream. Further write attempts will generate an **IOException**. |
| void flush( ) | Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers. |
| void write(byte *buffer*[ ]) | Writes an array of bytes to the invoking stream. |
| void write(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*]. |
| void write(int *b*) | Writes a single byte to the invoking stream. The byte written is the low-order byte of *b*. |
| void writeObject(Object *obj*) | Writes object *obj* to the invoking stream. |

**Table 20-6** The Methods Defined by **ObjectOutput**

## ObjectOutputStream

The **ObjectOutputStream** class extends the **OutputStream** class and implements the **ObjectOutput** interface. It is responsible for writing objects to a stream. One constructor of this class is shown here:

ObjectOutputStream(OutputStream *outStream*) throws IOException

The argument *outStream* is the output stream to which serialized objects will be written. Closing an **ObjectOutputStream** automatically closes the underlying stream specified by *outStream*.

Several commonly used methods in this class are shown in Table 20-7. They will throw an **IOException** on error conditions. There is also an inner class to **ObjectOuputStream** called **PutField**. It facilitates the writing of persistent fields, and its use is beyond the scope of this book.

| Method | Description |
|---|---|
| void close( ) | Closes the invoking stream. Further write attempts will generate an **IOException**. The underlying stream is also closed. |
| void flush( ) | Finalizes the output state so any buffers are cleared. That is, it flushes the output buffers. |
| void write(byte *buffer*[ ]) | Writes an array of bytes to the invoking stream. |
| void write(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*]. |
| void write(int *b*) | Writes a single **byte** to the invoking stream. The byte written is the low-order byte of *b*. |
| void writeBoolean(boolean *b*) | Writes a **boolean** to the invoking stream. |
| void writeByte(int *b*) | Writes a **byte** to the invoking stream. The byte written is the low-order byte of *b*. |
| void writeBytes(String *str*) | Writes the bytes representing *str* to the invoking stream. |
| void writeChar(int *c*) | Writes a **char** to the invoking stream. |
| void writeChars(String *str*) | Writes the characters in *str* to the invoking stream. |
| void writeDouble(double *d*) | Writes a **double** to the invoking stream. |
| void writeFloat(float *f*) | Writes a **float** to the invoking stream. |
| void writeInt(int *i*) | Writes an **int** to the invoking stream. |
| void writeLong(long *l*) | Writes a **long** to the invoking stream. |
| final void writeObject(Object *obj*) | Writes *obj* to the invoking stream. |
| void writeShort(int *i*) | Writes a **short** to the invoking stream. |

**Table 20-7** A Sampling of Commonly Used Methods Defined by **ObjectOutputStream**

## ObjectInput

The **ObjectInput** interface extends the **DataInput** and **AutoCloseable** interfaces and defines the methods shown in Table 20-8. It supports object serialization. Note especially the **readObject( )** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**.

## ObjectInputStream

The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface. **ObjectInputStream** is responsible for reading objects from a stream. One constructor of this class is shown here:

ObjectInputStream(InputStream *inStream*) throws IOException

The argument *inStream* is the input stream from which serialized objects should be read. Closing an **ObjectInputStream** automatically closes the underlying stream specified by *inStream*.

Several commonly used methods in this class are shown in Table 20-9. They will throw an **IOException** on error conditions. The **readObject( )** method can also throw **ClassNotFoundException**. There is also an inner class to **ObjectInputStream** called **GetField**. It facilitates the reading of persistent fields, and its use is beyond the scope of this book.



| Method | Description |
|---|---|
| int available( ) | Returns the number of bytes that are now available in the input buffer. |
| void close( ) | Closes the invoking stream. Further read attempts will generate an **IOException**. |
| int read( ) | Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ]) | Attempts to read up to *buffer.length* bytes into *buffer*, returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes that were successfully read. –1 is returned when the end of the file is encountered. |
| Object readObject( ) | Reads an object from the invoking stream. |
| long skip(long *numBytes*) | Ignores (that is, skips) *numBytes* bytes in the invoking stream, returning the number of bytes actually ignored. |

**Table 20-8** The Methods Defined by **ObjectInput**

| Method | Description |
|---|---|
| int available( ) | Returns the number of bytes that are now available in the input buffer. |
| void close( ) | Closes the invoking stream. Further read attempts will generate an **IOException**. The underlying stream is also closed. |
| int read( ) | Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered. |
| int read(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered. |
| Boolean readBoolean( ) | Reads and returns a **boolean** from the invoking stream. |
| byte readByte( ) | Reads and returns a **byte** from the invoking stream. |
| char readChar( ) | Reads and returns a **char** from the invoking stream. |
| double readDouble( ) | Reads and returns a **double** from the invoking stream. |
| float readFloat( ) | Reads and returns a **float** from the invoking stream. |
| void readFully(byte *buffer*[ ]) | Reads *buffer.length* bytes into *buffer*. Returns only when all bytes have been read. |
| void readFully(byte *buffer*[ ],<br>int *offset*,<br>int *numBytes*) | Reads *numBytes* bytes into *buffer* starting at *buffer*[*offset*]. Returns only when *numBytes* have been read. |
| int readInt( ) | Reads and returns an **int** from the invoking stream. |
| long readLong( ) | Reads and returns a **long** from the invoking stream. |
| final Object readObject( ) | Reads and returns an object from the invoking stream. |
| short readShort( ) | Reads and returns a **short** from the invoking stream. |
| int readUnsignedByte( ) | Reads and returns an unsigned **byte** from the invoking stream. |
| int readUnsignedShort( ) | Reads and returns an unsigned **short** from the invoking stream. |

**Table 20-9** Commonly Used Methods Defined by **ObjectInputStream**

## A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String**, **int**, and **double**. This is the information we want to save and restore.

A **FileOutputStream** is created that refers to a file named "serial", and an **ObjectOutputStream** is created for that file stream. The **writeObject( )** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed.

A **FileInputStream** is then created that refers to the file named "serial", and an **ObjectInputStream** is created for that file stream. The **readObject( )** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.

