CHAPTER

20 Input/Output: Exploring java.io

This chapter explores **java.io**, which provides support for I/O operations. Chapter 13 presented an overview of Java's I/O system, including basic techniques for reading and writing files, handling I/O exceptions, and closing a file. Here, we will examine the Java I/O system in greater detail.

As all programmers learn early on, most programs cannot accomplish their goals without accessing external data. Data is retrieved from an *input* source. The results of a program are sent to an *output* destination. In Java, these sources or destinations are defined very broadly. For example, a network connection, memory buffer, or disk file can be manipulated by the Java I/O classes. Although physically different, these devices are all handled by the same abstraction: the *stream*. An I/O stream, as explained in Chapter 13, is a logical entity that either produces or consumes information. An I/O stream is linked to a physical device by the Java I/O system. All I/O streams behave in the same manner, even if the actual physical devices they are linked to differ.

NOTE The stream-based I/O system packaged in **java.io** and described in this chapter has been part of Java since its original release and is widely used. However, beginning with version 1.4, a second I/O system was added to Java. It is called NIO (which was originally an acronym for New I/O). NIO is packaged in **java.nio** and its subpackages. The NIO system is described in Chapter 21.

NOTE It is important not to confuse the I/O streams used by the I/O system discussed here with the new stream API added by JDK 8. Although conceptually related, they are two different things. Therefore, when the term *stream* is used in this chapter, it refers to an I/O stream.

The I/O Classes and Interfaces

The I/O classes defined by **java.io** are listed here:

BufferedInputStream	FileWriter	PipedOutputStream
BufferedOutputStream	FilterInputStream	PipedReader

BufferedReader	FilterOutputStream	PipedWriter
BufferedWriter	FilterReader	PrintStream
ByteArrayInputStream	FilterWriter	PrintWriter
ByteArrayOutputStream	InputStream	PushbackInputStream
CharArrayReader	InputStreamReader	PushbackReader
CharArrayWriter	LineNumberReader	RandomAccessFile
Console	ObjectInputStream	Reader
DataInputStream	ObjectInputStream.GetField	SequenceInputStream
DataOutputStream	ObjectOutputStream	SerializablePermission
File	ObjectOutputStream.PutField	StreamTokenizer
FileDescriptor	ObjectStreamClass	StringReader
FileInputStream	ObjectStreamField	StringWriter
FileOutputStream	OutputStream	Writer
FilePermission	OutputStreamWriter	
FileReader	PipedInputStream	

The **java.io** package also contains two deprecated classes that are not shown in the preceding table: **LineNumberInputStream** and **StringBufferInputStream**. These classes should not be used for new code.

The following interfaces are defined by **java.io**:

Closeable	FileFilter	ObjectInputValidation
DataInput	FilenameFilter	ObjectOutput
DataOutput	Flushable	ObjectStreamConstants
Externalizable	ObjectInput	Serializable

As you can see, there are many classes and interfaces in the **java.io** package. These include byte and character streams, and object serialization (the storage and retrieval of objects). This chapter examines several commonly used I/O components. We begin our discussion with one of the most distinctive I/O classes: **File**.

File

Although most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

NOTE The **Path** interface and **Files** class, which are part of the NIO system, offer a powerful alternative to **File** in many cases. See Chapter 21 for details.

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a **File** with one additional property—a list of filenames that can be examined by the **list( )** method.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

Here, *directoryPath* is the path name of the file; *filename* is the name of the file or subdirectory; *dirObj* is a **File** object that specifies a directory; and *uriObj* is a **URI** object that describes a file.

The following example creates three files: **f1**, **f2**, and **f3**. The first **File** object is constructed with a directory path as the only argument. The second includes two arguments—the path and the filename. The third includes the file path assigned to **f1** and a filename; **f3** refers to the same file as **f2**.

```
File f1 = new File("/");
File f2 = new File("/","autoexec.bat");
File f3 = new File(f1,"autoexec.bat");
```

NOTE Java does the right thing with path separators between UNIX and Windows conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly. Remember, if you are using the Windows convention of a backslash character (\), you will need to use its escape sequence (\\) within a string.

