Suppose `monkey.txt` contains the name `Harold` and `wolf.txt` contains the name `Howler`. The previous code prints `1` in that case because the second position is different, and we use zero-based indexing in Java. Given those values, what do you think this code prints?

```
System.out.println(Files.mismatch(
   Path.of("/animals/wolf.txt"),
   Path.of("/animals/monkey.txt")));
```

The answer is the same as the previous example. The code prints `1` again. The `mismatch()` method is symmetric and returns the same result regardless of the order of the parameters.

Introducing I/O Streams

Now that we have the basics out of the way, let's move on to I/O streams, which are far more interesting. In this section, we show you how to use I/O streams to read and write data. The "I/O" refers to the nature of how data is accessed, either by reading the data from a resource (input) or by writing the data to a resource (output).

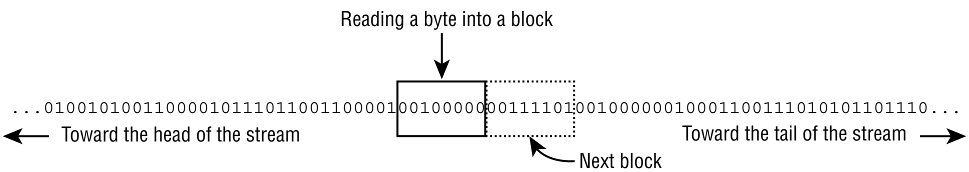
> **NOTE** When we refer to *I/O streams* in this chapter, we are referring to the ones found in the `java.io` API. If we just say *streams*, it means the ones from Chapter 10. We agree that the naming can be a bit confusing!

Understanding I/O Stream Fundamentals

The contents of a file may be accessed or written via an I/O *stream*, which is a list of data elements presented sequentially. An I/O stream can be conceptually thought of as a long, nearly never-ending stream of water with data presented one wave at a time.

We demonstrate this principle in Figure 14.5. The I/O stream is so large that once we start reading it, we have no idea where the beginning or the end is. We just have a pointer to our current position in the I/O stream and read data one block at a time.

FIGURE 14.5 Visual representation of an I/O stream



Each type of I/O stream segments data into a wave or block in a particular way. For example, some I/O stream classes read or write data as individual bytes. Other I/O stream classes read or write individual characters or strings of characters. On top of that, some I/O stream classes read or write larger groups of bytes or characters at a time, specifically those with the word `Buffered` in their name.

NOTE: Although the `java.io` API is full of I/O streams that handle characters, strings, groups of bytes, and so on, nearly all are built on top of reading or writing an individual byte or an array of bytes at a time. Higher-level I/O streams exist for convenience as well as performance.

Although I/O streams are commonly used with file I/O, they are more generally used to handle the reading/writing of any sequential data source. For example, you might construct a Java application that submits data to a website using an output stream and reads the result via an input stream.

I/O Streams Can Be Big

When writing code where you don't know what the I/O stream size will be at runtime, it may be helpful to visualize an I/O stream as being so large that all of the data contained in it could not possibly fit into memory. For example, a 1 TB file could not be stored entirely in memory by most computer systems (at the time this book is being written). The file can still be read and written by a program with very little memory, since the I/O stream allows the application to focus on only a small portion of the overall I/O stream at any given time.

Learning I/O Stream Nomenclature

The `java.io` API provides numerous classes for creating, accessing, and manipulating I/O streams—so many that it tends to overwhelm many new Java developers. Stay calm! We review the major differences between each I/O stream class and show you how to distinguish between them.

Even if you come across a particular I/O stream on the exam that you do not recognize, the name of the I/O stream often gives you enough information to understand exactly what it does.

The goal of this section is to familiarize you with common terminology and naming conventions used with I/O streams. Don't worry if you don't recognize the particular stream class names used in this section or their function; we cover how to use them in detail in this chapter.

Storing Data as Bytes

Data is stored in a file system (and memory) as a `0` or `1`, called a *bit*. Since it's really hard for humans to read/write data that is just `0`s and `1`s, they are grouped into a set of 8 bits, called a *byte*.

What about the Java `byte` primitive type? As you learn later, when we use I/O streams, values are often read or written using `byte` values and arrays.

Byte Streams vs. Character Streams

The `java.io` API defines two sets of I/O stream classes for reading and writing I/O streams: byte I/O streams and character I/O streams. We use both types of I/O streams throughout this chapter.

Differences between Byte and Character I/O Streams

- Byte I/O streams read/write binary data (`0`s and `1`s) and have class names that end in `InputStream` or `OutputStream`.
- Character I/O streams read/write text data and have class names that end in `Reader` or `Writer`.

The API frequently includes similar classes for both byte and character I/O streams, such as `FileInputStream` and `FileReader`. The difference between the two classes is based on how the bytes are read or written.

It is important to remember that even though character I/O streams do not contain the word `Stream` in their class name, they are still I/O streams. The use of `Reader`/`Writer` in the name is just to distinguish them from byte streams.

NOTE

Throughout the chapter, we refer to both `InputStream` and `Reader` as *input streams*, and we refer to both `OutputStream` and `Writer` as *output streams*.

The byte I/O streams are primarily used to work with binary data, such as an image or executable file, while character I/O streams are used to work with text files. For example, you can use a `Writer` class to output a `String` value to a file without necessarily having to worry about the underlying character encoding of the file.

The *character encoding* determines how characters are encoded and stored in bytes in an I/O stream and later read back or decoded as characters. Although this may sound simple, Java supports a wide variety of character encodings, ranging from ones that may use one byte for Latin characters, `UTF-8` and `ASCII` for example, to using two or more bytes per character, such as `UTF-16`. For the exam, you don't need to memorize the character encodings, but you should be familiar with the names.

