CHAPTER 20

The I/O Package

From a programmer's point of view,
the user is a peripheral that types when you issue a `read` *request.*
—Peter Williams

THE Java platform includes a number of packages that are concerned with the movement of data into and out of programs. These packages differ in the kinds of abstractions they provide for dealing with I/O (input/output).

The `java.io` package defines I/O in terms of *streams.* Streams are ordered sequences of data that have a *source* (input streams) or *destination* (output streams). The I/O classes isolate programmers from the specific details of the underlying operating system, while enabling access to system resources through files and other means. Most stream types (such as those dealing with files) support the methods of some basic interfaces and abstract classes, with few (if any) additions. The best way to understand the I/O package is to start with the basic interfaces and abstract classes.

The `java.nio` package and its subpackages define I/O in terms of *buffers* and *channels.* Buffers are data stores (similar to arrays) that can be read from or written to. Channels represent connections to entities capable of performing I/O operations, including buffers, files, and sockets. The "n" in `nio` is commonly understood as meaning "new" (the `nio` package predates the original stream-based `io` package), but it originally stood for "non-blocking" because one of the key differences between channel-based I/O and stream-based I/O is that channels allow for non-blocking I/O operations, as well as interruptible blocking operations. This is a powerful capability that is critical in the design of high throughput server-style applications.

The `java.net` package provides specific support for network I/O, based around the use of sockets, with an underlying stream or channel-based model.

This chapter is mainly concerned with the stream-based model of the `java.io` package. A short introduction to some of the capabilities of the `java.nio` package is given in “A Taste of New I/O” on page 565, but the use of non-blocking I/O and the `java.net` network I/O are advanced topics, beyond the scope of this book.

20.1 Streams Overview

The package `java.io` has two major parts: *character streams* and *byte streams.* Characters are 16-bit UTF-16 characters, whereas bytes are (as always) 8 bits. I/O is either text-based or data-based (binary). Text-based I/O works with streams of human-readable characters, such as the source code for a program. Data-based I/O works with streams of binary data, such as the bit pattern for an image. The character streams are used for text-based I/O, while byte streams are used for data-based I/O. Streams that work with bytes cannot properly carry characters, and some character-related issues are not meaningful with byte streams—though the byte streams can also be used for older text-based protocols that use 7- or 8-bit characters. The byte streams are called *input streams* and *output streams,* and the character streams are called *readers* and *writers.* For nearly every input stream there is a corresponding output stream, and for most input or output streams there is a corresponding reader or writer character stream of similar functionality, and vice versa.

Because of these overlaps, this chapter describes the streams in fairly general terms. When we talk simply about streams, we mean any of the streams. When we talk about input streams or output streams, we mean the byte variety. The character streams are referred to as readers and writers. For example, when we talk about the `Buffered` streams we mean the entire family of `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, and `BufferedWriter`. When we talk about `Buffered` byte streams we mean both `BufferedInputStream` and `BufferedOutputStream`. When we talk about `Buffered` character streams, we mean `BufferedReader` and `BufferedWriter`.

The classes and interfaces in `java.io` can be broadly split into five groups:

- The general classes for building different types of byte and character streams—input and output streams, readers and writers, and classes for converting between them—are covered in Section 20.2 through to Section 20.4.
- A range of classes that define various types of streams—filtered streams, buffered streams, piped streams, and some specific instances of those streams, such as a line number reader and a stream tokenizer—are discussed in Section 20.5.
- The data stream classes and interfaces for reading and writing primitive values and strings are discussed in Section 20.6.

- Classes and interfaces for interacting with files in a system independent manner are discussed in Section 20.7.
- The classes and interfaces that form the *object serialization* mechanism, which transforms objects into byte streams and allows objects to be reconstituted from the data read from a byte stream, are discussed in Section 20.8.

Some of the output streams provide convenience methods for producing formatted output, using instances of the `java.util.Formatter` class. You get formatted input by binding an input stream to a `java.util.Scanner` object. Details of formatting and scanning are covered in Chapter 22.

The `IOException` class is used by many methods in `java.io` to signal exceptional conditions. Some extended classes of `IOException` signal specific problems, but most problems are signaled by an `IOException` object with a descriptive string. Details are provided in Section 20.9 on page 563. Any method that throws an `IOException` will do so when an error occurs that is directly related to the stream. In particular, invoking a method on a closed stream may result in an `IOException`. Unless there are particular circumstances under which the `IOException` will be thrown, this exception is not documented for each individual method of each class.

Similarly, `NullPointerException` and `IndexOutOfBoundsException` can be expected to be thrown whenever a `null` reference is passed to a method, or a supplied index accesses outside of an array. Only those situations where this does not occur are explicitly documented.

All code presented in this chapter uses the types in `java.io`, and every example has imported `java.io.*` even when there is no explicit `import` statement in the code.

20.2 Byte Streams

The `java.io` package defines abstract classes for basic byte input and output streams. These abstract classes are then extended to provide several useful stream types. Stream types are almost always paired: For example, where there is a `FileInputStream` to read from a file, there is usually a `FileOutputStream` to write to a file.

Before you can learn about specific kinds of input and output byte streams, it is important to understand the basic `InputStream` and `OutputStream` abstract classes. The type tree for the byte streams of `java.io` in Figure 20–1 shows the type hierarchy of the byte streams.

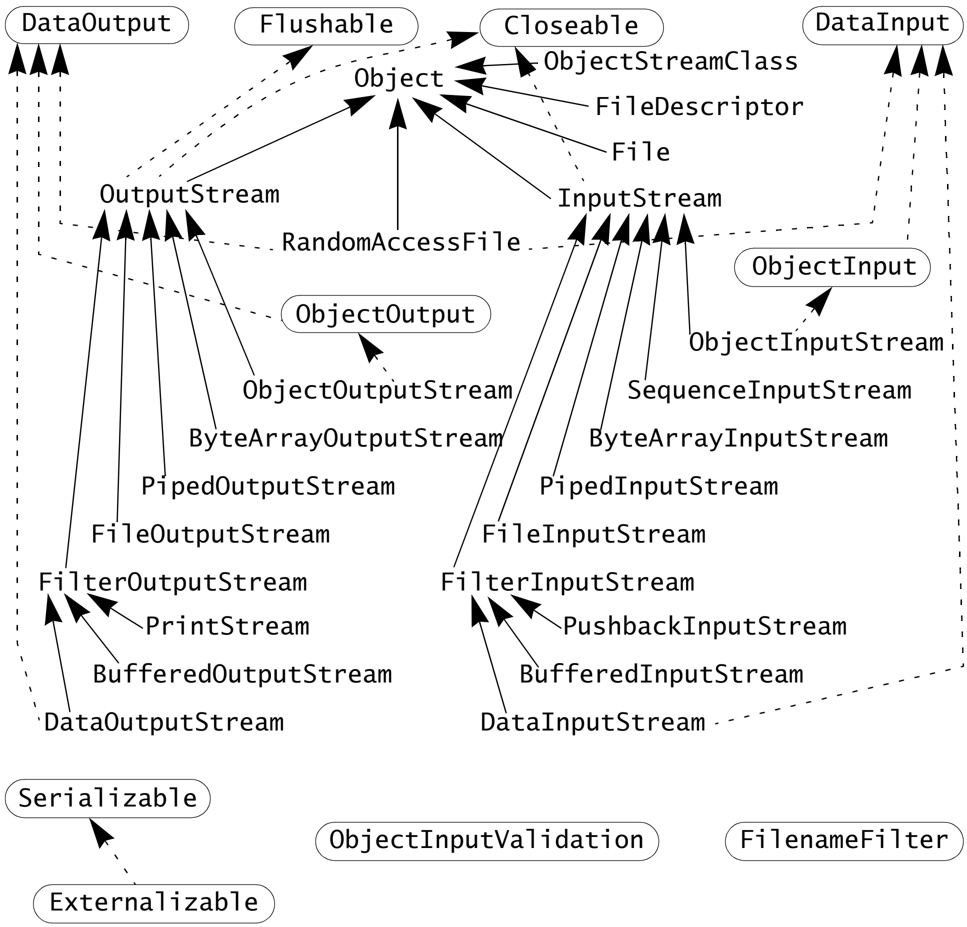


FIGURE 20–1: ***Type Tree for Byte Streams in `java.io`***

All byte streams have some things in common. For example, all streams support the notion of being open or closed. You open a stream when you create it, and can read or write while it is open. You close a stream with its `close` method, defined in the `Closeable`[1] interface. Closing a stream releases resources (such as file descriptors) that the stream may have used and that should be reclaimed as soon as they are no longer needed. If a stream is not explicitly closed it will hold on to these resources. A stream class could define a `finalize` method to release these resources during garbage collection but, as you learned on page 449, that could be too late. You should usually close streams when you are done with them.

[1] Yes, another misspelling.

All byte streams also share common synchronization policies and concurrent behavior. These are discussed in Section 20.5.1 on page 515.

20.2.1 `InputStream`

The abstract class `InputStream` declares methods to read bytes from a particular source. `InputStream` is the superclass of most byte input streams in `java.io`, and has the following methods:

`public abstract int` **`read()`** `throws IOException`

Reads a single byte of data and returns the byte that was read, as an integer in the range 0 to 255, not –128 to 127; in other words, the byte value is treated as unsigned. If no byte is available because the end of the stream has been reached, the value –1 is returned. This method blocks until input is available, the end of stream is found, or an exception is thrown. The `read` method returns an `int` instead of an actual `byte` value because it needs to return all valid byte values plus a flag value to indicate the end of stream. This requires more values than can fit in a `byte` and so the larger `int` is used.

`public int` **`read(byte[] buf, int offset, int count)`**
`throws IOException`

Reads into a part of a `byte` array. The maximum number of bytes read is `count`. The bytes are stored from `buf[offset]` up to a maximum of `buf[offset+count-1]`—all other values in `buf` are left unchanged. The number of bytes actually read is returned. If no bytes are read because the end of the stream was found, the value –1 is returned. If `count` is zero then no bytes are read and zero is returned. This method blocks until input is available, the end of stream is found, or an exception is thrown. If the first byte cannot be read for any reason other than reaching the end of the stream—in particular, if the stream has already been closed—an `IOException` is thrown. Once a byte has been read, any failure that occurs while trying to read subsequent bytes is not reported with an exception but is treated as encountering the end of the stream—the method completes normally and returns the number of bytes read before the failure occurred.

`public int` **`read(byte[] buf)`** `throws IOException`

Equivalent to `read(buf, 0, buf.length)`.

`public long` **`skip(long count)`** `throws IOException`

Skips as many as `count` bytes of input or until the end of the stream is found. Returns the actual number of bytes skipped. If `count` is negative, no bytes are skipped.

`public int` **`available()`** `throws IOException`

Returns the number of bytes that can be read (or skipped over) without blocking. The default implementation returns zero.

`public void` **`close()`** `throws IOException`

Closes the input stream. This method should be invoked to release any resources (such as file descriptors) associated with the stream. Once a stream has been closed, further operations on the stream will throw an `IOException`. Closing a previously closed stream has no effect. The default implementation of `close` does nothing.

The implementation of `InputStream` requires only that a subclass provide the single-byte variant of `read` because the other `read` methods are defined in terms of this one. Most streams, however, can improve performance by overriding other methods as well. The default implementations of `available` and `close` will usually need to be overridden as appropriate for a particular stream.

The following program demonstrates the use of input streams to count the total number of bytes in a file, or from `System.in` if no file is specified:

```
import java.io.*;

class CountBytes {
    public static void main(String[] args)
        throws IOException
    {
        InputStream in;
        if (args.length == 0)
            in = System.in;
        else
            in = new FileInputStream(args[0]);

        int total = 0;
        while (in.read() != -1)
            total++;

        System.out.println(total + " bytes");
    }
}
```

This program takes a filename from the command line. The variable `in` represents the input stream. If a file name is not provided, it uses the standard input stream `System.in`; if one is provided, it creates an object of type `FileInputStream`, which is a subclass of `InputStream`.

The `while` loop counts the total number of bytes in the file. At the end, the results are printed. Here is the output of the program when used on itself:

```
318 bytes
```

You might be tempted to set `total` using `available`, but it won't work on many kinds of streams. The `available` method returns the number of bytes that can be read *without blocking*. For a file, the number of bytes available is usually its entire contents. If `System.in` is a stream associated with a keyboard, the answer can be as low as zero; when there is no pending input, the next `read` will block.

20.2.2 **`OutputStream`**

The abstract class `OutputStream` is analogous to `InputStream`; it provides an abstraction for writing bytes to a destination. Its methods are:

`public abstract void` **`write(int b)`** `throws IOException`

Writes `b` as a byte. The byte is passed as an `int` because it is often the result of an arithmetic operation on a byte. Expressions involving bytes are type `int`, so making the parameter an `int` means that the result can be passed without a cast to `byte`. Note, however, that only the lowest 8 bits of the integer are written. This method blocks until the byte is written.

`public void` **`write(byte[] buf, int offset, int count)`** `throws IOException`

Writes part of an array of bytes, starting at `buf[offset]` and writing `count` bytes. This method blocks until the bytes have been written.

`public void` **`write(byte[] buf)`** `throws IOException`

Equivalent to `write(buf, 0, buf.length)`.

`public void` **`flush()`** `throws IOException`

Flushes the stream. If the stream has buffered any bytes from the various `write` methods, `flush` writes them immediately to their destination. Then, if that destination is another stream, it is also flushed. One `flush` invocation will flush all the buffers in a chain of streams. If the stream is not buffered, `flush` may do nothing—the default implementation. This method is defined in the `Flushable` interface.

`public void` **`close()`** `throws IOException`

Closes the output stream. This method should be invoked to release any resources (such as file descriptors) associated with the stream. Once a stream has been closed, further operations on the stream will throw an `IOException`. Closing a previously closed stream has no effect.The default implementation of `close` does nothing.

The implementation of `OutputStream` requires only that a subclass provide the single-byte variant of `write` because the other `write` methods are defined in terms of this one. Most streams, however, can improve performance by overriding other methods as well. The default implementations of `flush` and `close` will usually need to be overridden as appropriate for a particular stream—in particular, buffered streams may need to flush when closed.

Here is a program that copies its input to its output, translating one particular byte value to a different one along the way. The `TranslateByte` program takes two parameters: a `from` byte and a `to` byte. Bytes that match the value in the string `from` are translated into the value in the string `to`.

```
import java.io.*;

class TranslateByte {
    public static void main(String[] args)
        throws IOException
    {
        byte from = (byte) args[0].charAt(0);
        byte to   = (byte) args[1].charAt(0);
        int b;
        while ((b = System.in.read()) != -1)
            System.out.write(b == from ? to : b);
    }
}
```

For example, if we invoked the program as

```
java TranslateByte b B
```

and entered the text `abracadabra!`, we would get the output

```
aBracadaBra!
```

Manipulating data from a stream after it has been read, or before it is written, is often achieved by writing `Filter` streams, rather than hardcoding the manipulation in a program. You'll learn about filters in Section 20.5.2 on page 516.

Exercise 20.1: Rewrite the `TranslateByte` program as a method that translates the contents of an `InputStream` onto an `OutputStream`, in which the mapping and the streams are parameters. For each type of `InputStream` and `OutputStream` you read about in this chapter, write a new `main` method that uses the translation method to operate on a stream of that type. If you have paired input and output streams, you can cover both in one `main` method.