File defines many methods that obtain the standard properties of a **File** object. For example, **getName( )** returns the name of the file; **getParent( )** returns the name of the parent directory; and **exists( )** returns **true** if the file exists, **false** if it does not. The following example demonstrates several of the **File** methods. It assumes that a directory called **java** exists off the root directory and that it contains a file called **COPYRIGHT**.

```
// Demonstrate File.
import java.io.File;

class FileDemo {
  static void p(String s) {
    System.out.println(s);
  }

  public static void main(String args[]) {
    File f1 = new File("/java/COPYRIGHT");

    p("File Name: " + f1.getName());
    p("Path: " + f1.getPath());
    p("Abs Path: " + f1.getAbsolutePath());
    p("Parent: " + f1.getParent());
    p(f1.exists() ? "exists" : "does not exist");
    p(f1.canWrite() ? "is writeable" : "is not writeable");
    p(f1.canRead() ? "is readable" : "is not readable");
    p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
    p(f1.isFile() ? "is normal file" : "might be a named pipe");
    p(f1.isAbsolute() ? "is absolute" : "is not absolute");
    p("File last modified: " + f1.lastModified());
    p("File size: " + f1.length() + " Bytes");
  }
}
```

This program will produce output similar to this:

```
File Name: COPYRIGHT
Path: \java\COPYRIGHT
Abs Path: C:\java\COPYRIGHT
Parent: \java
exists
is writeable
is readable
is not a directory
is normal file
is not absolute
File last modified: 1282832030047
File size: 695 Bytes
```

Most of the **File** methods are self-explanatory. **isFile( )** and **isAbsolute( )** are not. **isFile( )** returns **true** if called on a file and **false** if called on a directory. Also, **isFile( )** returns **false** for some special files, such as device drivers and named pipes, so this method can be used to make sure the file will behave as a file. The **isAbsolute( )** method returns **true** if the file has an absolute path and **false** if its path is relative.

File includes two useful utility methods of special interest. The first is **renameTo( )**, shown here:

boolean renameTo(File *newName*)

Here, the filename specified by *newName* becomes the new name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed (if you attempt to rename a file so that it uses an existing filename, for example).

The second utility method is **delete( )**, which deletes the disk file represented by the path of the invoking **File** object. It is shown here:

boolean delete( )

You can also use **delete( )** to delete a directory if the directory is empty. **delete( )** returns **true** if it deletes the file and **false** if the file cannot be removed.

Here are some other **File** methods that you will find helpful:

Method	Description
void deleteOnExit( )	Removes the file associated with the invoking object when the Java Virtual Machine terminates.
long getFreeSpace( )	Returns the number of free bytes of storage available on the partition associated with the invoking object.
long getTotalSpace( )	Returns the storage capacity of the partition associated with the invoking object.
long getUsableSpace( )	Returns the number of usable free bytes of storage available on the partition associated with the invoking object.
boolean isHidden( )	Returns **true** if the invoking file is hidden. Returns **false** otherwise.
boolean setLastModified(long *millisec*)	Sets the time stamp on the invoking file to that specified by *millisec*, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC).
boolean setReadOnly( )	Sets the invoking file to read-only.

Methods also exist to mark files as readable, writable, and executable. Because **File** implements the **Comparable** interface, the method **compareTo( )** is also supported.

JDK 7 added a method to **File** called **toPath( )**, which is shown here:

Path toPath( )

toPath( ) returns a **Path** object that represents the file encapsulated by the invoking **File** object. (In other words, **toPath( )** converts a **File** into a **Path**.) **Path** is packaged in **java.nio.file** and is part of NIO. Thus, **toPath( )** forms a bridge between the older **File** class and the newer **Path** interface. (See Chapter 21 for a discussion of **Path**.)

Directories

A directory is a **File** that contains a list of other files and directories. When you create a **File** object that is a directory, the **isDirectory( )** method will return **true**. In this case, you can call **list( )** on that object to extract the list of other files and directories inside. It has two forms. The first is shown here:

String[ ] list( )

The list of files is returned in an array of **String** objects.

The program shown here illustrates how to use **list( )** to examine the contents of a directory:

```
// Using directories.
import java.io.File;

class DirList {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    if (f1.isDirectory()) {
      System.out.println("Directory of " + dirname);
      String s[] = f1.list();

      for (int i=0; i < s.length; i++) {
        File f = new File(dirname + "/" + s[i]);
        if (f.isDirectory()) {
          System.out.println(s[i] + " is a directory");
        } else {
          System.out.println(s[i] + " is a file");
        }
      }
    } else {
      System.out.println(dirname + " is not a directory");
    }
  }
}
```

Here is sample output from the program. (Of course, the output you see will be different, based on what is in the directory.)

```
Directory of /java
bin is a directory
lib is a directory
demo is a directory
COPYRIGHT is a file
README is a file
index.html is a file
include is a directory
src.zip is a file
src is a directory
```

Using FilenameFilter

You will often want to limit the number of files returned by the **list( )** method to include only those files that match a certain filename pattern, or *filter*. To do this, you must use a second form of **list( )**, shown here:

String[ ] list(FilenameFilter *FFObj*)

In this form, *FFObj* is an object of a class that implements the **FilenameFilter** interface.

FilenameFilter defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *directory*, String *filename*)

The **accept( )** method returns **true** for files in the directory specified by *directory* that should be included in the list (that is, those that match the *filename* argument) and returns **false** for those files that should be excluded.