Note that **MyClass** is defined to implement the **Serializable** interface. If this is not done, a **NotSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be **transient**. That data is then not saved during serialization.

```
// A serialization demo.
// This program uses try-with-resources. It requires JDK 7 or later.

import java.io.*;

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

    // Object serialization

    try ( ObjectOutputStream objOStrm =
            new ObjectOutputStream(new FileOutputStream("serial")) )
    {
      MyClass object1 = new MyClass("Hello", -7, 2.7e10);
      System.out.println("object1: " + object1);

      objOStrm.writeObject(object1);
    }
    catch(IOException e) {
      System.out.println("Exception during serialization: " + e);
    }

    // Object deserialization

    try ( ObjectInputStream objIStrm =
            new ObjectInputStream(new FileInputStream("serial")) )
    {
      MyClass object2 = (MyClass)objIStrm.readObject();
      System.out.println("object2: " + object2);
    }
    catch(Exception e) {
      System.out.println("Exception during deserialization: " + e);
    }
  }
}

class MyClass implements Serializable {
  String s;
  int i;
  double d;

  public MyClass(String s, int i, double d) {
    this.s = s;
    this.i = i;
    this.d = d;
  }
```

```
  public String toString() {
    return "s=" + s + "; i=" + i + "; d=" + d;
  }
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical. The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

## Stream Benefits

The streaming interface to I/O in Java provides a clean abstraction for a complex and often cumbersome task. The composition of the filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements. Java programs written to adhere to the abstract, high-level **InputStream**, **OutputStream**, **Reader**, and **Writer** classes will function properly in the future even when new and improved concrete stream classes are invented. As you will see in Chapter 22, this model works very well when we switch from a file system–based set of streams to the network and socket streams. Finally, serialization of objects plays an important role in many types of Java programs. Java's serialization I/O classes provide a portable solution to this sometimes tricky task.

CHAPTER

# 21 Exploring NIO

Beginning with version 1.4, Java has provided a second I/O system called NIO (which is short for *New I/O*). It supports a buffer-oriented, channel-based approach to I/O operations. With the release of JDK 7, the NIO system was greatly expanded, providing enhanced support for file-handling and file system features. In fact, so significant were the changes that the term *NIO.2* is often used. Because of the capabilities supported by the NIO file classes, NIO is expected to become an increasingly important approach to file handling. This chapter explores several of the key features of the NIO system.

## The NIO Classes

The NIO classes are contained in the packages shown here:

| Package | Purpose |
|---|---|
| java.nio | Top-level package for the NIO system. Encapsulates various types of buffers that contain data operated upon by the NIO system. |
| java.nio.channels | Supports channels, which are essentially open I/O connections. |
| java.nio.channels.spi | Supports service providers for channels. |
| java.nio.charset | Encapsulates character sets. Also supports encoders and decoders that convert characters to bytes and bytes to characters, respectively. |
| java.nio.charset.spi | Supports service providers for character sets. |
| java.nio.file | Provides support for files. |
| java.nio.file.attribute | Provides support for file attributes. |
| java.nio.file.spi | Supports service providers for file systems. |

Before we begin, it is important to emphasize that the NIO subsystem does not replace the stream-based I/O classes found in **java.io**, which are discussed in Chapter 20, and good working knowledge of the stream-based I/O in **java.io** is helpful to understanding NIO.

**NOTE** This chapter assumes that you have read the overview of I/O given in Chapter 13 and the discussion of stream-based I/O supplied in Chapter 20.

# NIO Fundamentals

The NIO system is built on two foundational items: buffers and channels. A *buffer* holds data. A *channel* represents an open connection to an I/O device, such as a file or a socket. In general, to use the NIO system, you obtain a channel to an I/O device and a buffer to hold data. You then operate on the buffer, inputting or outputting data as needed. The following sections examine buffers and channels in more detail.

## Buffers

Buffers are defined in the **java.nio** package. All buffers are subclasses of the **Buffer** class, which defines the core functionality common to all buffers: current position, limit, and capacity. The *current position* is the index within the buffer at which the next read or write operation will take place. The current position is advanced by most read or write operations. The *limit* is the index value one past the last valid location in the buffer. The *capacity* is the number of elements that the buffer can hold. Often the limit equals the capacity of the buffer. **Buffer** also supports mark and reset. **Buffer** defines several methods, which are shown in Table 21-1.

| Method | Description |
|---|---|
| abstract Object array( ) | If the invoking buffer is backed by an array, returns a reference to the array. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown. |
| abstract int arrayOffset( ) | If the invoking buffer is backed by an array, returns the index of the first element. Otherwise, an **UnsupportedOperationException** is thrown. If the array is read-only, a **ReadOnlyBufferException** is thrown. |
| final int capacity( ) | Returns the number of elements that the invoking buffer is capable of holding. |
| final Buffer clear( ) | Clears the invoking buffer and returns a reference to the buffer. |
| final Buffer flip( ) | Sets the invoking buffer's limit to the current position and resets the current position to 0. Returns a reference to the buffer. |
| abstract boolean hasArray( ) | Returns **true** if the invoking buffer is backed by a read/write array and **false** otherwise. |
| final boolean hasRemaining( ) | Returns **true** if there are elements remaining in the invoking buffer. Returns **false** otherwise. |

**Table 21-1** The Methods Defined by **Buffer**

| Method | Description |
|---|---|
| abstract boolean isDirect( ) | Returns **true** if the invoking buffer is direct, which means I/O operations act directly upon it. Returns **false** otherwise. |
| abstract boolean isReadOnly( ) | Returns **true** if the invoking buffer is read-only. Returns **false** otherwise. |
| final int limit( ) | Returns the invoking buffer's limit. |
| final Buffer limit(int *n*) | Sets the invoking buffer's limit to *n*. Returns a reference to the buffer. |
| final Buffer mark( ) | Sets the mark and returns a reference to the invoking buffer. |
| final int position( ) | Returns the current position. |
| final Buffer position(int *n*) | Sets the invoking buffer's current position to *n*. Returns a reference to the buffer. |
| int remaining( ) | Returns the number of elements available before the limit is reached. In other words, it returns the limit minus the current position. |
| final Buffer reset( ) | Resets the current position of the invoking buffer to the previously set mark. Returns a reference to the buffer. |
| final Buffer rewind( ) | Sets the position of the invoking buffer to 0. Returns a reference to the buffer. |

**Table 21-1** The Methods Defined by **Buffer** *(continued)*

From **Buffer**, the following specific buffer classes are derived, which hold the type of data that their names imply:

| | | | |
|---|---|---|---|
| ByteBuffer | CharBuffer | DoubleBuffer | FloatBuffer |
| IntBuffer | LongBuffer | MappedByteBuffer | ShortBuffer |

**MappedByteBuffer** is a subclass of **ByteBuffer** and is used to map a file to a buffer.

All of the aforementioned buffers provide various **get( )** and **put( )** methods, which allow you to get data from a buffer or put data into a buffer. (Of course, if a buffer is read-only, then **put( )** operations are not available.) Table 21-2 shows the **get( )** and **put( )** methods defined by **ByteBuffer**. The other buffer classes have similar methods. All buffer classes also support methods that perform various buffer operations. For example, you can allocate a buffer manually using **allocate( )**. You can wrap an array inside a buffer using **wrap( )**. You can create a subsequence of a buffer using **slice( )**.

## Channels

Channels are defined in **java.nio.channels**. A channel represents an open connection to an I/O source or destination. Channels implement the **Channel** interface. It extends **Closeable**, and it extends **AutoCloseable**. By implementing **AutoCloseable**, channels can be managed

| Method | Description |
|---|---|
| abstract byte get( ) | Returns the byte at the current position. |
| ByteBuffer get(byte *vals*[ ]) | Copies the invoking buffer into the array referred to by *vals*. Returns a reference to the buffer. If there are not *vals*.**length** elements remaining in the buffer, a **BufferUnderflowException** is thrown. |
| ByteBuffer get(byte *vals*[ ], int *start*, int *num*) | Copies *num* elements from the invoking buffer into the array referred to by *vals*, beginning at the index specified by *start*. Returns a reference to the buffer. If there are not *num* elements remaining in the buffer, a **BufferUnderflowException** is thrown. |
| abstract byte get(int *idx*) | Returns the byte at the index specified by *idx* within the invoking buffer. |
| abstract ByteBuffer put(byte *b*) | Copies *b* into the invoking buffer at the current position. Returns a reference to the buffer. If the buffer is full, a **BufferOverflowException** is thrown. |
| final ByteBuffer put(byte *vals*[ ] ) | Copies all elements of *vals* into the invoking buffer, beginning at the current position. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. |
| ByteBuffer put(byte *vals*[ ], int *start*, int *num*) | Copies *num* elements from *vals*, beginning at *start*, into the invoking buffer. Returns a reference to the buffer. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. |
| ByteBuffer put(ByteBuffer *bb*) | Copies the elements in *bb* to the invoking buffer, beginning at the current position. If the buffer cannot hold all of the elements, a **BufferOverflowException** is thrown. Returns a reference to the buffer. |
| abstract ByteBuffer put(int *idx*, byte *b*) | Copies *b* into the invoking buffer at the location specified by *idx*. Returns a reference to the buffer. |

**Table 21-2** The **get( )** and **put( )** Methods Defined for **ByteBuffer**

with a **try**-with-resources statement. When used in a **try**-with-resources block, a channel is closed automatically when it is no longer needed. (See Chapter 13 for a discussion of **try**-with-resources.)

One way to obtain a channel is by calling **getChannel( )** on an object that supports channels. For example, **getChannel( )** is supported by the following I/O classes:

| | | |
|---|---|---|
| DatagramSocket | FileInputStream | FileOutputStream |
| RandomAccessFile | ServerSocket | Socket |

The specific type of channel returned depends upon the type of object **getChannel( )** is called on. For example, when called on a **FileInputStream**, **FileOutputStream**, or **RandomAccessFile**, **getChannel( )** returns a channel of type **FileChannel**. When called on a **Socket**, **getChannel( )** returns a **SocketChannel**.

Another way to obtain a channel is to use one of the **static** methods defined by the **Files** class. For example, using **Files**, you can obtain a byte channel by calling **newByteChannel( )**. It returns a **SeekableByteChannel**, which is an interface implemented by **FileChannel**. (The **Files** class is examined in detail later in this chapter.)

Channels such as **FileChannel** and **SocketChannel** support various **read( )** and **write( )** methods that enable you to perform I/O operations through the channel. For example, here are a few of the **read( )** and **write( )** methods defined for **FileChannel**.

| Method | Description |
|---|---|
| abstract int read(ByteBuffer *bb*) throws IOException | Reads bytes from the invoking channel into *bb* until the buffer is full or there is no more input. Returns the number of bytes actually read. Returns –1 at end-of-stream. |
| abstract int read(ByteBuffer *bb*, long *start*) throws IOException | Beginning at the file location specified by *start*, reads bytes from the invoking channel into *bb* until the buffer is full or there is no more input. The current position is unchanged. Returns the number of bytes actually read or –1 if *start* is beyond the end of the file. |
| abstract int write(ByteBuffer *bb*) throws IOException | Writes the contents of *bb* to the invoking channel, starting at the current position. Returns the number of bytes written. |
| abstract int write(ByteBuffer *bb*, long *start*) throws IOException | Beginning at the file location specified by *start*, writes the contents of *bb* to the invoking channel. The current position is unchanged. Returns the number of bytes written. |

All channels support additional methods that give you access to and control over the channel. For example, **FileChannel** supports methods to get or set the current position, transfer information between file channels, obtain the current size of the channel, and lock the channel, among others. **FileChannel** provides a **static** method called **open( )**, which opens a file and returns a channel to it. This provides another way to obtain a channel. **FileChannel** also provides the **map( )** method, which lets you map a file to a buffer.

## Charsets and Selectors

Two other entities used by NIO are charsets and selectors. A *charset* defines the way that bytes are mapped to characters. You can encode a sequence of characters into bytes using an *encoder*. You can decode a sequence of bytes into characters using a *decoder*. Charsets, encoders, and decoders are supported by classes defined in the **java.nio.charset** package. Because default encoders and decoders are provided, you will not often need to work explicitly with charsets.

A *selector* supports key-based, non-blocking, multiplexed I/O. In other words, selectors enable you to perform I/O through multiple channels. Selectors are supported by classes defined in the **java.nio.channels** package. Selectors are most applicable to socket-backed channels.

