18 Files and Streams

Supplementary Objectives

- Write code that uses objects of the `File` class to navigate the file system.
- Distinguish between byte and character streams, and identify the roots of their inheritance hierarchies.
- Select valid constructor arguments for `FilterInputStream` and `FilterOutputStream` subclasses from a list of classes in the `java.io` package.
- Write appropriate code to read, write and update files using `FileInputStream`, `FileOutputStream` and `RandomAccessFile` objects.
- Write code that uses the classes `DataOutputStream` and `DataInputStream` for writing and reading Java primitive values.
- Write code that uses objects of the classes `InputStreamReader` and `OutputStreamWriter` to translate between Unicode and either platform default or other character encodings.
- Write code to set up (buffered) readers and writers for text files.
- Write code that uses the classes `ObjectOutputStream` and `ObjectInputStream` for writing and reading objects.

18.1 Input and Output

The `java.io` package provides an extensive library of classes for dealing with input and output. Java provides *streams* as a general mechanism for dealing with data I/O. Streams implement *sequential access* of data. There are two kinds of streams: *byte streams* and *character streams*. An *input stream* is an object that an application can use to read a sequence of data, and an *output stream* is an object that an application can use to write a sequence of data. An *input stream* acts as a *source* of data, and an *output stream* acts as a *destination* of data. The following entities can act as both input and output streams:

- an array of bytes or characters
- a file
- a *pipe*
- a network connection

Streams can be *chained* with *filters* to provide new functionality. In addition to dealing with bytes and characters, streams are provided for input and output of Java primitive values and objects. The `java.io` package also provides support for *random access* of files, and a general interface to interact with the file system of the host platform.

18.2 `File` Class

The `File` class provides a general machine-independent interface to the file system of the underlying platform. A `File` object represents the pathname of a file or directory in the host file system. An application can use the functionality provided by the `File` class for handling files and directories in the file system. The `File` class is *not* meant for handling the contents of files. For that purpose, there are the `FileInputStream`, `FileOutputStream` and `RandomAccessFile` classes, which are discussed later in this chapter.

The pathname for a file or directory is specified using the naming conventions of the host system. However, the `File` class defines platform-dependent constants that can be used to handle file and directory names in a platform-independent way:

```
public static final char separatorChar
public static final String separator
```

Defines the character or string that separates the directory and the file components in a pathname. This separator is `'/'`, `'\'` or `':'` for Unix, Windows and Macintosh, respectively.

```
public static final char pathSeparatorChar
public static final String pathSeparator
```

Defines the character or string that separates the file or directory names in a "path list". This character is `':'` or `';'` for Unix and Windows, respectively.

Some examples of pathnames are:

`/book/chapter1`	on Unix
`C:\book\chapter1`	on Windows
`HD:book:chapter1`	on Macintosh

Some examples of path lists are:

`/book:/manual:/draft`	on Unix
`C:\book;D:\manual;A:\draft`	on Windows

Files and directories can be referenced using both absolute and relative pathnames, but the pathname must follow the conventions of the host platform. On Unix platforms, a pathname is absolute if its first character is the separator character. On Windows platforms, a path is absolute if the ASCII `'\'` is the first character, or follows the volume name (e.g., `C:`), in a pathname. On the Macintosh, a pathname is absolute if it begins with a name followed by a colon. Java programs should not rely on system-specific pathname conventions. The `File` class provides facilities to construct pathnames in a platform-independent way.

The `File` class has various constructors for assigning a file or a directory to an object of the `File` class. Creating a `File` object does not mean creation of any file or directory based on the pathname specified. A `File` instance, called the *abstract pathname,* is a representation of the pathname of a file and directory. The pathname cannot be changed once the `File` object is created.

`File(String pathname)`

The pathname (of a file or a directory) can be an absolute pathname or a pathname relative to the current directory. An empty string as argument results in an abstract pathname for the current directory.

```
// "/book/chapter1" - absolute pathname of a file
File chap1 = new File(File.separator + "book" +
                      File.separator + "chapter1");
// "draft/chapters" - relative pathname of a directory
File draftChapters = new File("draft" + File.separator + "chapters");
```

`File(String directoryPathname, String filename)`

This creates a `File` object whose pathname is as follows: `directoryPathname` + separator + `filename`.

```
// "/book/chapter1" - absolute pathname of a file
File updatedChap1 = new File(File.separator + "book", "chapter1");
```

`File(File directory, String filename)`

If the `directory` argument is `null`, the resulting `File` object represents a file in the current directory. If the directory argument is not `null`, it creates a `File` object that represents a file in the given directory. The pathname of the file is then the pathname of the `directory` `File` object + separator + `filename`.

```
// "chapter13" - relative pathname of a file
File parent = null;
File chap13 = new File(parent, "chapter13");

// "draft/chapters/chapter13" - relative pathname of a file
File draftChapters = new File("draft" + File.separator + "chapters");
File updatedChap13 = new File(draftChapters, "chapter13");
```

An object of the `File` class provides a handle to a file or directory in the file system, and can be used to create, rename, and delete the entry.

A `File` object can also be used to query the file system for information about a file or directory:

- whether the entry exists
- whether the `File` object represents a file or directory
- whether the entry has read or write access
- get pathname information about the file or directory
- list all entries under a directory in the file system

Many methods of the `File` class throw a `SecurityException` in the case of a security violation, for example if read or write access is denied. Some also return a boolean value to indicate whether the operation was successful.

Querying the File System

The `File` class provides a number of methods for obtaining the platform-dependent representation of a pathname and its components.

`String getName()`

Returns the name of the file entry, excluding the specification of the directory in which it resides.

On Unix, the name part of `"/book/chapters/one"` is `"one"`.

On Windows platforms, the name part of `"c:\java\bin\javac"` is `"javac"`.

On the Macintosh, the name part of `"HD:java-tools:javac"` is `"javac"`.

`String getPath()`

The method returns the (absolute or relative) pathname of the file represented by the `File` object.

`String getAbsolutePath()`

If the `File` object represents an absolute pathname then this pathname is returned, otherwise the returned pathname is constructed by concatenating the current directory pathname, the separator character and the pathname of the `File` object.

`String getCanonicalPath() throws IOException`

Also platform-dependent, the canonical path usually specifies a pathname in which all relative references have been completely resolved.

For example, if the `File` object represented the absolute pathname `"c:\book\chapter1"` on Windows, then this pathname would be returned by these methods. On the other hand, if the `File` object represented the relative pathname `"..\book\chapter1"` and the current directory had the absolute pathname `"c:\documents"`, the pathname returned by the `getPath()`, `getAbsolutePath()` and `getCanonicalPath()` methods would be `"..\book\chapter1"`, `"c:\documents\..\book\chapter1"` and `"c:\book\chapter1"`, respectively.

`String getParent()`

The parent part of the pathname of this `File` object is returned if one exists, otherwise the `null` value is returned. The parent part is generally the prefix obtained from the pathname after deleting the file or directory name component found after the last occurrence of the separator character. However, this is not true for all platforms.

On Unix, the parent part of `"/book/chapter1"` is `"/book"`, whose parent part is `"/"`, which in turn has no parent.

On Windows platforms, the parent part of `"c:\java-tools"` is `"c:\"`, which in turn has no parent.

On the Macintosh, the parent part of `"HD:java-tools"` is `"HD:"`, which in turn has no parent.

`boolean isAbsolute()`

Whether a `File` object represents an absolute pathname can be determined using this method.

The following three methods can be used to query the file system about the modification time of a file or directory, determine the size (in bytes) of a file, and ascertain whether two pathnames are identical.

`long lastModified()`

The modification time returned is encoded as a `long` value, and should only be compared with other values returned by this method.

`long length()`

Returns the size (in bytes) of the file represented by the `File` object.

`boolean equals(Object obj)`

This method only compares the pathnames of the `File` objects, and returns `true` if they are identical.

File or Directory Existence

A `File` object is created using a pathname. Whether this pathname denotes an entry that actually exists in the file system can be checked using the `exists()` method:

```
boolean exists()
```

Since a `File` object can represent a file or a directory, the following methods can be used to distinguish whether a given `File` object represents a file or a directory respectively:

```
boolean isFile()
boolean isDirectory()
```

Read and Write Access

To check whether the specified file has write and read access, the following methods can be used. They throw a `SecurityException` if general access is not allowed, i.e. the application is not even allowed to check whether it can read or write a file.

```
boolean canWrite()
boolean canRead()
```

Listing Directory Entries

The entries in a specified directory can be obtained as a table of file names or abstract pathnames, using the following `list()` methods. The current directory and the parent directory are excluded from the list.

```
String[] list()
String[] list(FilenameFilter filter)
File[]   listFiles()
File[]   listFiles(FilenameFilter filter)
File[]   listFiles(FileFilter filter)
```

The `filter` argument can be used to specify a *filter* that determines whether an entry should be included in the list. These methods return `null` if the abstract pathname does not denote a directory, or if an I/O error occurs. A filter is an object of a class that implements either of these two interfaces:

```
interface FilenameFilter {
    boolean accept(File currentDirectory, String entryName);
}
interface FileFilter {
    boolean accept(File pathname);
}
```

The `list()` methods call the `accept()` methods of the filter for each entry, to determine whether the entry should be included in the list.

Creating New Files and Directories

The `File` class can be used to create files and directories. A file can be created whose pathname is specified in a `File` object using the following method:

```
boolean createNewFile() throws IOException
```

It creates a new, empty file named by the abstract pathname if, and only if, a file with this name does not already exist. The returned value is `true` if the file was successfully created, `false` if the file already exists. Any I/O error results in an `IOException`.

A directory whose pathname is specified in a `File` object can be created using the following methods:

```
boolean mkdir()
boolean mkdirs()
```

The `mkdirs()` method creates any intervening parent directories in the pathname of the directory to be created.

Renaming Files and Directories

A file or a directory can be renamed, using the following method which takes the new pathname from its argument:

```
boolean renameTo(File dest)
```

Deleting Files and Directories

A file or a directory can be deleted using the following method. In the case of a directory, it must be empty before it can be deleted.

```
boolean delete()
```

Example 18.1 *Listing Files Under a Directory*

```
import java.io.*;

public class DirectoryLister {
    public static void main(String args[]) {
        File entry;
        if (args.length == 0) {               // (1)
            System.err.println("Please specify a directory name.");
            return;
        }
        entry = new File(args[0]);            // (2) user specified

        listDirectory(entry);
    }
```

```
    public static void listDirectory(File entry) {
        try {
            if (!entry.exists()) {                                          // (3)
                System.out.println(entry.getName() + " not found.");
                return;
            }

            if (entry.isFile()) {
                // Write the pathname of the file
                System.out.println(entry.getCanonicalPath());               // (4)
            } else if (entry.isDirectory()) {
                // Create list of entries for this directory
                String[] fileName = entry.list();                           // (5)
                if (fileName == null) return;
                for (int i = 0; i<fileName.length; i++) {
                    // Create a File object for the entry
                    File item = new File(entry.getPath(), fileName[i]); // (6)
                    // List it by a recursive call.
                    listDirectory(item);                                    // (7)
                }
            }
        } catch(IOException e) { System.out.println("Error: " + e); }
    }
}
```

Running the program on a Windows platform:

```
java DirectoryLister D:\docs\JC-Book\special
```

produces the following output:

```
D:\docs\JC-Book\special\book19990308\JC-14-applets.fm
D:\docs\JC-Book\special\book19990308\JC-16-swing.fm
D:\docs\JC-Book\special\JC-11-awtlayout.fm
```

The class `DirectoryLister` in Example 18.1 lists all entries in a directory specified in the command line. If no directory is given, an error message is written. This is shown at (1) and (2). In the method `listDirectory()`, each entry is tested to see if it exists, as shown at (3). The entry could be an alias (*symbolic link* in Unix or *shortcut* in Windows terminology) and its destination might not exist. The method determines whether the entry is a file, in which case the absolute pathname is listed, as shown at (4). In the case of a directory, an array of entry names is created, as shown at (5). For each entry in the directory, a `File` object is created, as shown at (6). The method `listDirectory()` is called recursively for each entry, as shown at (7).

18.3 Byte Streams: Input Streams and Output Streams

The abstract classes `InputStream` and `OutputStream` are the root of the inheritance hierarchies for handling the reading and writing of *bytes* (Figure 18.1). Their subclasses, implementing different kinds of input and output streams, override the

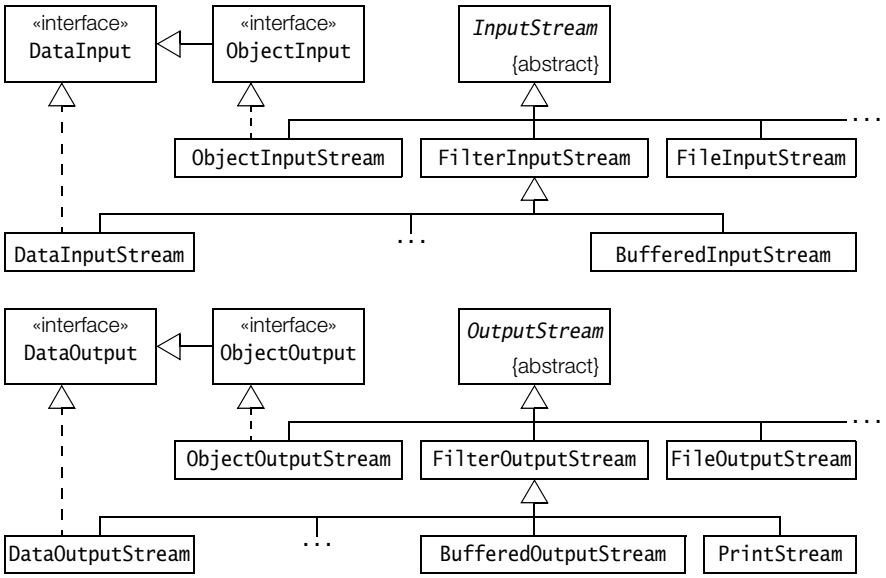

Figure 18.1 *Byte Stream Inheritance Hierarchies*

following methods from the `InputStream` and `OutputStream` classes to customize the reading and writing of bytes, respectively:

```
int read() throws IOException
int read(byte[] b) throws IOException
int read(byte[] b, int off, int len) throws IOException
```

```
void write(int b) throws IOException
void write(byte[] b) throws IOException
void write(byte[] b, int off, int len) throws IOException
```

Note that the first `read()` method read a byte, but returns an `int` value. The byte read resides in the eight least significant bits of the `int`, while the remaining bits in the `int` are zeroed out. The `read()` methods return the value –1 when the end of stream is reached. The first `write()` method takes an `int` as argument, but truncates it down to the eight least significant bits before writing it out as a byte.

A stream should be closed when no longer needed, to free system resources:

```
void close() throws IOException
void flush() throws IOException
```

Closing an output stream automatically *flushes* the stream, meaning that any data in its internal buffer is written out. An output stream can also be manually flushed by calling the second method.

Read and write operations on streams are synchronous (*blocking*) operations, i.e. a call to a read or write method does not return before a byte has been read or written.

Many methods in the classes contained in the `java.io` package throw an `IOException`. A calling method must either catch the exception explicitly, or specify it in a `throws` clause.

Table 18.1 and Table 18.2 give an overview of the byte streams.

Table 18.1 *Input Streams*

`ByteArrayInputStream`	Data is read from a byte array that must be specified.
`FileInputStream`	Data is read as bytes from a file. The file acting as the input stream can be specified by a `File` object, a `FileDescriptor` or a `String` file name.
`FilterInputStream`	Superclass of all input stream filters. An input filter must be chained to an underlying input stream.
`BufferedInputStream`	A filter that buffers the bytes read from an underlying input stream. The underlying input stream must be specified, and an optional buffer size can be included.
`DataInputStream`	A filter that allows the binary representation of Java primitive values to be read from an underlying input stream. The underlying input stream must be specified.
`PushbackInputStream`	A filter that allows bytes to be "unread" from an underlying input stream. The number of bytes to be unread can optionally be specified.
`ObjectInputStream`	Allows binary representations of Java objects and Java primitive values to be read from a specified input stream.
`PipedInputStream`	Reads bytes from a `PipedOutputStream` to which it must be connected. The `PipedOutputStream` can optionally be specified when creating the `PipedInputStream`.
`SequenceInputStream`	Allows bytes to be read sequentially from two or more input streams consecutively. This should be regarded as concatenating the contents of several input streams into a single continuous input stream.

Table 18.2 *Output Streams*

`ByteArrayOutputStream`	Data is written to a byte array. The size of the byte array created can be specified.
`FileOutputStream`	Data is written as bytes to a file. The file acting as the output stream can be specified by a `File` object, a `FileDescriptor` or a `String` file name.
`FilterOutputStream`	Superclass of all output stream filters. An output filter must be chained to an underlying output stream.
`BufferedOutputStream`	A filter that buffers the bytes written to an underlying output stream. The underlying output stream must be specified, and an optional buffer size can be given.

Table 18.2 *Output Streams (continued)*

`DataOutputStream`	A filter that allows the binary representation of Java primitive values to be written to an underlying output stream. The underlying output stream must be specified.
`ObjectOutputStream`	Allows the binary representation of Java objects and Java primitive values to be written to a specified underlying output stream.
`PipedOutputStream`	Writes bytes to a `PipedInputStream` to which it must be connected. The `PipedInputStream` can optionally be specified when creating the `PipedOutputStream`.

File Streams

The classes `FileInputStream` and `FileOutputStream` define byte input and output streams that are connected to files. Data can only be read or written as a sequence of bytes.

An input stream for reading bytes can be created using the following constructors:

```
FileInputStream(String name) throws FileNotFoundException
FileInputStream(File file) throws FileNotFoundException
FileInputStream(FileDescriptor fdObj)
```

The file can be specified by its name, through a `File` or a `FileDescriptor` object.

If the file does not exist, a `FileNotFoundException` is thrown. If it exists, it is set to be read from the beginning. A `SecurityException` is thrown if the file does not have read access.

An output stream for writing bytes can be created using the following constructors:

```
FileOutputStream(String name) throws FileNotFoundException
FileOutputStream(String name, boolean append) throws FileNotFoundException
FileOutputStream(File file) throws IOException
FileOutputStream(FileDescriptor fdObj)
```

The file can be specified by its name, through a `File` object or using a `FileDescriptor` object.

If the file does not exist, it is created. If it exists, its contents are reset, unless the appropriate constructor is used to indicate that output should be appended to the file. A `SecurityException` is thrown if the file does not have write access or it cannot be created.

The `FileInputStream` class provides an implementation for the `read()` methods in its superclass `InputStream`. Similarly, the `FileOutputStream` class provides an implementation for the `write()` methods in its superclass `OutputStream`.

Example 18.2 demonstrates usage of writing and reading bytes to and from file streams. It copies the contents of one file to another file. The input and the output file names are specified on the command line. The streams are created as shown at (1) and (2). The input file is read a byte at a time and written straight to the output file, as shown in the `try` block at (3). The streams are explicitly closed, as shown at (4). Note that most of the code consists of `try-catch` constructs to handle the various exceptions. The example could be optimized by using buffering, and reading and writing several bytes at a time.

Example 18.2 *Copy a File*

```
/* Copy a file.
   Command syntax: java CopyFile <from-file> <to-file>
*/

import java.io.*;

class CopyFile {
    public static void main(String args[]) {
        FileInputStream fromFile;
        FileOutputStream toFile;

        // Assign the files
        try {
            fromFile = new FileInputStream(args[0]);     // (1)
            toFile = new FileOutputStream(args[1]);      // (2)
        } catch(FileNotFoundException e) {
            System.err.println("File could not be copied: " + e);
            return;
        } catch(IOException e) {
            System.err.println("File could not be copied: " + e);
            return;
        } catch(ArrayIndexOutOfBoundsException e) {
            System.err.println("Usage: CopyFile <from-file> <to-file>");
            return;
        }

        // Copy bytes
        try {                                            // (3)
            int i = fromFile.read();
            while (i != -1) { // check end of file
                toFile.write(i);
                i = fromFile.read();
            }
        } catch(IOException e) {
            System.err.println("Error reading/writing.");
        }

        // Close the files
        try {                                            // (4)
            fromFile.close();
            toFile.close();
```

```
            } catch(IOException e) {
                System.err.println("Error closing file.");
            }
        }
    }
```

Filter Streams

A *filter* is a high-level stream that provides additional functionality to an underlying stream to which it is chained. The data from the underlying stream is manipulated in some way by the filter. The `FilterInputStream` and `FilterOutputStream` classes, together with their subclasses, define input and output filter streams. Subclasses `BufferedInputStream` and `BufferedOutputStream` implement filters that respectively buffer input from, and output to, the underlying stream. Subclasses `DataInputStream` and `DataOutputStream` implement filters that allow Java primitive values to be read and written respectively to and from an underlying stream.

I/O of Java Primitive Values

The `java.io` package contains two interfaces: `DataInput` and `DataOutput`, that streams can implement to allow reading and writing of binary representations of Java primitive values (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`). Methods for writing binary representations of Java primitive values are named `writeX()`, where *X* is any Java primitive datatype. Methods for reading binary representations of Java primitive values are similarly named `readX()`. Table 18.3 gives an overview of the `readX()` and `writeX()` methods found in these two interfaces. Note the methods provided for reading and writing strings. Whereas the methods `readChar()` and `writeChar()` handle a single character, the methods `readLine()` and `writeChars()` handle a string of characters. The methods `readUTF()` and `writeUTF()` also read and write a string of characters, but use the UTF-8 character encoding (see Table 18.7 on page 570).

The filter streams `DataOutputStream` and `DataInputStream` implement `DataOutput` and `DataInput` interfaces respectively, and can be used to write and read binary representations of Java primitive values to and from an underlying stream. Both the `writeX()` and `readX()` methods throw an `IOException` in the event of an I/O error. Bytes can also be skipped from a `DataInput` stream, using the `skipBytes(int n)` method which skips `n` bytes. The following constructors can be used to set up filters for reading and writing Java primitive values respectively from an underlying stream:

```
DataInputStream(InputStream in)
DataOutputStream(OutputStream out)
```

For handling *character streams*, Java provides special streams called *readers* and *writers* which are discussed in Section 18.4.

Table 18.3 `DataInput` ***and*** `DataOutput` ***Interfaces***

Type	Methods in *DataInput*	Methods in *DataOutput*
`boolean`	`readBoolean()`	`writeBoolean(boolean v)`
`char`	`readChar()`	`writeChar(int v)`
`byte`	`readByte()`	`writeByte(int v)`
`short`	`readShort()`	`writeShort(int v)`
`int`	`readInt()`	`writeInt(int v)`
`long`	`readLong()`	`writeLong(long v)`
`float`	`readFloat()`	`writeFloat(float v)`
`double`	`readDouble()`	`writeDouble(double v)`
`String`	`readLine()`	`writeChars(String s)`
`String`	`readUTF()`	`writeUTF(String s)`

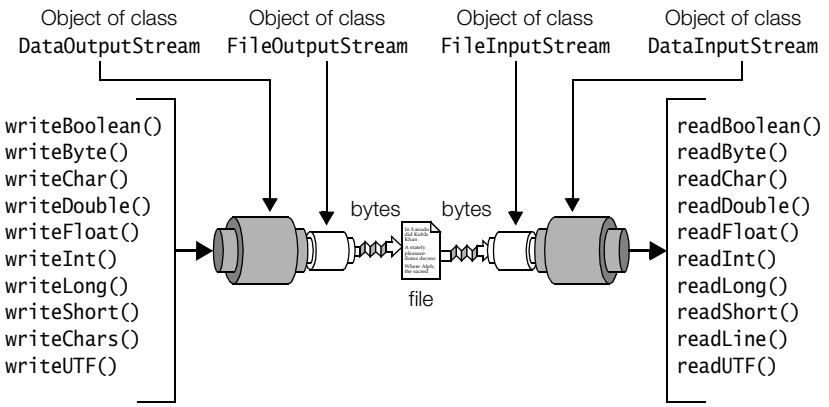

Figure 18.2 *Stream Chaining*

To write the binary representation of Java primitive values to a *file*, the following procedure can be used, which is also depicted in Figure 18.2.

1. Create a `FileOutputStream`:

```
FileOutputStream outputFile = new FileOutputStream("primitives.data");
```

2. Create a `DataOutputStream` which is chained to the `FileOutputStream`:

```
DataOutputStream outputStream = new DataOutputStream(outputFile);
```

3. Write Java primitive values using relevant `write`*X*`()` methods:

```
outputStream.writeBoolean(true);
outputStream.writeChar('A');                    // int written as Unicode char
outputStream.writeByte(Byte.MAX_VALUE);         // int written as 8-bits byte
outputStream.writeShort(Short.MIN_VALUE);       // int written as 16-bits short
outputStream.writeInt(Integer.MAX_VALUE);
```

```
outputStream.writeLong(Long.MIN_VALUE);
outputStream.writeFloat(Float.MAX_VALUE);
outputStream.writeDouble(Math.PI);
```

Note that in the case of `char`, `byte` and `short` datatypes, the `int` argument to the `writeX()` method is converted to the corresponding type, before it is written.

4. Close the filter stream, which also closes the underlying stream:

```
outputStream.close();
```

To read the binary representation of Java primitive values from a *file* the following procedure can be used, which is also depicted in Figure 18.2.

1. Create a `FileInputStream`:

```
FileInputStream inputFile = new FileInputStream("primitives.data");
```

2. Create a `DataInputStream` which is chained to the `FileInputStream`:

```
DataInputStream inputStream = new DataInputStream(inputFile);
```

3. Read Java primitive values in the same order they were written out, using relevant `readX()` methods :

```
boolean v = inputStream.readBoolean();
char    c = inputStream.readChar();
byte    b = inputStream.readByte();
short   s = inputStream.readShort();
int     i = inputStream.readInt();
long    l = inputStream.readLong();
float   f = inputStream.readFloat();
double  d = inputStream.readDouble();
```

4. Close the filter stream, which also closes the underlying stream:

```
inputStream.close();
```

Example 18.3 uses both the procedures described above: first to write and then to read some Java primitive values to and from a file. The values are also written to the terminal.

Example 18.3 *Reading and Writing Java Primitive Values*

```
import java.io.*;

public class JavaPrimitiveValues {
    public static void main(String args[]) throws IOException {
        // Create a FileOutputStream.
        FileOutputStream outputFile =
                 new FileOutputStream("primitives.data");

        // Create a DataOutputStream which is chained to the FileOutputStream.
        DataOutputStream outputStream = new DataOutputStream(outputFile);

        // Write Java primitive values.
        outputStream.writeBoolean(true);
        outputStream.writeChar('A');               // int written as Unicode char
        outputStream.writeByte(Byte.MAX_VALUE);    // int written as 8-bits byte
```

```
            outputStream.writeShort(Short.MIN_VALUE);  // int written as 16-bits short
            outputStream.writeInt(Integer.MAX_VALUE);
            outputStream.writeLong(Long.MIN_VALUE);
            outputStream.writeFloat(Float.MAX_VALUE);
            outputStream.writeDouble(Math.PI);

            // Close the output stream, which also closes the underlying stream.
            outputStream.close();

            // Create a FileInputStream.
            FileInputStream inputFile = new FileInputStream("primitives.data");

            // Create a DataInputStream which is chained to the FileInputStream.
            DataInputStream inputStream = new DataInputStream(inputFile);

            // Read Java primitive values in the same order they were written out.
            boolean v = inputStream.readBoolean();
            char    c = inputStream.readChar();
            byte    b = inputStream.readByte();
            short   s = inputStream.readShort();
            int     i = inputStream.readInt();
            long    l = inputStream.readLong();
            float   f = inputStream.readFloat();
            double  d = inputStream.readDouble();

            // Close the input stream, which also closes the underlying stream.
            inputStream.close();

            // Write the values read on the terminal
            System.out.println(v);
            System.out.println(c);
            System.out.println(b);
            System.out.println(s);
            System.out.println(i);
            System.out.println(l);
            System.out.println(f);
            System.out.println(d);
        }
    }
```

Output from the program:

```
true
A
127
-32768
2147483647
-9223372036854775808
3.4028235E38
3.141592653589793
```

Buffered Byte Streams

The filter classes `BufferedInputStream` and `BufferedOutputStream` implement *buffering of bytes* for input and output streams, respectively. Data is read and written in *blocks*

of bytes, rather than a single byte at a time. Buffering can enhance performance significantly. These filter classes only provide methods for reading and writing bytes. A buffering filter must be chained to an underlying stream:

```
BufferedInputStream(InputStream in)
BufferedOutputStream(OutputStream out)
```

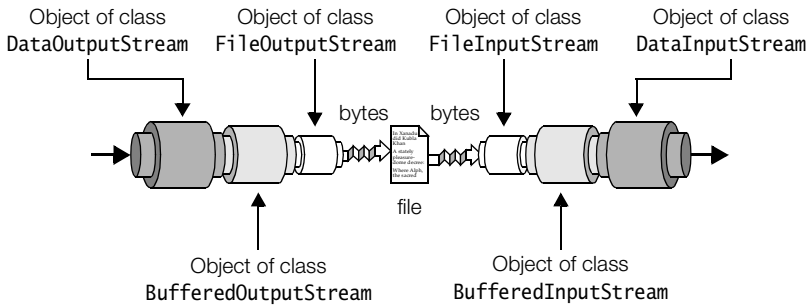

Figure 18.3 *Buffering Byte Streams*

Other filters can be chained to byte buffering filters to provide buffering of data. For example, during the writing of binary representations of Java primitive values to a file, bytes can be buffered (see Figure 18.3):

```
FileOutputStream outputFile = new FileOutputStream("primitives.data");
BufferedOutputStream bufferedOutput = new BufferedOutputStream(outputFile);
DataOutputStream outputStream = new DataOutputStream(bufferedOutput);
```

Values are now written using the `DataOutputStream` `outputStream`, with the buffering of bytes being provided by the `BufferedOutputStream` `bufferedOutput`.

Likewise, during the reading of binary representations of Java primitive values from a file, bytes can be buffered (see Figure 18.3):

```
FileInputStream inputFile = new FileInputStream("primitives.data");
BufferedInputStream bufferedInput = new BufferedInputStream(inputFile);
DataInputStream inputStream = new DataInputStream(bufferedInput);
```

Values are now read using the `DataInputStream` `inputStream`, with the buffering of bytes being provided by the `BufferedInputStream` `bufferedInput`.

Comparison of Byte Output Streams and Input Streams

Usually an output stream has a corresponding input stream of the same type. The table below shows the correspondence between byte output and input streams. Note that not all classes have a corresponding counterpart.

Table 18.4 *Comparing Output Streams and Input Streams*

OutputStreams	InputStreams
`ByteArrayOutputStream`	`ByteArrayInputStream`
`FileOutputStream`	`FileInputStream`
`FilterOutputStream`	`FilterInputStream`
`BufferedOutputStream`	`BufferedInputStream`
`DataOutputStream`	`DataInputStream`
No counterpart	`PushbackInputStream`
`ObjectOutputStream`	`ObjectInputStream`
`PipedOutputStream`	`PipedInputStream`
No counterpart	`SequenceInputStream`

Review questions

18.1 Which of these can act both as an input stream and as an output stream, based on the classes provided by the `java.io` package?

Select all valid answers.

(a) A file
(b) A network connection
(c) A pipe
(d) A string
(e) An array of chars

18.2 Which of these statements about the constant named `separator` of the `File` class are true?

Select all valid answers.

(a) The variable is of type `char`.
(b) The variable is of type `String`.
(c) It can be assumed that the value of the variable always is the character `'/'`.
(d) It can be assumed that the value of the variable always is one of `'/'`, `'\'` or `':'`.
(e) The separator can consist of more than one character.

18.3 Which one of these methods in the `File` class will return the name of the entry, excluding the specification of the directory in which it resides?

Select the one right answer.

(a) `getAbsolutePath()`
(b) `getName()`
(c) `getParent()`
(d) `getPath()`
(e) None of the above.

18.4 What will the method `length()` in the class `File` return?

Select the one right answer.

(a) The number of characters in the file.
(b) The number of kilobytes in the file.
(c) The number of lines in the file.
(d) The number of words in the file.
(e) None of the above.

18.5 A file is readable but not writable on the file system of the host. What will be the result of calling the method `canWrite()` on a `File` object representing this file?

Select the one right answer.

(a) A `SecurityException` is thrown.
(b) The boolean value `false` is returned.
(c) The boolean value `true` is returned.
(d) The file is modified from being unwritable to being writable.
(e) None of the above.

18.6 What is the type of the parameter given to the method `renameTo()` in the class `File`?

Select the one right answer.

(a) `File`
(b) `FileDescriptor`
(c) `FileNameFilter`
(d) `String`
(e) `char[]`

18.7 If `write(0x01234567)` is called on an instance of `OutputStream`, what will be written to the destination of the stream?

Select the one right answer.

(a) The bytes `0x01`, `0x23`, `0x34`, `0x45` and `0x67`, in that order.
(b) The bytes `0x67`, `0x45`, `0x34`, `0x23` and `0x01`, in that order.
(c) The byte `0x01`.
(d) The byte `0x67`.
(e) None of the above.

18.8 Given the following code, under which circumstances will the method return `false`?

```
public static boolean test(InputStream is) throws IOException {
    int value = is.read();
    return value == (value & 0xff);
}
```

Select all valid answers.

(a) A character of more than 8 bits was read from the stream.
(b) An I/O error occurred.
(c) Never.
(d) The end of the input was reached in the input stream.

18.9 Which of these classes provides methods for writing binary representations of primitive Java types?

Select all valid answers.

(a) `DataOutputStream`
(b) `FileOutputStream`
(c) `ObjectOutputStream`
(d) `PrintStream`
(e) `BufferedOutputStream`

18.4 Character Streams: Readers and Writers

A *character encoding* is a scheme for representing characters. Java programs represent characters internally in the 16-bit Unicode character encoding, but the host platform might use another character encoding to represent characters externally. For example, the ASCII (American Standard Code for Information Interchange) character encoding is widely used to represent characters on many platforms. However, it is only one small subset of the Unicode standard.

The abstract classes `Reader` and `Writer` are the roots of the inheritance hierarchies for streams that read and write *Unicode characters* using a specific character encoding.

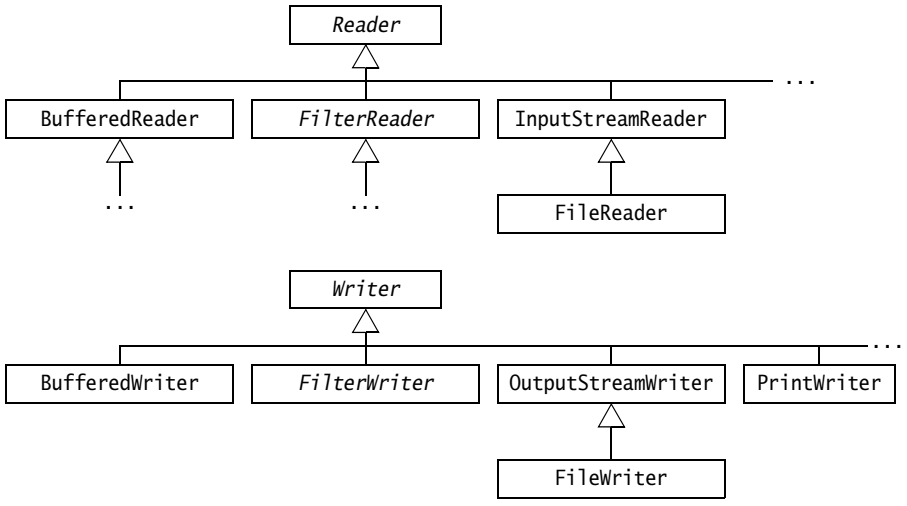

Figure 18.4 *Character Stream Inheritance Hierarchies*

A *reader* is an input character stream that reads a sequence of Unicode characters, and a *writer* is an output character stream that writes a sequence of Unicode characters. Character encodings are used by readers and writers to convert between external encoding and internal Unicode characters. Table 18.5 and Table 18.6 give an overview of the character streams found in the `java.io` package.

Table 18.5 *Readers*

`BufferedReader`	A reader that buffers the characters read from an underlying reader. The underlying reader must be specified, and an optional buffer size can be given.
`LineNumberReader`	A buffered reader that reads characters from an underlying reader while keeping track of the number of *lines* read. The underlying reader must be specified, and an optional buffer size can be given.
`CharArrayReader`	Characters are read from a character array that must be specified.
`FilterReader`	Abstract superclass of all character input stream *filters*. A `FilterReader` must be chained to an underlying reader which must be specified.
`PushbackReader`	A filter that allows characters to be "unread" from a character input stream. A `PushbackReader` must be chained to an underlying reader which must be specified. The number of characters to be unread can optionally be specified.
`InputStreamReader`	Characters are read from a byte input stream which must be specified. The default character encoding is used if no character encoding is explicitly specified.
`FileReader`	Reads characters from a file using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileInputStream` for the file.
`PipedReader`	Reads characters from a `PipedWriter` to which it must be connected. The `PipedWriter` can optionally be specified when creating the `PipedReader`.
`StringReader`	Characters are read from a `String` which must be specified.

Readers use the following methods for reading Unicode characters:

```
int read() throws IOException
int read(char cbuf[]) throws IOException
int read(char cbuf[], int off, int len) throws IOException
```

Note that the `read()` methods read an `int` in the range 0 to 65535 (0x0000–0xFFFF), i.e. a Unicode character. The value –1 is returned if the end of file has been reached.

```
long skip(long n) throws IOException
```

A reader can skip over characters using the `skip()` method.

Table 18.6 *Writers*

`BufferedWriter`	A writer that buffers the characters before writing them to an underlying writer. The underlying writer must be specified, and an optional buffer size can be specified.
`CharArrayWriter`	Characters are written to a character array that grows dynamically. The size of the character array initially created can be specified.
`FilterWriter`	Abstract superclass of all character output stream filters. The `java.io` package does not have any concrete character output stream filters.
`OutputStreamWriter`	Characters are written to a byte output stream which must be specified. The default character encoding is used if no explicit character encoding is specified.
`FileWriter`	Writes characters to a file, using the default character encoding. The file can be specified by a `File` object, a `FileDescriptor`, or a `String` file name. It automatically creates a `FileOutputStream` for the file.
`PipedWriter`	Writes characters to a `PipedReader`, to which it must be connected. The `PipedReader` can optionally be specified when creating the `PipedWriter`.
`PrintWriter`	A filter that allows *textual* representations of Java objects and Java primitive values to be written to an underlying output stream or writer. The underlying output stream or writer must be specified.
`StringWriter`	Characters are written to a `StringBuffer`. The initial size of the `StringBuffer` created can be specified.

Writers use the following methods for writing Unicode characters:

```
void write(int c) throws IOException
```

The `write()` method takes an `int` as argument, but only writes out the least significant 16 bits.

```
void write(char[] cbuf) throws IOException
void write(String str) throws IOException
void write(char[] cbuf, int off, int len) throws IOException
void write(String str, int off, int len) throws IOException
```

These methods write the characters from an array of characters or a string.

```
void close() throws IOException
void flush() throws IOException
```

Like byte streams, a character stream should be closed when no longer needed, to free system resources. Closing a character output stream automatically *flushes* the stream, and a character output stream can also be manually flushed.

Like byte streams, many methods of the character stream classes throw an `IOException` that a calling method must either catch explicitly, or specify in a `throws` clause.