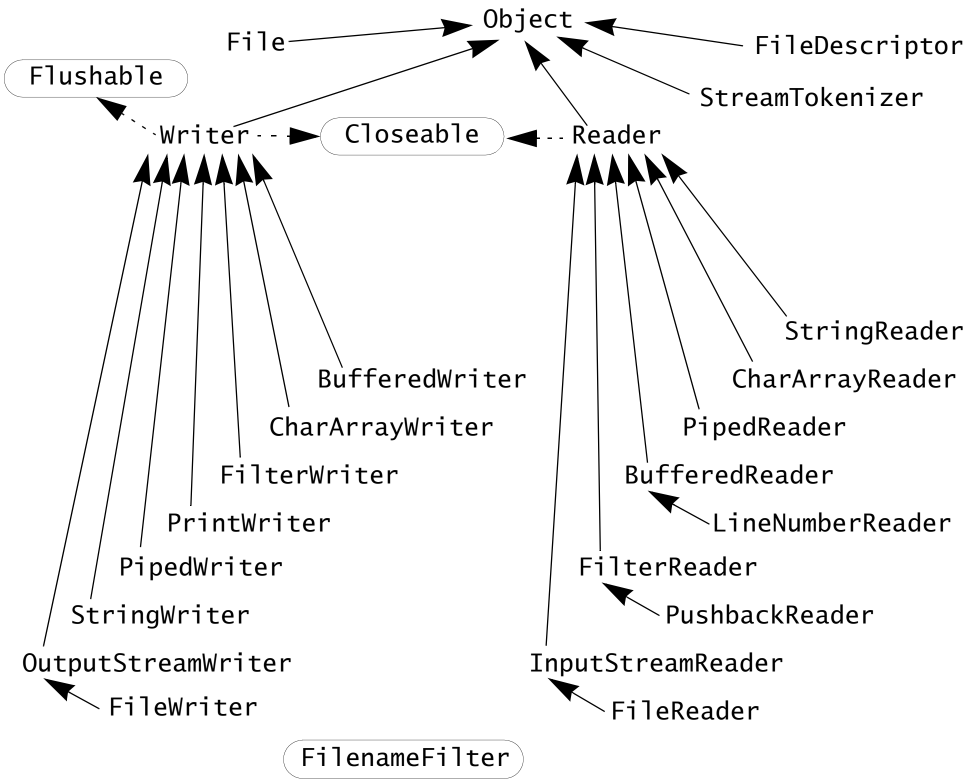


FIGURE 20–2: ***Type Tree for Character Streams in*** **`java.io`**

20.3 Character Streams

The abstract classes for reading and writing streams of characters are `Reader` and `Writer`. Each supports methods similar to those of its byte stream counterpart—`InputStream` and `OutputStream`, respectively. For example, `InputStream` has a `read` method that returns a `byte` as the lowest 8 bits of an `int`, and `Reader` has a `read` method that returns a `char` as the lowest 16 bits of an `int`. And where `OutputStream` has methods that write `byte` arrays, `Writer` has methods that write `char` arrays. The character streams were designed after the byte streams to provide full support for working with Unicode characters, and in the process the contracts of the classes were improved to make them easier to work with. The type tree for the character streams of `java.io` appears in Figure 20–2.

As with the byte streams, character streams should be explicitly closed to release resources associated with the stream. Character stream synchronization policies are discussed in Section 20.5.1 on page 515.

20.3.1 Reader

The abstract class `Reader` provides a character stream analogous to the byte stream `InputStream` and the methods of `Reader` essentially mirror those of `InputStream`:

`public int` **`read()`** `throws IOException`

Reads a single character and returns it as an integer in the range 0 to 65535. If no character is available because the end of the stream has been reached, the value –1 is returned. This method blocks until input is available, the end of stream is found, or an exception is thrown.

`public abstract int` **`read(char[] buf, int offset, int count)`** `throws IOException`

Reads into a part of a `char` array. The maximum number of characters to read is `count`. The read characters are stored from `buf[offset]` up to a maximum of `buf[offset+count-1]`—all other values in `buf` are left unchanged. The number of characters actually read is returned. If no characters are read because the end of the stream was found, –1 is returned. If `count` is zero then no characters are read and zero is returned. This method blocks until input is available, the end of stream is found, or an exception is thrown. If the first character cannot be read for any reason other than finding the end of the stream—in particular, if the stream has already been closed—an `IOException` is thrown. Once a character has been read, any failure that occurs while trying to read characters does not cause an exception, but is treated just like finding the end of the stream—the method completes normally and returns the number of characters read before the failure occurred.

`public int` **`read(char[] buf)`** `throws IOException`

Equivalent to `read(buf, 0, buf.length)`.

`public int` **`read(java.nio.CharBuffer buf)`** `throws IOException`

Attempts to read as many characters as possible into the specified character buffer, without overflowing it. The number of characters actually read is returned. If no characters are read because the end of the stream was found, –1 is returned. This is equivalent to reading into an array that has the same length as the buffer has available capacity, and then copying the array into the buffer. This method is defined in the `java.lang.Readable` interface, and has no counterpart in `InputStream`.

`public long` **`skip(long count)`** `throws IOException`

Skips as many as `count` characters of input or until the end of the stream is found. Returns the actual number of characters skipped. The value of `count` must not be negative.

`public boolean` **`ready()`** `throws IOException`

Returns `true` if the stream is ready to read; that is, there is at least one character available to be read. Note that a return value of `false` does not guarantee that the next invocation of `read` will block because data could have become available by the time the invocation occurs.

`public abstract void` **`close()`** `throws IOException`

Closes the stream. This method should be invoked to release any resources (such as file descriptors) associated with the stream. Once a stream has been closed, further operations on the stream will throw an `IOException`. Closing a previously closed stream has no effect.

The implementation of `Reader` requires that a subclass provide an implementation of both the `read` method that reads into a char array, and the `close` method. Many subclasses will be able to improve performance if they also override some of the other methods.

There are a number of differences between `Reader` and `InputStream`. With `Reader` the fundamental reading method reads into a char array and the other read methods are defined in terms of this method. In contrast the `InputStream` class uses the single-byte `read` method as its fundamental reading method. In the `Reader` class subclasses must implement the abstract `close` method in contrast to inheriting an empty implementation—many stream classes will at least need to track whether or not they have been closed and so `close` will usually need to be overridden. Finally, where `InputStream` had an `available` method to tell you how much data was available to read, `Reader` simply has a `ready` method that tells you if there is any data.

As an example, the following program counts the number of whitespace characters in a character stream:

```
import java.io.*;

class CountSpace {
    public static void main(String[] args)
        throws IOException
    {
        Reader in;
        if (args.length == 0)
            in = new InputStreamReader(System.in);
```

```
            else
                in = new FileReader(args[0]);
            int ch;
            int total;
            int spaces = 0;
            for (total = 0; (ch = in.read()) != -1; total++) {
                if (Character.isWhitespace((char) ch))
                    spaces++;
            }
            System.out.println(total + " chars, "
                + spaces + " spaces");
        }
    }
```

This program takes a filename from the command line. The variable `in` represents the character stream. If a filename is not provided, the standard input stream, `System.in`, is used after wrapping it in an `InputStreamReader`, which converts an input byte stream into an input character stream; if a filename is provided, an object of type `FileReader` is created, which is a subclass of `Reader`.

The `for` loop counts the total number of characters in the file and the number of spaces, using the `Character` class's `isWhitespace` method to test whether a character is whitespace. At the end, the results are printed. Here is the output of the program when used on itself:

```
453 chars, 111 spaces
```

20.3.2 **`Writer`**

The abstract class `Writer` provides a stream analogous to `OutputStream` but designed for use with characters instead of bytes. The methods of `Writer` essentially mirror those of `OutputStream`, but add some other useful forms of `write`:

`public void` **`write(int ch)`** `throws IOException`

Writes ch as a character. The character is passed as an `int` but only the lowest 16 bits of the integer are written. This method blocks until the character is written.

`public abstract void` **`write(char[] buf, int offset, int count)`** `throws IOException`

Writes part of an array of characters, starting at `buf[offset]` and writing `count` characters. This method blocks until the characters have been written.

`public void` **`write(char[] buf)`** `throws IOException`

Equivalent to `write(buf, 0, buf.length)`.

`public void` **`write(String str, int offset, int count)`**
`throws IOException`

Writes `count` characters from the string `str` onto the stream, starting with `str.charAt(offset)`.

`public void` **`write(String str)`** `throws IOException`

Equivalent to `write(str, 0, str.length())`.

`public abstract void` **`flush()`** `throws IOException`

Flushes the stream. If the stream has buffered any characters from the various `write` methods, `flush` immediately writes them to their destination. Then, if that destination is another stream, it is also flushed. One `flush` invocation will flush all the buffers in a chain of streams. If a stream is not buffered `flush` will do nothing.

`public abstract void` **`close()`** `throws IOException`

Closes the stream, flushing if necessary. This method should be invoked to release any resources (such as file descriptors) associated with the stream. Once a stream has been closed, further operations on the stream will throw an `IOException`. Closing a previously closed stream has no effect.

Subclasses of `Writer` must implement the array writing variant of `write`, the `close` method, and the `flush` method. All other `Writer` methods are implemented in terms of these three. This contrasts with `OutputStream` which uses the single-byte variant of `write` method as the fundamental writing method, and which provides default implementations of `flush` and `close`. As with `Reader`, many subclasses can improve performance if they also override other methods.

`Writer` also implements the `java.lang.Appendable` interface—see page 332. The `append(char c)` method is equivalent to `write(c)`; the `append` methods that take a `CharSequence` are equivalent to passing the `String` representations of the `CharSequence` objects to the `write(String str)` method.

20.3.3 Character Streams and the Standard Streams

The standard streams `System.in`, `System.out`, and `System.err` existed before the character streams were invented, so these streams are byte streams even though logically they should be character streams. This situation creates some anomalies. It is impossible, for example, to replace `System.in` with a `LineNumberReader` to keep track of the standard input stream's current line number. By attaching an `InputStreamReader`—an object that converts a byte input stream to a character input stream—to `System.in`, you can create a `LineNumberReader` object to keep track of the current line number (see "`LineNumberReader`" on page 527). But `System.in` is an `InputStream`, so you cannot replace it with a `LineNumberReader`, which is a type of `Reader`, not an `InputStream`.

`System.out` and `System.err` are `PrintStream` objects. `PrintStream` has been replaced by its equivalent character-based version `PrintWriter`. Generally, you should avoid creating `PrintStream` objects directly. You'll learn about the `Print` stream classes in Section 20.5.8 on page 525.

20.4 InputStreamReader and OutputStreamWriter

The conversion streams `InputStreamReader` and `OutputStreamWriter` translate between character and byte streams using either a specified character set encoding or the default encoding for the local system. These classes are the "glue" that lets you use existing 8-bit character encodings for local character sets in a consistent, platform-independent fashion. An `InputStreamReader` object is given a byte input stream as its source and produces the corresponding UTF-16 characters. An `OutputStreamWriter` object is given a byte output stream as its destination and produces encoded byte forms of the UTF-16 characters written on it. For example, the following code would read bytes encoded under ISO 8859-6 for Arabic characters, translating them into the appropriate UTF-16 characters:

```
public Reader readArabic(String file) throws IOException {
    InputStream fileIn = new FileInputStream(file);
    return new InputStreamReader(fileIn, "iso-8859-6");
}
```

By default, these conversion streams will work in the platform's default character set encoding, but other encodings can be specified. Encoding values were discussed in "Character Set Encoding" on page 320; they can be represented by name or a `Charset`, or by a `CharsetDecoder` or `CharsetEncoder` object from the `java.nio.charset` package.

`public` **`InputStreamReader(InputStream in)`**

Creates an `InputStreamReader` to read from the given `InputStream` using the default character set encoding.

`public` **`InputStreamReader(InputStream in, Charset c)`**

Creates an `InputStreamReader` to read from the given `InputStream` using the given character set encoding.

`public` **`InputStreamReader(InputStream in, CharsetDecoder c)`**

Creates an `InputStreamReader` to read from the given `InputStream` using the given character set decoder.

`public` **`InputStreamReader(InputStream in, String enc)`**
`throws UnsupportedEncodingException`

Creates an `InputStreamReader` to read from the given `InputStream` using the named character set encoding. If the named encoding is not supported an `UnsupportedEncodingException` is thrown.

`public` **`OutputStreamWriter(OutputStream out)`**

Creates an `OutputStreamWriter` to write to the given `OutputStream` using the default character set encoding.

`public` **`OutputStreamWriter(OutputStream out, Charset c)`**

Creates an `OutputStreamWriter` to write to the given `OutputStream` using the given character set encoding.

`public` **`OutputStreamWriter(OutputStream out, CharsetEncoder c)`**

Creates an `OutputStreamWriter` to write to the given `OutputStream` using the given character set encoder.

`public` **`OutputStreamWriter(OutputStream out, String enc)`**
`throws UnsupportedEncodingException`

Creates an `OutputStreamWriter` to write to the given `OutputStream` using the named character set encoding. If the named encoding is not supported an `UnsupportedEncodingException` is thrown.

The `read` methods of `InputStreamReader` simply read bytes from their associated `InputStream` and convert them to characters using the appropriate encoding for that stream. Similarly, the `write` methods of `OutputStreamWriter` take the supplied characters, convert them to bytes with the appropriate encoding, and write them to the associated `OutputStream`.

In both classes, closing the conversion stream also closes the associated byte stream. This may not always be desirable—such as when you are converting the standard streams—so consider carefully when closing conversion streams.

Both classes also support the method `getEncoding`, which returns a string representing either the historical or canonical name of the stream's character encoding, or `null` if the stream has been closed.

The `FileReader` and `FileWriter` classes are subclasses of these conversion streams. This helps you read and write local files correctly in a consistent, Unicode-savvy fashion using the local encoding. However, if the default local encoding isn't what you need, you must use an explicit `InputStreamReader` or `OutputStreamWriter` object. You will learn about the file related streams in more detail in Section 20.7 on page 540.

You can also use the data output stream you will learn about in Section 20.6.2 on page 539 to write characters as bytes using a specific Unicode encoding.

There is no `ReaderInputStream` class to translate characters to bytes, nor a `WriterOutputStream` class to translate bytes to characters.

20.5 A Quick Tour of the Stream Classes

The `java.io` package defines several types of streams. The stream types usually have input/output pairs, and most have both byte stream and character stream variants. Some of these streams define general behavioral properties. For example:

- `Filter` streams are abstract classes representing streams with some filtering operation applied as data is read or written by another stream. For example, a `FilterReader` object gets input from another `Reader` object, processes (filters) the characters in some manner, and returns the filtered result. You build sequences of filtered streams by chaining various filters into one large filter. Output can be filtered similarly (Section 20.5.2).
- `Buffered` streams add buffering so that `read` and `write` need not, for example, access the file system for every invocation. The character variants of these streams also add the notion of line-oriented text (Section 20.5.3).
- `Piped` streams are pairs such that, say, characters written to a `PipedWriter` can be read from a `PipedReader` (Section 20.5.4).

A group of streams, called *in-memory streams,* allow you to use in-memory data structures as the source or destination for a stream:

- `ByteArray` streams use a `byte` array (Section 20.5.5).
- `CharArray` streams use a `char` array (Section 20.5.6).
- `String` streams use string types (Section 20.5.7).

The I/O package also has input and output streams that have no output or input counterpart:

- The `Print` streams provide `print` and `println` methods for formatting printed data in human-readable text form (Section 20.5.8).
- `LineNumberReader` is a buffered reader that tracks the line numbers of the input (characters only) (Section 20.5.9).
- `SequenceInputStream` converts a sequence of `InputStream` objects into a single `InputStream` so that a list of concatenated input streams can be treated as a single input stream (bytes only) (Section 20.5.10).

There are also streams that are useful for building parsers:

- `Pushback` streams add a pushback buffer you can use to put back data when you have read too far (Section 20.5.11).
- The `StreamTokenizer` class breaks a `Reader` into a stream of tokens—recognizable “words”— that are often needed when parsing user input (characters only) (Section 20.5.12).

These classes can be extended to create new kinds of stream classes for specific applications.

Each of these stream types is described in the following sections. Before looking at these streams in detail, however, you need to learn something about the synchronization behavior of the different streams.

20.5.1 Synchronization and Concurrency

Both the byte streams and the characters streams define synchronization policies though they do this in different ways. The concurrent behavior of the stream classes is not fully specified but can be broadly described as follows.

Each byte stream class synchronizes on the current stream object when performing operations that must be free from interference. This allows multiple threads to use the same streams yet still get well-defined behavior when invoking individual stream methods. For example, if two threads each try to read data from a stream in chunks of `n` bytes, then the data returned by each read operation will contain up to `n` bytes that appeared consecutively in the stream. Similarly, if two threads are writing to the same stream then the bytes written in each write operation will be sent consecutively to the stream, not intermixed at random points.

The character streams use a different synchronization strategy from the byte streams. The character streams synchronize on a protected `lock` field which, by default, is a reference to the stream object itself. However, both `Reader` and `Writer` provide a protected constructor that takes an object for `lock` to refer to. Some subclasses set the `lock` field to refer to a different object. For example, the `StringWriter` class that writes its character into a `StringBuffer` object sets its `lock` object to be the `StringBuffer` object. If you are writing a reader or writer, you should set the `lock` field to an appropriate object if `this` is not appropriate. Conversely, if you are extending an existing reader or writer you should always synchronize on `lock` and not `this`.

In many cases, a particular stream object simply wraps another stream instance and delegates the main stream methods to that instance, forming a chain of connected streams, as is the case with `Filter` streams. In this case, the synchronization behavior of the method will depend on the ultimate stream object being wrapped. This will only become an issue if the wrapping class needs to perform some additional action that must occur atomically with respect to the main stream action. In most cases filter streams simply manipulate data before writing it to, or after reading it from, the wrapped stream, so synchronization is not an issue.

Most input operations will block until data is available, and it is also possible that output stream operations can block trying to write data—the ultimate source or destination could be a stream tied to a network socket. To make the threads performing this blocking I/O more responsive to cancellation requests an implementation may respond to `Thread` interrupt requests (see page 365) by unblocking the thread and throwing an `InterruptedIOException`. This exception can report the number of bytes transferred before the interruption occurred—if the code that throws it sets the value.

For single byte transfers, interrupting an I/O operation is quite straightforward. In general, however, the state of a stream after a thread using it is interrupted is problematic. For example, suppose you use a particular stream to read HTTP requests across the network. If a thread reading the next request is interrupted after reading two bytes of the header field in the request packet, the next thread reading from that stream will get invalid data unless the stream takes steps to prevent this. Given the effort involved in writing classes that can deal effectively with these sorts of situations, most implementations *do not* allow a thread to be interrupted until the main I/O operation has completed, so you cannot rely on blocking I/O being interruptible. The interruptible channels provided in the `java.nio` package support interruption by closing the stream when any thread using the stream is interrupted—this ensures that there are no issues about what would next be read.

Even when interruption cannot be responded to during an I/O operation many systems will check for interruption at the start and/or end of the operation and throw the `InterruptedIOException` then. Also, if a thread is blocked on a stream when the stream is closed by another thread, most implementations will unblock the blocked thread and throw an `IOException`.

20.5.2 `Filter` Streams

`Filter` streams—`FilterInputStream`, `FilterOutputStream`, `FilterReader`, and `FilterWriter`—help you chain streams to produce composite streams of greater utility. Each filter stream is bound to another stream to which it delegates the actual input or output actions. `Filter` streams get their power from the ability to filter—process—what they read or write, transforming the data in some way.

`Filter` byte streams add new constructors that accept a stream of the appropriate type (input or output) to which to connect. `Filter` character streams similarly add a new constructor that accepts a character stream of the appropriate type (reader or writer). However, many character streams already have constructors that take another character stream, so those `Reader` and `Writer` classes can act as filters even if they do not extend `FilterReader` or `FilterWriter`.

The following shows an input filter that converts characters to uppercase:

```
public class UppercaseConvertor extends FilterReader {
    public UppercaseConvertor(Reader in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toUpperCase((char)c));
    }

    public int read(char[] buf, int offset, int count)
        throws IOException
    {
        int nread = super.read(buf, offset, count);
        int last = offset + nread;
        for (int i = offset; i < last; i++)
            buf[i] = Character.toUpperCase(buf[i]);
        return nread;
    }
}
```

We override each of the `read` methods to perform the actual read and then convert the characters to upper case. The actual reading is done by invoking an appropriate superclass method. Note that we don't invoke `read` on the stream `in` itself—this would bypass any filtering performed by our superclass. Note also that we have to watch for the end of the stream. In the case of the no-arg `read` this means an explicit test, but in the array version of `read`, a return value of –1 will prevent the `for` loop from executing. In the array version of `read` we also have to be careful to convert to uppercase only those characters that we stored in the buffer.

We can use our uppercase convertor as follows:

```
public static void main(String[] args)
    throws IOException
{
    StringReader src = new StringReader(args[0]);
    FilterReader f = new UppercaseConvertor(src);
```

```
        int c;
        while ((c = f.read()) != -1)
            System.out.print((char)c);
        System.out.println();
    }
```

We use a string as our data source by using a `StringReader` (see Section 20.5.7 on page 523). The `StringReader` is then wrapped by our `UppercaseConvertor`. Reading from the filtered stream converts all the characters from the string stream into uppercase. For the input `"no lowercase"` we get the output:

```
NO LOWERCASE
```

You can chain any number of `Filter` byte or character streams. The original source of input can be a stream that is not a `Filter` stream. You can use an `InputStreamReader` to convert a byte input stream to a character input stream.

`Filter` output streams can be chained similarly so that data written to one stream will filter and write data to the next output stream. All the streams, from the first to the next-to-last, must be `Filter` output stream objects, but the last stream can be any kind of output stream. You can use an `OutputStreamWriter` to convert a character output stream to a byte output stream.

Not all classes that are `Filter` streams actually alter the data. Some classes are behavioral filters, such as the buffered streams you'll learn about next, while others provide a new interface for using the streams, such as the print streams. These classes are `Filter` streams because they can form part of a filter chain.

Exercise 20.2: Rewrite the `TranslateByte` class as a filter.

Exercise 20.3: Create a pair of `Filter` stream classes that encrypt bytes using any algorithm you choose—such as XORing the bytes with some value—with your `DecryptInputStream` able to decrypt the bytes that your `EncryptOutputStream` class creates.

Exercise 20.4: Create a subclass of `FilterReader` that will return one line of input at a time via a method that blocks until a full line of input is available.

20.5.3 Buffered Streams

The `Buffered` stream classes—`BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, and `BufferedWriter`—buffer their data to avoid every `read` or `write` going directly to the next stream. These classes are often used in conjunction with File streams—accessing a disk file is much slower than using a memory buffer, and buffering helps reduce file accesses.

Each of the Buffered streams supports two constructors: One takes a reference to the wrapped stream and the size of the buffer to use, while the other only takes a reference to the wrapped stream and uses a default buffer size.

When read is invoked on an empty Buffered input stream, it invokes read on its source stream, fills the buffer with as much data as is available—only blocking if it needs the data being waited for—and returns the requested data from that buffer. Future read invocations return data from that buffer until its contents are exhausted, and that causes another read on the source stream. This process continues until the source stream is exhausted.

Buffered output streams behave similarly. When a write fills the buffer, the destination stream's write is invoked to empty the buffer. This buffering can turn many small write requests on the Buffered stream into a single write request on the underlying destination.

Here is how to create a buffered output stream to write bytes to a file:

```
new BufferedOutputStream(new FileOutputStream(path));
```

You create a FileOutputStream with the path, put a BufferedOutputStream in front of it, and use the buffered stream object. This scheme enables you to buffer output destined for the file.

You must retain a reference to the FileOutputStream object if you want to invoke methods on it later because there is no way to obtain the downstream object from a Filter stream. However, you should rarely need to work with the downstream object. If you do keep a reference to a downstream object, you must ensure that the first upstream object is flushed before operating on the downstream object because data written to upper streams may not have yet been written all the way downstream. Closing an upstream object also closes all downstream objects, so a retained reference may cease to be usable.

The Buffered character streams also understand lines of text. The newLine method of BufferedWriter writes a line separator to the stream. Each system defines what constitutes a line separator in the system String property line.separator, which need not be a single character. You should use newLine to end lines in text files that may be read by humans on the local system (see "System Properties" on page 663).

The method readLine in BufferedReader returns a line of text as a String. The method readLine accepts any of the standard set of line separators: line feed (\n), carriage return (\r), or carriage return followed by line feed (\r\n). This implies that you should never set line.separator to use any other sequence. Otherwise, lines terminated by newLine would not be recognized by readLine. The string returned by readLine does not include the line separator. If the end of stream is encountered before a line separator, then the text read to that point is returned. If only the end of stream is encountered `readLine` returns `null`.

20.5.4 Piped Streams

`Piped` streams—`PipedInputStream`, `PipedOutputStream`, `PipedReader`, and `PipedWriter`—are used as input/output pairs; data written on the output stream of a pair is the data read on the input stream. The pipe maintains an internal buffer with an implementation-defined capacity that allows writing and reading to proceed at different rates—there is no way to control the size of the buffer.

Pipes provide an I/O-based mechanism for communicating data between different threads. The only safe way to use `Piped` streams is with two threads: one for reading and one for writing. Writing on one end of the pipe blocks the thread when the pipe fills up. If the writer and reader are the same thread, that thread will block permanently. Reading from a pipe blocks the thread if no input is available.

To avoid blocking a thread forever when its counterpart at the other end of the pipe terminates, each pipe keeps track of the identity of the most recent reader and writer threads. The pipe checks to see that the thread at the other end is alive before blocking the current thread. If the thread at the other end has terminated, the current thread will get an `IOException`.

The following example uses a pipe stream to connect a `TextGenerator` thread with a thread that wants to read the generated text. First, the text generator:

```
class TextGenerator extends Thread {
    private Writer out;

    public TextGenerator(Writer out) {
        this.out = out;
    }

    public void run() {
        try {
            try {
                for (char c = 'a'; c <= 'z'; c++)
                    out.write(c);
            } finally {
                out.close();
            }
        } catch (IOException e) {
            getUncaughtExceptionHandler().
                uncaughtException(this, e);
```

```
        }
    }
}
```

The `TextGenerator` simply writes to the output stream passed to its constructor. In the example that stream will actually be a piped stream to be read by the main thread:

```
class Pipe {
    public static void main(String[] args)
        throws IOException
    {
        PipedWriter out = new PipedWriter();
        PipedReader in = new PipedReader(out);
        TextGenerator data = new TextGenerator(out);
        data.start();
        int ch;
        while ((ch = in.read()) != -1)
            System.out.print((char) ch);
        System.out.println();
    }
}
```

We create the `Piped` streams, making the `PipedWriter` a parameter to the constructor for the `PipedReader`. The order is unimportant: The input pipe could be a parameter to the output pipe. What is important is that an input/output pair be attached to each other. We create the new `TextGenerator` object, with the `PipedWriter` as the output stream for the generated characters. Then we loop, reading characters from the text generator and writing them to the system output stream. At the end, we make sure that the last line of output is terminated.

`Piped` streams need not be connected when they are constructed—there is a no-arg constructor—but can be connected at a later stage via the `connect` method. `PipedReader.connect` takes a `PipedWriter` parameter and vice versa. As with the constructor, it does not matter whether you connect `x` to `y`, or `y` to `x`, the result is the same. Trying to use a `Piped` stream before it is connected or trying to connect it when it is already connected results in an `IOException`.

20.5.5 `ByteArray` Byte Streams

You can use arrays of bytes as the source or destination of byte streams by using `ByteArray` streams. The `ByteArrayInputStream` class uses a `byte` array as its input source, and reading on it can never block. It has two constructors:

`public` **`ByteArrayInputStream(byte[] buf, int offset, int count)`**
Creates a `ByteArrayInputStream` from the specified array of bytes using only the part of `buf` from `buf[offset]` to `buf[offset+count-1]` or the end of the array, whichever is smaller. The input array is used directly, not copied, so you should take care not to modify it while it is being used as an input source.

`public` **`ByteArrayInputStream(byte[] buf)`**
Equivalent to `ByteArrayInputStream(buf, 0, buf.length)`.

The `ByteArrayOutputStream` class provides a dynamically growing `byte` array to hold output. It adds constructors and methods:

`public` **`ByteArrayOutputStream()`**
Creates a new `ByteArrayOutputStream` with a default initial array size.

`public` **`ByteArrayOutputStream(int size)`**
Creates a new `ByteArrayOutputStream` with the given initial array size.

`public int` **`size()`**
Returns the number of bytes generated thus far by output to the stream.

`public byte[]` **`toByteArray()`**
Returns a copy of the bytes generated thus far by output to the stream. When you are finished writing into a `ByteArrayOutputStream` via upstream filter streams, you should flush the upstream objects before using `toByteArray`.

`public void` **`reset()`**
Resets the stream to reuse the current buffer, discarding its contents.

`public String` **`toString()`**
Returns the current contents of the buffer as a `String`, translating bytes into characters according to the default character encoding.

`public String` **`toString(String enc)`**
`throws UnsupportedEncodingException`
Returns the current contents of the buffer as a `String`, translating bytes into characters according to the specified character encoding. If the encoding is not supported an `UnsupportedEncodingException` is thrown.

`public void` **`writeTo(OutputStream out)`** `throws IOException`
Writes the current contents of the buffer to the stream `out`.

20.5.6 `CharArray` Character Streams

The `CharArray` character streams are analogous to the `ByteArray` byte streams—they let you use `char` arrays as a source or destination without ever blocking. You construct `CharArrayReader` objects with an array of `char`:

`public` **`CharArrayReader(char[] buf, int offset, int count)`**
Creates a `CharArrayReader` from the specified array of characters using only the subarray of `buf` from `buf[offset]` to `buf[offset+count-1]` or the end of the array, whichever is smaller. The input array is used directly, not copied, so you should take care not to modify it while it is being used as an input source.

`public` **`CharArrayReader(char[] buf)`**
Equivalent to `CharArrayReader(buf, 0, buf.length)`.

The `CharArrayWriter` class provides a dynamically growing `char` array to hold output. It adds constructors and methods:

`public` **`CharArrayWriter()`**
Creates a new `CharArrayWriter` with a default initial array size.

`public` **`CharArrayWriter(int size)`**
Creates a new `CharArrayWriter` with the given initial array size.

`public int` **`size()`**
Returns the number of characters generated thus far by output to the stream.

`public char[]` **`toCharArray()`**
Returns a copy of the characters generated thus far by output to the stream. When you are finished writing into a `CharArrayWriter` via upstream filter streams, you should flush the upstream objects before using `toCharArray`.

`public void` **`reset()`**
Resets the stream to reuse the current buffer, discarding its contents.

`public String` **`toString()`**
Returns the current contents of the buffer as a `String`.

`public void` **`writeTo(Writer out)`** `throws IOException`
Writes the current contents of the buffer to the stream `out`.

20.5.7 `String` Character Streams

The `StringReader` reads its characters from a `String` and will never block. It provides a single constructor that takes the string from which to read. For example, the following program factors numbers read either from the command line or `System.in`:

```
class Factor {
    public static void main(String[] args) {
        if (args.length == 0) {
            factorNumbers(new InputStreamReader(System.in));
```

```
            } else {
                for (String str : args) {
                    StringReader in = new StringReader(str);
                    factorNumbers(in);
                }
            }
        }
        // ... definition of factorNumbers ...
    }
```

If the command is invoked without parameters, `factorNumbers` parses numbers from the standard input stream. When the command line contains some arguments, a `StringReader` is created for each parameter, and `factorNumbers` is invoked on each one. The parameter to `factorNumbers` is a stream of characters containing numbers to be parsed; it does not know whether they come from the command line or from standard input.

`StringWriter` lets you write results into a buffer that can be retrieved as a `String` or `StringBuffer` object. It adds the following constructors and methods:

`public` **`StringWriter()`**

Creates a new `StringWriter` with a default initial buffer size.

`public` **`StringWriter(int size)`**

Creates a new `StringWriter` with the specified initial buffer size. Providing a good initial size estimate for the buffer will improve performance in many cases.

`public StringBuffer` **`getBuffer()`**

Returns the actual `StringBuffer` being used by this stream. Because the actual `StringBuffer` is returned, you should take care not to modify it while it is being used as an output destination.

`public String` **`toString()`**

Returns the current contents of the buffer as a `String`.

The following code uses a `StringWriter` to create a string that contains the output of a series of `println` calls on the contents of an array:

```
public static String arrayToStr(Object[] objs) {
    StringWriter strOut = new StringWriter();
    PrintWriter out = new PrintWriter(strOut);
    for (int i = 0; i < objs.length; i++)
        out.println(i + ": " + objs[i]);
    return strOut.toString();
}
```

20.5.8 Print Streams

The Print streams—PrintStream and PrintWriter—provide methods that make it easy to write the values of primitive types and objects to a stream, in a human-readable text format—as you have seen in many examples. The Print streams provide print and println methods for the following types:

```
char    int    float   Object  boolean
char[]  long   double  String
```

These methods are much more convenient than the raw stream write methods. For example, given a float variable f and a PrintStream reference out, the call out.print(f) is equivalent to

```
out.write(String.valueOf(f).getBytes());
```

The println method appends a line separator after writing its argument to the stream—a simple println with no parameters ends the current line. The line separator string is defined by the system property line.separator and is not necessarily a single newline character (\n).

Each of the Print streams acts as a Filter stream, so you can filter data on its way downstream.

The PrintStream class acts on byte streams while the PrintWriter class acts on character streams. Because printing is clearly character-related output, the PrintWriter class is the class you should use. However, for historical reasons System.out and System.err are PrintStreams that use the default character set encoding—these are the only PrintStream objects you should use. We describe only the PrintWriter class, though PrintStream provides essentially the same interface.

PrintWriter has eight constructors.

public **PrintWriter(Writer out, boolean autoflush)**
Creates a new PrintWriter that will write to the stream out. If autoflush is true, println invokes flush. Otherwise, println invocations are treated like any other method, and flush is not invoked. Autoflush behavior cannot be changed after the stream is constructed.

public **PrintWriter(Writer out)**
Equivalent to PrintWriter(out, false).

public **PrintWriter(OutputStream out, boolean autoflush)**
Equivalent to
PrintWriter(new OutputStreamWriter(out), autoflush).

public **PrintWriter(OutputStream out)**
: Equivalent to `PrintWriter(new OutputStreamWriter(out), false)`.

public **PrintWriter(File file)** throws FileNotFoundException
: Equivalent to `PrintWriter(new OutputStreamWriter(fos))`, where `fos` is a `FileOutputStream` created with the given file.

public **PrintWriter(File file, String enc)**
  throws FileNotFoundException, UnsupportedEncodingException
: Equivalent to `PrintWriter(new OutputStreamWriter(fos, enc))`, where `fos` is a `FileOutputStream` created with the given file.

public **PrintWriter(String filename)** throws FileNotFoundException
: Equivalent to `PrintWriter(new OutputStreamWriter(fos))`, where `fos` is a `FileOutputStream` created with the given file name.

public **PrintWriter(String filename, String enc)**
  throws FileNotFoundException, UnsupportedEncodingException
: Equivalent to `PrintWriter(new OutputStreamWriter(fos, enc))`, where `fos` is a `FileOutputStream` created with the given file name.

The `Print` streams implement the `Appendable` interface which allows them to be targets for a `Formatter`. Additionally, the following convenience methods are provided for formatted output—see "`Formatter`" on page 624 for details:

public PrintWriter **format(String format, Object... args)**
: Acts like `new Formatter(this).format(format, args)`, but a new `Formatter` need not be created for each call. The current `PrintWriter` is returned.

public PrintWriter
  format(Locale l, String format, Object... args)
: Acts like `new Formatter(this, l).format(format, args)`, but a new `Formatter` need not be created for each call. The current `PrintWriter` is returned. Locales are described in Chapter 24.

There are two `printf` methods that behave exactly the same as the `format` methods—`printf` stands for "print formatted" and is an old friend from the `C` programming language.

One important characteristic of the `Print` streams is that none of the output methods throw `IOException`. If an error occurs while writing to the underlying stream the methods simply return normally. You should check whether an error occurred by invoking the boolean method `checkError`—this flushes the stream and checks its error state. Once an error has occurred, there is no way to clear it. If any of the underlying stream operations result in an `InterruptedIOException`, the error state is not set, but instead the current thread is re-interrupted using `Thread.currentThread().interrupt()`.

20.5.9 `LineNumberReader`

The `LineNumberReader` stream keeps track of line numbers while reading text. As usual a line is considered to be terminated by any one of a line feed (`\n`), a carriage return (`\r`), or a carriage return followed immediately by a linefeed (`\r\n`).

The following program prints the line number where the first instance of a particular character is found in a file:

```
import java.io.*;

class FindChar {
    public static void main(String[] args)
        throws IOException
    {
        if (args.length != 2)
            throw new IllegalArgumentException(
                                    "need char and file");

        int match = args[0].charAt(0);
        FileReader fileIn = new FileReader(args[1]);
        LineNumberReader in = new LineNumberReader(fileIn);
        int ch;
        while ((ch = in.read()) != -1) {
            if (ch == match) {
                System.out.println("'" + (char)ch +
                    "' at line " + in.getLineNumber());
                return;
            }
        }
        System.out.println((char)match + " not found");
    }
}
```

This program creates a `FileReader` named `fileIn` to read from the named file and then inserts a `LineNumberReader`, named `in`, before it. `LineNumberReader` objects get their characters from the reader they are attached to, keeping track of line numbers as they read. The `getLineNumber` method returns the current line number; by default, lines are counted starting from *zero*. When this program is run on itself looking for the letter `'I'`, its output is

```
'I' at line 4
```

You can set the current line number with `setLineNumber`. This could be useful, for example, if you have a file that contains several sections of information. You could use `setLineNumber` to reset the line number to 1 at the start of each section so that problems would be reported to the user based on the line numbers within the section instead of within the file.

`LineNumberReader` is a `BufferedReader` that has two constructors: One takes a reference to the wrapped stream and the size of the buffer to use, while the other only takes a reference to the wrapped stream and uses a default buffer size.

Exercise 20.5: Write a program that reads a specified file and searches for a specified word, printing each line number and line in which the word is found.

20.5.10 SequenceInputStream

The `SequenceInputStream` class creates a single input stream from reading one or more byte input streams, reading the first stream until its end of input and then reading the next one, and so on through the last one. `SequenceInputStream` has two constructors: one for the common case of two input streams that are provided as the two parameters to the constructor, and the other for an arbitrary number of input streams using the `Enumeration` abstraction (described in "`Enumeration`" on page 617). `Enumeration` is an interface that provides an ordered iteration through a list of objects. For `SequenceInputStream`, the enumeration should contain only `InputStream` objects. If it contains anything else, a `ClassCastException` will be thrown when the `SequenceInputStream` tries to get that object from the list.

The following example program concatenates all its input to create a single output. This program is similar to a simple version of the UNIX utility `cat`—if no files are named, the input is simply forwarded to the output. Otherwise, the program opens all the files and uses a `SequenceInputStream` to model them as a single stream. Then the program writes its input to its output:

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

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

```
            InputStream in; // stream to read characters from
            if (args.length == 0) {
                in = System.in;
            } else {
                InputStream fileIn, bufIn;
                List<InputStream> inputs =
                    new ArrayList<InputStream>(args.length);
                for (String arg : args) {
                    fileIn = new FileInputStream(arg);
                    bufIn = new BufferedInputStream(fileIn);
                    inputs.add(bufIn);
                }
                Enumeration<InputStream> files =
                    Collections.enumeration(inputs);
                in = new SequenceInputStream(files);
            }
            int ch;
            while ((ch = in.read()) != -1)
                System.out.write(ch);
        }
        // ...
    }
```

If there are no parameters, we use `System.in` for input. If there are parameters, we create an `ArrayList` large enough to hold as many `BufferedInputStream` objects as there are command-line arguments (see "`ArrayList`" on page 582). Then we create a stream for each named file and add the stream to the `inputs` list. When the loop is finished, we use the `Collections` class's `enumeration` method to get an `Enumeration` object for the list elements. We use this `Enumeration` in the constructor for `SequenceInputStream` to create a single stream that concatenates all the streams for the files into a single `InputStream` object. A simple loop then reads all the bytes from that stream and writes them on `System.out`.

You could instead write your own implementation of `Enumeration` whose `nextElement` method creates a `FileInputStream` for each argument on demand, closing the previous stream, if any.

20.5.11 Pushback Streams

A `Pushback` stream lets you push back, or "unread," characters or bytes when you have read too far. `Pushback` is typically useful for breaking input into tokens. Lexical scanners, for example, often know that a token (such as an identifier) has ended only when they have read the first character that follows it. Having seen that character, the scanner must push it back onto the input stream so that it is available as the start of the next token. The following example uses PushbackInputStream to report the longest consecutive sequence of any single byte in its input:

```
import java.io.*;

class SequenceCount {
    public static void main(String[] args)
        throws IOException
    {
        PushbackInputStream
           in = new PushbackInputStream(System.in);
        int max = 0;    // longest sequence found
        int maxB = -1;  // the byte in that sequence
        int b;          // current byte in input

        do {
            int cnt;
            int b1 = in.read(); // 1st byte in sequence
            for (cnt = 1; (b = in.read()) == b1; cnt++)
                continue;
            if (cnt > max) {
                max = cnt; // remember length
                maxB = b1; // remember which byte value
            }
            in.unread(b);  // pushback start of next seq
        } while (b != -1); // until we hit end of input

        System.out.println(max + " bytes of " + maxB);
    }
}
```

We know that we have reached the end of one sequence only when we read the first byte of the next sequence. We push this byte back using unread so that it is read again when we repeat the do loop for the next sequence.

Both PushbackInputStream and PushbackReader support two constructors: One takes a reference to the wrapped stream and the size of the pushback buffer to create, while the other only takes a reference to the wrapped stream and uses a pushback buffer with space for one piece of data (byte or char as appropriate). Attempting to push back more than the specified amount of data will cause an IOException.

Each `Pushback` stream has three variants of `unread`, matching the variants of `read`. We illustrate the character version of `PushbackReader`, but the byte equivalents for `PushbackInputStream` have the same behavior:

`public void` **`unread(int c)`** `throws IOException`

Pushes back the single character `c`. If there is insufficient room in the pushback buffer an `IOException` is thrown.

`public void` **`unread(char[] buf, int offset, int count)`**
`throws IOException`

Pushes back the characters in the specified subarray. The first character pushed back is `buf[offset]` and the last is `buf[offset+count-1]`. The subarray is prepended to the front of the pushback buffer, such that the next character to be read will be that at `buf[offset]`, then `buf[offset+1]`, and so on. If the pushback buffer is full an `IOException` is thrown.

`public void` **`unread(char[] buf)`** `throws IOException`

Equivalent to `unread(buf, 0, buf.length)`.

For example, after two consecutive `unread` calls on a `PushbackReader` with the characters `'1'` and `'2'`, the next two characters read will be `'2'` and `'1'` because `'2'` was pushed back second. Each `unread` call sets its own list of characters by prepending to the buffer, so the code

```
pbr.unread(new char[] {'1', '2'});
pbr.unread(new char[] {'3', '4'});
for (int i = 0; i < 4; i++)
    System.out.println(i + ": " + (char)pbr.read());
```

produces the following lines of output:

```
0: 3
1: 4
2: 1
3: 2
```

Data from the last `unread` (the one with `'3'` and `'4'`) is read back first, and within that `unread` the data comes from the beginning of the array through to the end. When that data is exhausted, the data from the first `unread` is returned in the same order. The `unread` method copies data into the pushback buffer, so changes made to an array after it is used with `unread` do not affect future calls to `read`.

20.5.12 StreamTokenizer

Tokenizing input text is a common application, and the java.io package provides a StreamTokenizer class for simple tokenization. A more general facility for scanning and converting input text is provided by the java.util.Scanner class—see “Scanner” on page 641.

You can tokenize a stream by creating a StreamTokenizer with a Reader object as its source and then setting parameters for the scan. A scanner loop invokes nextToken, which returns the token type of the next token in the stream. Some token types have associated values that are found in fields in the StreamTokenizer object.

This class is designed primarily to parse programming language-style input; it is not a general tokenizer. However, many configuration files look similar enough to programming languages that they can be parsed by this tokenizer. When designing a new configuration file or other data, you can save work if you make it look enough like a language to be parsed with StreamTokenizer.

When nextToken recognizes a token, it returns the token type as its value and also sets the ttype field to the same value. There are four token types:

- TT_WORD: A word was scanned. The String field sval contains the word that was found.
- TT_NUMBER: A number was scanned. The double field nval contains the value of the number. Only decimal floating-point numbers (with or without a decimal point) are recognized. The tokenizer does not understand 3.4e79 as a floating-point number, nor 0xffff as a hexadecimal number.
- TT_EOL: An end-of-line was found.
- TT_EOF: The end-of-file was reached.

The input text is assumed to consist of bytes in the range \u0000 to \u00FF—Unicode characters outside this range are not handled correctly. Input is composed of both *special* and *ordinary* characters. Special characters are those that the tokenizer treats specially—namely whitespace, characters that make up numbers, characters that make up words, and so on. Any other character is considered ordinary. When an ordinary character is the next in the input, its token type is itself. For example, if the character '¿' is encountered in the input and is not special, the token return type (and the ttype field) is the int value of the character '¿'.

As one example, let's look at a method that sums the numeric values in a character stream it is given:

```
static double sumStream(Reader source) throws IOException {
    StreamTokenizer in = new StreamTokenizer(source);
    double result = 0.0;
    while (in.nextToken() != StreamTokenizer.TT_EOF) {
        if (in.ttype == StreamTokenizer.TT_NUMBER)
            result += in.nval;
    }
    return result;
}
```

We create a StreamTokenizer object from the reader and then loop, reading tokens from the stream, adding all the numbers found into the burgeoning result. When we get to the end of the input, we return the final sum.

Here is another example that reads an input source, looking for attributes of the form name=value, and stores them as attributes in AttributedImpl objects, described in "Implementing Interfaces" on page 127:

```
public static Attributed readAttrs(Reader source)
    throws IOException
{
    StreamTokenizer in = new StreamTokenizer(source);
    AttributedImpl attrs = new AttributedImpl();
    Attr attr = null;
    in.commentChar('#');    // '#' is ignore-to-end comment
    in.ordinaryChar('/');   // was original comment char
    while (in.nextToken() != StreamTokenizer.TT_EOF) {
        if (in.ttype == StreamTokenizer.TT_WORD) {
            if (attr != null) {
                attr.setValue(in.sval);
                attr = null;        // used this one up
            } else {
                attr = new Attr(in.sval);
                attrs.add(attr);
            }
        } else if (in.ttype == '=') {
            if (attr == null)
                throw new IOException("misplaced '='");
        } else {
            if (attr == null)       // expected a word
```

```
                    throw new IOException("bad Attr name");
                attr.setValue(new Double(in.nval));
                attr = null;
            }
        }
        return attrs;
    }
```

The attribute file uses `'#'` to mark comments. Ignoring these comments, the stream is searched for a string token followed by an optional `'='` followed by a word or number. Each such attribute is put into an `Attr` object, which is added to a set of attributes in an `AttributedImpl` object. When the file has been parsed, the set of attributes is returned.

Setting the comment character to `'#'` sets its character class. The tokenizer recognizes several character classes that are set by the following methods:

`public void` **`wordChars(int low, int hi)`**
: Characters in this range are word characters: They can be part of a `TT_WORD` token. You can invoke this several times with different ranges. A word consists of one or more characters inside any of the legal ranges.

`public void` **`whitespaceChars(int low, int hi)`**
: Characters in this range are whitespace. Whitespace is ignored, except to separate tokens such as two consecutive words. As with the `wordChars` range, you can make several invocations, and the union of the invocations is the set of whitespace characters.

`public void` **`ordinaryChars(int low, int hi)`**
: Characters in this range are ordinary. An ordinary character is returned as itself, not as a token. This removes any special significance the characters may have had as comment characters, delimiters, word components, whitespace, or number characters. In the above example, we used `ordinaryChar` to remove the special comment significance of the `'/'` character.

`public void` **`ordinaryChar(int ch)`**
: Equivalent to `ordinaryChars(ch, ch)`.

`public void` **`commentChar(int ch)`**
: The character `ch` starts a single-line comment—characters after `ch` up to the next end-of-line are treated as one run of whitespace.

`public void` **`quoteChar(int ch)`**
: Matching pairs of the character `ch` delimit `String` constants. When a `String` constant is recognized, the character `ch` is returned as the token, and

the field `sval` contains the body of the string with surrounding `ch` characters removed. When string constants are read, some of the standard \ processing is applied (for example, `\t` can be in the string). The string processing in `StreamTokenizer` is a subset of the language's strings. In particular, you cannot use `\uxxxx`, `\'`, `\"`, or (unfortunately) `\Q`, where `Q` is the quote character `ch`. You can have more than one quote character at a time on a stream, but strings must start and end with the same quote character. In other words, a string that starts with one quote character ends when the next instance of that same quote character is found. If a different quote character is found in between, it is simply part of the string.

`public void` **`parseNumbers()`**

Specifies that numbers should be parsed as double-precision floating-point numbers. When a number is found, the stream returns a type of `TT_NUMBER`, leaving the value in `nval`. There is no way to turn off just this feature—to turn it off you must either invoke `ordinaryChars` for all the number-related characters (don't forget the decimal point and minus sign) or invoke `resetSyntax`.

`public void` **`resetSyntax()`**

Resets the syntax table so that all characters are ordinary. If you do this and then start reading the stream, `nextToken` always returns the next character in the stream, just as when you invoke `InputStream.read`.

There are no methods to ask the character class of a given character or to add new classes of characters. Here are the default settings for a newly created `StreamTokenizer` object:

```
wordChars('a', 'z');      // lower case ASCII letters
wordChars('A', 'Z');      // upper case ASCII letters
wordChars(128 + 32, 255); // "high" non-ASCII values
whitespaceChars(0, ' ');  // ASCII control codes
commentChar('/');
quoteChar('"');
quoteChar('\'');
parseNumbers();
```

This leaves the ordinary characters consisting of most of the punctuation and arithmetic characters (`;`, `:`, `[`, `{`, `+`, `=`, and so forth).

The changes made to the character classes are cumulative, so, for example, invoking `wordChars` with two different ranges of characters defines both ranges as word characters. To replace a range you must first mark the old range as ordinary and then add the new range. Resetting the syntax table clears all settings, so if you want to return to the default settings, for example, you must manually make the invocations listed above.

Other methods control the basic behavior of the tokenizer:

`public void` **`eolIsSignificant(boolean flag)`**

If `flag` is `true`, ends of lines are significant and `TT_EOL` may be returned by `nextToken`. If `false`, ends of lines are treated as whitespace and `TT_EOL` is never returned. The default is `false`.

`public void` **`slashStarComments(boolean flag)`**

If `flag` is `true`, the tokenizer recognizes `/*...*/` comments. This occurs independently of settings for any comment characters. The default is `false`.

`public void` **`slashSlashComments(boolean flag)`**

If `flag` is `true`, the tokenizer recognizes `//` to end-of-line comments. This occurs independently of the settings for any comment characters. The default is `false`.

`public void` **`lowerCaseMode(boolean flag)`**

If `flag` is `true`, all characters in `TT_WORD` tokens are converted to their lowercase equivalent if they have one (using `String.toLowerCase`). The default is `false`. Because of the case issues described in "`Character`" on page 192, you cannot reliably use this for Unicode string equivalence—two tokens might be equivalent but have different lowercase representations. Use `String.equalsIgnoreCase` for reliable case-insensitive comparison.

There are three miscellaneous methods:

`public void` **`pushBack()`**

Pushes the previously returned token back into the stream. The next invocation of `nextToken` returns the same token again instead of proceeding to the next token. There is only a one-token pushback; multiple consecutive invocations to `pushBack` are equivalent to one invocation.

`public int` **`lineno()`**

Returns the current line number. Usually used for reporting errors you detect.

`public String` **`toString()`**

Returns a `String` representation of the last returned stream token, including its line number.

Exercise 20.6: Write a program that takes input of the form *`name op value`*, where *`name`* is one of three words of your choosing, *`op`* is `+`, `-`, or `=`, and *`value`* is a number. Apply each operator to the named value. When input is exhausted, print the three values. For extra credit, use the `HashMap` class that was used for `AttributedImpl` so that you can use an arbitrary number of named values.

20.6 The Data Byte Streams

Reading and writing text characters is useful, but you also frequently need to transmit the binary data of specific types across a stream. The `DataInput` and `DataOutput` interfaces define methods that transmit primitive types across a stream. The classes `DataInputStream` and `DataOutputStream` provide a default implementation for each interface. We cover the interfaces first, followed by their implementations.

20.6.1 `DataInput` and `DataOutput`

The interfaces for data input and output streams are almost mirror images. The parallel read and write methods for each type are

Read	**Write**	**Type**
`readBoolean`	`writeBoolean`	`boolean`
`readChar`	`writeChar`	`char`
`readByte`	`writeByte`	`byte`
`readShort`	`writeShort`	`short`
`readInt`	`writeInt`	`int`
`readLong`	`writeLong`	`long`
`readFloat`	`writeFloat`	`float`
`readDouble`	`writeDouble`	`double`
`readUTF`	`writeUTF`	`String` (in UTF format)

String values are read and written using a modified form of the UTF-8 character encoding. This differs from standard UTF-8 in three ways: the null byte (`\u0000`) is encoded in a 2-byte format so that the encoded string does not have embedded null bytes; only 1-byte, 2-byte, or 3-byte formats are used; and supplementary characters are encoded using surrogate pairs. Encoding Unicode characters into bytes is necessary in many situations because of the continuing transition from 8-bit to 16-bit character sets.

In addition to these paired methods, `DataInput` has several methods of its own, some of which are similar to those of `InputStream`:

`public abstract void` **`readFully(byte[] buf, int offset, int count)`**
`throws IOException`

Reads into part of a `byte` array. The maximum number of bytes read is `count`. The bytes are stored from `buf[offset]` up to a maximum of `buf[offset+count-1]`. If `count` is zero then no bytes are read. This method blocks until input is available, the end of the file (that is, stream) is found—in which case an `EOFException` is thrown—or an exception is thrown because of an I/O error.

`public abstract void` **`readFully(byte[] buf)`** `throws IOException`

Equivalent to `readFully(buf, 0, buf.length)`.

`public abstract int` **`skipBytes(int count)`** `throws IOException`

Attempts to skip over `count` bytes, discarding any bytes skipped over. Returns the actual number of bytes skipped. This method never throws an `EOFException`.

`public abstract int` **`readUnsignedByte()`** `throws IOException`

Reads one input byte, zero-extends it to type `int`, and returns the result, which is therefore in the range 0 through 255. This method is suitable for reading a byte written by the `writeByte` method of `DataOutput` if the argument to `writeByte` was a value in the range 0 through 255.

`public abstract int` **`readUnsignedShort()`** `throws IOException`

Reads two input bytes and returns an `int` value in the range 0 through 65535. The first byte read is made the high byte. This method is suitable for reading bytes written by the `writeShort` method of `DataOutput` if the argument to `writeShort` was a value in the range 0 through 65535.

The `DataInput` interface methods usually handle end-of-file (stream) by throwing `EOFException` when it occurs. `EOFException` extends `IOException`.

The `DataOutput` interface supports signatures equivalent to the three forms of `write` in `OutputStream` and with the same specified behavior. Additionally, it provides the following unmirrored methods:

`public abstract void` **`writeBytes(String s)`** `throws IOException`

Writes a `String` as a sequence of bytes. The upper byte in each character is lost, so unless you are willing to lose data, use this method only for strings that contain characters between `\u0000` and `\u00ff`.

`public abstract void` **`writeChars(String s)`** `throws IOException`

Writes a `String` as a sequence of `char`. Each character is written as two bytes with the high byte written first.

There are no `readBytes` or `readChars` methods to read the same number of characters written by a `writeBytes` or `writeChars` invocation, therefore you must use a loop on `readByte` or `readChar` to read strings written with these methods. To do that you need a way to determine the length of the string, perhaps by writing the length of the string first, or by using an end-of-sequence character to mark its end. You could use `readFully` to read a full array of bytes if you wrote the length first, but that won't work for `writeChars` because you want `char` values, not `byte` values.

20.6.2 The Data Stream Classes

For each Data interface there is a corresponding Data stream. In addition, the RandomAccessFile class implements both the input and output Data interfaces (see Section 20.7.2 on page 541). Each Data class is an extension of its corresponding Filter class, so you can use Data streams to filter other streams. Each Data class has constructors that take another appropriate input or output stream. For example, you can use the filtering to write data to a file by putting a DataOutputStream in front of a FileOutputStream object. You can then read the data by putting a DataInputStream in front of a FileInputStream object:

```
public static void writeData(double[] data, String file)
    throws IOException
{
    OutputStream fout = new FileOutputStream(file);
    DataOutputStream out = new DataOutputStream(fout);
    out.writeInt(data.length);
    for (double d : data)
        out.writeDouble(d);
    out.close();
}

public static double[] readData(String file)
    throws IOException
{
    InputStream fin = new FileInputStream(file);
    DataInputStream in = new DataInputStream(fin);
    double[] data = new double[in.readInt()];
    for (int i = 0; i < data.length; i++)
        data[i] = in.readDouble();
    in.close();
    return data;
}
```

The writeData method first opens the file and writes the array length. It then loops, writing the contents of the array. The file can be read into an array with readData. These methods can be rewritten more simply using the Object streams you will learn about in Section 20.8 on page 549.

Exercise 20.7: Add a method to the Attr class of Chapter 3 that writes the contents of an object to a DataOutputStream and add a constructor that will read the state from a DataInputStream.

20.7 Working with Files

The `java.io` package provides a number of classes that help you work with files in the underlying system. The `File` stream classes allow you to read from and write to files and the `FileDescriptor` class allows the system to represent underlying file system resources as objects. `RandomAccessFile` lets you deal with files as randomly accessed streams of bytes or characters. Actual interaction with the local file system is through the `File` class, which provides an abstraction of file pathnames, including path component separators, and useful methods to manipulate file names.

20.7.1 `File` Streams and `FileDescriptor`

The `File` streams—`FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter`—allow you to treat a file as a stream for input or output. Each type is instantiated with one of three constructors:

- A constructor that takes a `String` that is the name of the file.
- A constructor that takes a `File` object that refers to the file (see Section 20.7.3 on page 543).
- A constructor that takes a `FileDescriptor` object (see below).

If a file does not exist, the input streams will throw a `FileNotFoundException`. Accessing a file requires a security check and a `SecurityException` is thrown if you do not have permission to access that file—see "Security" on page 677.

With a byte or character output stream, the first two constructor types create the file if it does not exist, or truncate it if it does exist. You can control truncation by using the overloaded forms of these two constructors that take a second argument: a `boolean` that, if `true`, causes each individual write to append to the file. If this `boolean` is `false`, the file will be truncated and new data added. If the file does not exist, the file will be created and the `boolean` will be ignored.

The byte `File` streams also provide a `getChannel` method for integration with the `java.nio` facilities. It returns a `java.nio.channels.FileChannel` object for accessing the file.

A `FileDescriptor` object represents a system-dependent value that describes an open file. You can get a file descriptor object by invoking `getFD` on a `File` byte stream—you cannot obtain the file descriptor from `File` character streams. You can test the validity of a `FileDescriptor` by invoking its boolean `valid` method—file descriptors created directly with the no-arg constructor of `FileDescriptor` are not valid.

`FileDescriptor` objects create a new `File` stream to the same file as another stream without needing to know the file's pathname. You must be careful to avoid unexpected interactions between two streams doing different things with the same file. You cannot predict what happens, for example, when two threads write to the same file using two different `FileOutputStream` objects at the same time.

The `flush` method of `FileOutputStream` and `FileWriter` guarantees that the buffer is flushed to the underlying file. It does not guarantee that the data is committed to disk—the underlying file system may do its own buffering. You can guarantee that the data is committed to disk by invoking the `sync` method on the file's `FileDescriptor` object, which will either force the data to disk or throw a `SyncFailedException` if the underlying system cannot fulfill this contract.

20.7.2 `RandomAccessFile`

The `RandomAccessFile` class provides a more sophisticated file mechanism than the `File` streams do. A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the *file pointer;* input operations read bytes starting at the file pointer and advance the file pointer past the bytes read. If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written.

`RandomAccessFile` is not a subclass of `InputStream`, `OutputStream`, `Reader`, or `Writer` because it can do both input and output and can work with both characters and bytes. The constructor has a parameter that declares whether the stream is for input or for both input and output.

`RandomAccessFile` supports `read` and `write` methods of the same names and signatures as the byte streams. For example, `read` returns a single byte. `RandomAccessFile` also implements the `DataInput` and `DataOutput` interfaces (see page 537) and so can be used to read and write data types supported in those interfaces. Although you don't have to learn a new set of method names and semantics for the same kinds of tasks you do with the other streams, you cannot use a `RandomAccessFile` where any of the other streams are required.

The constructors for `RandomAccessFile` are

`public` **`RandomAccessFile(String name, String mode)`**
`throws FileNotFoundException`

Creates a random access file stream to read from, and optionally write to, a file with the specified name. The basic mode can be either `"r"` or `"rw"` for read or read/write, respectively. Variants of `"rw"` mode provide additional semantics: `"rws"` mode specifies that on each write the file contents and metadata (file size, last modification time, etc.) are written synchronously through to the disk; `"rwd"` mode specifies that only the file contents are written synchronously to the disk. Specifying any other mode will get you an `IllegalArgumentException`. If the mode contains `"rw"` and the file does not exist, it will be created or, if that fails, a `FileNotFoundException` is thrown.

`public` **`RandomAccessFile(File file, String mode)`**
`throws FileNotFoundException`

Creates a random access file stream to read from, and optionally write to, the file specified by the `File` argument. Modes are the same as for the `String`-based constructor.

Since accessing a file requires a security check, these constructors could throw a `SecurityException` if you do not have permission to access the file in that mode—see "Security" on page 677.

The "random access" in the name of the class refers to the ability to set the read/write file pointer to any position in the file and then perform operations. The additional methods in `RandomAccessFile` to support this functionality are:

`public long` **`getFilePointer()`** `throws IOException`

Returns the current location of the file pointer (in bytes) from the beginning of the file.

`public void` **`seek(long pos)`** `throws IOException`

Sets the file pointer to the specified number of bytes from the beginning of the file. The next byte written or read will be the `pos`th byte in the file, where the initial byte is the 0^{th}. If you position the file pointer beyond the end of the file and write to the file, the file will grow.

`public int` **`skipBytes(int count)`** `throws IOException`

Attempts to advance the file pointer `count` bytes. Any bytes skipped over can be read later after `seek` is used to reposition the file pointer. Returns the actual number of bytes skipped. This method is guaranteed never to throw an `EOFException`. If `count` is negative, no bytes are skipped.

`public long` **`length()`** `throws IOException`

Returns the file length.

`public void` **`setLength(long newLength)`** `throws IOException`

Sets the length of the file to `newLength`. If the file is currently shorter, the file is grown to the given length, filled in with any byte values the implementation chooses. If the file is currently longer, the data beyond this position is discarded. If the current position (as returned by `getFilePointer`) is greater than `newLength`, the position is set to `newLength`.

You can access the `FileDescriptor` for a `RandomAccessFile` by invoking its `getFD` method. You can obtain a `FileChannel` for a `RandomAccessFile` by invoking its `getChannel` method.

Exercise 20.8: Write a program that reads a file with entries separated by lines starting with `%%` and creates a table file with the starting position of each such entry. Then write a program that uses that table to print a random entry (see the `Math.random` method described in "`Math` and `StrictMath`" on page 657).

20.7.3 The `File` Class

The `File` class (not to be confused with the file streams) provides several common manipulations that are useful with file names. It provides methods to separate pathnames into subcomponents and to ask the file system about the file a pathname refers to.

A `File` object actually represents a path, not necessarily an underlying file. For example, to find out whether a pathname represents an existing file, you create a `File` object with the pathname and then invoke `exists` on that object.

A path is separated into directory and file parts by a `char` stored in the static field `separatorChar` and available as a `String` in the static field `separator`. The last occurrence of this character in the path separates the pathname into directory and file components. (*Directory* is the term used on most systems; some systems call such an entity a "folder" instead.)

`File` objects are created with one of four constructors:

`public` **`File(String path)`**
Creates a `File` object to manipulate the specified `path`.

`public` **`File(String dirName, String name)`**
Creates a `File` object for the file `name` in the directory named `dirName`. If `dirName` is `null`, only `name` is used. If `dirName` is an empty string, `name` is resolved against a system dependent default directory. Otherwise, this is equivalent to using `File(dirName + File.separator + name)`.

`public` **`File(File fileDir, String name)`**
Creates a `File` object for the file `name` in the directory named by the `File` object `fileDir`. Equivalent to using `File(fileDir.getPath(), name)`.

`public` **`File(java.net.URI uri)`**
Creates a `File` object for the pathname represented by the given `file:` URI (Uniform Resource Identifier). If the given URI is not a suitable file URI then `IllegalArgumentException` is thrown.

Five "get" methods retrieve information about the components of a `File` object's pathname. The following code invokes each of them after creating a `File` object for the file `"FileInfo.java"` in the `"ok"` subdirectory of the parent of the current directory (specified by `".."`):

```
File src = new File(".." + File.separator + "ok",
                    "FileInfo.java");
System.out.println("getName() = " + src.getName());
System.out.println("getPath() = " + src.getPath());
System.out.println("getAbsolutePath() = "
    + src.getAbsolutePath());
System.out.println("getCanonicalPath() = "
    + src.getCanonicalPath());
System.out.println("getParent() = " + src.getParent());
```

And here is the output:

```
getName() = FileInfo.java
getPath() = ../ok/FileInfo.java
getAbsolutePath() = /vob/java_prog/src/../ok/FileInfo.java
getCanonicalPath() = /vob/java_prog/ok/FileInfo.java
getParent() = ../ok
```

The canonical path is defined by each system. Usually, it is a form of the absolute path with relative components (such as `".."` to refer to the parent directory) renamed and with references to the current directory removed. Unlike the other "get" methods, `getCanonicalPath` can throw `IOException` because resolving path components can require calls to the underlying file system, which may fail.

The methods `getParentFile`, `getAbsoluteFile`, and `getCanonicalFile` are analogous to `getParent`, `getAbsolutePath`, and `getCanonicalPath`, but they return `File` objects instead of strings.

You can convert a `File` to a `java.net.URL` or `java.net.URI` object by invoking `toURL` or `toURI`, respectively.

The overriding method `File.equals` deserves mention. Two `File` objects are considered equal if they have the same path, not if they refer to the same underlying file system object. You cannot use `File.equals` to test whether two `File` objects denote the same file. For example, two `File` objects may refer to the same file but use different relative paths to refer to it, in which case they do not compare equal. Relatedly, you can compare two files using the `compareTo` method, which returns a number less than, equal to, or greater than zero as the current file's pathname is lexicographically less than, equal to, or greater than the pathname of the argument `File`. The `compareTo` method has two overloaded forms: one takes a `File` argument and the other takes an `Object` argument and so implements the `Comparable` interface.

Several boolean tests return information about the underlying file:

- `exists` returns `true` if the file exists in the file system.
- `canRead` returns `true` if a file exists and can be read.
- `canWrite` returns `true` if the file exists and can be written.
- `isFile` returns `true` if the file is not a directory or other special type of file.
- `isDirectory` returns `true` if the file is a directory.
- `isAbsolute` returns `true` if the path is an absolute pathname.
- `isHidden` returns `true` if the path is one normally hidden from users on the underlying system.

All the methods that inspect or modify the actual file system are security checked and can throw `SecurityException` if you don't have permission to perform the operation. Methods that ask for the filename itself are not security checked.

`File` objects have many other methods for manipulating files and directories. There are methods to inspect and manipulate the current file:

`public long` **`lastModified()`**

Returns a `long` value representing the time the file was last modified or zero if the file does not exist.

`public long` **`length()`**

Returns the file length in bytes, or zero if the file does not exist.

`public boolean` **`renameTo(File newName)`**

Renames the file, returning `true` if the rename succeeded.

`public boolean` **`delete()`**

Deletes the file or directory named in this `File` object, returning `true` if the deletion succeeded. Directories must be empty before they are deleted.

There are methods to create an underlying file or directory named by the current `File`:

`public boolean` **`createNewFile()`**

Creates a new empty file, named by this `File`. Returns `false` if the file already exists or if the file cannot be created. The check for the existence of the file and its subsequent creation is performed atomically with respect to other file system operations.

`public boolean` **`mkdir()`**

Creates a directory named by this `File`, returning `true` on success.

`public boolean` **`mkdirs()`**

Creates all directories in the path named by this `File`, returning `true` if all were created. This is a way to ensure that a particular directory is created, even if it means creating other directories that don't currently exist above it in the directory hierarchy. Note that some of the directories may have been created even if `false` is returned.

However, files are usually created by `FileOutputStream` or `FileWriter` objects or `RandomAccessFile` objects, not using `File` objects.

Two methods let you change the state of the underlying file, assuming that one exists:

`public boolean` **`setLastModified(long time)`**

Sets the "last modified" time for the file or returns `false` if it cannot do so.

`public boolean` **`setReadOnly()`**

Makes the underlying file unmodifiable in the file system or returns `false` if it cannot do so. The file remains unmodifiable until it is deleted or externally marked as modifiable again—there is no method for making it modifiable again.

There are methods for listing the contents of directories and finding out about root directories:

`public String[]` **`list()`**

Lists the files in this directory. If used on something that isn't a directory, it returns `null`. Otherwise, it returns an array of file names. This list includes all files in the directory except the equivalent of `"."` and `".."` (the current and parent directory, respectively).

`public String[]` **`list(FilenameFilter filter)`**

Uses `filter` to selectively list files in this directory (see `FilenameFilter` described in the next section).

`public static File[]` **`listRoots()`**

Returns the available filesystem roots, that is, roots of local hierarchical file systems. Windows platforms, for example, have a root directory for each active drive; UNIX platforms have a single `/` root directory. If none are available, the array has zero elements.

The methods `listFiles() and listFiles(FilenameFilter)` are analogous to `list() and list(FilenameFilter)`, but return arrays of `File` objects instead of arrays of strings. The method `listFiles(FileFilter)` is analogous to the `list` that uses a `FilenameFilter`.

Three methods relate primarily to temporary files (sometimes called "scratch files")—those files you need to create during a run of your program for storing data, or to pass between passes of your computation, but which are not needed after your program is finished.

`public static File` **`createTempFile(String prefix, String suffix, File directory)`** `throws IOException`

Creates a new empty file in the specified directory, using the given prefix and suffix strings to generate its name. If this method returns successfully then it is guaranteed that the file denoted by the returned abstract pathname did not exist before this method was invoked, and neither this method nor any of its variants will return the same abstract pathname again in the current invocation of the virtual machine. The `prefix` argument must be at least three characters long, otherwise an `IllegalArgumentException` is thrown. It is recommended that the prefix be a short, meaningful string such as `"hjb"` or `"mail"`. The `suffix` argument may be `null`, in which case the suffix `".tmp"` will be used. Note that since there is no predefined separator between the file name and the suffix, any separator, such as `'.'`, must be part of the suffix. If the `directory` argument is `null` then the system-dependent default temporary-file directory will be used. The default temporary-file directory is specified by the system property `java.io.tmpdir`.

`public static File` **`createTempFile(String prefix, String suffix)`** `throws IOException`

Equivalent to `createTempFile(prefix, suffix, null)`.

`public void` **`deleteOnExit()`**

Requests the system to remove the file when the virtual machine terminates—see "Shutdown" on page 672. This request only applies to a normal termination of the virtual machine and cannot be revoked once issued.

When a temporary file is created, the prefix and the suffix may first be adjusted to fit the limitations of the underlying platform. If the prefix is too long then it will be truncated, but its first three characters will always be preserved. If the suffix is too long then it too will be truncated, but if it begins with a period (`.`) then the period and the first three characters following it will always be preserved. Once these adjustments have been made the name of the new file will be generated by concatenating the prefix, five or more internally generated characters, and the suffix. Temporary files are not automatically deleted on exit, although you will often invoke `deleteOnExit` on `File` objects returned by `createTempFile`.

Finally, the character `File.pathSeparatorChar` and its companion string `File.pathSeparator` represent the character that separates file or directory names in a search path. For example, UNIX separates components in the program search path with a colon, as in `".:/bin:/usr/bin"`, so `pathSeparatorChar` is a colon on UNIX systems.

Exercise 20.9: Write a method that, given one or more pathnames, will print all the information available about the file it represents (if any).

Exercise 20.10: Write a program that uses a `StreamTokenizer` object to break an input file into words and counts the number of times each word occurs in the file, printing the result. Use a `HashMap` to keep track of the words and counts.

20.7.4 `FilenameFilter` and `FileFilter`

The `FilenameFilter` interface provides objects that filter unwanted files from a list. It supports a single method:

`boolean` **`accept(File dir, String name)`**

Returns `true` if the file named `name` in the directory `dir` should be part of the filtered output.

Here is an example that uses a `FilenameFilter` object to list only directories:

```
import java.io.*;

class DirFilter implements FilenameFilter {
    public boolean accept(File dir, String name) {
        return new File(dir, name).isDirectory();
    }

    public static void main(String[] args) {
        File dir = new File(args[0]);
        String[] files = dir.list(new DirFilter());
        System.out.println(files.length + " dir(s):");
        for (String file : files)
            System.out.println("\t" + file);
    }
}
```

First we create a `File` object to represent a directory specified on the command line. Then we create a `DirFilter` object and pass it to `list`. For each name in the directory, `list` invokes the `accept` method on the filtering object and includes the name in the list if the filtering object returns `true`. For our `accept` method, `true` means that the named file is a directory.

The `FileFilter` interface is analogous to `FilenameFilter`, but works with a single `File` object:

`boolean` **`accept(File pathname)`**

Returns `true` if the file represented by `pathname` should be part of the filtered output.

Exercise 20.11: Using `FilenameFilter` or `FileFilter`, write a program that takes a directory and a suffix as parameters and prints all files it can find that have that suffix.

20.8 Object Serialization

The ability to save objects in a byte stream that can be transferred across the network (perhaps for use in remote method invocations), saved to disk in a file or database, and later reconstituted to form a live object, is an essential aspect of many real-world applications.

The process of converting an object's representation into a stream of bytes is known as *serialization,* while reconstituting an object from a byte stream is *deserialization.* When talking about the classes, interfaces, and language features involved in this overall process, we generally just use the term *serialization* and understand that it includes deserialization as well.

A number of classes and interfaces are involved with serialization. You have already learned about the basic mechanisms for reading and writing primitive types and strings using the `Data` stream classes (see page 537). This section covers the object byte streams—`ObjectInputStream` and `ObjectOutputStream`—that allow you to serialize and deserialize complete objects. Various other classes and interfaces provide specific support for the serialization process. In addition, the field modifier `transient` provides a language-level means of marking data that should not be serialized.

20.8.1 The `Object` Byte Streams

The `Object` streams—`ObjectInputStream` and `ObjectOutputStream`—allow you to read and write object graphs in addition to the well-known types (primitives, strings, and arrays). By "object graph" we mean that when you use `writeObject` to write an object to an `ObjectOutputStream`, bytes representing the object—including all other objects that it references—are written to the stream. This process of transforming an object into a stream of bytes is called *seri-*

alization. Because the serialized form is expressed in bytes, not characters, the `Object` streams have no `Reader` or `Writer` forms.

When bytes encoding a serialized graph of objects are read by the method `readObject` of `ObjectInputStream`—that is, *deserialized*—the result is a graph of objects equivalent to the input graph.

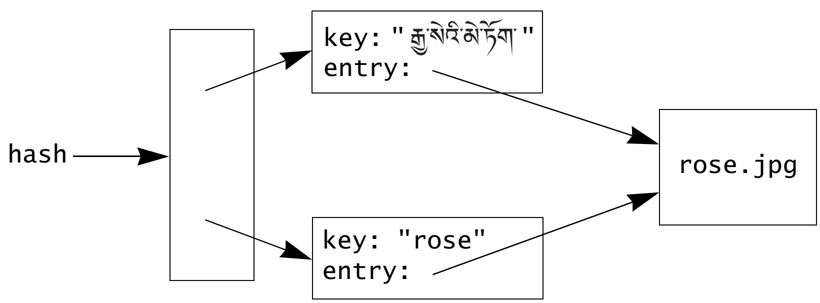
Suppose, for example, that you have a `HashMap` object that you wish to store into a file for future use. You could write the graph of objects that starts with the hash map this way:

```
FileOutputStream fileOut = new FileOutputStream("tab");
ObjectOutputStream out = new ObjectOutputStream(fileOut);
HashMap<?,?> hash = getHashMap();
out.writeObject(hash);
```

As you can see, this approach is quite straightforward. The single `writeObject` on `hash` writes the entire contents of the hash map, including all entries, all the objects that the entries refer to, and so on, until the entire graph of interconnected objects has been visited. A new copy of the hash map could be reconstituted from the serialized bytes:

```
FileInputStream fileIn = new FileInputStream("tab");
ObjectInputStream in = new ObjectInputStream(fileIn);
HashMap<?,?> newHash = (HashMap<?,?>) in.readObject();
```

Serialization preserves the integrity of the graph itself. Suppose, for example, that in a serialized hash map, an object was stored under two different keys:



When the serialized hash map is deserialized, the two analogous entries in the new copy of the hash map will have references to a single copy of the `rose.jpg` object, not references to two separate copies of `rose.jpg`.[2]

[2] The first `key` field is the word “rose” in Tibetan.

Sometimes, however, sharing objects in this way is not what is desired. In that case you can use `ObjectOutputStream`'s `writeUnshared` method to write the object as a new distinct object, rather than using a reference to an existing serialization of that object. Any object written into the graph by `writeUnshared` will only ever have one reference to it in the serialized data. The `readUnshared` method of `ObjectInputStream` reads an object that is expected to be unique. If the object is actually a reference to an existing deserialized object then an `ObjectStreamException` is thrown; similarly, if the deserialization process later tries to create a second reference to an object returned by `readUnshared`, an `ObjectStreamException` is thrown. These uniqueness checks only apply to the actual object passed to `writeUnshared` or read by `readUnshared`, not to any objects they refer to.

20.8.2 Making Your Classes `Serializable`

When an `ObjectOutputStream` writes a serialized object, the object must implement the `Serializable` marker interface. This marker interface declares that the class is designed to have its objects serialized.

Being serializable can be quite simple. The default serialization process is to serialize each field of the object that is neither `transient` nor `static`. Primitive types and strings are written in the same encoding used by `DataOutputStream`; objects are serialized by calling `writeObject`. With default serialization, all serialized fields that are object references must refer to serializable object types. Default serialization also requires either that your superclass have a no-arg constructor (so that deserialization can invoke it) or that it also be `Serializable` (in which case declaring your class to implement `Serializable` is redundant but harmless). For most classes this default serialization is sufficient, and the entire work necessary to make a class serializable is to mark it as such by declaring that it implements the `Serializable` interface:

```
public class Name implements java.io.Serializable {
    private String name;
    private long id;
    private transient boolean hashSet = false;
    private transient int hash;
    private static long nextID = 0;

    public Name(String name) {
        this.name = name;
        synchronized (Name.class) {
            id = nextID++;
```

```
        }
    }

    public int hashCode() {
        if (!hashSet) {
            hash = name.hashCode();
            hashSet = true;
        }
        return hash;
    }

    // ... override equals, provide other useful methods
}
```

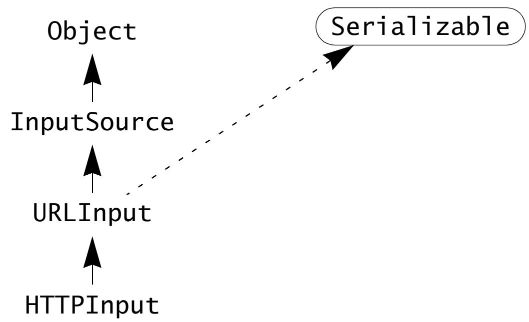
The class `Name` can be written to an `ObjectOutputStream` either directly with `writeObject`, or indirectly if it is referenced by an object written to such a stream. The `name` and `id` fields will be written to the stream; the fields `nextID`, `hashSet`, and `hash` will not be written, `nextID` because it is `static` and the others because they are declared `transient`. Because `hash` is a cached value that can easily be recalculated from `name`, there is no reason to consume the time and space it takes to write it to the stream.

Default deserialization reads the values written during serialization. Static fields in the class are left untouched—if the class needs to be loaded then the normal initialization of the class takes place, giving the static fields an initial value. Each transient field in the reconstituted object is set to the default value for its type. When a `Name` object is deserialized, the newly created object will have `name` and `id` set to the same values as those of the original object, the static field `nextID` will remain untouched, and the transient fields `hashSet` and `hash` will have their default values (`false` and `0`). These defaults work because when `hashSet` is `false` the value of `hash` will be recalculated.

You will occasionally have a class that is generally serializable but has specific instances that are not serializable. For example, a container might itself be serializable but contain references to objects that are not serializable. Any attempt to serialize a non-serializable object will throw a `NotSerializableException`.

20.8.3 Serialization and Deserialization Order

Each class is responsible for properly serializing its own state—that is, its fields. Objects are serialized and deserialized down the type tree—from the highest-level class that is `Serializable` to the most specific class. This order is rarely important when you're serializing, but it can be important when you're deserializing. Let us consider the following type tree for an `HTTPInput` class:



When deserializing an `HTTPInput` object, `ObjectInputStream` first allocates memory for the new object and then finds the first `Serializable` class in the object's type hierarchy—in this case `URLInput`. The stream invokes the no-arg constructor of that class's superclass (the object's last non-serializable class), which in this case is `InputSource`. If other state from the superclass must be preserved, `URLInput` is responsible for serializing that state and restoring it on deserialization. If your non-serializable superclass has state, you will almost certainly need to customize the first serializable class (see the next section). If the first serializable class directly extends `Object` (as the earlier `Name` class did), customizing is easy because `Object` has no state to preserve or restore.

Once the first serializable class has finished with its part of its superclass's state, it will set its own state from the stream. Then `ObjectInputStream` will walk down the type tree, deserializing the state for each class using `readObject`. When `ObjectInputStream` reaches the bottom of the type tree, the object has been completely deserialized.

As the stream is deserialized, other serialized objects will be found that were referenced from the object currently being deserialized. These other objects are deserialized as they are encountered. Thus, if `URLInput` had a reference to a `HashMap`, that hash map and its contents would be deserialized before the `HTTPInput` part of the object was deserialized.

Before any of this can happen, the relevant classes must first be loaded. This requires finding a class of the same name as the one written and checking to see that it is the same class. You'll learn about versioning issues shortly. Assuming it is the same class, the class must be loaded. If the class is not found or cannot be loaded for any reason, `readObject` will throw a `ClassNotFoundException`.

20.8.4 Customized Serialization

The default serialization methods work for many classes but not for all of them. For some classes default deserialization may be improper or inefficient. The HashMap class is an example of both problems. Default serialization would write all the data structures for the hash map, including the hash codes of the entries. This serialization is both wrong and inefficient.

It is wrong because hash codes may be different for deserialized entries. This will be true, for example, of entries using the default hashCode implementation.

It is inefficient because a hash map typically has a significant number of empty buckets. There is no point in serializing empty buckets. It would be more efficient to serialize the referenced keys and entries and rebuild a hash map from them than to serialize the entire data structure of the map.

For these reasons, java.util.HashMap provides private writeObject and readObject methods.[3] These methods are invoked by ObjectOutputStream and ObjectInputStream, respectively, when it is time to serialize or deserialize a HashMap object. These methods are invoked only on classes that provide them, and the methods are responsible only for the class's own state, including any state from non-serializable superclasses. A class's writeObject and readObject methods, if provided, should *not* invoke the superclass's readObject or writeObject method. Object serialization differs in this way from clone and finalize.

Let us suppose, for example, that you wanted to improve the Name class so that it didn't have to check whether the cached hash code was valid each time. You could do this by setting hash in the constructor, instead of lazily when it is asked for. But this causes a problem with serialization—since hash is transient it does not get written as part of serialization (nor should it), so when you are deserializing you need to explicitly set it. This means that you have to implement readObject to deserialize the main fields and then set hash, which implies that you have to implement writeObject so that you know how the main fields were serialized.

```
public class BetterName implements Serializable {
    private String name;
    private long id;
    private transient int hash;
```

[3] These methods are private because they should never be overridden and they should never be invoked by anyone using or subclassing your class. The serialization mechanism gains access to these private methods using reflection to disable the language level access control (see page 426). Of course this can only happen if the current security policy allows it—see "Security" on page 677.

```
    private static long nextID = 0;

    public BetterName(String name) {
        this.name = name;
        synchronized (BetterName.class) {
            id = nextID++;
        }
        hash = name.hashCode();
    }

    private void writeObject(ObjectOutputStream out)
        throws IOException
    {
        out.writeUTF(name);
        out.writeLong(id);
    }

    private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException
    {
        name = in.readUTF();
        id = in.readLong();
        hash = name.hashCode();
    }

    public int hashCode() {
        return hash;
    }

    // ... override equals, provide other useful methods
}
```

We use `writeObject` to write out each of the non-static, non-transient fields. It declares that it can throw `IOException` because the write methods it invokes can do so, and, if one does throw an exception, the serialization must be halted. When `readObject` gets the values from the stream, it can then set `hash` properly. It, too, must declare that it throws `IOException` because the read methods it invokes can do so, and this should stop deserialization. The `readObject` method must declare that it throws `ClassNotFoundException` because, in the general case, deserializing fields of the current object could require other classes to be loaded—though not in the example.

There is one restriction on customized serialization: You cannot directly set a `final` field within `readObject` because `final` fields can only be set in initializers or constructors. For example, if `name` was declared `final` the class `BetterName` would not compile. You will need to design your classes with this restriction in mind when considering custom serialization. The default serialization mechanism can bypass this restriction because it uses native code. This means that default serialization works fine with classes that have `final` fields. For custom serialization it is possible to use reflection to set a final field—see "Final Fields" on page 420—but the security restrictions for doing this means that it is seldom applicable. One circumstance in which it is applicable, for example, is if your classes are required to be installed as a standard extension and so have the necessary security privileges—see "Security Policies" on page 680.

The `readObject` and `writeObject` methods for `BetterName` show that you can use the methods of `DataInput` and `DataOutput` to transmit arbitrary data on the stream. However, the actual implementations replicate the default serialization and then add the necessary setup for `hash`. The read and write invocations of these methods could have been replaced with a simple invocation of methods that perform default serialization and deserialization:

```
private void writeObject(ObjectOutputStream out)
    throws IOException
{
    out.defaultWriteObject();
}

private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException
{
    in.defaultReadObject();
    hash = name.hashCode();
}
```

In fact, as you may have surmised, given that `writeObject` performs nothing but default serialization, we need not have implemented it at all.

A `writeObject` method can throw `NotSerializableException` if a particular object is not serializable. For example, in rare cases, objects of a class might be generally serializable, but a particular object might contain sensitive data.

You will occasionally find that an object cannot be initialized properly until the graph of which it is a part has been completely deserialized. You can have the `ObjectInputStream` invoke a method of your own devising by calling the stream's `registerValidation` method with a reference to an object that implements the interface `ObjectInputValidation`. When deserialization of the top-level object at the head of the graph is complete, your object's `validateObject` method will be invoked to make any needed validation operation or check.

Normally, an object is serialized as itself on the output stream, and a copy of the same type is reconstituted during deserialization. You will find a few classes for which this is not correct. For example, if you have a class that has objects that are supposed to be unique in each virtual machine for each unique value (so that == will return `true` if and only if `equals` also would return `true`), you would need to resolve an object being deserialized into an equivalent one in the local virtual machine. You can control these by providing `writeReplace` and `readResolve` methods of the following forms and at an appropriate access level:

<access> `Object` **`writeReplace()`** `throws ObjectStreamException`
: Returns an object that will replace the current object during serialization. Any object may be returned including the current one.

<access> `Object` **`readResolve()`** `throws ObjectStreamException`
: Returns an object that will replace the current object during deserialization. Any object may be returned including the current one.

In our example, `readResolve` would check to find the local object that was equivalent to the one just deserialized—if it exists it will be returned, otherwise we can register the current object (for use by `readResolve` in the future) and return `this`. These methods can be of any accessibility; they will be used if they are accessible to the object type being serialized. For example, if a class has a private `readResolve` method, it only affects deserialization of objects that are exactly of its type. A package-accessible `readResolve` affects only subclasses within the same package, while public and protected `readResolve` methods affect objects of all subclasses.

20.8.5 Object Versioning

Class implementations change over time. If a class's implementation changes between the time an object is serialized and the time it is deserialized, the `ObjectInputStream` can detect this change. When the object is written, the *serial version UID* (unique identifier), a 64-bit `long` value, is written with it. By default, this identifier is a secure hash of the full class name, superinterfaces, and members—the facts about the class that, if they change, signal a possible class incompatibility. Such a hash is essentially a fingerprint—it is nearly impossible for two different classes to have the same UID.

When an object is read from an `ObjectInputStream`, the serial version UID is also read. An attempt is then made to load the class. If no class with the same name is found or if the loaded class's UID does not match the UID in the stream, `readObject` throws an `InvalidClassException`. If the versions of all the classes in the object's type are found and all the UIDs match, the object can be deserialized.

This assumption is very conservative: Any change in the class creates an incompatible version. Many class changes are less drastic than this. Adding a cache to a class can be made compatible with earlier versions of the serialized form, as can adding optional behavior or values. Rather then relying on the default serial version UID, any serializable class should explicitly declare its own serial version UID value. Then when you make a change to a class that can be compatible with the serialized forms of earlier versions of the class, you can explicitly declare the serial version UID for the earlier class. A serial version UID is declared as follows:

```
private static final
    long serialVersionUID = -1307795172754062330L;
```

The `serialVersionUID` field must be a static, final field of type `long`. It should also be private since it is only applied to the declaring class. The value of `serialVersionUID` is provided by your development system. In many development systems, it is the output of a command called `serialver`. Other systems have different ways to provide you with this value, which is the serial version UID of the class before the first incompatible modification. (Nothing prevents you from using any number as this UID if you stamp it from the start, but it is usually a really bad idea. Your numbers will not be as carefully calculated to avoid conflict with other classes as the secure hash is.)

Now when the `ObjectInputStream` finds your class and compares the UID with that of the older version in the file, the UIDs will be the same even though the implementation has changed. If you invoke `defaultReadObject`, only those fields that were present in the original version will be set. Other fields will be left in their default state. If `writeObject` in the earlier version of the class wrote values on the field without using `defaultWriteObject`, you must read those values. If you try to read more values than were written, you will get an `EOFException`, which can inform you that you are deserializing an older form that wrote less information. If possible, you should design classes with a class version number instead of relying on an exception to signal the version of the original data.

When an object is written to an `ObjectOutputStream`, the `Class` object for that object is also written. Because `Class` objects are specific to each virtual machine, serializing the actual `Class` object would not be helpful. So `Class` objects on a stream are replaced by `ObjectStreamClass` objects that contain the information necessary to find an equivalent class when the object is deserialized.

This information includes the class's full name and its serial version UID. Unless you create one, you will never directly see an `ObjectStreamClass` object.

As a class evolves it is possible that a new superclass is introduced for that class. If an older serialized form of the class is deserialized it will not contain any serialized data for that superclass. Rather than making this an error, the system will set all fields declared by the superclass to their default initialized values. To override this default behavior, the new superclass (which must implement `Serializable`, of course) can declare the following method:

```
private void readObjectNoData() throws ObjectStreamException
```

If, as an object is deserialized, the serialized data lists the superclass as a known superclass then the superclass's `readObject` method will be invoked (if it exists), otherwise the superclass's `readObjectNoData` method will be invoked. The `readObjectNoData` method can then set appropriate values in the object's superclass fields.

20.8.6 Serialized Fields

The default serialization usually works well, but for more sophisticated classes and class evolution you may need to access the original fields. For example, suppose you were representing a rectangle in a geometric system by using two opposite corners. You would have four fields: `x1`, `y1`, `x2`, and `y2`. If you later want to use a corner, plus width and height, you would have four different fields: `x`, `y`, `width`, and `height`. Assuming default serialization of the four original fields you would also have a compatibility problem: the rectangles that were already serialized would have the old fields instead of the new ones. To solve this problem you could maintain the serialized format of the original class and convert between the old and new fields as you encounter them in `readObject` or `writeObject`. You do this using *serialized field* types to view the serialized form as an abstraction and to access individual fields:

```
public class Rectangle implements Serializable {
    private static final
        long serialVersionUID = -1307795172754062330L;
    private static final
        ObjectStreamField[] serialPersistentFields = {
            new ObjectStreamField("x1", Double.TYPE),
            new ObjectStreamField("y1", Double.TYPE),
            new ObjectStreamField("x2", Double.TYPE),
            new ObjectStreamField("y2", Double.TYPE),
        };
```

```
        private transient double x, y, width, height;

        private void readObject(ObjectInputStream in)
            throws IOException, ClassNotFoundException
        {
            ObjectInputStream.GetField fields;
            fields = in.readFields();
            x = fields.get("x1", 0.0);
            y = fields.get("y1", 0.0);
            double x2 = fields.get("x2", 0.0);
            double y2 = fields.get("y2", 0.0);
            width = (x2 - x);
            height = (y2 - y);
        }

        private void writeObject(ObjectOutputStream out)
            throws IOException
        {
            ObjectOutputStream.PutField fields;
            fields = out.putFields();
            fields.put("x1", x);
            fields.put("y1", y);
            fields.put("x2", x + width);
            fields.put("y2", y + height);
            out.writeFields();
        }
    }
```

`Rectangle` keeps the `serialVersionUID` of the original version to declare that the versions are compatible. Changing fields that would be used by default serialization is otherwise considered to be an incompatible change.

To represent each of the old fields that will be found in the serialized data, you create an `ObjectStreamField` object. You construct each `ObjectStreamField` object by passing in the name of the field it represents, and the `Class` object for the type of the field it represents. An overloaded constructor also takes a `boolean` argument that specifies whether the field refers to an unshared object—that is, one written by `writeUnshared` or read by `readUnshared`. The serialization mechanism needs to know where to find these `ObjectStreamField` objects, so they must be defined in the static, final array called `serialPersistentFields`.

The fields `x`, `y`, `width`, and `height` are marked `transient` because they are not serialized—during serialization these new fields must be converted into appropriate values of the original fields so that we preserve the serialized form. So `writeObject` uses an `ObjectOutputStream.PutField` object to write out the old form, using `x` and `y` as the old `x1` and `y1`, and calculating `x2` and `y2` from the rectangle's `width` and `height`. Each `put` method takes a field name as one argument and a value for that field as the other—the type of the value determines which overloaded form of `put` is invoked (one for each primitive type and `Object`). In this way the default serialization of the original class has been emulated and the serialized format preserved.

When a `Rectangle` object is deserialized, the reverse process occurs. Our `readObject` method gets an `ObjectInputStream.GetField` that allows access to fields by name from the serialized object. There is a `get` method for returning each primitive type, and one for returning an `Object` reference. Each `get` method takes two parameters: the name of the field and a value to return if it is not defined in the serialized object. The return value's type chooses which overload of `get` is used: A `short` return value will use the `get` that returns a `short`, for example. In our example, all values are `double`: We get the `x1` and `y1` fields to use for one corner of the rectangle, and the old `x2` and `y2` fields to calculate `width` and `height`.

Using the above technique the new `Rectangle` class can deserialize old rectangle objects and a new serialized rectangle can be deserialized by the original `Rectangle` class, provided that both virtual machines are using compatible versions of the serialization stream protocol. The stream protocol defines the actual layout of serialized objects in the stream regardless of whether they use default serialization or the serialized field objects. This means that the serialized form of an object is not dependent on, for example, the order in which you invoke `put`, nor do you have to know the order in which to invoke `get`—you can use `get` or `put` to access fields in any order any number of times.

20.8.7 The `Externalizable` Interface

The `Externalizable` interface extends `Serializable`. A class that implements `Externalizable` takes complete control over its serialized state, assuming responsibility for all the data of its superclasses, any versioning issues, and so on. You may need this, for example, when a repository for serialized objects mandates restrictions on the form of those objects that are incompatible with the provided serialization mechanism. The `Externalizable` interface has two methods:

```
public interface Externalizable extends Serializable {
    void writeExternal(ObjectOutput out)
        throws IOException;
```

```
    void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException;
}
```

These methods are invoked when the object is serialized and deserialized, respectively. They are normal public methods, so the exact type of the object determines which implementation will be used. Subclasses of an externalizable class will often need to invoke their superclass's implementation before serializing or deserializing their own state—in contrast to classes that use normal serialization.

You should note that the methods of the interface are public and so can be invoked by anyone at anytime. In particular, a malicious program might invoke `readExternal` to make an object overwrite its state from some serialized stream, possibly with invented content. If you are designing classes where such security counts you have to take this into account either by not using `Externalizable` or by writing your `readExternal` method to be only invoked once, and never at all if the object was created via one of your constructors.

20.8.8 Documentation Comment Tags

As you can see from the `Rectangle` code, the serialized form of an object can be an important thing, separate from its runtime form. This can happen over time due to evolution, or by initial design when the runtime form is not a good serialized form. When you write serializable classes that others will reimplement, you should document the persistent form so that other programmer's can properly reimplement the serialized form as well as the runtime behavior. You do this with the special javadoc tags `@serial`, `@serialField`, and `@serialData`.

Use `@serial` to document fields that use default serialization. For example, the original `Rectangle` class could have looked like this:

```
/** X-coordinate of one corner.
 *  @serial */
private double x1;
/** Y-coordinate of one corner.
 *  @serial */
private double y1;
/** X-coordinate of opposite corner.
 *  @serial */
private double x2;
/** Y-coordinate of opposite corner.
 *  @serial */
private double y2;
```

The @serial tag can include a description of the meaning of the field. If none is given (as above), then the description of the runtime field will be used. The javadoc tool will add all @serial information to a page, known as the *serialized form page.*

The @serial tag can also be applied to a class or package with the single argument include or exclude, to control whether serialization information is documented for that class or package. By default public and protected types are included, otherwise they are excluded. A class level @serial tag overrides a package level @serial tag.

The @serialField tag documents fields that are created by GetField and PutField invocations, such as those in our Rectangle example. The tag takes first the field name, then its type, and then a description. For example:

```
/** @serialField x1 double X-coordinate of one corner. */
/** @serialField y1 double Y-coordinate of one corner. */
/** @serialField x2 double X-coordinate of other corner. */
/** @serialField y2 double Y-coordinate of other corner. */
private transient double x, y, width, height;
```

You use the @serialData tag in the doc comment for a writeObject method to document any additional data written by the method. You can also use @serialData to document anything written by an Externalizable class's writeExternal method.

20.9 The IOException Classes

Every I/O-specific error detected by classes in java.io is signaled by an IOException or a subclass. Most I/O classes are designed to be general, so most of the exceptions cannot be listed specifically. For example, InputStream methods that throw IOException cannot detail which particular exceptions might be thrown, because any particular input stream class might throw a subclass of IOException for particular error conditions relevant to that stream. And the filter input and output streams pass through exceptions only from their downstream objects, which can also be of other stream types.

The specific subclasses of IOException used in the java.io package are

CharConversionException extends IOException

Thrown when a character conversion problem occurs in a character stream operation that must convert local character codes to Unicode or vice versa.

EOFException extends IOException

Thrown when the end of the file (stream) is detected while reading.

`FileNotFoundException` `extends IOException`

Thrown when the attempt to access the file specified by a given pathname fails—presumably because the file does not exist.

`InterruptedIOException` `extends IOException`

Thrown when a blocking I/O operation detects that the current thread has been interrupted before or during the operation. In principle, except for the `Print` stream methods, interrupting a thread should cause this exception if the thread is performing a blocking I/O operation. In practice most implementations only check for interruption before performing an operation and do not respond to interruption during the operation (see page 515) so you cannot rely on the ability to interrupt a blocked thread. This exception is also used to signify that a time-out occurred during network I/O.

`InvalidClassException` `extends ObjectStreamException`

Thrown when the serialization mechanism detects a problem with a class: The serial version of the class does not match that read from the stream, the class contains unknown data types, or the class does not have an accessible no-arg constructor when needed.

`InvalidObjectException` `extends ObjectStreamException`

Thrown when the `validateObject` method cannot make the object valid, thus aborting the deserialization.

`NotActiveException` `extends ObjectStreamException`

Thrown when a serialization method, such as `defaultReadObject`, is invoked when serialization is not under way on the stream.

`NotSerializableException` `extends ObjectStreamException`

Thrown either by the serialization mechanism or explicitly by a class when a class cannot be serialized.

`ObjectStreamException` `extends IOException`

The superclass for all the `Object` stream related exceptions.

`OptionalDataException` `extends ObjectStreamException`

Thrown when the optional data (that is, not part of default serialization) in the object input stream is corrupt or was not read by the reading method.

`StreamCorruptedException` `extends ObjectStreamException`

Thrown when internal object stream state is missing or invalid.

`SyncFailedException` `extends IOException`

Thrown by `FileDescriptor.sync` when the data cannot be guaranteed to have been written to the underlying media.

`UnsupportedEncodingException` `extends IOException`

Thrown when an unknown character encoding is specified.

`UTFDataFormatException` `extends IOException`

Thrown by `DataInputStream.readUTF` when the string it is reading has malformed UTF syntax.

`WriteAbortedException` `extends ObjectStreamException`

Thrown when an exception occurred during a serialization write operation.

In addition to these specific exceptions, other exceptional conditions in `java.io` are signaled with an `IOException` containing a string that describes the specific error encountered. For example, using a `Piped` stream object that has never been connected throws an exception object with a detail string such as `"Pipe not connected"`, and trying to push more than the allowed number of characters onto a `PushbackReader` throws an exception with the string `"Pushback buffer overflow"`. Such exceptions are difficult to catch explicitly, so this style of exception reporting is not in favor. Specific exception subtypes should be created for each category of exceptional circumstance.

20.10 A Taste of New I/O

The `java.nio` package ("New I/O") and its subpackages give you access to high performance I/O, albeit with more complexity. Instead of a simple stream model you have control over buffers, channels, and other abstractions to let you get maximum speed for your I/O needs. This is recommended only for those who have a demonstrated need.

The model for rapid I/O is to use buffers to walk through channels of primitive types. Buffers are containers for data and are associated with channels that connect to external data sources. There are buffer types for all primitive types: A `FloatBuffer` works with `float` values, for example. The `ByteBuffer` is more general; it can handle any primitive type with methods such as `getFloat` and `putLong`. `MappedByteBuffer` helps you map a large file into memory for quick access. You can use character set decoders and encoders to translate buffers of bytes to and from Unicode.

Channels come from objects that access external data, namely files and sockets. `FileInputStream` has a `getChannel` method that returns a channel for that stream, as do `RandomAccessFile`, `java.net.Socket`, and others.

Here is some code that will let you efficiently access a large text file in a specified encoding:

```
public static int count(File file, String charSet, char ch)
    throws IOException
{
    Charset charset = Charset.forName(charSet);
```

```
        CharsetDecoder decoder = charset.newDecoder();
        FileInputStream fis = new FileInputStream(file);
        FileChannel fc = fis.getChannel();

        // Get the file's size and then map it into memory
        long size = fc.size();
        MappedByteBuffer bb =
            fc.map(FileChannel.MapMode.READ_ONLY, 0, size);
        CharBuffer cb = decoder.decode(bb);
        int count = 0;
        for (int i = 0; i < size && i < Integer.MAX_VALUE; i++)
            if (cb.charAt(i) == ch)
                count++;
        fc.close();
        return count;
    }
```

We use a FileInputStream to get a channel for the file. Then we create a mapped buffer for the entire file. What a "mapped buffer" does may vary with the platform, but for large files (greater than a few tens of kilobytes) you can assume that it will be at least as efficient as streaming through the data, and nearly certainly much more efficient. We then get a decoder for the specified character set, which gives us a CharBuffer from which to read.[4]

The CharBuffer not only lets you read (decoded) characters from the file, it also acts as a CharSequence and, therefore, can be used with the regular expression mechanism.

In addition to high-performance I/O, the new I/O package also provides a different programming model that allows for non-blocking I/O operations to be performed. This is an advanced topic well beyond the scope of this book, but suffice it to say that this allows a small number of threads to efficiently manage a large number of simultaneous I/O connections.

There is also a reliable file locking mechanism: You can lock a FileChannel and receive a java.nio.channels.FileLock object that represents either a shared or exclusive lock on a file. You can release the FileLock when you are done with it.

Nothing has really happened until it has been recorded.
—Virginia Woolf

---

4 Note that there is an unfortunate discrepancy between the ability to map huge files and the fact that the returned buffer has a capacity that is limited to Integer.MAX_VALUE.