The **OnlyExt** class, shown next, implements **FilenameFilter**. It will be used to modify the preceding program to restrict the visibility of the filenames returned by **list( )** to files with names that end in the file extension specified when the object is constructed.

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
  String ext;

  public OnlyExt(String ext) {
    this.ext = "." + ext;
  }

  public boolean accept(File dir, String name) {
    return name.endsWith(ext);
  }
}
```

The modified directory listing program is shown here. Now it will only display files that use the **.html** extension.

```
// Directory of .HTML files.
import java.io.*;

class DirListOnly {
  public static void main(String args[]) {
    String dirname = "/java";
    File f1 = new File(dirname);
    FilenameFilter only = new OnlyExt("html");
    String s[] = f1.list(only);

    for (int i=0; i < s.length; i++) {
      System.out.println(s[i]);
    }
  }
}
```

The listFiles( ) Alternative

There is a variation to the **list( )** method, called **listFiles( )**, which you might find useful. The signatures for **listFiles( )** are shown here:

File[ ] listFiles( )
File[ ] listFiles(FilenameFilter *FFObj*)
File[ ] listFiles(FileFilter *FObj*)

These methods return the file list as an array of **File** objects instead of strings. The first method returns all files, and the second returns those files that satisfy the specified **FilenameFilter**. Aside from returning an array of **File** objects, these two versions of **listFiles( )** work like their equivalent **list( )** methods.

The third version of **listFiles( )** returns those files with path names that satisfy the specified **FileFilter**. **FileFilter** defines only a single method, **accept( )**, which is called once for each file in a list. Its general form is given here:

boolean accept(File *path*)

The **accept( )** method returns **true** for files that should be included in the list (that is, those that match the *path* argument) and **false** for those that should be excluded.

Creating Directories

Another two useful **File** utility methods are **mkdir( )** and **mkdirs( )**. The **mkdir( )** method creates a directory, returning **true** on success and **false** on failure. Failure can occur for various reasons, such as the path specified in the **File** object already exists, or the directory cannot be created because the entire path does not exist yet. To create a directory for which no path exists, use the **mkdirs( )** method. It creates both a directory and all the parents of the directory.

The AutoCloseable, Closeable, and Flushable Interfaces

There are three interfaces that are quite important to the stream classes. Two are **Closeable** and **Flushable**. They are defined in **java.io** and were added by JDK 5. The third, **AutoCloseable**, was added by JDK 7. It is packaged in **java.lang**.

AutoCloseable provides support for the **try**-with-resources statement, which automates the process of closing a resource. (See Chapter 13.) Only objects of classes that implement **AutoCloseable** can be managed by **try**-with-resources. **AutoCloseable** is discussed in Chapter 17, but it is reviewed here for convenience. The **AutoCloseable** interface defines only the **close( )** method:

void close( ) throws Exception

This method closes the invoking object, releasing any resources that it may hold. It is called automatically at the end of a **try**-with-resources statement, thus eliminating the need to explicitly call **close( )**. Because this interface is implemented by all of the I/O classes that open a stream, all such streams can be automatically closed by a **try**-with-resources statement. Automatically closing a stream ensures that it is properly closed when it is no longer needed, thus preventing memory leaks and other problems.

The **Closeable** interface also defines the **close( )** method. Objects of a class that implement **Closeable** can be closed. Beginning with JDK 7, **Closeable** extends **AutoCloseable**. Therefore, any class that implements **Closeable** also implements **AutoCloseable**.

Objects of a class that implements **Flushable** can force buffered output to be written to the stream to which the object is attached. It defines the **flush( )** method, shown here:

void flush( ) throws IOException

Flushing a stream typically causes buffered output to be physically written to the underlying device. This interface is implemented by all of the I/O classes that write to a stream.

I/O Exceptions

Two exceptions play an important role in I/O handling. The first is **IOException**. As it relates to most of the I/O classes described in this chapter, if an I/O error occurs, an **IOException** is thrown. In many cases, if a file cannot be opened, a **FileNotFoundException** is thrown. **FileNotFoundException** is a subclass of **IOException**, so both can be caught with a single **catch** that catches **IOException**. For brevity, this is the approach used by most of the sample code in this chapter. However, in your own applications, you might find it useful to **catch** each exception separately.

Another exception class that is sometimes important when performing I/O is **SecurityException**. As explained in Chapter 13, in situations in which a security manager is present, several of the file classes will throw a **SecurityException** if a security violation occurs when attempting to open a file. By default, applications run via **java** do not use a security manager. For that reason, the I/O examples in this book do not need to watch for a possible **SecurityException**. However, applets will use the security manager provided by the browser, and file I/O performed by an applet could generate a **SecurityException**. In such a case, you will need to handle this exception.

Two Ways to Close a Stream

In general, a stream must be closed when it is no longer needed. Failure to do so can lead to memory leaks and resource starvation. The techniques used to close a stream were described in Chapter 13, but because of their importance, they warrant a brief review here before the stream classes are examined.

Beginning with JDK 7, there are two basic ways in which you can close a stream. The first is to explicitly call **close( )** on the stream. This is the traditional approach that has been used since the original release of Java. With this approach, **close( )** is typically called within a **finally** block. Thus, a simplified skeleton for the traditional approach is shown here:

```
try {
  // open and access file
} catch( I/O-exception) {
  // ...
} finally {
  // close the file
}
```

This general technique (or variation thereof) is common in code that predates JDK 7.

The second approach to closing a stream is to automate the process by using the **try**-with-resources statement that was added by JDK 7 (and, of course, supported by JDK 8). The **try**-with-resources statement is an enhanced form of **try** that has the following form:

```
try (resource-specification) {
   // use the resource
}
```

Here, *resource-specification* is a statement or statements that declares and initializes a resource, such as a file or other stream-related resource. It consists of a variable declaration in which the variable is initialized with a reference to the object being managed. When the

try block ends, the resource is automatically released. In the case of a file, this means that the file is automatically closed. Thus, there is no need to call **close( )** explicitly.

Here are three key points about the **try**-with-resources statement:

- Resources managed by **try**-with-resources must be objects of classes that implement **AutoCloseable**.
- The resource declared in the **try** is implicitly **final**.
- You can manage more than one resource by separating each declaration by a semicolon.

Also, remember that the scope of the declared resource is limited to the **try**-with-resources statement.

The principal advantage of **try**-with-resources is that the resource (in this case, a stream) is closed automatically when the **try** block ends. Thus, it is not possible to forget to close the stream, for example. The **try**-with-resources approach also typically results in shorter, clearer, easier-to-maintain source code.

Because of its advantages, **try**-with-resources is expected to be used extensively in new code. As a result, most of the code in this chapter (and in this book) will use it. However, because a large amount of older code still exists, it is important for all programmers to also be familiar with the traditional approach to closing a stream. For example, you will quite likely have to work on legacy code that uses the traditional approach or in an environment that uses an older version of Java. There may also be times when the automated approach is not appropriate because of other aspects of your code. For this reason, a few I/O examples in this book will demonstrate the traditional approach so you can see it in action.

One last point: The examples that use **try**-with-resources must be compiled by a modern version of Java. They won't work with an older compiler. The examples that use the traditional approach can be compiled by older versions of Java.

REMEMBER Because **try**-with-resources streamlines the process of releasing a resource and eliminates the possibility of accidentally forgetting to release a resource, it is the approach recommended for new code when its use is appropriate.

The Stream Classes

Java's stream-based I/O is built upon four abstract classes: **InputStream**, **OutputStream**, **Reader**, and **Writer**. These classes were briefly discussed in Chapter 13. They are used to create several concrete stream subclasses. Although your programs perform their I/O operations through concrete subclasses, the top-level classes define the basic functionality common to all stream classes.

InputStream and **OutputStream** are designed for byte streams. **Reader** and **Writer** are designed for character streams. The byte stream classes and the character stream classes form separate hierarchies. In general, you should use the character stream classes when working with characters or strings and use the byte stream classes when working with bytes or other binary objects.

In the remainder of this chapter, both the byte- and character-oriented streams are examined.

The Byte Streams

The byte stream classes provide a rich environment for handling byte-oriented I/O. A byte stream can be used with any type of object, including binary data. This versatility makes byte streams important to many types of programs. Since the byte stream classes are topped by **InputStream** and **OutputStream**, our discussion begins with them.

InputStream

InputStream is an abstract class that defines Java's model of streaming byte input. It implements the **AutoCloseable** and **Closeable** interfaces. Most of the methods in this class will throw an **IOException** when an I/O error occurs. (The exceptions are **mark( )** and **markSupported( )**.) Table 20-1 shows the methods in **InputStream**.

NOTE Most of the methods described in Table 20-1 are implemented by the subclasses of **InputStream**. The **mark( )** and **reset( )** methods are exceptions; notice their use, or lack thereof, by each subclass in the discussions that follow.

OutputStream

OutputStream is an abstract class that defines streaming byte output. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Most of the methods defined by this class return **void** and throw an **IOException** in the case of I/O errors. Table 20-2 shows the methods in **OutputStream**.

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )** / **reset( )** are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

Table 20-1 The Methods Defined by **InputStream**

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with an expression without having to cast it back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

Table 20-2 The Methods Defined by **OutputStream**

FileInputStream

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file. Two commonly used constructors are shown here:

FileInputStream(String *filePath*)
FileInputStream(File *fileObj*)

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

The following example creates two **FileInputStream**s that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")
File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allows you to closely examine the file using the **File** methods, before attaching it to an input stream. When a **FileInputStream** is created, it is also opened for reading. **FileInputStream** overrides six of the methods in the abstract class **InputStream**. The **mark( )** and **reset( )** methods are not overridden, and any attempt to use **reset( )** on a **FileInputStream** will generate an **IOException**.

The next example shows how to read a single byte, an array of bytes, and a subrange of an array of bytes. It also illustrates how to use **available( )** to determine the number of bytes remaining and how to use the **skip( )** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory. Notice that it uses the **try**-with-resources statement to automatically close the file when it is no longer needed.

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

import java.io.*;
```

```
class FileInputStreamDemo {
  public static void main(String args[]) {
    int size;

    // Use try-with-resources to close the stream.
    try ( FileInputStream f =
           new FileInputStream("FileInputStreamDemo.java") ) {

      System.out.println("Total Available Bytes: " +
                         (size = f.available()));

      int n = size/40;
      System.out.println("First " + n +
                         " bytes of the file one read() at a time");
      for (int i=0; i < n; i++) {
        System.out.print((char) f.read());
      }

      System.out.println("\nStill Available: " + f.available());

      System.out.println("Reading the next " + n +
                         " with one read(b[])");
      byte b[] = new byte[n];
      if (f.read(b) != n) {
        System.err.println("couldn't read " + n + " bytes.");
      }

      System.out.println(new String(b, 0, n));
      System.out.println("\nStill Available: " + (size = f.available()));
      System.out.println("Skipping half of remaining bytes with skip()");
      f.skip(size/2);
      System.out.println("Still Available: " + f.available());

      System.out.println("Reading " + n/2 + " into the end of array");
      if (f.read(b, n/2, n/2) != n/2) {
        System.err.println("couldn't read " + n/2 + " bytes.");
      }

      System.out.println(new String(b, 0, b.length));
      System.out.println("\nStill Available: " + f.available());
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1785
First 44 bytes of the file one read() at a time
// Demonstrate FileInputStream.
// This pr
Still Available: 1741
```

```
Reading the next 44 with one read(b[])
ogram uses try-with-resources. It requires J

Still Available: 1697
Skipping half of remaining bytes with skip()
Still Available: 849
Reading 22 into the end of array
ogram uses try-with-rebyte[n];
      if (

Still Available: 827
```

This somewhat contrived example demonstrates how to read three ways, to skip input, and to inspect the amount of data available on a stream.

NOTE The preceding example and the other examples in this chapter handle any I/O exceptions that might occur as described in Chapter 13. See Chapter 13 for details and alternatives.

FileOutputStream

FileOutputStream creates an **OutputStream** that you can use to write bytes to a file. It implements the **AutoCloseable**, **Closeable**, and **Flushable** interfaces. Four of its constructors are shown here:

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

They can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode.

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

The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes( )** method to extract the byte array equivalent. It then creates three files. The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileOutputStream.
// This program uses the traditional approach to closing a file.

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();
    FileOutputStream f0 = null;
    FileOutputStream f1 = null;
    FileOutputStream f2 = null;

    try {
      f0 = new FileOutputStream("file1.txt");
      f1 = new FileOutputStream("file2.txt");
      f2 = new FileOutputStream("file3.txt");

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

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

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    } finally {
      try {
        if(f0 != null) f0.close();
      } catch(IOException e) {
        System.out.println("Error Closing file1.txt");
      }
      try {
        if(f1 != null) f1.close();
      } catch(IOException e) {
        System.out.println("Error Closing file2.txt");
      }
      try {
        if(f2 != null) f2.close();
      } catch(IOException e) {
        System.out.println("Error Closing file3.txt");
      }
    }
  }
}
```

Here are the contents of each file after running this program. First, **file1.txt**:

```
Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.
```

Next, **file2.txt**:

```
Now is the time for all good men
 to come to the aid of their country
 and pay their due taxes.
```

Finally, **file3.txt**:

```
nd pay their due taxes.
```

As the comment at the top of the program states, the preceding program shows an example that uses the traditional approach to closing a file when it is no longer needed. This approach is required by all versions of Java prior to JDK 7 and is widely used in legacy code. As you can see, quite a bit of rather awkward code is required to explicitly call **close( )** because each call could generate an **IOException** if the close operation fails. This program can be substantially improved by using the new **try**-with-resources statement. For comparison, here is the revised version. Notice that it is much shorter and streamlined:

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

import java.io.*;

class FileOutputStreamDemo {
  public static void main(String args[]) {
    String source = "Now is the time for all good men\n"
      + " to come to the aid of their country\n"
      + " and pay their due taxes.";
    byte buf[] = source.getBytes();

    // Use try-with-resources to close the files.
    try (FileOutputStream f0 = new FileOutputStream("file1.txt");
         FileOutputStream f1 = new FileOutputStream("file2.txt");
         FileOutputStream f2 = new FileOutputStream("file3.txt") )
    {

      // write to first file
      for (int i=0; i < buf.length; i += 2) f0.write(buf[i]);

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

      // write to third file
      f2.write(buf, buf.length-buf.length/4, buf.length/4);
    } catch(IOException e) {
      System.out.println("An I/O Error Occurred");
    }
  }
}
```

ByteArrayInputStream

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

ByteArrayInputStream(byte *array* [ ])
ByteArrayInputStream(byte *array* [ ], int *start*, int *numBytes*)

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

The **close( )** method has no effect on a **ByteArrayInputStream**. Therefore, it is not necessary to call **close( )** on a **ByteArrayInputStream**, but doing so is not an error.

The following example creates a pair of **ByteArrayInputStream**s, initializing them with the byte representation of the alphabet:

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

class ByteArrayInputStreamDemo {
  public static void main(String args[]) {
    String tmp = "abcdefghijklmnopqrstuvwxyz";
    byte b[] = tmp.getBytes();

    ByteArrayInputStream input1 = new ByteArrayInputStream(b);
    ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
  }
}
```

The **input1** object contains the entire lowercase alphabet, whereas **input2** contains only the first three letters.

A **ByteArrayInputStream** implements both **mark( )** and **reset( )**. However, if **mark( )** has not been called, then **reset( )** sets the stream pointer to the start of the stream—which, in this case, is the start of the byte array passed to the constructor. The next example shows how to use the **reset( )** method to read the same input twice. In this case, the program reads and prints the letters "abc" once in lowercase and then again in uppercase.

```
import java.io.*;

class ByteArrayInputStreamReset {
  public static void main(String args[]) {
    String tmp = "abc";
    byte b[] = tmp.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(b);

    for (int i=0; i<2; i++) {
      int c;
      while ((c = in.read()) != -1) {
        if (i == 0) {
          System.out.print((char) c);
        } else {
          System.out.print(Character.toUpperCase((char) c));
        }
      }
      System.out.println();
      in.reset();
    }
  }
}
```

This example first reads each character from the stream and prints it as-is in lowercase. It then resets the stream and begins reading again, this time converting each character to uppercase before printing. Here's the output:

```
    abc
    ABC
```

ByteArrayOutputStream

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

ByteArrayOutputStream( )
ByteArrayOutputStream(int *numBytes*)

In the first form, a buffer of 32 bytes is created. In the second, a buffer is created with a size equal to that specified by *numBytes*. The buffer is held in the protected **buf** field of **ByteArrayOutputStream**. The buffer size will be increased automatically, if needed. The number of bytes held by the buffer is contained in the protected **count** field of **ByteArrayOutputStream**.

The **close( )** method has no effect on a **ByteArrayOutputStream**. Therefore, it is not necessary to call **close( )** on a **ByteArrayOutputStream**, but doing so is not an error.

The following example demonstrates **ByteArrayOutputStream**:

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

import java.io.*;

class ByteArrayOutputStreamDemo {
  public static void main(String args[]) {
    ByteArrayOutputStream f = new ByteArrayOutputStream();
    String s = "This should end up in the array";
    byte buf[] = s.getBytes();

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

    System.out.println("Buffer as a string");
    System.out.println(f.toString());
    System.out.println("Into array");
    byte b[] = f.toByteArray();
    for (int i=0; i<b.length; i++) System.out.print((char) b[i]);

    System.out.println("\nTo an OutputStream()");

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

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

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

When you run the program, you will create the following output. Notice how after the call to **reset( )**, the three X's end up at the beginning.

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To an OutputStream()
Doing a reset
XXX
```

This example uses the **writeTo( )** convenience method to write the contents of **f** to **test.txt**. Examining the contents of the **test.txt** file created in the preceding example shows the result we expected:

```
This should end up in the array
```

Filtered Byte Streams

Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. These streams are typically accessed by methods that are expecting a generic stream, which is a superclass of the filtered streams. Typical extensions are buffering, character translation, and raw data translation. The filtered byte streams are **FilterInputStream** and **FilterOutputStream**. Their constructors are shown here:

FilterOutputStream(OutputStream *os*)
FilterInputStream(InputStream *is*)

The methods provided in these classes are identical to those in **InputStream** and **OutputStream**.

Buffered Byte Streams

For the byte-oriented streams, a *buffered stream* extends a filtered stream class by attaching a memory buffer to the I/O stream. This buffer allows Java to do I/O operations on more than a byte at a time, thereby improving performance. Because the buffer is available, skipping, marking, and resetting of the stream become possible. The buffered byte stream classes are **BufferedInputStream** and **BufferedOutputStream**. **PushbackInputStream** also implements a buffered stream.

BufferedInputStream

Buffering I/O is a very common performance optimization. Java's **BufferedInputStream** class allows you to "wrap" any **InputStream** into a buffered stream to improve performance.

BufferedInputStream has two constructors:

BufferedInputStream(InputStream *inputStream*)
BufferedInputStream(InputStream *inputStream*, int *bufSize*)

The first form creates a buffered stream using a default buffer size. In the second, the size of the buffer is passed in *bufSize*. Use of sizes that are multiples of a memory page, a disk block, and so on, can have a significant positive impact on performance. This is, however, implementation-dependent. An optimal buffer size is generally dependent on the host operating system, the amount of memory available, and how the machine is configured. To make good use of buffering doesn't necessarily require quite this degree of sophistication. A good guess for a size is around 8,192 bytes, and attaching even a rather small buffer to an I/O stream is always a good idea. That way, the low-level system can read blocks of data from the disk or network and store the results in your buffer. Thus, even if you are reading the data a byte at a time out of the **InputStream**, you will be manipulating fast memory most of the time.

Buffering an input stream also provides the foundation required to support moving backward in the stream of the available buffer. Beyond the **read( )** and **skip( )** methods implemented in any **InputStream**, **BufferedInputStream** also supports the **mark( )** and **reset( )** methods. This support is reflected by **BufferedInputStream.markSupported( )** returning **true**.

The following example contrives a situation where we can use **mark( )** to remember where we are in an input stream and later use **reset( )** to get back there. This example is parsing a stream for the HTML entity reference for the copyright symbol. Such a reference begins with an ampersand (&) and ends with a semicolon (;) without any intervening whitespace. The sample input has two ampersands to show the case where the **reset( )** happens and where it does not.

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

import java.io.*;

class BufferedInputStreamDemo {
  public static void main(String args[]) {
    String s = "This is a &copy; copyright symbol " +
      "but this is &copy not.\n";
    byte buf[] = s.getBytes();

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

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

Notice that this example uses **mark(32)**, which preserves the mark for the next 32 bytes read (which is enough for all entity references). Here is the output produced by this program:

```
    This is a (c) copyright symbol but this is &copy not.
```

BufferedOutputStream

A **BufferedOutputStream** is similar to any **OutputStream** with the exception that the **flush( )** method is used to ensure that data buffers are written to the stream being buffered. Since the point of a **BufferedOutputStream** is to improve performance by reducing the number of times the system actually writes data, you may need to call **flush( )** to cause any data that is in the buffer to be immediately written.

Unlike buffered input, buffering output does not provide additional functionality. Buffers for output in Java are there to increase performance. Here are the two available constructors:

BufferedOutputStream(OutputStream *outputStream*)
BufferedOutputStream(OutputStream *outputStream*, int *bufSize*)

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

PushbackInputStream

One of the novel uses of buffering is the implementation of pushback. *Pushback* is used on an input stream to allow a byte to be read and then returned (that is, "pushed back") to the stream. The **PushbackInputStream** class implements this idea. It provides a mechanism to "peek" at what is coming from an input stream without disrupting it.

PushbackInputStream has the following constructors:

PushbackInputStream(InputStream *inputStream*)
PushbackInputStream(InputStream *inputStream*, int *numBytes*)

The first form creates a stream object that allows one byte to be returned to the input stream. The second form creates a stream that has a pushback buffer that is *numBytes* long. This allows multiple bytes to be returned to the input stream.

Beyond the familiar methods of **InputStream**, **PushbackInputStream** provides **unread( )**, shown here:

void unread(int *b*)
void unread(byte *buffer* [ ])
void unread(byte *buffer*, int *offset*, int *numBytes*)

The first form pushes back the low-order byte of *b*. This will be the next byte returned by a subsequent call to **read( )**. The second form pushes back the bytes in *buffer*. The third form pushes back *numBytes* bytes beginning at *offset* from *buffer*. An **IOException** will be thrown if there is an attempt to push back a byte when the pushback buffer is full.

Here is an example that shows how a programming language parser might use a **PushbackInputStream** and **unread( )** to deal with the difference between the **= =** operator for comparison and the **=** operator for assignment:

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

import java.io.*;

class PushbackInputStreamDemo {
  public static void main(String args[]) {
    String s = "if (a == 4) a = 0;\n";
    byte buf[] = s.getBytes();
    ByteArrayInputStream in = new ByteArrayInputStream(buf);
    int c;
    try ( PushbackInputStream f = new PushbackInputStream(in) )
    {
      while ((c = f.read()) != -1) {
        switch(c) {
        case '=':
          if ((c = f.read()) == '=')
            System.out.print(".eq.");
          else {
            System.out.print("<-");
            f.unread(c);
          }
          break;
        default:
          System.out.print((char) c);
          break;
        }
      }
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    }
  }
}
```

Here is the output for this example. Notice that == was replaced by **".eq."** and = was replaced by **"<–"**.

```
    if (a .eq. 4) a <- 0;
```

CAUTION **PushbackInputStream** has the side effect of invalidating the **mark( )** or **reset( )** methods of the **InputStream** used to create it. Use **markSupported( )** to check any stream on which you are going to use **mark( )**/**reset( )**.

SequenceInputStream

The **SequenceInputStream** class allows you to concatenate multiple **InputStream**s. The construction of a **SequenceInputStream** is different from any other **InputStream**. A **SequenceInputStream** constructor uses either a pair of **InputStream**s or an **Enumeration** of **InputStream**s as its argument:

SequenceInputStream(InputStream *first*, InputStream *second*)
SequenceInputStream(Enumeration <? extends InputStream> *streamEnum*)

Operationally, the class fulfills read requests from the first **InputStream** until it runs out and then switches over to the second one. In the case of an **Enumeration**, it will continue through all of the **InputStream**s until the end of the last one is reached. When the end of each file is reached, its associated stream is closed. Closing the stream created by **SequenceInputStream** causes all unclosed streams to be closed.

Here is a simple example that uses a **SequenceInputStream** to output the contents of two files. For demonstration purposes, this program uses the traditional technique used to close a file. As an exercise, you might want to try changing it to use the **try**-with-resources statement.

```
// Demonstrate sequenced input.
// This program uses the traditional approach to closing a file.

import java.io.*;
import java.util.*;

class InputStreamEnumerator implements Enumeration<FileInputStream> {
  private Enumeration<String> files;

  public InputStreamEnumerator(Vector<String> files) {
    this.files = files.elements();
  }

  public boolean hasMoreElements() {
    return files.hasMoreElements();
  }

  public FileInputStream nextElement() {
    try {
      return new FileInputStream(files.nextElement().toString());
    } catch (IOException e) {
      return null;
    }
  }
}

class SequenceInputStreamDemo {
  public static void main(String args[]) {
    int c;
    Vector<String> files = new Vector<String>();

    files.addElement("file1.txt");
    files.addElement("file2.txt");
    files.addElement("file3.txt");
    InputStreamEnumerator ise = new InputStreamEnumerator(files);
    InputStream input = new SequenceInputStream(ise);

    try {
      while ((c = input.read()) != -1)
        System.out.print((char) c);
    } catch(NullPointerException e) {
      System.out.println("Error Opening File.");
    } catch(IOException e) {
      System.out.println("I/O Error: " + e);
    } finally {
      try {
        input.close();
      } catch(IOException e) {
        System.out.println("Error Closing SequenceInputStream");
      }
    }
  }
}
```

This example creates a **Vector** and then adds three filenames to it. It passes that vector of names to the **InputStreamEnumerator** class, which is designed to provide a wrapper on the vector where the elements returned are not the filenames but, rather, open **FileInputStream**s on those names. The **SequenceInputStream** opens each file in turn, and this example prints the contents of the files.

Notice in **nextElement( )** that if a file cannot be opened, **null** is returned. This results in a **NullPointerException**, which is caught in **main( )**.

PrintStream

The **PrintStream** class provides all of the output capabilities we have been using from the **System** file handle, **System.out**, since the beginning of the book. This makes **PrintStream** one of Java's most often used classes. It implements the **Appendable**, **AutoCloseable**, **Closeable**, and **Flushable** interfaces.

PrintStream defines several constructors. The ones shown next have been specified from the start:

PrintStream(OutputStream *outputStream*)
PrintStream(OutputStream *outputStream*, boolean *autoFlushingOn*)
PrintStream(OutputStream *outputStream*, boolean *autoFlushingOn* String *charSet*)
    throws UnsupportedEncodingException

Here, *outputStream* specifies an open **OutputStream** that will receive output. The *autoFlushingOn* parameter controls whether the output buffer is automatically flushed every time a newline (**\n**) character or a byte array is written or when **println( )** is called. If *autoFlushingOn* is **true**, flushing automatically takes place. If it is **false**, flushing is not automatic. The first constructor does not automatically flush. You can specify a character encoding by passing its name in *charSet*.

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

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

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

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*[ ], int *offset*, 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*[ ], int *offset*, 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*[ ], int *offset*, 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*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
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*[ ], int *offset*, 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.