Character Encodings

Every platform has a *default* character encoding that can be used by readers and writers to convert between external encodings and internal Unicode characters. Readers and writers can also explicitly specify which encoding schemes to use for reading and writing. Some common encoding schemes are given in Table 18.7.

Table 18.7 *Encoding Schemes*

Encoding Name	Character Set Name
`8859_1`	ISO Latin-1 (subsumes ASCII)
`8859_2`	ISO Latin-2
`8859_3`	ISO Latin-3
`8859_4`	ISO Latin/Cyrillic
`UTF8`	Standard UTF-8 (UCS Transformation Format; UCS stands for Universal Character Set) (subsumes ASCII)

Not all Unicode characters can be represented in other encoding schemes. In that case, the `'?'` character is usually used to denote any such character in the resulting output, during translation from Unicode.

The raw 16-bit Unicode is not particularly space efficient for storing characters derived from the Latin alphabet, because the majority of the characters can be represented by one byte (same as ASCII), making the higher byte in the 16-bit Unicode superfluous. For this reason, Unicode characters are usually encoded externally, using the UTF8 encoding which has a multi-byte encoding format. It represents ASCII characters as one-byte characters but uses multiple bytes for others. The readers and writers can correctly and efficiently translate between UTF8 and Unicode.

The class `OutputStreamWriter` implements writers that can translate Unicode characters into bytes, using a character encoding which can be either the default encoding of the host platform or an encoding that is explicitly specified, and write the resulting bytes to a byte output stream:

```
OutputStreamWriter(OutputStream out)
```

This creates a writer that uses the default character encoding.

```
OutputStreamWriter(OutputStream out, String encodingName)
       throws UnsupportedEncodingException
```

This creates a writer that uses the specified character encoding.

The class `InputStreamReader` implements readers that can read bytes in the default character encoding or a particular character encoding from an input stream, and translate them to Unicode characters:

```
InputStreamReader(InputStream in)
```

This creates a reader that reads bytes in the default character encoding.

```
InputStreamReader(InputStream in, String encodingName)
       throws UnsupportedEncodingException
```

This creates a reader that reads bytes in the specified character encoding.

An `InputStreamReader` or an `OutputStreamWriter` can be queried about the encoding scheme it uses:

```
String getEncoding()
```

The `OutputStreamWriter` and the `InputStreamReader` classes provide methods for writing and reading individual characters and arrays of characters to and from byte streams. The `OutputStreamWriter` class in addition provides a method for writing strings to byte output streams.

The rest of this section provides examples that illustrate readers and writers for handling text files, including textual representation of Java primitive values and objects, and usage of character encodings.

Print Writers

The capabilities of the `OutputStreamWriter` and the `InputStreamReader` classes are limited, as they primarily write and read characters.

In order to write textual representation of Java primitive values and objects, a `PrintWriter` should be chained to either a writer or a byte output stream, using one of the following constructors:

```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
```

The `autoFlush` argument specifies whether the `PrintWriter` should be flushed when any `println()` method of the `PrintWriter` class is called.

When the underlying writer is specified, the character encoding supplied by the underlying writer is used. However, an `OutputStream` has no notion of any character encoding, so the necessary intermediate `OutputStreamWriter` is automatically created, which will convert characters into bytes, using the default character encoding.

The `PrintWriter` class provides the following methods for writing textual representation of Java primitive values and objects:

Table 18.8 *Print Methods of the* `PrintWriter` *Class*

print()-methods	*println*-methods
	`println()`
`print(boolean b)`	`println(boolean b)`
`print(char c)`	`println(char c)`
`print(int i)`	`println(int i)`
`print(long l)`	`println(long l)`
`print(float f)`	`println(float f)`
`print(double d)`	`println(double d)`
`print(char[] s)`	`println(char[] s)`
`print(String s)`	`println(String s)`
`print(Object obj)`	`println(Object obj)`

The `println()` methods write the text representation of their argument to the underlying stream, and then append a *line-separator*. The `println()` methods use the correct platform-dependent line-separator. For example, on Unix platforms the line-separator is `'\n'` (linefeed), while on Windows platforms it is `"\r\n"` (carriage return + linefeed) and on the Macintosh it is `'\r'` (carriage return).

The `print()` methods create a textual representation of an object by calling the `toString()` method on the object. The `print()` methods do not throw any `IOException`. Instead, the `checkError()` method of the `PrintWriter` class must be called to check for errors.

Writing Text Files

When writing text to a file using the default character encoding, the following three procedures for setting up a `PrintWriter` are equivalent.

Setting up a `PrintWriter` based on an `OutputStreamWriter` which is chained to a `FileOutputStream` (Figure 18.5a):

1. Create a `FileOutputStream`:

   ```
   FileOutputStream outputFile = new FileOutputStream("info.txt");
   ```

2. Create an `OutputStreamWriter` which is chained to the `FileOutputStream`:

   ```
   OutputStreamWriter outputStream = new OutputStreamWriter(outputFile);
   ```

   The `OutputStreamWriter` uses the default character encoding for writing the characters to the file.

3. Create a `PrintWriter` which is chained to the `OutputStreamWriter`:

   ```
   PrintWriter printWriter1 = new PrintWriter(outputStream, true);
   ```

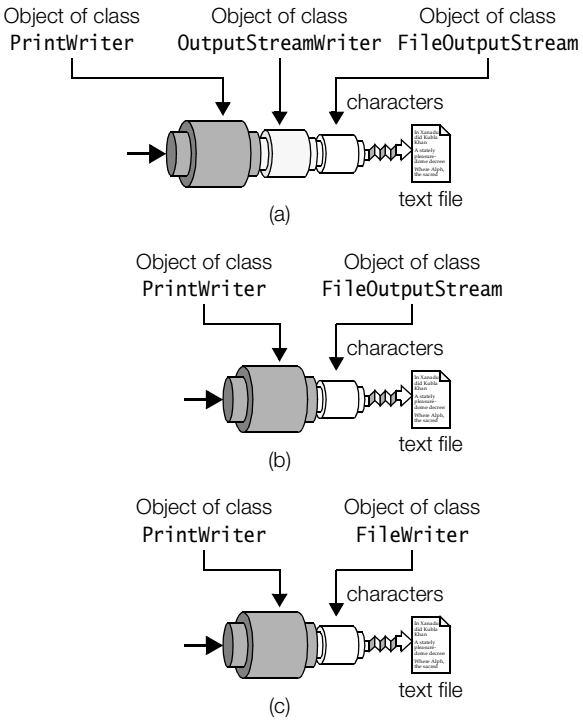

Figure 18.5 *Setting up a Print Writer*

Setting up a `PrintWriter` based on a `FileOutputStream` (Figure 18.5b):

1. Create a `FileOutputStream`:

   ```
   FileOutputStream outputFile = new FileOutputStream("info.txt");
   ```

2. Create a `PrintWriter` which is chained to the `FileOutputStream`:

   ```
   PrintWriter printWriter2 = new PrintWriter(outputFile, true);
   ```

   The intermediate `OutputStreamWriter` to convert the characters using the default encoding is automatically supplied.

Setting up a `PrintWriter` based on a `FileWriter` (Figure 18.5c):

1. Create a `FileWriter` which is a subclass of `OutputStreamWriter`:

   ```
   FileWriter fileWriter = new FileWriter("info.txt");
   ```

   This is equivalent to having an `OutputStreamWriter` chained to a `FileOutputStream` for writing the characters to the file, as shown in Figure 18.5a.

2. Create a `PrintWriter` which is chained to the `FileWriter`:

   ```
   PrintWriter printWriter3 = new PrintWriter(fileWriter, true);
   ```

If a specific character encoding is desired for the writer, then the first procedure (Figure 18.5a) must be used, the encoding being specified for the `OutputStreamWriter`:

```
FileOutputStream   outputFile   = new FileOutputStream("info.txt");
OutputStreamWriter outputStream = new OutputStreamWriter(outputFile, "8859_1");
PrintWriter        printWriter4 = new PrintWriter(outputStream, true);
```

This writer will use the 8859_1 character encoding to write the characters to the file. A `BufferedWriter` can be used to improve the efficiency of writing to the underlying stream.

Reading Text Files

Java primitive values and objects cannot be read directly from their textual representation. Characters must be read and converted to the relevant values explicitly. One common strategy is to write *lines of text* and tokenize the characters as they are read, a line at a time.

When reading characters from a file using the default character encoding, the following two procedures for setting up an `InputStreamReader` are equivalent.

Set up an `InputStreamReader` which is chained to a `FileInputStream` (Figure 18.6a):

1. Create a `FileInputStream`:

   ```
   FileInputStream inputFile = new FileInputStream("info.txt");
   ```

2. Create an `InputStreamReader` which is chained to the `FileInputStream`:

   ```
   InputStreamReader reader = new InputStreamReader(inputFile);
   ```

   The `InputStreamReader` uses the default character encoding for reading the characters from the file.

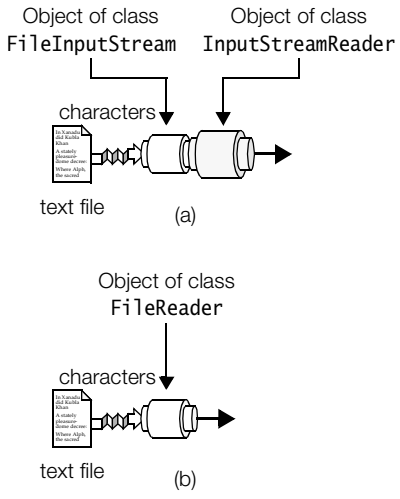

Figure 18.6 *Setting up Readers*

Set up a `FileReader` which is a subclass of `InputStreamReader` (Figure 18.6b):

1. Create a `FileReader`:

   ```
   FileReader fileReader = new FileReader("info.txt");
   ```

   This is equivalent to having an `InputStreamReader` chained to a `FileInputStream` for reading the characters from the file, using the default character encoding.

If a specific character encoding is desired for the reader, then the first procedure must be used (Figure 18.6a), the encoding being specified for the `InputStreamReader`:

```
FileInputStream inputFile = new FileInputStream("info.txt");
InputStreamReader reader = new InputStreamReader(inputFile, "8859_1");
```

This reader will use the 8859_1 character encoding to read the characters from the file.

Buffered Character Streams

To improve the efficiency of I/O operations, readers and writers can buffer their input and output. For this purpose, a `BufferedWriter` or a `BufferedReader` can be chained to the underlying writer or reader, respectively:

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
BufferedReader(Reader in)
BufferedReader(Reader in, int size)
```

The default buffer size is used, unless the buffer size is explicitly specified.

The `BufferedReader` class provides the method `readLine()` to read a line of text from the underlying reader:

```
String readLine() throws IOException
```

The `null` value is returned when the end of input is reached. The returned string must explicitly be converted to other values.

The `BufferedWriter` class provides the method `newLine()` for writing the platform-dependent line-separator.

Using Buffered Writers

The following code creates a `PrintWriter` whose output is buffered, and the characters are written using the 8859_1 character encoding (Figure 18.7a):

```
FileOutputStream   outputFile      = new FileOutputStream("info.txt");
OutputStreamWriter outputStream    = new OutputStreamWriter(outputFile, "8859_1");
BufferedWriter     bufferedWriter1 = new BufferedWriter(outputStream);
PrintWriter        printWriter1    = new PrintWriter(bufferedWriter1, true);
```

The following code creates a `PrintWriter` whose output is buffered, and the characters are written using the default character encoding (Figure 18.7b):

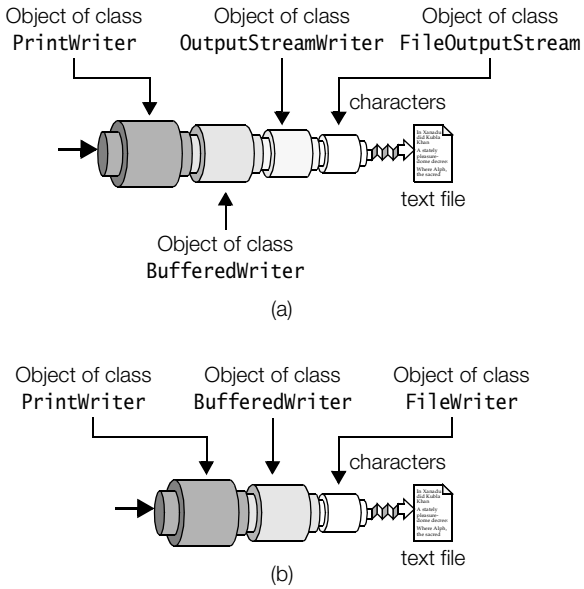

Figure 18.7 *Buffered Writers*

```
FileWriter     fileWriter      = new FileWriter("info.txt");
BufferedWriter bufferedWriter2 = new BufferedWriter(fileWriter);
PrintWriter    printWriter2    = new PrintWriter(bufferedWriter2, true);
```

Note that in both cases the `PrintWriter` is used to write the characters. The `BufferedWriter` is sandwiched between the `PrintWriter` and the underlying `OutputStreamWriter`.

Using Buffered Readers

The following code creates a `BufferedReader` that can be used to read text lines from a file, using the 8859_1 character encoding (Figure 18.8a):

```
FileInputStream   inputFile       = new FileInputStream("info.txt");
InputStreamReader reader          = new InputStreamReader(inputFile, "8859_1");
BufferedReader    bufferedReader1 = new BufferedReader(reader);
```

The following code creates a `BufferedReader` that can be used to read text lines from a file, using the default character encoding (Figure 18.8b):

```
FileReader     fileReader      = new FileReader("lines.txt");
BufferedReader bufferedReader2 = new BufferedReader(fileReader);
```

Note that in both cases the `BufferedReader` object is used to read the text lines.

In contrast to Example 18.3, which demonstrated the reading and writing of binary representations of primitive data values, Example 18.4 shows the reading and writing of textual representations of primitive data values.

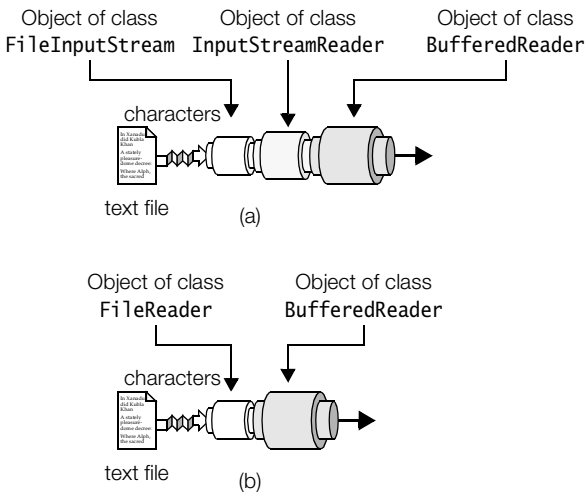

Figure 18.8 *Buffered Readers*

The `CharEncodingDemo` class in Example 18.4 writes textual representations of Java primitive values, using the 8859_1 character encoding (Figure 18.7a). The `PrintWriter` is buffered. Its underlying writer uses the specified encoding, as shown at (1). Values are written out with one value on each line, as shown at (2), and the writer is closed, as shown at (3). The example uses the same character encoding to read the values. A `BufferedReader` is created (Figure 18.8a). Its underlying reader uses the specified encoding, as shown at (4). The values are read in the same order they were written out, one value per line. The line is explicitly converted to an appropriate value, as shown at (5). The `BufferedReader` is closed, as shown at (6), and the values are echoed on the terminal, as shown at (7). Note the exceptions that are specified in the `throws` clause of the `main()` method.

Example 18.4 *Demonstrating Readers and Writers, and Character Encoding*

```
import java.io.*;

public class CharEncodingDemo {

    public static void main(String args[])
                    throws IOException, NumberFormatException {
        // character encoding.                                          (1)
        FileOutputStream outputFile = new FileOutputStream("info.txt");
        OutputStreamWriter writer = new OutputStreamWriter(outputFile, "8859_1");
        BufferedWriter bufferedWriter1 = new BufferedWriter(writer);
        PrintWriter printWriter = new PrintWriter(bufferedWriter1, true);
        System.out.println("Writing using encoding: " + writer.getEncoding());

        // Print Java primitive values, one on each line.               (2)
        printWriter.println(true);
        printWriter.println('A');
        printWriter.println(Byte.MAX_VALUE);
```

```
            printWriter.println(Short.MIN_VALUE);
            printWriter.println(Integer.MAX_VALUE);
            printWriter.println(Long.MIN_VALUE);
            printWriter.println(Float.MAX_VALUE);
            printWriter.println(Math.PI);

            // Close the writer, which also closes the underlying stream      (3)
            printWriter.close();

            // Create a BufferedReader which uses 8859_1 character encoding   (4)
            FileInputStream inputFile = new FileInputStream("info.txt");
            InputStreamReader reader = new InputStreamReader(inputFile, "8859_1");
            BufferedReader bufferedReader = new BufferedReader(reader);
            System.out.println("Reading using encoding: " + reader.getEncoding());

            // Read Java primitive values in the same order they              (5)
            // were written out, one on each line
            boolean v = bufferedReader.readLine().equals("true")? true : false;
            char    c = bufferedReader.readLine().charAt(0);
            byte    b = (byte) Integer.parseInt(bufferedReader.readLine());
            short   s = (short) Integer.parseInt(bufferedReader.readLine());
            int     i = Integer.parseInt(bufferedReader.readLine());
            long    l = Long.parseLong(bufferedReader.readLine());
            float   f = Float.parseFloat(bufferedReader.readLine());
            double  d = Double.parseDouble(bufferedReader.readLine());

            // Close the reader, which also closes the underlying stream      (6)
            bufferedReader.close();

            // Write the values read on the terminal                          (7)
            System.out.println("Values:");
            System.out.println(v);
            System.out.println(c);
            System.out.println(b);
            System.out.println(s);
            System.out.println(i);
            System.out.println(l);
            System.out.println(f);
            System.out.println(d);
        }
    }
```

Output from the program:

```
Writing using encoding: ISO8859_1
Reading using encoding: ISO8859_1
Values:
true
A
127
-32768
2147483647
-9223372036854775808
3.4028235E38
3.141592653589793
```

Terminal I/O

The *standard output* stream (usually the screen) is represented by the `PrintStream` object `System.out`. The *standard input* stream (usually the keyboard) is represented by the `InputStream` object `System.in`. In other words, it is a byte input stream. The *standard error* stream (also usually the screen) is represented by `System.err` which is another object of the `PrintStream` class. The `PrintStream` class is now mostly deprecated, but its `print()` methods, which act as corresponding `print()` methods from the `PrintWriter` class, can still be used to write output to `System.out` and `System.err`. In other words, both `System.out` and `System.err` act like `PrintWriter`, but in addition they have `write()` methods for writing bytes.

In order to read and translate characters correctly and efficiently, `System.in` should be chained to an `InputStreamReader` that in turn should be buffered:

```
InputStreamReader inStream = new InputStreamReader(System.in);
BufferedReader stdInStream = new BufferedReader(inStream);
```

In this case, the default character encoding is used to translate the characters.

In Example 18.5, a `BufferedReader` is chained to an `InputStreamReader` that in turn is chained to `System.in`, as shown at (1). This allows the characters from the standard input stream to be buffered and read using the default character encoding. The `BufferedReader` in the example always reads a whole line at a time from the terminal. If a line of text is requested, the whole line read is returned, as shown at (3). If an `int` is to be read, the line is parsed to an `int`, as shown at (5). If a `double` is to be read, the line is parsed to a `double`, as shown at (7). Note the exception handling that is necessary to read a line of characters and ensure that it contains a valid numerical value.

The Java class libraries provide a class named `java.text.NumberFormat` that can be used to format numeric values according to a specified locale. At (8), the example uses a `NumberFormat` object created to format values according to the locale `java.util.Locale.US`.

Example 18.5 *Demonstrating Terminal I/O*

```
import java.io.*;
import java.text.*;
import java.util.*;

public final class Stdin {

    // A BufferedReader chained to an InputStreamReader chained to an InputStream.
    private static BufferedReader reader = new BufferedReader(         // (1)
            new InputStreamReader(System.in)
    );

    // Read one line of text from the terminal and return it as a string.
    public static String readLine() {                                  // (2)
        while (true) try {
            return reader.readLine();                                  // (3)
```

```
        } catch(IOException ioe) {
            reportError(ioe);
        }
    }

    // Read one integer value from the terminal.
    public static int readInteger() {                                   // (4)
        while (true) try {
            return Integer.parseInt(reader.readLine());                 // (5)
        } catch (IOException ioe) {
            reportError(ioe);
        } catch(NumberFormatException nfe) {
            reportError(nfe);
        }
    }

    // Read one double value from the terminal.
    public static double readDouble() {                                 // (6)
        while (true) try {
            return Double.parseDouble(reader.readLine());               // (7)
        } catch(IOException ioe) {
            reportError(ioe);
        } catch(NumberFormatException nfe) {
            reportError(nfe);
        }
    }

    private static void reportError(Exception e) {
        System.err.println("Error in input: " + e);
        System.err.println("Please re-enter data.");
    }

    public static void main(String args[]) {
        System.out.println("Input a string:");
        String str = Stdin.readLine();
        System.out.println("Input an integer:");
        int i = Stdin.readInteger();
        System.out.println("Input a double:");
        double d = Stdin.readDouble();

        NumberFormat formatter = NumberFormat.getInstance(Locale.US);  // (8)
        System.out.println("Data read:");
        System.out.println(str);
        System.out.println(formatter.format(i));
        System.out.println(formatter.format(d));
    }
}
```

Output from the program:

```
Input a string:
Habari
Input an integer:
0201596148
Input a double:
47.584152
```

```
Data read:
Habari
201,596,148
47.584
```

Comparison of Character Writers and Readers

Usually a writer has a corresponding reader. Table 18.9 shows the correspondence between character output and character input streams. Note that not all classes have a corresponding counterpart.

Table 18.9 *Correspondence between Writers and Readers*

Writers	Readers
`BufferedWriter`	`BufferedReader`
No counterpart	`LineNumberReader`
`CharArrayWriter`	`CharArrayReader`
`FilterWriter`	`FilterReader`
No counterpart	`PushbackReader`
`OutputStreamWriter`	`InputStreamReader`
`FileWriter`	`FileReader`
`PipedWriter`	`PipedReader`
`PrintWriter`	*No counterpart*
`StringWriter`	`StringReader`

Comparison of Byte Streams and Character Streams

It is instructive to see which byte streams correspond to which character streams. Table 18.10 shows the correspondence between byte and character streams. Note that not all classes have a corresponding counterpart.

Table 18.10 *Correspondence between Byte Streams and Character Streams*

Byte Streams	Character Streams
`OutputStream` `InputStream`	`Writer` `Reader`
`ByteArrayOutputStream` `ByteArrayInputStream`	`CharArrayWriter` `CharArrayReader`
No counterpart *No counterpart*	`OutputStreamWriter` `InputStreamReader`

Table 18.10 *Correspondence between Byte Streams and Character Streams (continued)*

Byte Streams	Character Streams
`FileOutputStream` `FileInputStream`	`FileWriter` `FileReader`
`FilterOutputStream` `FilterInputStream`	`FilterWriter` `FilterReader`
`BufferedOutputStream` `BufferedInputStream`	`BufferedWriter` `BufferedReader`
`PrintStream`	`PrintWriter`
`DataOutputStream` `DataInputStream`	*No counterpart* *No counterpart*
`ObjectOutputStream` `ObjectInputStream`	*No counterpart* *No counterpart*
`PipedOutputStream` `PipedInputStream`	`PipedWriter` `PipedReader`
No counterpart *No counterpart*	`StringWriter` `StringReader`
No counterpart	`LineNumberReader`
`PushbackInputStream`	`PushbackReader`
`SequenceInputStream`	*No counterpart*

Review questions

18.10 Which of these are valid parameter types for the `write()` methods of the `Writer` class?

Select all valid answers.

(a) Type `String`
(b) Type `char`
(c) Type `char[]`
(d) Type `int`

18.11 What is the default encoding for an `OutputStreamWriter`?

Select the one right answer.

(a) 8859_1
(b) UTF8
(c) Unicode
(d) The default is system-dependent.
(e) The default is not system-dependent, but is none of the above.

18.12 Which of these integer types do not have their own `print()` method in the `PrintWriter` class?

Select all valid answers.

(a) `byte`
(b) `char`
(c) `int`
(d) `long`
(e) All have their own `print()` method.

18.13 How can one access the standard error stream?

Select all valid answers.

(a) It is accessed as a member of the class `System.err`.
(b) It is accessed as a static variable named `out` in the class `System`.
(c) It is accessed as a static variable named `err` in the class `System`.
(d) It is accessed as a static variable named `err` in the class `Runtime`.
(e) It is returned by a method in the class `System`.

18.5 Random Access for Files

The `RandomAccessFile` class implements *direct access for files*, i.e. bytes can be read from or written to any specified location in a file. The `RandomAccessFile` class inherits directly from the `Object` class. It implements both the `DataInput` and `DataOutput` interfaces, meaning that Java primitive values can be written and read from a random access file. However, note that objects of the `RandomAccessFile` class cannot be chained with streams.

A random access file must be created and assigned to a file, before it can be used.

```
RandomAccessFile(String name, String mode) throws IOException
RandomAccessFile(File file, String mode) throws IOException
```

The file is specified by a file name or by a `File` object. The mode argument must be equal to either `"r"` (for reading) or `"rw"` (for both reading and writing), otherwise an `IllegalArgumentException` is thrown. Note that opening the file for writing does not reset the contents of the file. The file should have the access specified in the constructor.

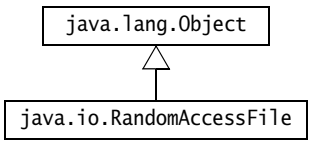

Figure 18.9 *Random Access File Inheritance Hierarchy*

An `IOException` is thrown if an I/O error occurs, most notably when the mode is `"r"` and the file does not exist. However, if the mode is `"rw"` and the file does not exist, a new empty file is created. Regardless of the mode, if the file does exist, its file pointer is set to the beginning of the file.

A `SecurityException` is thrown if the application does not have the necessary access rights.

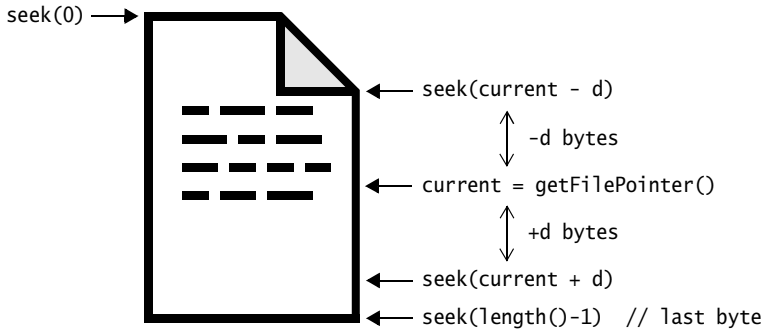

Figure 18.10 *Positioning the File Pointer for Direct File Access*

A *file pointer* indicates the next location in the file where bytes can be read from or written to. The current position of the file pointer can be obtained by using the `getFilePointer()` method:

```
long getFilePointer() throws IOException
```

The number of bytes in the file can be obtained by using the `length()` method:

```
long length() throws IOException
```

The file pointer can be positioned using the `seek()` method:

```
void seek(long offset) throws IOException
```

The `offset` argument specifies the position from the beginning of the file, the first byte being at position 0. The position will be the target of the next read or write operation. See Figure 18.10.

When a random access file is no longer needed, it should be closed, to free the resources:

```
void close() throws IOException
```

Example 18.6 illustrates usage of random access files. The program creates a file and writes the byte representation of the squares of numbers from 0 to 9. It then reads the squares of odd numbers back from the file, using direct access. The squares are represented as `int` values. The file is then extended with the squares of numbers from 10 to 19, and again the squares of odd numbers are read from the file. In the method `createFile()`, the initial file is created using a `RandomAccessFile`

object with "rw" mode. The squares of odd numbers are read in the method readFile() using a RandomAccessFile object with "r" mode, which opens the file for direct read access. The numbers are read after the current file pointer value is incremented with the size of an int value, thereby reading every other integer from the file, as shown at (1).

The file is extended in the method extendFile(). The file is opened for direct read and write access. The file pointer is first positioned at the end of the file, before writing the new numbers as shown at (2).

The output from the program shows that only squares of odd numbers were read from the file.

Example 18.6 *Random Access File*

```
import java.io.*;

public class RandomAccessDemo {
    static String fileName = "new-numbers.data";

    final static int INT_SIZE = 4;

    public static void main(String args[]) {
        try {
            RandomAccessDemo random = new RandomAccessDemo();
            random.createFile();
            random.readFile();
            random.extendFile();
            random.readFile();
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }

    // Create a file with squares of numbers from 0 to 9.
    public void createFile() throws IOException {
        File dataFile = new File(fileName);
        RandomAccessFile outputFile = new RandomAccessFile(dataFile, "rw");
        for (int i = 0; i < 10; i++)
            outputFile.writeInt(i*i);
        outputFile.close();
    }

    // Read every other number from the file i.e. the squares of odd numbers
    public void readFile() throws IOException {
        File dataFile = new File(fileName);
        RandomAccessFile inputFile = new RandomAccessFile(dataFile, "r");
        System.out.println("Squares of odd numbers from the file:");
        long length = inputFile.length();
        for (int i = INT_SIZE; i < length; i += 2 * INT_SIZE) {
            inputFile.seek(i);                                          // (1)
            System.out.println(inputFile.readInt());
        }
        inputFile.close();
    }
```

```
    // Extend the file with squares from 10 to 19.
    public void extendFile() throws IOException {
        RandomAccessFile outputFile = new RandomAccessFile(fileName, "rw");
        outputFile.seek(outputFile.length());                          // (2)
        for (int i = 10; i < 20; i++)
            outputFile.writeInt(i*i);
        outputFile.close();
    }
}
```

Output from the program:

```
Squares of odd numbers from the file:
1
9
25
49
81
Squares of odd numbers from the file:
1
9
25
49
81
121
169
225
289
361
```

Review questions

18.14 Which of the these are valid access mode specifiers for a constructor of the `RandomAccessFile` class?

Select all valid answers.

(a) `""`
(b) `"r"`
(c) `"rw"`
(d) `"w"`
(e) `"wr"`
(f) `null`

18.15 Which of the following method calls would, if executed on a `RandomAccessFile` object, position the file pointer so that reading the last byte of the file could be done with a single call to `read()`?

Select the one right answer.

(a) `seek(length())`
(b) `seek(length()+1)`

(c) `seek(length()+2)`
(d) `seek(length()-1)`
(e) `seek(length()-2)`

18.6 Object Serialization

Object serialization allows an object to be transformed into a sequence of bytes that can later be re-created (*deserialized*) into the original object. After deserialization the object has the same state as it had when it was serialized, barring any data members that were not serializable. Java provides this facility through the `ObjectInput` and `ObjectOutput` interfaces, which allow the reading and writing of objects from and to streams. These interfaces extend the `DataInput` and `DataOutput` interfaces respectively.

The `ObjectOutputStream` class implements the `ObjectOutput` interface. This means that the `ObjectOutputStream` class provides methods to write objects as well as bytes, text and Java primitive values. Similarly `ObjectInputStream` class implements the `ObjectInput` interface. This means that the `ObjectInputStream` class provides methods to read objects as well as bytes, text and Java primitive values. Figure 18.11 gives an overview of how these classes can be chained and the methods they provide.

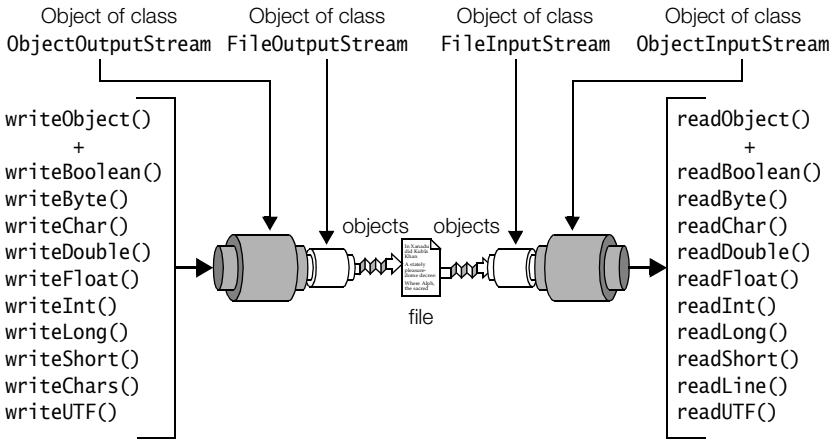

Figure 18.11 *Object Stream Chaining*

`ObjectOutputStream` Class

The class `ObjectOutputStream` can write objects to any stream that is a subclass of the `OutputStream`, for example to a file or a network connection (socket). An `ObjectOutputStream` must be chained to an `OutputStream`, using the following constructor:

```
ObjectOutputStream(OutputStream out) throws IOException
```

For example, in order to store objects in a file and thus provide persistent storage for objects, an `ObjectOutputStream` can be chained to a `FileOutputStream`:

```
FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
ObjectOutputStream outputStream = new ObjectOutputStream(outputFile);
```

Objects can be written to the stream, using the `writeObject()` method of the `ObjectOutputStream` class:

```
final void writeObject(Object obj) throws IOException
```

The `writeObject()` method can be used to write *any* object to a stream, including strings and arrays, as long as the object supports the `java.io.Serializable` interface, which is a marker interface with no methods. The `String` class and all array types implement the `Serializable` interface. A serializable object can be any compound object containing references to other objects, and all constituent objects that are serializable are serialized recursively when the compound object is written out. Each object is written out once during serialization. The following information is included when an object is serialized:

- the class information needed to reconstruct the object.
- the values of all serializable non-transient and non-static members, including those that are inherited.

`ObjectInputStream` Class

An `ObjectInputStream` is used to restore (*deserialize*) objects that have previously been serialized using an `ObjectOutputStream`. An `ObjectInputStream` must be chained to an `InputStream`, using the following constructor:

```
ObjectInputStream(InputStream in)
        throws IOException, StreamCorruptedException
```

For example, in order to restore objects from a file, an `ObjectInputStream` can be chained to a `FileInputStream`:

```
FileInputStream inputFile = new FileInputStream("obj-storage.dat");
ObjectInputStream inputStream = new ObjectInputStream(inputFile);
```

The method `readObject()` of the `ObjectInputStream` class is used to read an object from the stream:

```
final Object readObject()
        throws OptionalDataException, ClassNotFoundException, IOException
```

Note that the reference returned is of type `Object` regardless of the actual type of the retrieved object, and can be cast to the desired type. Objects and values must be read in the same order as when they were serialized.

Serializable, non-transient data members of an object, including those data members that are inherited, are restored to the values they had at the time of serialization. For compound objects containing references to other objects, the constituent objects are read to re-create the whole object structure. In order to deserialize objects, the appropriate classes must be available at runtime. Note that new objects are created during deserialization, so that no existing objects are overwritten.

The class `ObjectSerializationDemo` in Example 18.7 serializes some objects in the `writeData()` method at (1), and then deserializes them in the `readData()` method at (2). The `readData()` method also writes the data to the standard output stream.

The `writeData()` method writes the following: an array of strings (`strArray`), a `long` value (`num`), an array of `int` values (`intArray`), and lastly a `String` object (`commonStr`) which is shared with the array of strings, `strArray`. However, this shared `String` object is actually only serialized once. Duplication is automatically avoided when the same object is serialized several times. Note that the array elements and the characters in a `String` object are not written out explicitly one by one. It is enough to specify the object reference in the `writeObject()` method. The method also recursively goes through the array of strings, `strArray`, serializing each `String` object in the array.

The method `readData()` deserializes the data in the order in which it was written. An explicit cast is needed to convert the reference of a deserialized object to the right type. Note that new objects are created by the `readObject()` method, and that an object created during the deserialization process has the same state as the object that was serialized.

Example 18.7 *Object Serialization*

```
// Reading and Writing Objects
import java.io.*;

public class ObjectSerializationDemo {

    void writeData() {                                          // (1)
        try {
            // Setup the Output stream
            FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
            ObjectOutputStream outputStream = new ObjectOutputStream(outputFile);

            // Write data
            String[] strArray = {"Seven", "Eight", "Six"};
            long num = 2001;
            int[] intArray = {1, 3, 1949};
            String commonStr = strArray[2];

            outputStream.writeObject(strArray);
            outputStream.writeLong(num);
            outputStream.writeObject(intArray);
            outputStream.writeObject(commonStr);
```

```
            // Close the stream
            outputStream.flush();
            outputStream.close();
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }

    void readData() {                                    // (2)
        try {
            // Setup the Input stream
            FileInputStream inputFile = new FileInputStream("obj-storage.dat");
            ObjectInputStream inputStream = new ObjectInputStream(inputFile);

            // Read data
            String[] strArray = (String[]) inputStream.readObject();
            long num = inputStream.readLong();
            int[] intArray = (int[]) inputStream.readObject();
            String commonStr = (String) inputStream.readObject();

            // Write data on standard output stream
            for (int i = 0; i < strArray.length; i++) {
                System.out.print(strArray[i] + "\t");
            }
            System.out.println();
            System.out.println(num);
            for (int i = 0; i < intArray.length; i++) {
                System.out.print(intArray[i] + "\t");
            }
            System.out.println();
            System.out.println(commonStr);

            // Close the stream
            inputStream.close();
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }

    public static void main(String args[]) {
        ObjectSerializationDemo demo = new ObjectSerializationDemo();
        demo.writeData();
        demo.readData();
    }
}
```

Output from the program:

```
Seven   Eight   Six
2001
1       3       1949
Six
```

Review questions

18.16 How many methods are defined in the `Serializable` interface?

Select the one right answer.

(a) None
(b) One
(c) Two
(d) Three
(e) None of the above.

18.17 Which of the following best describes the data an `ObjectOutputStream` can write?

Select the one right answer.

(a) Bytes and other primitive Java types.
(b) Object hierarchies.
(c) Object hierarchies and primitive Java types.
(d) Single objects.
(e) Single objects and primitive Java types.

Chapter summary

The following information was included in this chapter:

- Discussion of the `File` class, which provides an interface to the host file system.
- Byte streams, as represented by the `InputStream` and `OutputStream` classes.
- File streams, as represented by the `FileInputStream` and `FileOutputStream` classes.
- Reading and writing Java primitive values using the `DataInputStream` and `DataOutputStream` classes.
- Buffering byte streams for improved efficiency, using the `BufferedInputStream` and `BufferedOutputStream` classes.
- Character streams, as represented by the `Reader` and `Writer` classes.
- Usage of character encodings, including Unicode and UTF8, by the `InputStreamReader` and `OutputStreamWriter` classes.
- Reading and writing text files.
- Buffered character streams, as represented by the `BufferedReader` and `BufferedWriter` classes.
- Terminal I/O using `System.in`, `System.out` and `System.err`.
- Random access files for direct access I/O.
- Object serialization: reading and writing objects.

Programming exercise

18.1 Write a program that reads text from a source using one encoding, and writes the text to a destination using another encoding. The program should have four optional arguments:

- First argument, if present, should specify the encoding of the source. The default source encoding should be `"8859_1"`.
- Second argument, if present, should specify the encoding of the destination. The default destination encoding should be `"UTF8"`.
- Third argument, if present, should specify a source file. If no argument is given, the standard input should be used.
- Fourth argument, if present, should specify a destination file. If no argument is given, the standard output should be used.

Use buffering, and read and write 512 bytes at a time to make the program efficient.

Errors should be written to the standard error stream.