We will not use charsets or selectors in this chapter, but you might find them useful in your own applications.

# Enhancements Added to NIO by JDK 7

Beginning with JDK 7, the NIO system was substantially expanded and enhanced. In addition to support for the **try**-with-resources statement (which provides automatic resource management), the improvements included three new packages (**java.nio.file**, **java.nio.file.attribute**, and **java.nio.file.spi**); several new classes, interfaces, and methods; and direct support for stream-based I/O. The additions have greatly expanded the ways in which NIO can be used, especially with files. Several of the key additions are described in the following sections.

## The Path Interface

Perhaps the single most important addition to the NIO system is the **Path** interface because it encapsulates a path to a file. As you will see, **Path** is the glue that binds together many of the NIO.2 file-based features. It describes a file's location within the directory structure. **Path** is packaged in **java.nio.file**, and it inherits the following interfaces: **Watchable**, **Iterable<Path>**, and **Comparable<Path>**. **Watchable** describes an object that can be monitored for changes. The **Iterable** and **Comparable** interfaces were described earlier in this book.

**Path** declares a number of methods that operate on the path. A sampling is shown in Table 21-3. Pay special attention to the **getName( )** method. It is used to obtain an element in a path. It works using an index. At index zero is the part of the path nearest the root, which is the leftmost element in a path. Subsequent indexes specify elements to the right of the root. The number of elements in a path can be obtained by calling **getNameCount( )**. If you want to obtain a string representation of the entire path, simply call **toString( )**. Notice that you can resolve a relative path into an absolute path by using the **resolve( )** method.

| Method | Description |
|---|---|
| boolean endsWith(String *path*) | Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**. |
| boolean endsWith(Path *path*) | Returns **true** if the invoking **Path** ends with the path specified by *path*. Otherwise, returns **false**. |
| Path getFileName( ) | Returns the filename associated with the invoking **Path**. |
| Path getName(int *idx*) | Returns a **Path** object that contains the name of the path element specified by *idx* within the invoking object. The leftmost element is at index 0. This is the element nearest the root. The rightmost element is at **getNameCount( )** – 1. |
| int getNameCount( ) | Returns the number of elements beyond the root directory in the invoking **Path**. |
| Path getParent( ) | Returns a **Path** that contains the entire path except for the name of the file specified by the invoking **Path**. |
| Path getRoot( ) | Returns the root of the invoking **Path**. |

**Table 21-3** A Sampling of Methods Specified by **Path**

| Method | Description |
|---|---|
| boolean isAbsolute( ) | Returns **true** if the invoking **Path** is absolute. Otherwise, returns **false**. |
| Path resolve(Path *path*) | If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified. |
| Path resolve(String *path*) | If *path* is absolute, *path* is returned. Otherwise, if *path* does not contain a root, *path* is prefixed by the root specified by the invoking **Path** and the result is returned. If *path* is empty, the invoking **Path** is returned. Otherwise, the behavior is unspecified. |
| boolean startsWith(String *path*) | Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**. |
| boolean startsWith(Path *path*) | Returns **true** if the invoking **Path** starts with the path specified by *path*. Otherwise, returns **false**. |
| Path toAbsolutePath( ) | Returns the invoking **Path** as an absolute path. |
| String toString( ) | Returns a string representation of the invoking **Path**. |

**Table 21-3** A Sampling of Methods Specified by **Path** *(continued)*

One other point: When updating legacy code that uses the **File** class defined by **java.io**, it is possible to convert a **File** instance into a **Path** instance by calling **toPath( )** on the **File** object. Furthermore, it is possible to obtain a **File** instance by calling the **toFile( )** method defined by **Path**.

## The Files Class

Many of the actions that you perform on a file are provided by **static** methods within the **Files** class. The file to be acted upon is specified by its **Path**. Thus, the **Files** methods use a **Path** to specify the file that is being operated upon. **Files** contains a wide array of functionality. For example, it has methods that let you open or create a file that has the specified path. You can obtain information about a **Path**, such as whether it is executable, hidden, or read-only. **Files** also supplies methods that let you copy or move files. A sampling is shown in Table 21-4. In addition to **IOException**, several other exceptions are possible. JDK 8 adds these four methods to **Files**: **list( )**, **walk( )**, **lines( )**, and **find( )**. All return a **Stream** object. These methods help integrate NIO with the new stream API defined by JDK 8 and described in Chapter 29.

Notice that several of the methods in Table 21-4 take an argument of type **OpenOption**. This is an interface that describes how to open a file. It is implemented by the **StandardOpenOption** class, which defines an enumeration that has the values shown in Table 21-5.

| Method | Description |
|---|---|
| static Path copy(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException | Copies the file specified by *src* to the location specified by *dest*. The *how* parameter specifies how the copy will take place. |
| static Path createDirectory(Path *path*, FileAttribute<?> ... *attribs*) throws IOException | Creates the directory whose path is specified by *path*. The directory attributes are specified by *attribs*. |
| static Path createFile(Path *path*, FileAttribute<?> ... *attribs*) throws IOException | Creates the file whose path is specified by *path*. The file attributes are specified by *attribs*. |
| static void delete(Path *path*) throws IOException | Deletes the file whose path is specified by *path*. |
| static boolean exists(Path *path*, LinkOption ... *opts*) | Returns **true** if the file specified by *path* exists and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static boolean isDirectory(Path *path*, LinkOption ... *opts*) | Returns **true** if *path* specifies a directory and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static boolean isExecutable(Path *path*) | Returns **true** if the file specified by *path* is executable and **false** otherwise. |
| static boolean isHidden(Path *path*) throws IOException | Returns **true** if the file specified by *path* is hidden and **false** otherwise. |
| static boolean isReadable(Path *path*) | Returns **true** if the file specified by *path* can be read from and **false** otherwise. |
| static boolean isRegularFile(Path *path*, LinkOption ... *opts*) | Returns **true** if *path* specifies a file and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static boolean isWritable(Path *path*) | Returns **true** if the file specified by *path* can be written to and **false** otherwise. |
| static Path move(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException | Moves the file specified by *src* to the location specified by *dest*. The *how* parameter specifies how the move will take place. |
| static SeekableByteChannel newByteChannel(Path *path*, OpenOption ... *how*) throws IOException | Opens the file specified by *path*, as specified by *how*. Returns a **SeekableByteChannel** to the file. This is a byte channel whose current position can be changed. **SeekableByteChannel** is implemented by **FileChannel**. |
| static DirectoryStream<Path> newDirectoryStream(Path *path*) throws IOException | Opens the directory specified by *path*. Returns a **DirectoryStream** linked to the directory. |

**Table 21-4** A Sampling of Methods Defined by **Files**

| Method | Description |
|---|---|
| static InputStream<br>newInputStream(Path *path*,<br>OpenOption ... *how*)<br>throws IOException | Opens the file specified by *path*, as specified by *how*. Returns an **InputStream** linked to the file. |
| static OutputStream<br>newOutputStream(Path *path*,<br>OpenOption ... *how*)<br>throws IOException | Opens the file specified by the invoking object, as specified by *how*. Returns an **OutputStream** linked to the file. |
| static boolean<br>notExists(Path *path*,<br>LinkOption ... *opts*) | Returns **true** if the file specified by *path* does *not* exist and **false** otherwise. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static <A extends BasicFileAttributes> A<br>readAttributes(Path *path*,<br>Class<A> *attribType*,<br>LinkOption ... *opts*)<br>throws IOException | Obtains the attributes associated with the file specified by *path*. The type of attributes to obtain is passed in *attribType*. If *opts* is not specified, then symbolic links are followed. To prevent the following of symbolic links, pass **NOFOLLOW_LINKS** to *opts*. |
| static long size(Path *path*)<br>throws IOException | Returns the size of the file specified by *path*. |

**Table 21-4** A Sampling of Methods Defined by **Files** *(continued)*

| Value | Meaning |
|---|---|
| APPEND | Causes output to be written to the end of the file. |
| CREATE | Creates the file if it does not already exist. |
| CREATE_NEW | Creates the file only if it does not already exist. |
| DELETE_ON_CLOSE | Deletes the file when it is closed. |
| DSYNC | Causes changes to the file to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed. |
| READ | Opens the file for input operations. |
| SPARSE | Indicates to the file system that the file is sparse, meaning that it may not be completely filled with data. If the file system does not support sparse files, this option is ignored. |
| SYNC | Causes changes to the file or its metadata to be immediately written to the physical file. Normally, changes to a file are buffered by the file system in the interest of efficiency, being written to the file only as needed. |
| TRUNCATE_EXISTING | Causes a preexisting file opened for output to be reduced to zero length. |
| WRITE | Opens the file for output operations. |

**Table 21-5** The Standard Open Options

## The Paths Class

Because **Path** is an interface, not a class, you can't create an instance of **Path** directly through the use of a constructor. Instead, you obtain a **Path** by a calling a method that returns one. Frequently, you do this by using the **get( )** method defined by the **Paths** class. There are two forms of **get( )**. The one used in this chapter is shown here:

static Path get(String *pathname*, String ... *parts*)

It returns a **Path** that encapsulates the specified path. The path can be specified in two ways. First, if *parts* is not used, then the path must be specified in its entirety by *pathname*. Alternatively, you can pass the path in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. In either case, if the path specified is syntactically invalid, **get( )** will throw an **InvalidPathException**.

The second form of **get( )** creates a **Path** from a **URI**. It is shown here:

static Path get(URI *uri*)

The **Path** corresponding to *uri* is returned.

It is important to understand that creating a **Path** to a file does not open or create a file. It simply creates an object that encapsulates the file's directory path.

## The File Attribute Interfaces

Associated with a file is a set of attributes. These attributes include such things as the file's time of creation, the time of its last modification, whether the file is a directory, and its size. NIO organizes file attributes into several different interfaces. Attributes are represented by a hierarchy of interfaces defined in **java.nio.file.attribute**. At the top is **BasicFileAttributes**. It encapsulates the set of attributes that are commonly found in a variety of file systems. The methods defined by **BasicFileAttributes** are shown in Table 21-6.

| Method | Description |
|---|---|
| FileTime creationTime( ) | Returns the time at which the file was created. If creation time is not provided by the file system, then an implementation-dependent value is returned. |
| Object fileKey( ) | Returns the file key. If not supported, **null** is returned. |
| boolean isDirectory( ) | Returns **true** if the file represents a directory. |
| boolean isOther( ) | Returns **true** if the file is not a file, symbolic link, or a directory. |
| boolean isRegularFile( ) | Returns **true** if the file is a normal file, rather than a directory or symbolic link. |
| boolean isSymbolicLink( ) | Returns **true** if the file is a symbolic link. |
| FileTime lastAccessTime( ) | Returns the time at which the file was last accessed. If the time of last access is not provided by the file system, then an implementation-dependent value is returned. |
| FileTime lastModifiedTime( ) | Returns the time at which the file was last modified. If the time of last modification is not provided by the file system, then an implementation-dependent value is returned. |
| long size( ) | Returns the size of the file. |

**Table 21-6** The Methods Defined by **BasicFileAttributes**

From **BasicFileAttributes** two interfaces are derived: **DosFileAttributes** and **PosixFileAttributes**. **DosFileAttributes** describes those attributes related to the FAT file system as first defined by DOS. It defines the methods shown here:

| Method | Description |
|---|---|
| boolean isArchive( ) | Returns **true** if the file is flagged for archiving and **false** otherwise. |
| boolean isHidden( ) | Returns **true** if the file is hidden and **false** otherwise. |
| boolean isReadOnly( ) | Returns **true** if the file is read-only and **false** otherwise. |
| boolean isSystem( ) | Returns **true** if the file is flagged as a system file and **false** otherwise. |

**PosixFileAttributes** encapsulates attributes defined by the POSIX standards. (POSIX stands for *Portable Operating System Interface*.) It defines the methods shown here:

| Method | Description |
|---|---|
| GroupPrincipal group( ) | Returns the file's group owner. |
| UserPrincipal owner( ) | Returns the file's owner. |
| Set<PosixFilePermission> permissions( ) | Returns the file's permissions. |

There are various ways to access a file's attributes. First, you can obtain an object that encapsulates a file's attributes by calling **readAttributes( )**, which is a **static** method defined by **Files**. One of its forms is shown here:

```
static <A extends BasicFileAttributes>
    A readAttributes(Path path, Class<A> attrType, LinkOption... opts)
            throws IOException
```

This method returns a reference to an object that specifies the attributes associated with the file passed in *path*. The specific type of attributes is specified as a **Class** object in the *attrType* parameter. For example, to obtain the basic file attributes, pass **BasicFileAttributes.class** to *attrType*. For DOS attributes, use **DosFileAttributes.class**, and for POSIX attributes, use **PosixFileAttributes.class**. Optional link options are passed via *opts*. If not specified, symbolic links are followed. The method returns a reference to requested attributes. If the requested attribute type is not available, **UnsupportedOperationException** is thrown. Using the object returned, you can access the file's attributes.

A second way to gain access to a file's attributes is to call **getFileAttributeView( )** defined by **Files**. NIO defines several attribute view interfaces, including **AttributeView**, **BasicFileAttributeView**, **DosFileAttributeView**, and **PosixFileAttributeView**, among others. Although we won't be using attribute views in this chapter, they are a feature that you may find helpful in some situations.

In some cases, you won't need to use the file attribute interfaces directly because the **Files** class offers **static** convenience methods that access several of the attributes. For example, **Files** includes methods such as **isHidden( )** and **isWritable( )**.

It is important to understand that not all file systems support all possible attributes. For example, the DOS file attributes apply to the older FAT file system as first defined by DOS. The attributes that will apply to a wide variety of file systems are described by **BasicFileAttributes**. For this reason, these attributes are used in the examples in this chapter.

### The FileSystem, FileSystems, and FileStore Classes

You can easily access the file system through the **FileSystem** and **FileSystems** classes packaged in **java.nio.file**. In fact, by using the **newFileSystem( )** method defined by **FileSystems**, it is even possible to obtain a new file system. The **FileStore** class encapsulates the file storage system. Although these classes are not used directly in this chapter, you may find them helpful in your own applications.

## Using the NIO System

This section illustrates how to apply the NIO system to a variety of tasks. Before beginning, it is important to emphasize that with the release of JDK 7, the NIO subsystem was greatly expanded. As a result, its uses have also been greatly expanded. As mentioned, the enhanced version is sometimes referred to as NIO.2. Because the features added by NIO.2 are so substantial, they have changed the way that much NIO-based code is written and have increased the types of tasks to which NIO can be applied. Because of its importance, most of the remaining discussion and examples in this chapter utilize NIO.2 features and, therefore, require JDK 7, JDK 8, or later. However, at the end of the chapter is a brief description of pre-JDK 7 code. This will be of aid to those programmers working in pre-JDK 7 environments or maintaining older code.

---

**REMEMBER** Most of the examples in this chapter require JDK 7 or later.

In the past, the primary purpose of NIO was channel-based I/O, and this is still a very important use. However, you can now use NIO for stream-based I/O and for performing file-system operations. As a result, the discussion of using NIO is divided into three parts:

- Using NIO for channel-based I/O
- Using NIO for stream-based I/O
- Using NIO for path and file system operations

Because the most common I/O device is the disk file, the rest of this chapter uses disk files in the examples. Because all file channel operations are byte-based, the type of buffers that we will be using are of type **ByteBuffer**.

Before you can open a file for access via the NIO system, you must obtain a **Path** that describes the file. One way to do this is to call the **Paths.get( )** factory method, which was described earlier. The form of **get( )** used in the examples is shown here:

static Path get(String *pathname*, String ... *parts*)

Recall that the path can be specified in two ways. It can be passed in pieces, with the first part passed in *pathname* and the subsequent elements specified by the *parts* varargs parameter. Alternatively, the entire path can be specified in *pathname* and *parts* is not used. This is the approach used by the examples.

### Use NIO for Channel-Based I/O

An important use of NIO is to access a file via a channel and buffers. The following sections demonstrate some techniques that use a channel to read from and write to a file.

## Reading a File via a Channel

There are several ways to read data from a file using a channel. Perhaps the most common way is to manually allocate a buffer and then perform an explicit read operation that loads that buffer with data from the file. It is with this approach that we begin.

Before you can read from a file, you must open it. To do this, first create a **Path** that describes the file. Then use this **Path** to open the file. There are various ways to open the file depending on how it will be used. In this example, the file will be opened for byte-based input via explicit input operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel( )**. The **newByteChannel( )** method has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. The **Path** that describes the file is passed in *path*. The *how* parameter specifies how the file will be opened. Because it is a varargs parameter, you can specify zero or more comma-separated arguments. (The valid values were discussed earlier and shown in Table 21-5.) If no arguments are specified, the file is opened for input operations. **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel** class. When the default file system is used, the returned object can be cast to **FileChannel**. You must close the channel after you have finished with it. Since all channels, including **FileChannel**, implement **AutoCloseable**, you can use a **try**-with-resources statement to close the file automatically instead of calling **close( )** explicitly. This approach is used in the examples.

Next, you must obtain a buffer that will be used by the channel either by wrapping an existing array or by allocating the buffer dynamically. The examples use allocation, but the choice is yours. Because file channels operate on byte buffers, we will use the **allocate( )** method defined by **ByteBuffer** to obtain the buffer. It has this general form:

```
static ByteBuffer allocate(int cap)
```

Here, *cap* specifies the capacity of the buffer. A reference to the buffer is returned.

After you have created the buffer, call **read( )** on the channel, passing a reference to the buffer. The version of **read( )** that we will use is shown next:

```
int read(ByteBuffer buf) throws IOException
```

Each time it is called, **read( )** fills the buffer specified by *buf* with data from the file. The reads are sequential, meaning that each call to **read( )** reads the next buffer's worth of bytes from the file. The **read( )** method returns the number of bytes actually read. It returns –1 when there is an attempt to read at the end of the file.

The following program puts the preceding discussion into action by reading a file called **test.txt** through a channel using explicit input operations:

```
// Use Channel I/O to read a file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;
```

```
public class ExplicitChannelRead {
  public static void main(String args[]) {
    int count;
    Path filepath = null;

    // First, obtain a path to the file.
    try {
      filepath = Paths.get("test.txt");
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
      return;
    }

    // Next, obtain a channel to that file within a try-with-resources block.
    try ( SeekableByteChannel fChan = Files.newByteChannel(filepath) )
    {

      // Allocate a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen as characters.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);

      System.out.println();
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

Here is how the program works. First, a **Path** object is obtained that contains the relative path to a file called **test.txt**. A reference to this object is assigned to **filepath**. Next, a channel connected to the file is obtained by calling **newByteChannel( )**, passing in **filepath**. Because no open option is specified, the file is opened for reading. Notice that this channel is the object managed by the **try**-with-resources statement. Thus, the channel is automatically closed when the block ends. The program then calls the **allocate( )** method of **ByteBuffer** to allocate a buffer that will hold the contents of the file when it is read. A reference to this buffer is stored in **mBuf**. The contents of the file are then read, one buffer at a time, into **mBuf** through a call to **read( )**. The number of bytes read is stored in **count**. Next, the

buffer is rewound through a call to **rewind( )**. This call is necessary because the current position is at the end of the buffer after the call to **read( )**. It must be reset to the start of the buffer in order for the bytes in **mBuf** to be read by calling **get( )**. (Recall that **get( )** is defined by **ByteBuffer**.) Because **mBuf** is a byte buffer, the values returned by **get( )** are bytes. They are cast to **char** so the file can be displayed as text. (Alternatively, it is possible to create a buffer that encodes the bytes into characters and then read that buffer.) When the end of the file has been reached, the value returned by **read( )** will be –1. When this occurs, the program ends, and the channel is automatically closed.

As a point of interest, notice that the program obtains the **Path** within one **try** block and then uses another **try** block to obtain and manage a channel linked to that path. Although there is nothing wrong, per se, with this approach, in many cases, it can be streamlined so that only one **try** block is needed. In this approach, the calls to **Paths.get( )** and **newByteChannel( )** are sequenced together. For example, here is a reworked version of the program that uses this approach:



```
// A more compact way to open a channel. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Here, the channel is opened on the Path returned by Paths.get().
    // There is no need for the filepath variable.
    try ( SeekableByteChannel fChan =
            Files.newByteChannel(Paths.get("test.txt")) )
    {
      // Allocate a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(128);

      do {
        // Read a buffer.
        count = fChan.read(mBuf);

        // Stop when end of file is reached.
        if(count != -1) {

          // Rewind the buffer so that it can be read.
          mBuf.rewind();

          // Read bytes from the buffer and show
          // them on the screen as characters.
          for(int i=0; i < count; i++)
            System.out.print((char)mBuf.get());
        }
      } while(count != -1);
```

```
      System.out.println();
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

In this version, the variable **filepath** is not needed and both exceptions are handled by the same **try** statement. Because this approach is more compact, it is the approach used in the rest of the examples in this chapter. Of course, in your own code, you may encounter situations in which the creation of a **Path** object needs to be separate from the acquisition of a channel. In these cases, the previous approach can be used.

Another way to read a file is to map it to a buffer. The advantage is that the buffer automatically contains the contents of the file. No explicit read operation is necessary. To map and read the contents of a file, follow this general procedure. First, obtain a **Path** object that encapsulates the file as previously described. Next, obtain a channel to that file by calling **Files.newByteChannel( )**, passing in the **Path** and casting the returned object to **FileChannel**. As explained, **newByteChannel( )** returns a **SeekableByteChannel**. When using the default file system, this object can be cast to **FileChannel**. Then, map the channel to a buffer by calling **map( )** on the channel. The **map( )** method is defined by **FileChannel**. This is why the cast to **FileChannel** is needed. The **map( )** function is shown here:

```
MappedByteBuffer map(FileChannel.MapMode how,
                     long pos, long size) throws IOException
```

The **map( )** method causes the data in the file to be mapped into a buffer in memory. The value in *how* determines what type of operations are allowed. It must be one of these values:

| MapMode.READ_ONLY | MapMode.READ_WRITE | MapMode.PRIVATE |
|---|---|---|

For reading a file, use **MapMode.READ_ONLY**. To read and write, use **MapMode.READ_WRITE**. **MapMode.PRIVATE** causes a private copy of the file to be made, and changes to the buffer do not affect the underlying file. The location within the file to begin mapping is specified by *pos*, and the number of bytes to map are specified by *size*. A reference to this buffer is returned as a **MappedByteBuffer**, which is a subclass of **ByteBuffer**. Once the file has been mapped to a buffer, you can read the file from that buffer. Here is an example that illustrates this approach:

```
// Use a mapped file to read a file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

```
    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan =
          (FileChannel) Files.newByteChannel(Paths.get("test.txt")) )
    {

      // Get the size of the file.
      long fSize = fChan.size();

      // Now, map the file into a buffer.
      MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_ONLY, 0, fSize);

      // Read and display bytes from buffer.
      for(int i=0; i < fSize; i++)
        System.out.print((char)mBuf.get());

      System.out.println();

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

In the program, a **Path** to the file is created and then opened via **newByteChannel( )**. The channel is cast to **FileChannel** and stored in **fChan**. Next, the size of the file is obtained by calling **size( )** on the channel. Then, the entire file is mapped into memory by calling **map( )** on **fChan** and a reference to the buffer is stored in **mBuf**. Notice that **mBuf** is declared as a reference to a **MappedByteBuffer**. The bytes in **mBuf** are read by calling **get( )**.

## Writing to a File via a Channel

As is the case when reading from a file, there are also several ways to write data to a file using a channel. We will begin with one of the most common. In this approach, you manually allocate a buffer, write data to that buffer, and then perform an explicit write operation to write that data to a file.

Before you can write to a file, you must open it. To do this, first obtain a **Path** that describes the file and then use this **Path** to open the file. In this example, the file will be opened for byte-based output via explicit output operations. Therefore, this example will open the file and establish a channel to it by calling **Files.newByteChannel( )**. As shown in the previous section, the **newByteChannel( )** method has this general form:

```
static SeekableByteChannel newByteChannel(Path path, OpenOption ... how)
    throws IOException
```

It returns a **SeekableByteChannel** object, which encapsulates the channel for file operations. To open a file for output, the *how* parameter must specify **StandardOpenOption.WRITE**. If you want to create the file if it does not already exist, then you must also specify **StandardOpenOption.CREATE**. (Other options, which are shown in Table 21-5, are also available.) As explained in the previous section, **SeekableByteChannel** is an interface that describes a channel that can be used for file operations. It is implemented by the **FileChannel**

class. When the default file system is used, the return object can be cast to **FileChannel**. You must close the channel after you have finished with it.

Here is one way to write to a file through a channel using explicit calls to **write( )**. First, obtain a **Path** to the file and then open it with a call to **newByteChannel( )**, casting the result to **FileChannel**. Next, allocate a byte buffer and write data to that buffer. Before the data is written to the file, call **rewind( )** on the buffer to set its current position to zero. (Each output operation on the buffer increases the current position. Thus, it must be reset prior to writing to the file.) Then, call **write( )** on the channel, passing in the buffer. The following program demonstrates this procedure. It writes the alphabet to a file called **test.txt**.

```
// Write to a file using NIO. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan = (FileChannel)
            Files.newByteChannel(Paths.get("test.txt"),
                                 StandardOpenOption.WRITE,
                                 StandardOpenOption.CREATE) )
    {
      // Create a buffer.
      ByteBuffer mBuf = ByteBuffer.allocate(26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

      // Reset the buffer so that it can be written.
      mBuf.rewind();

      // Write the buffer to the output file.
      fChan.write(mBuf);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
      System.exit(1);
    }
  }
}
```

It is useful to emphasize an important aspect of this program. As mentioned, after data is written to **mBuf**, but before it is written to the file, a call to **rewind( )** on **mBuf** is made. This is necessary in order to reset the current position to zero after data has been written to **mBuf**. Remember, each call to **put( )** on **mBuf** advances the current position. Therefore,

it is necessary for the current position to be reset to the start of the buffer before calling **write( )**. If this is not done, **write( )** will think that there is no data in the buffer.

Another way to handle the resetting of the buffer between input and output operations is to call **flip( )** instead of **rewind( )**. The **flip( )** method sets the value of the current position to zero and the limit to the previous current position. In the preceding example, because the capacity of the buffer equals its limit, **flip( )** could have been used instead of **rewind( )**. However, the two methods are not interchangeable in all cases.

In general, you must reset the buffer between read and write operations. For example, assuming the preceding example, the following loop will write the alphabet to the file three times. Pay special attention to the calls to **rewind( )**.

```
for(int h=0; h<3; h++) {
  // Write some bytes to the buffer.
  for(int i=0; i<26; i++)
    mBuf.put((byte)('A' + i));

  // Rewind the buffer so that it can be written.
  mBuf.rewind();

  // Write the buffer to the output file.
  fChan.write(mBuf);

  // Rewind the buffer so that it can be written to again.
  mBuf.rewind();
}
```

Notice that **rewind( )** is called between each read and write operation.

One other thing about the program warrants mentioning: When the buffer is written to the file, the first 26 bytes in the file will contain the output. If the file **test.txt** was preexisting, then after the program executes, the first 26 bytes of **test.txt** will contain the alphabet, but the remainder of the file will remain unchanged.

Another way to write to a file is to map it to a buffer. The advantage to this approach is that the data written to the buffer will automatically be written to the file. No explicit write operation is necessary. To map and write the contents of a file, we will use this general procedure. First, obtain a **Path** object that encapsulates the file and then create a channel to that file by calling **Files.newByteChannel( )**, passing in the **Path**. Cast the reference returned by **newByteChannel( )** to **FileChannel**. Next, map the channel to a buffer by calling **map( )** on the channel. The **map( )** method was described in detail in the previous section. It is summarized here for your convenience. Here is its general form:

```
MappedByteBuffer map(FileChannel.MapMode how,
                     long pos, long size) throws IOException
```

The **map( )** method causes the data in the file to be mapped into a buffer in memory. The value in *how* determines what type of operations are allowed. For writing to a file, *how* must be **MapMode.READ_WRITE**. The location within the file to begin mapping is specified by *pos*, and the number of bytes to map are specified by *size*. A reference to this buffer is returned. Once the file has been mapped to a buffer, you can write data to that buffer, and it will automatically be written to the file. Therefore, no explicit write operations to the channel are necessary.

Here is the preceding program reworked so that a mapped file is used. Notice that in the call to **newByteChannel( )**, the open option **StandardOpenOption.READ** has been added. This is because a mapped buffer can either be read-only or read/write. Thus, to write to the mapped buffer, the channel must be opened as read/write.

```
// Write to a mapped file. Requires JDK 7 or later.

import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

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

    // Obtain a channel to a file within a try-with-resources block.
    try ( FileChannel fChan = (FileChannel)
          Files.newByteChannel(Paths.get("test.txt"),
                   StandardOpenOption.WRITE,
                   StandardOpenOption.READ,
                   StandardOpenOption.CREATE) )
    {

      // Then, map the file into a buffer.
      MappedByteBuffer mBuf = fChan.map(FileChannel.MapMode.READ_WRITE, 0, 26);

      // Write some bytes to the buffer.
      for(int i=0; i<26; i++)
        mBuf.put((byte)('A' + i));

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

As you can see, there are no explicit write operations to the channel itself. Because **mBuf** is mapped to the file, changes to **mBuf** are automatically reflected in the underlying file.

## Copying a File Using NIO

NIO simplifies several types of file operations. Although we can't examine them all, an example will give you an idea of what is available. The following program copies a file using a call to a single NIO method: **copy( )**, which is a **static** method defined by **Files**. It has several forms. Here is the one we will be using:

static Path copy(Path *src*, Path *dest*, CopyOption ... *how*) throws IOException

The file specified by *src* is copied to the file specified by *dest*. How the copy is performed is specified by *how*. Because it is a varargs parameter, it can be missing. If specified, it can be one or more of these values, which are valid for all file systems:

| | |
|---|---|
| StandardCopyOption.COPY_ATTRIBUTES | Request that the file's attributes be copied. |
| StandardLinkOption.NOFOLLOW_LINKS | Do not follow symbolic links. |
| StandardCopyOption.REPLACE_EXISTING | Overwrite a preexisting file. |

Other options may be supported, depending on the implementation.

The following program demonstrates **copy( )**. The source and destination files are specified on the command line, with the source file specified first. Notice how short the program is. You might want to compare this version of the file copy program to the one found in Chapter 13. As you will find, the part of the program that actually copies the file is substantially shorter in the NIO version shown here.



```
// Copy a file using NIO. Requires JDK 7 or later.
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class NIOCopy {

  public static void main(String args[]) {

    if(args.length != 2) {
      System.out.println("Usage: Copy from to");
      return;
    }

    try {
      Path source = Paths.get(args[0]);
      Path target = Paths.get(args[1]);

      // Copy the file.
      Files.copy(source, target, StandardCopyOption.REPLACE_EXISTING);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch (IOException e) {
      System.out.println("I/O Error " + e);
    }
  }
}
```

## Use NIO for Stream-Based I/O

Beginning with NIO.2, you can use NIO to open an I/O stream. Once you have a **Path**, open a file by calling **newInputStream( )** or **newOutputStream( )**, which are **static** methods defined by **Files**. These methods return a stream connected to the specified file. In either case, the stream can then be operated on in the way described in Chapter 20, and the same techniques apply. The advantage of using **Path** to open a file is that all of the features defined by NIO are available for your use.

To open a file for stream-based input, use **Files.newInputStream( )**. It is shown here:

static InputStream newInputStream(Path *path*, OpenOption ... *how*)
throws IOException

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an input stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.READ** were passed.

Once opened, you can use any of the methods defined by **InputStream**. For example, you can use **read( )** to read bytes from the file.

The following program demonstrates the use of NIO-based stream I/O. It reworks the **ShowFile** program from Chapter 13 so that it uses NIO features to open the file and obtain a stream. As you can see, it is very similar to the original, except for the use of **Path** and **newInputStream( )**.

```
/* Display a text file using stream-based, NIO code.
   Requires JDK 7 or later.

   To use this program, specify the name
   of the file that you want to see.
   For example, to see a file called TEST.TXT,
   use the following command line.

   java ShowFile TEST.TXT
*/

import java.io.*;
import java.nio.file.*;

class ShowFile {
  public static void main(String args[])
  {
    int i;

    // First, confirm that a filename has been specified.
    if(args.length != 1) {
      System.out.println("Usage: ShowFile filename");
      return;
    }

    // Open the file and obtain a stream linked to it.
    try ( InputStream fin = Files.newInputStream(Paths.get(args[0])) )
    {
      do {
        i = fin.read();
        if(i != -1) System.out.print((char) i);
      } while(i != -1);

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(IOException e) {
```

```
      System.out.println("I/O Error "  + e);
    }
  }
}
```

Because the stream returned by **newInputStream( )** is a normal stream, it can be used like any other stream. For example, you can wrap the stream inside a buffered stream, such as a **BufferedInputStream**, to provide buffering, as shown here:

```
new BufferedInputStream(Files.newInputStream(Paths.get(args[0])))
```

Now, all reads will be automatically buffered.

To open a file for output, use **Files.newOutputStream( )**. It is shown here:

static OutputStream newOutputStream(Path *path*, OpenOption ... *how*)
throws IOException

Here, *path* specifies the file to open and *how* specifies how the file will be opened. It must be one or more of the values defined by **StandardOpenOption**, described earlier. (Of course, only those options that relate to an output stream will apply.) If no options are specified, then the file is opened as if **StandardOpenOption.WRITE**, **StandardOpenOption.CREATE**, and **StandardOpenOption.TRUNCATE_EXISTING** were passed.

The methodology for using **newOutputStream( )** is similar to that shown previously for **newInputStream( )**. Once opened, you can use any of the methods defined by **OutputStream**. For example, you can use **write( )** to write bytes to the file. You can also wrap the stream inside a **BufferedOutputStream** to buffer the stream.

The following program shows **newOutputStream( )** in action. It writes the alphabet to a file called **test.txt**. Notice the use of buffered I/O.

```
// Demonstrate NIO-based, stream output. Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;

class NIOStreamWrite {
  public static void main(String args[])
  {
    // Open the file and obtain a stream linked to it.
    try ( OutputStream fout =
          new BufferedOutputStream(
                Files.newOutputStream(Paths.get("test.txt"))) )
    {
      // Write some bytes to the stream.
      for(int i=0; i < 26; i++)
        fout.write((byte)('A' + i));

    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

## Use NIO for Path and File System Operations

At the beginning of Chapter 20, the **File** class in the **java.io** package was examined. As explained there, the **File** class deals with the file system and with the various attributes associated with a file, such as whether a file is read-only, hidden, and so on. It was also used to obtain information about a file's path. Although the **File** class is still perfectly acceptable, the interfaces and classes defined by NIO.2 offer a better way to perform these functions. The benefits include support for symbolic links, better support for directory tree traversal, and improved handling of metadata, among others. The following sections show samples of two common file system operations: obtaining information about a path and file and getting the contents of a directory.

---

**REMEMBER** If you want to update older code that uses **java.io.File** to the new **Path** interface, you can use the **toPath( )** method to obtain a **Path** instance from a **File** instance.

### Obtain Information About a Path and a File

Information about a path can be obtained by using methods defined by **Path**. Some attributes associated with the file described by a **Path** (such as whether or not the file is hidden) are obtained by using methods defined by **Files**. The **Path** methods used here are **getName( )**, **getParent( )**, and **toAbsolutePath( )**. Those provided by **Files** are **isExecutable( )**, **isHidden( )**, **isReadable( )**, **isWritable( )**, and **exists( )**. These are summarized in Tables 21-3 and 21-4, shown earlier.

---

**CAUTION** Methods such as **isExecutable( ), isReadable( ), isWritable( ),** and **exists( )** must be used with care because the state of the file system may change after the call, in which case a program malfunction could occur. Such a situation could have security implications.

Other file attributes are obtained by requesting a list of attributes by calling **Files.readAttributes( )**. In the program, this method is called to obtain the **BasicFileAttributes** associated with a file, but the general approach applies to other types of attributes.

The following program demonstrates several of the **Path** and **Files** methods, along with several methods provided by **BasicFileAttributes**. This program assumes that a file called **test.txt** exists in a directory called **examples**, which must be a subdirectory of the current directory.

```
// Obtain information about a path and a file.
// Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;

class PathDemo {
  public static void main(String args[]) {
    Path filepath = Paths.get("examples\\test.txt");
```

```
    System.out.println("File Name: " + filepath.getName(1));
    System.out.println("Path: " + filepath);
    System.out.println("Absolute Path: " + filepath.toAbsolutePath());
    System.out.println("Parent: " + filepath.getParent());

    if(Files.exists(filepath))
      System.out.println("File exists");
    else
      System.out.println("File does not exist");

    try {
      if(Files.isHidden(filepath))
        System.out.println("File is hidden");
      else
        System.out.println("File is not hidden");
    } catch(IOException e) {
        System.out.println("I/O Error: " + e);
    }

    Files.isWritable(filepath);
    System.out.println("File is writable");


    Files.isReadable(filepath);
    System.out.println("File is readable");

    try {
      BasicFileAttributes attribs =
        Files.readAttributes(filepath, BasicFileAttributes.class);

      if(attribs.isDirectory())
        System.out.println("The file is a directory");
      else
        System.out.println("The file is not a directory");

      if(attribs.isRegularFile())
        System.out.println("The file is a normal file");
      else
        System.out.println("The file is not a normal file");

      if(attribs.isSymbolicLink())
        System.out.println("The file is a symbolic link");
      else
        System.out.println("The file is not a symbolic link");

      System.out.println("File last modified: " + attribs.lastModifiedTime());
      System.out.println("File size: " + attribs.size() + " Bytes");
    } catch(IOException e) {
      System.out.println("Error reading attributes: " + e);
    }
  }
}
```

If you execute this program from a directory called **MyDir**, which has a subdirectory called **examples**, and the **examples** directory contains the **test.txt** file, then you will see output similar to that shown here. (Of course, the information you see will differ.)

```
File Name: test.txt
Path: examples\test.txt
Absolute Path: C:\MyDir\examples\test.txt
Parent: examples
File exists
File is not hidden
File is writable
File is readable
The file is not a directory
The file is a normal file
The file is not a symbolic link
File last modified: 2014-01-01T18:20:46.380445Z
File size: 18 Bytes
```

If you are using a computer that supports the FAT file system (i.e., the DOS file system), then you might want to try using the methods defined by **DosFileAttributes**. If you are using a POSIX-compatible system, then try using **PosixFileAttributes**.

## List the Contents of a Directory

If a path describes a directory, then you can read the contents of that directory by using **static** methods defined by **Files**. To do this, you first obtain a directory stream by calling **newDirectoryStream( )**, passing in a **Path** that describes the directory. One form of **newDirectoryStream( )** is shown here:

```
static DirectoryStream<Path> newDirectoryStream(Path dirPath)
                throws IOException
```

Here, *dirPath* encapsulates the path to the directory. The method returns a **DirectoryStream<Path>** object that can be used to obtain the contents of the directory. It will throw an **IOException** if an I/O error occurs and a **NotDirectoryException** (which is a subclass of **IOException**) if the specified path is not a directory. A **SecurityException** is also possible if access to the directory is not permitted.

**DirectoryStream<Path>** implements **AutoCloseable**, so it can be managed by a **try**-with-resources statement. It also implements **Iterable<Path>**. This means that you can obtain the contents of the directory by iterating over the **DirectoryStream** object. When iterating, each directory entry is represented by a **Path** instance. An easy way to iterate over a **DirectoryStream** is to use a for-each style **for** loop. It is important to understand, however, that the iterator implemented by **DirectoryStream<Path>** can be obtained only once for each instance. Thus, the **iterator( )** method can be called only once, and a for-each loop can be executed only once.

The following program displays the contents of a directory called **MyDir**:

```
// Display a directory. Requires JDK 7 or later.

import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
```

```
class DirList {
  public static void main(String args[]) {
    String dirname = "\\MyDir";

    // Obtain and manage a directory stream within a try block.
    try ( DirectoryStream<Path> dirstrm =
            Files.newDirectoryStream(Paths.get(dirname)) )
    {
      System.out.println("Directory of " + dirname);

      // Because DirectoryStream implements Iterable, we
      // can use a "foreach" loop to display the directory.
      for(Path entry : dirstrm) {
        BasicFileAttributes attribs =
             Files.readAttributes(entry, BasicFileAttributes.class);

        if(attribs.isDirectory())
          System.out.print("<DIR> ");
        else
          System.out.print("      ");

        System.out.println(entry.getName(1));
      }
    } catch(InvalidPathException e) {
      System.out.println("Path Error " + e);
    } catch(NotDirectoryException e) {
      System.out.println(dirname + " is not a directory.");
    } catch (IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is sample output from the program:

```
Directory of \MyDir
      DirList.class
      DirList.java
<DIR> examples
      Test.txt
```

You can filter the contents of a directory in two ways. The easiest is to use this version of **newDirectoryStream( )**:

static DirectoryStream<Path> newDirectoryStream(Path *dirPath*, String *wildcard*)
throws IOException

In this version, only files that match the wildcard filename specified by *wildcard* will be obtained. For *wildcard*, you can specify either a complete filename or a *glob*. A *glob* is a string that defines a general pattern that will match one or more files using the familiar * and ? wildcard characters. These match zero or more of any character and any one character, respectively. The following are also recognized within a glob.