Character Encoding in Java

In Java, the character encoding can be specified using the `Charset` class by passing a name value to the static `Charset.forName()` method, such as in the following examples:

```
Charset usAsciiCharset = Charset.forName("US-ASCII");
Charset utf8Charset = Charset.forName("UTF-8");
Charset utf16Charset = Charset.forName("UTF-16");
```

Java supports numerous character encodings, each specified by a different standard name value.

Input vs. Output Streams

Most `InputStream` classes have a corresponding `OutputStream` class, and vice versa. For example, the `FileOutputStream` class writes data that can be read by a `FileInputStream`. If you understand the features of a particular `Input` or `Output` stream class, you should naturally know what its complementary class does.

It follows, then, that most `Reader` classes have a corresponding `Writer` class. For example, the `FileWriter` class writes data that can be read by a `FileReader`.

There are exceptions to this rule. For the exam, you should know that `PrintWriter` has no accompanying `PrintReader` class. Likewise, the `PrintStream` is an `OutputStream` that has no corresponding `InputStream` class. It also does not have `Output` in its name. We discuss these classes later in this chapter.

Low-Level vs. High-Level Streams

Another way that you can familiarize yourself with the `java.io` API is by segmenting I/O streams into low-level and high-level streams.

A *low-level stream* connects directly with the source of the data, such as a file, an array, or a `String`. Low-level I/O streams process the raw data or resource and are accessed in a direct and unfiltered manner. For example, a `FileInputStream` is a class that reads file data one byte at a time.

Alternatively, a *high-level stream* is built on top of another I/O stream using wrapping. *Wrapping* is the process by which an instance is passed to the constructor of another class, and operations on the resulting instance are filtered and applied to the original instance. For example, take a look at the `FileReader` and `BufferedReader` objects in the following sample code:

```
try (var br = new BufferedReader(new FileReader("zoo-data.txt"))) {
   System.out.println(br.readLine());
}
```

In this example, `FileReader` is the low-level I/O stream, whereas `BufferedReader` is the high-level I/O stream that takes a `FileReader` as input. Many operations on the high-level I/O stream pass through as operations to the underlying low-level I/O stream, such as `read()` or `close()`. Other operations override or add new functionality to the low-level I/O stream methods. The high-level I/O stream may add new methods, such as `readLine()`, as well as performance enhancements for reading and filtering the low-level data.

High-level I/O streams can also take other high-level I/O streams as input. For example, although the following code might seem a little odd at first, the style of wrapping an I/O stream is quite common in practice:

```
try (var ois = new ObjectInputStream(
      new BufferedInputStream(
         new FileInputStream("zoo-data.txt")))) {
   System.out.print(ois.readObject());
}
```

In this example, the low-level `FileInputStream` interacts directly with the file, which is wrapped by a high-level `BufferedInputStream` to improve performance. Finally, the entire object is wrapped by another high-level `ObjectInputStream`, which allows us to interpret the data as a Java object.

For the exam, the only low-level stream classes you need to be familiar with are the ones that operate on files. The rest of the nonabstract stream classes are all high-level streams.

Stream Base Classes

The `java.io` library defines four abstract classes that are the parents of all I/O stream classes defined within the API: `InputStream`, `OutputStream`, `Reader`, and `Writer`.

The constructors of high-level I/O streams often take a reference to the abstract class. For example, `BufferedWriter` takes a `Writer` object as input, which allows it to take any subclass of `Writer`.

One common area where the exam likes to play tricks on you is mixing and matching I/O stream classes that are not compatible with each other. For example, take a look at each of the following examples and see whether you can determine why they do not compile:

```
new BufferedInputStream(new FileReader("z.txt"));  // DOES NOT COMPILE
new BufferedWriter(new FileOutputStream("z.txt")); // DOES NOT COMPILE
new ObjectInputStream(
   new FileOutputStream("z.txt"));                 // DOES NOT COMPILE
new BufferedInputStream(new InputStream());        // DOES NOT COMPILE
```

The first two examples do not compile because they mix `Reader`/`Writer` classes with `InputStream`/`OutputStream` classes, respectively. The third example does not compile because we are mixing an `OutputStream` with an `InputStream`. Although it is possible to read data from an `InputStream` and write it to an `OutputStream`, wrapping the I/O stream is not the way to do so. As you see later in this chapter, the data must be copied over. Finally, the last example does not compile because `InputStream` is an abstract class, and therefore you cannot create an instance of it.

Decoding I/O Class Names

Pay close attention to the name of the I/O class on the exam, as decoding it often gives you context clues as to what the class does. For example, without needing to look it up, it should be clear that `FileReader` is a class that reads data from a file as characters or strings. Furthermore, `ObjectOutputStream` sounds like a class that writes object data to a byte stream.

Table 14.7 lists the abstract base classes that all I/O streams inherit from.

TABLE 14.7 The `java.io` abstract stream base classes

Class name	Description
`InputStream`	Abstract class for all input byte streams
`OutputStream`	Abstract class for all output byte streams
`Reader`	Abstract class for all input character streams
`Writer`	Abstract class for all output character streams

Table 14.8 lists the concrete I/O streams that you should be familiar with for the exam. Note that most of the information about each I/O stream, such as whether it is an input or output stream or whether it accesses data using bytes or characters, can be decoded by the name alone.

TABLE 14.8 The `java.io` concrete I/O stream classes

Class name	Low/ High level	Description
`FileInputStream`	Low	Reads file data as bytes
`FileOutputStream`	Low	Writes file data as bytes
`FileReader`	Low	Reads file data as characters
`FileWriter`	Low	Writes file data as characters
`BufferedInputStream`	High	Reads byte data from existing `InputStream` in buffered manner, which improves efficiency and performance

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: