Class name	Low/ High level	Description
`BufferedOutputStream`	High	Writes byte data to existing `OutputStream` in buffered manner, which improves efficiency and performance
`BufferedReader`	High	Reads character data from existing `Reader` in buffered manner, which improves efficiency and performance
`BufferedWriter`	High	Writes character data to existing `Writer` in buffered manner, which improves efficiency and performance
`ObjectInputStream`	High	Deserializes primitive Java data types and graphs of Java objects from existing `InputStream`
`ObjectOutputStream`	High	Serializes primitive Java data types and graphs of Java objects to existing `OutputStream`
`PrintStream`	High	Writes formatted representations of Java objects to binary stream
`PrintWriter`	High	Writes formatted representations of Java objects to character stream

Keep Table 14.7 and Table 14.8 handy as you learn more about I/O streams in this chapter. We discuss these in more detail, including examples of each.

Reading and Writing Files

There are a number of ways to read and write from a file. We show them in this section by copying one file to another.

Using I/O Streams

I/O streams are all about reading/writing data, so it shouldn't be a surprise that the most important methods are `read()` and `write()`. Both `InputStream` and `Reader` declare a `read()` method to read byte data from an I/O stream. Likewise, `OutputStream` and `Writer` both define a `write()` method to write a byte to the stream:

The following `copyStream()` methods show an example of reading all of the values of an `InputStream` and `Reader` and writing them to an `OutputStream` and `Writer`, respectively. In both examples, `-1` is used to indicate the end of the stream.

```
void copyStream(InputStream in, OutputStream out) throws IOException {
   int b;
   while ((b = in.read()) != -1) {
      out.write(b);
   }
}

void copyStream(Reader in, Writer out) throws IOException {
   int b;
   while ((b = in.read()) != -1) {
      out.write(b);
   }
}
```

Hold on. We said we are reading and writing bytes, so why do the methods use `int` instead of `byte`? Remember, the `byte` data type has a range of 256 characters. They needed an extra value to indicate the end of an I/O stream. The authors of Java decided to use a larger data type, `int`, so that special values like `-1` would indicate the end of an I/O stream. The output stream classes use `int` as well, to be consistent with the input stream classes.

Reading and writing one byte at a time isn't a particularly efficient way of doing this. Luckily, there are overloaded methods for reading and writing multiple bytes at a time. The `offset` and `length` values are applied to the array itself. For example, an `offset` of `3` and `length` of `5` indicates that the stream should read up to five bytes/characters of data and put them into the array starting with position `3`. Let's look at an example:

```
10: void copyStream(InputStream in, OutputStream out) throws IOException {
11:    int batchSize = 1024;
12:    var buffer = new byte[batchSize];
13:    int lengthRead;
14:    while ((lengthRead = in.read(buffer, 0, batchSize)) > 0) {
15:       out.write(buffer, 0, lengthRead);
16:       out.flush();
17:    }
```

Instead of reading the data one byte at a time, we read and write up to `1024` bytes at a time on line 14. The return value `lengthRead` is critical for determining whether we are at the end of the stream and knowing how many bytes we should write into our output stream.

Unless our file happens to be a multiple of `1024` bytes, the last iteration of the `while` loop will write some value less than `1024` bytes. For example, if the buffer size is 1,024 bytes and the file size is 1,054 bytes, the last read will be only 30 bytes. If we ignored this return value and instead wrote 1,024 bytes, 994 bytes from the previous loop would be written to the end of the file.

We also added a `flush()` method on line 16 to reduce the amount of data lost if the application terminates unexpectedly. When data is written to an output stream, the underlying operating system does not guarantee that the data will make it to the file system immediately. The `flush()` method requests that all accumulated data be written immediately to disk. It is not without cost, though. Each time it is used, it may cause a noticeable delay in the application, especially for large files. Unless the data that you are writing is extremely critical, the `flush()` method should be used only intermittently. For example, it should not necessarily be called after every write, as it is in this example.

Equivalent methods exist on `Reader` and `Writer`, but they use `char` rather than `byte`, making the equivalent `copyStream()` method very similar.

The previous example makes reading and writing a file look like a lot to think about. That's because it only uses low-level I/O streams. Let's try again using high-level streams.

```
26: void copyTextFile(File src, File dest) throws IOException {
27:    try (var reader = new BufferedReader(new FileReader(src));
28:       var writer = new BufferedWriter(new FileWriter(dest))) {
29:       String line = null;
30:       while ((line = reader.readLine()) != null) {
31:          writer.write(line);
32:          writer.newLine();
33:       } } }
```

The key is to choose the most useful high-level classes. In this case, we are dealing with a `File`, so we want to use a `FileReader` and `FileWriter`. Both classes have constructors that can take either a `String` representing the location or a `File` directly.

If the source file does not exist, a `FileNotFoundException`, which inherits `IOException`, will be thrown. If the destination file already exists, this implementation will overwrite it. We can pass an optional `boolean` second parameter to `FileWriter` for an `append` flag if we want to change this behavior.

We also chose to use a `BufferedReader` and `BufferedWriter` so we can read a whole line at a time. This gives us the benefits of reading batches of characters on line 30 without having to write custom logic. Line 31 writes out the whole line of data at once. Since reading a line strips the line breaks, we add those back on line 32. Lines 27 and 28 demonstrate chaining constructors. The try-with-resources constructor takes care of closing all the objects in the chain.

Now imagine that we wanted `byte` data instead of characters. We would need to choose different high-level classes: `BufferedInputStream`, `BufferedOutputStream`, `FileInputStream`, and `FileOuputStream`. We would call `readAllBytes()` instead of `readLine()` and store the result in a `byte[]` instead of a `String`. Finally, we wouldn't need to handle new lines since the data is binary.

We can do a little better than `BufferedOutputStream` and `BufferedWriter` by using a `PrintStream` and `PrintWriter`. These classes contain four key methods. The `print()` and `println()` methods print data with and without a new line, respectively. There are also the `format()` and `printf()` methods, which we describe in the section on user interactions.

```
void copyTextFile(File src, File dest) throws IOException {
   try (var reader = new BufferedReader(new FileReader(src));
      var writer = new PrintWriter(new FileWriter(dest))) {
      String line = null;
      while ((line = reader.readLine()) != null)
          writer.println(line);
      }
}
```

While we used a `String`, there are numerous overloaded versions of `println()`, which take everything from primitives and `String` values to objects. Under the covers, these methods often just perform `String.valueOf()`.

The print stream classes have the distinction of being the only I/O stream classes we cover that do not have corresponding input stream classes. And unlike other `OutputStream` classes, `PrintStream` does not have `Output` in its name.

NOTE It may surprise you that you've been regularly using a `PrintStream` throughout this book. Both `System.out` and `System.err` are `PrintStream` objects. Likewise, `System.in`, often useful for reading user input, is an `InputStream`.

Unlike the majority of the other I/O streams we've covered, the methods in the print stream classes do not throw any checked exceptions. If they did, you would be required to catch a checked exception any time you called `System.out.print()`!

The line separator is `\n` or `\r\n`, depending on your operating system. The `println()` method takes care of this for you. If you need to get the character directly, either of the following will return it for you:

```
System.getProperty("line.separator");
System.lineSeparator();
```

Enhancing with *Files*

The NIO.2 APIs provide even easier ways to read and write a file using the `Files` class. Let's start by looking at three ways of copying a file by reading in the data and writing it back:

```
private void copyPathAsString(Path input, Path output) throws IOException {
   String string = Files.readString(input);
   Files.writeString(output, string);
}
private void copyPathAsBytes(Path input, Path output) throws IOException {
    byte[] bytes = Files.readAllBytes(input);
    Files.write(output, bytes);
}
private void copyPathAsLines(Path input, Path output) throws IOException {
   List<String> lines = Files.readAllLines(input);
   Files.write(output, lines);
}
```

That's pretty concise! You can read a `Path` as a `String`, a `byte` array, or a `List`. Be aware that the entire file is read at once for all three of these, thereby storing all of the contents of the file in memory at the same time. If the file is significantly large, you may trigger an `OutOfMemoryError` when trying to load all of it into memory. Luckily, there is an alternative. This time, we print out the file as we read it.

```
private void readLazily(Path path) throws IOException {
   try (Stream<String> s = Files.lines(path)) {
      s.forEach(System.out::println);
   }
}
```

Now the contents of the file are read and processed lazily, which means that only a small portion of the file is stored in memory at any given time. Taking things one step further, we can leverage other stream methods for a more powerful example.

```
try (var s = Files.lines(path)) {
   s.filter(f -> f.startsWith("WARN:"))
      .map(f -> f.substring(5))
      .forEach(System.out::println);
}
```

This sample code searches a log for lines that start with `WARN:`, outputting the text that follows. Assuming that the input file `sharks.log` is as follows:

```
INFO:Server starting
DEBUG:Processes available = 10
WARN:No database could be detected
DEBUG:Processes available reset to 0
WARN:Performing manual recovery
INFO:Server successfully started
```

Then the sample output would be the following:

```
No database could be detected
Performing manual recovery
```

As you can see, we have the ability to manipulate files in complex ways, often with only a few short expressions.

Files.readAllLines() vs. *Files.lines()*

For the exam, you need to know the difference between `readAllLines()` and `lines()`. Both of these examples compile and run:

```
Files.readAllLines(Paths.get("birds.txt")).forEach(System.out::println);
Files.lines(Paths.get("birds.txt")).forEach(System.out::println);
```

The first line reads the entire file into memory and performs a print operation on the result, while the second line lazily processes each line and prints it as it is read. The advantage of the second code snippet is that it does not require the entire file to be stored in memory at any time.

You should also be aware of when they are mixing incompatible types on the exam. Do you see why the following does not compile?

```
Files.readAllLines(Paths.get("birds.txt"))
   .filter(s -> s.length()> 2)
   .forEach(System.out::println);
```

The `readAllLines()` method returns a `List`, not a `Stream`, so the `filter()` method is not available.

Combining with *newBufferedReader()* and *newBufferedWriter()*

Sometimes you need to mix I/O streams and NIO.2. Conveniently, `Files` includes two convenience methods for getting I/O streams.

```
private void copyPath(Path input, Path output) throws IOException {
   try (var reader = Files.newBufferedReader(input);
        var writer = Files.newBufferedWriter(output)) {

      String line = null;
while ((line = reader.readLine()) != null)
   writer.write(line);
   writer.newLine();
} } }
```

You can wrap I/O stream constructors to produce the same effect, although it's a lot easier to use the factory method. The first method, `newBufferedReader()`, reads the file specified at the `Path` location using a `BufferedReader` object.

Reviewing Common Read and Write Methods

Table 14.9 reviews the `public` common I/O stream methods you should know for reading and writing. We also include `close()` and `flush()` since they are used when performing these actions. Table 14.10 does the same for common `public` NIO.2 read and write methods.

TABLE 14.9 Common I/O read and write methods

Class	Method name	Description
All input streams	`public int` **`read`**`()`	Reads single byte or returns `-1` if no bytes available.
`InputStream`	`public int` **`read`**`(byte[] b)`	Reads values into buffer. Returns number of bytes or characters read.
`Reader`	`public int` **`read`**`(char[] c)`	
`InputStream`	`public int` **`read`**`(byte[] b, int offset, int length)`	Reads up to `length` values into buffer starting from position `offset`. Returns number of bytes or characters read.
`Reader`	`public int` **`read`**`(char[] c, int offset, int length)`	
All output streams	`public void` **`write`**`(int b)`	Writes single byte.
`OutputStream`	`public void` **`write`**`(byte[] b)`	Writes array of values into stream.
`Writer`	`public void` **`write`**`(char[] c)`	
`OutputStream`	`public void` **`write`**`(byte[] b, int offset, int length)`	Writes `length` values from array into stream, starting with `offset` index.
`Writer`	`public void` **`write`**`(char[] c, int offset, int length)`	
`BufferedInputStream`	`public byte[]` **`readAllBytes`**`()`	Reads data in bytes.

TABLE 14.9 Common I/O read and write methods

Class	Method name	Description
`BufferedReader`	`public String` **`readLine`**`()`	Reads line of data.
`BufferedWriter`	`public void` **`write`**`( String line)`	Writes line of data.
`BufferedWriter`	`public void` **`newLine`**`()`	Writes new line.
All output streams	`public void` **`flush`**`()`	Flushes buffered data through stream.
All streams	`public void` **`close`**`()`	Closes stream and releases resources.

TABLE 14.10 Common `Files` NIO.2 read and write methods

Method Name	Description
`public static byte[]` **`readAllBytes`**`()`	Reads all data as bytes
`public static String` **`readString`**`()`	Reads all data into `String`
`public static List<String>` **`readAllLines`**`()`	Read all data into `List`
`public static Stream<String>` **`lines`**`()`	Lazily reads data
`public static void` **`write`**`(Path path, byte[] bytes)`	Writes array of bytes
`public static void` **`writeString`**`( Path path, String string)`	Writes `String`
`public static void` **`write`**`(Path path, List<String> list)`	Writes list of lines (technically, any `Iterable` of `CharSequence`, but you don't need to know that for the exam)

Serializing Data

Throughout this book, we have been managing our data model using classes, so it makes sense that we would want to save these objects between program executions. Data about our zoo animals' health wouldn't be particularly useful if it had to be entered every time the program runs!