Chapter 19
I/O

OCP EXAM OBJECTIVES COVERED IN THIS CHAPTER:

- **I/O (Fundamentals and NIO2)**
  - Read data from and write console and file data using I/O Streams
  - Use I/O Streams to read and write files
  - Read and write objects by using serialization

What can Java applications do outside the scope of managing objects and attributes in memory? How can they save data so that information is not lost every time the program is terminated? They use files, of course! You can design code that writes the current state of an application to a file every time the application is closed and then reloads the data when the application is executed the next time. In this manner, information is preserved between program executions.

This chapter focuses on using the `java.io` API to interact with files and streams. We start by describing how files and directories are organized within a file system and show how to access them with the `java.io.File` class. We then show how to read and write file data with the stream classes. We conclude this chapter by discussing ways of reading user input at runtime using the `Console` class.

In Chapter 20, “NIO.2,” we will revisit the discussion of files and directories and show how Java provides more powerful techniques for managing files.

NOTE

When we refer to streams in this chapter, we are referring to the I/O streams found in the `java.io` API (unless otherwise specified). I/O streams are completely unrelated to the streams you saw in Chapter 15, “Functional Programming.” We agree that the naming can be a bit confusing!

Understanding Files and Directories

We begin this chapter by reviewing what a file and directory are within a file system. We also present the `java.io.File` class and demonstrate how to use it to read and write file information.

Conceptualizing the File System

We start with the basics. Data is stored on persistent storage devices, such as hard disk drives and memory cards. A *file* is a record within the storage device that holds data. Files are organized into hierarchies using directories. A *directory* is a location that can contain files as well as other directories. When working with directories in Java, we often treat them like files. In fact, we use many of the same classes to operate on files and directories. For example, a file and directory both can be renamed with the same Java method.

To interact with files, we need to connect to the file system. The *file system* is in charge of reading and writing data within a computer. Different operating systems use different file systems to manage their data. For example, Windows-based systems use a different file system than Unix-based ones. For the exam, you just need to know how to issue commands using the Java APIs. The JVM will automatically connect to the local file system, allowing you to perform the same operations across multiple platforms.

Next, the *root directory* is the topmost directory in the file system, from which all files and directories inherit. In Windows, it is denoted with a drive name such as `c:\`, while on Linux it is denoted with a single forward slash, `/`.

Finally, a *path* is a `String` representation of a file or directory within a file system. Each file system defines its own path separator character that is used between directory entries. The value to the left of a separator is the parent of the value to the right of the separator. For example, the path value `/user/home/zoo.txt` means that the file `zoo.txt` is inside the `home` directory, with the `home` directory inside the `user` directory. You will see that paths can be absolute or relative later in this chapter.

We show how a directory and file system is organized in a hierarchical manner in Figure 19.1.

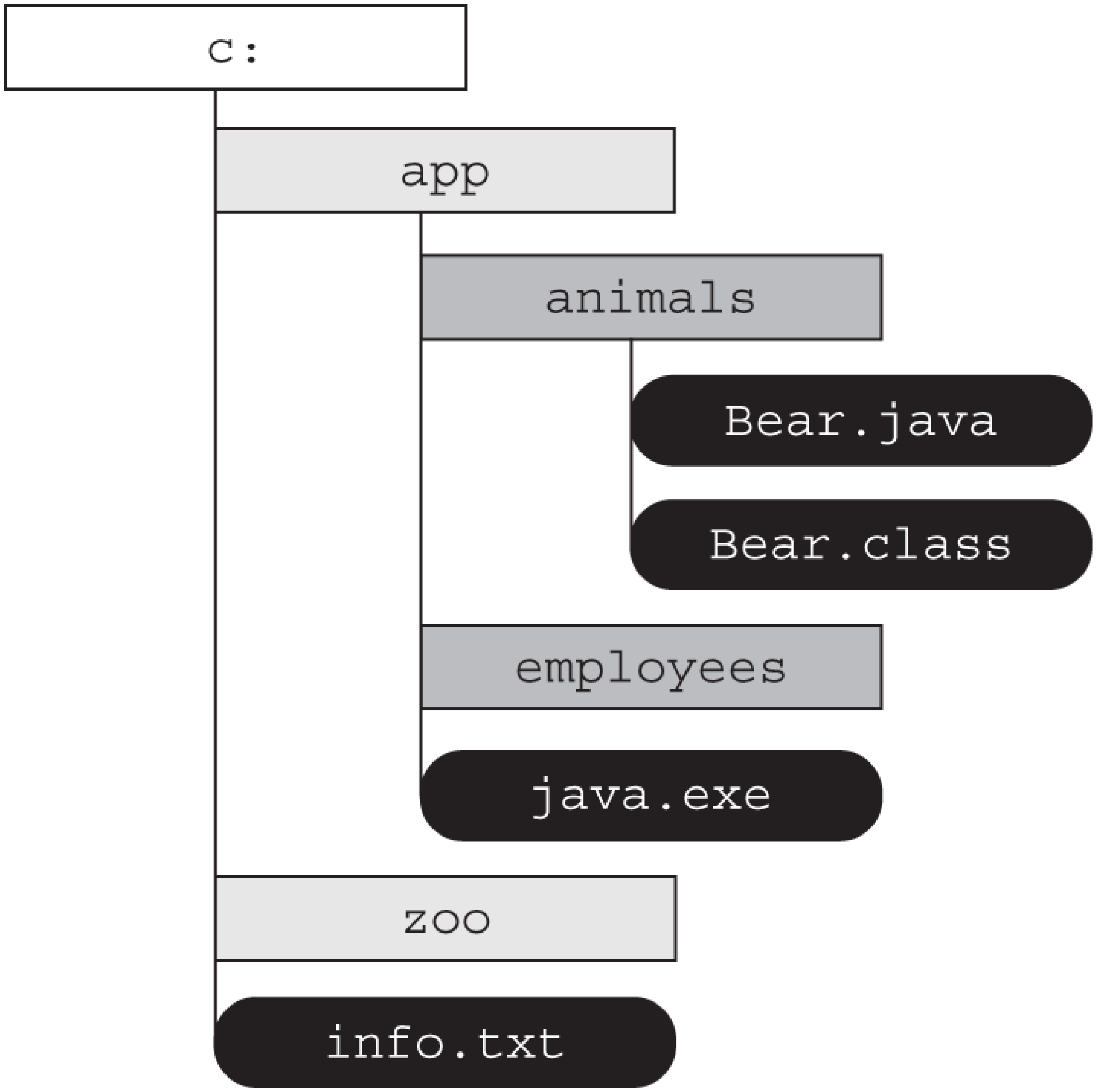

FIGURE 19.1 Directory and file hierarchy

This diagram shows the root directory, `c:` , as containing two directories, `app` and `zoo` , along with the file `info.txt` . Within the `app` directory, there are two more folders, `animals` and `employees` , along with the file `java.exe` . Finally, the `animals` directory contains two files, `Bear.java` and `Bear.class` .

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'll see later when we get to I/O streams, values are often read or written streams using `byte` values and arrays.

Real World Scenario

THE ASCII CHARACTERS

Using a little arithmetic (2^8), we see a byte can be set in one of 256 possible permutations. These 256 values form the alphabet for our computer system to be able to type characters like `a`, `#`, and `7`.

Historically, the 256 characters are referred to as ASCII characters, based on the encoding standard that defined them. Given all of the languages and emojis available today, 256 characters is actually pretty limiting. Many newer standards have been developed that rely on additional bytes to display characters.

Introducing the *File* Class

The first class that we will discuss is one of the most commonly used in the `java.io` API: the `java.io.File` class. The `File` class is used to read information about existing files and directories, list the contents of a directory, and create/delete files and directories.

An instance of a `File` class represents the path to a particular file or directory on the file system. The `File` class cannot read or write data within a file, although it can be passed as a reference to many stream classes to read or write data, as you will see in the next section.

NOTE

Remember, a `File` instance can represent a file or a directory.

Creating a File Object

A `File` object often is initialized with a `String` containing either an absolute or relative path to the file or directory within the file system. The *absolute path* of a file or directory is the full path from the root directory to the file or directory, including all subdirectories that contain the file or directory. Alternatively, the *relative path* of a file or directory is the path from the current working directory to the file or directory. For example, the following is an absolute path to the `stripes.txt` file:

```
/home/tiger/data/stripes.txt
```

The following is a relative path to the same file, assuming the user's current directory is set to `/home/tiger`:

```
data/stripes.txt
```

Different operating systems vary in their format of pathnames. For example, Unix-based systems use the forward slash, `/`, for paths, whereas Windows-based systems use the backslash, `\`, character. That said, many programming languages and file systems support both types of slashes when writing path statements. For convenience, Java offers two options to retrieve the local separator character: a system property and a `static` variable defined in the `File` class. Both of the following examples will output the separator character for the current environment:

```
System.out.println(System.getProperty("file.separator"));
System.out.println(java.io.File.separator);
```

The following code creates a `File` object and determines whether the path it references exists within the file system:

```
var zooFile1 = new File("/home/tiger/data/stripes.txt");
System.out.println(zooFile1.exists());  // true if the file exists
```

This example provides the absolute path to a file and outputs `true` or `false`, depending on whether the file exists. There are three `File` constructors you should know for the exam.

```
public <b>File(String pathname)</b>
public <b>File(File parent, String child)</b>
public <b>File(String parent, String child)</b>
```

The first one creates a `File` from a `String` path. The other two constructors are used to create a `File` from a parent and child path, such as the following:

```
File zooFile2 = new File("/home/tiger", "data/stripes.txt");

File parent = new File("/home/tiger");
File zooFile3 = new File(parent, "data/stripes.txt");
```

In this example, we create two new `File` instances that are equivalent to our earlier `zooFile1` instance. If the `parent` instance is `null`, then it would be skipped, and the method would revert to the single `String` constructor.

The *File* Object vs. the Actual File

When working with an instance of the `File` class, keep in mind that it only represents a path to a file. Unless operated upon, it is not connected to an actual file within the file system.

For example, you can create a new `File` object to test whether a file exists within the system. You can then call various methods to read file properties within the file system. There are also methods to modify the name or location of a file, as well as delete it.

The JVM and underlying file system will read or modify the file using the methods you call on the `File` class. If you try to operate on a file that does not exist or you do not have access to, some `File` methods will throw an exception. Other methods will return `false` if the file does not exist or the operation cannot be performed.

Working with a *File* Object

The `File` class contains numerous useful methods for interacting with files and directories within the file system. We present the most commonly used ones in Table 19.1. Although this table may seem like a lot of methods to learn, many of them are self-explanatory.

TABLE 19.1 Commonly used java.io.File methods

Method Name	Description
`boolean delete()`	Deletes the file or directory and returns `true` only if successful. If this instance denotes a directory, then the directory must be empty in order to be deleted.
`boolean exists()`	Checks if a file exists
`String getAbsolutePath()`	Retrieves the absolute name of the file or directory within the file system
`String getName()`	Retrieves the name of the file or directory.
`String getParent()`	Retrieves the parent directory that the path is contained in or `null` if there is none
`boolean isDirectory()`	Checks if a `File` reference is a directory within the file system
`boolean isFile()`	Checks if a `File` reference is a file within the file system
`long lastModified()`	Returns the number of milliseconds since the epoch (number of milliseconds since 12 a.m. UTC on January 1, 1970) when the file was last modified
`long length()`	Retrieves the number of bytes in the file
`File[] listFiles()`	Retrieves a list of files within a directory

Method Name	Description
`boolean mkdir()`	Creates the directory named by this path
`boolean mkdirs()`	Creates the directory named by this path including any nonexistent parent directories
`boolean renameTo(File dest)`	Renames the file or directory denoted by this path to `dest` and returns `true` only if successful

The following is a sample program that given a file path outputs information about the file or directory, such as whether it exists, what files are contained within it, and so forth:

```
var file = new File("c:\\data\\zoo.txt");
System.out.println("File Exists: " + file.exists());
if (file.exists()) {
   System.out.println("Absolute Path: " + file.getAbsolutePath());
   System.out.println("Is Directory: " + file.isDirectory());
   System.out.println("Parent Path: " + file.getParent());
   if (file.isFile()) {
      System.out.println("Size: " + file.length());
      System.out.println("Last Modified: " + file.lastModified());
   } else {
      for (File subfile : file.listFiles()) {
         System.out.println("   " + subfile.getName());
      }
   }
}
```

If the path provided did not point to a file, it would output the following:

```
File Exists: false
```

If the path provided pointed to a valid file, it would output something similar to the following:

```
File Exists: true
Absolute Path: c:\data\zoo.txt
Is Directory: false
Parent Path: c:\data
Size: 12382
Last Modified: 1606860000000
```

Finally, if the path provided pointed to a valid directory, such as `c:\data`, it would output something similar to the following:

```
File Exists: true
Absolute Path: c:\data
Is Directory: true
Parent Path: c:\
   employees.txt
   zoo.txt
   zoo-backup.txt
```

In these examples, you see that the output of an I/O-based program is completely dependent on the directories and files available at runtime in the underlying file system.

On the exam, you might get paths that look like files but are directories, or vice versa. For example, `/data/zoo.txt` could be a file or a directory, even though it has a file extension. Don't assume it is either unless the question tells you it is!

NOTE

In the previous example, we used two backslashes (`\\`) in the path `String`, such as `c:\\data\\zoo.txt`. When the compiler sees a `\\` inside a `String` expression, it interprets it as a single `\` value.

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 will 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).

Understanding I/O Stream Fundamentals

The contents of a file may be accessed or written via a *stream*, which is a list of data elements presented sequentially. Streams should 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 19.2. The 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 stream and read data one block at a time.

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

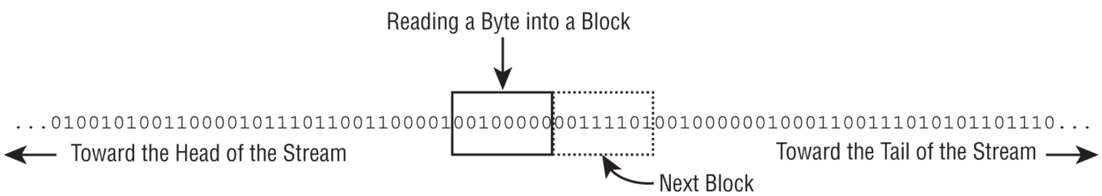

FIGURE 19.2 Visual representation of a stream

Real World Scenario

ALL JAVA I/O STREAMS USE BYTES

Although the `java.io` API is full of 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. The reason higher-level streams exist is for convenience, as well as performance.

For example, writing a file one byte at a time is time-consuming and slow in practice because the round-trip between the Java application and the file system is relatively expensive. By utilizing a `BufferedOutputStream`, the Java application can write a large chunk of bytes at a time, reducing the round-trips and drastically improving performance.

Although 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 stream size will be at runtime, it may be helpful to visualize a stream as being so large that all of the data contained in it could not possibly fit into memory. For example, a 1 terabyte 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 stream allows the application to focus on only a small portion of the overall stream at any given time.

Learning I/O Stream Nomenclature

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

Even if you come across a particular stream on the exam that you do not recognize, often the name of the stream 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 streams. Don't worry if you don't recognize the particular stream class names used in this section or their function; we'll be covering each in detail in the next part of the chapter.

Byte Streams vs. Character Streams

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

Differences between Byte and Character Streams

- Byte streams read/write binary data (`0` s and `1` s) and have class names that end in `InputStream` or `OutputStream`.
- Character 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 streams, such as `FileInputStream` and `FileReader`. The difference between the two classes is based on how the bytes of the stream are read or written.

It is important to remember that even though character 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 will refer to both `InputStream` and `Reader` as *input streams*, and we will refer to both `OutputStream` and `Writer` as *output streams*.

The byte streams are primarily used to work with binary data, such as an image or executable file, while character streams are used to work with text files. Since the byte stream classes can write all types of binary data, including strings, it follows that the character stream classes aren't strictly necessary. There are advantages, though, to using the character stream classes, as they are specifically focused on managing character and string data. 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 a 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 if you come across them on the exam.

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.

For character encoding, just remember that using a character stream is better for working with text data than a byte stream. The character stream classes were created for convenience, and you should certainly take advantage of them when possible.

Input vs. Output Streams

Most `InputStream` stream 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 will discuss these classes later this chapter.

Low-Level vs. High-Level Streams

Another way that you can familiarize yourself with the `java.io` API is by segmenting 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 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 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 stream reader, whereas `BufferedReader` is the high-level stream that takes a `FileReader` as input. Many operations on the high-level stream pass through as operations to the underlying low-level stream, such as `read()` or `close()`. Other operations override or add new functionality to the low-level stream methods. The high-level stream may add new methods, such as `readLine()`, as well as performance enhancements for reading and filtering the low-level data.

High-level streams can take other high-level streams as input. For example, although the following code might seem a little odd at first, the style of wrapping a 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, `FileInputStream` is the low-level stream that interacts directly with the file, which is wrapped by a high-level `BufferedInputStream` to improve performance. Finally, the entire object is wrapped by a 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.

Real World Scenario

USE BUFFERED STREAMS WHEN WORKING WITH FILES

As briefly mentioned, `Buffered` classes read or write data in groups, rather than a single byte or character at a time. The performance gain from using a `Buffered` class to access a low-level file stream cannot be overstated. Unless you are doing something very specialized in your application, you should always wrap a file stream with a `Buffered` class in practice.

One of the reasons that `Buffered` streams tend to perform so well in practice is that many file systems are optimized for sequential disk access. The more sequential bytes you read at a time, the fewer round-trips between the Java process and the file system, improving the access of your application. For example, accessing 1,600 sequential bytes is a lot faster than accessing 1,600 bytes spread across the hard drive.

Stream Base Classes

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

The constructors of high-level 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 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 stream is not the way to do so. As you will see later in this chapter, the data must be copied over, often iteratively. 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.

Review of java.io Class Name Properties

- A class with the word `InputStream` or `OutputStream` in its name is used for reading or writing binary or byte data, respectively.
- A class with the word `Reader` or `Writer` in its name is used for reading or writing character or string data, respectively.
- Most, but not all, input classes have a corresponding output class.
- A low-level stream connects directly with the source of the data.
- A high-level stream is built on top of another stream using wrapping.
- A class with `Buffered` in its name reads or writes data in groups of bytes or characters and often improves performance in sequential file systems.
- With a few exceptions, you only wrap a stream with another stream if they share the same abstract parent.

For the last rule, we'll cover some of those exceptions (like wrapping an `OutputStream` with a `PrintWriter`) later in the chapter.

Table 19.2 lists the abstract base classes that all I/O streams inherited from.

TABLE 19.2 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 19.3 lists the concrete I/O streams that you should be familiar with for the exam. Note that most of the information about each 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 19.3 The java.io concrete 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 an existing `InputStream` in a buffered manner, which improves efficiency and performance
`BufferedOutputStream`	High	Writes byte data to an existing `OutputStream` in a buffered manner, which improves efficiency and performance
`BufferedReader`	High	Reads character data from an existing `Reader` in a buffered manner, which improves efficiency and performance

Class Name	Low/High Level	Description
`BufferedWriter`	High	Writes character data to an existing `Writer` in a buffered manner, which improves efficiency and performance
`ObjectInputStream`	High	Deserializes primitive Java data types and graphs of Java objects from an existing `InputStream`
`ObjectOutputStream`	High	Serializes primitive Java data types and graphs of Java objects to an existing `OutputStream`
`PrintStream`	High	Writes formatted representations of Java objects to a binary stream
`PrintWriter`	High	Writes formatted representations of Java objects to a character stream

Keep Table 19.2 and Table 19.3 handy throughout this chapter. We will discuss these in more detail including examples of each.

Common I/O Stream Operations

While there are a lot of stream classes, many share a lot of the same operations. In this section, we'll review the common methods among various stream classes. In the next section, we'll cover specific stream classes.

Reading and Writing Data

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 the following method to read byte data from a stream:

```
// InputStream and Reader
public int read() throws IOException
```

Likewise, `OutputStream` and `Writer` both define the following method to write a byte to the stream:

```
// OutputStream and Writer
public void write(int b) throws IOException
```

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 a stream. The authors of Java decided to use a larger data type, `int`, so that special values like `-1` would indicate the end of a stream. The output stream classes use `int` as well, to be consistent with the input stream classes.

NOTE

Other stream classes you will learn about in this chapter throw exceptions to denote the end of the stream rather than a special value like `-1`.

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);
   }
}
```

NOTE

Most I/O stream methods declare a checked `IOException`. File or network resources that a stream relies on can disappear at any time, and our programs need be able to readily adapt to these outages.

The byte stream classes also include overloaded methods for reading and writing multiple bytes at a time.

```
// InputStream
public int read(byte[] b) throws IOException
public int read(byte[] b, int offset, int length) throws IOException

// OutputStream
public void write(byte[] b) throws IOException
public void write(byte[] b, int offset, int length) throws IOException
```

The `offset` and `length` are applied to the array itself. For example, an `offset` of `5` and `length` of `3` indicates that the stream should read up to `3` bytes of data and put them into the array starting with position `5`.

There are equivalent methods for the character stream classes that use `char` instead of `byte`.

```
// Reader
public int read(char[] c) throws IOException
public int read(char[] c, int offset, int length) throws IOException

// Writer
public void write(char[] c) throws IOException
public void write(char[] c, int offset, int length) throws IOException
```

We'll see examples of these methods later in the chapter.

Closing the Stream

All I/O streams include a method to release any resources within the stream when it is no longer needed.

```
// All I/O stream classes
public void close() throws IOException
```

Since streams are considered resources, it is imperative that all I/O streams be closed after they are used lest they lead to resource leaks. Since all I/O streams implement `Closeable`, the best way to do this is with a try-with-resources statement, which you saw in Chapter 16, “Exceptions, Assertions, and Localization.”

```
try (var fis = new FileInputStream("zoo-data.txt")) {
   System.out.print(fis.read());
}
```

In many file systems, failing to close a file properly could leave it locked by the operating system such that no other processes could read/write to it until the program is terminated. Throughout this chapter, we will close stream resources using the try-with-resources syntax since this is the preferred way of closing resources in Java. We will also use `var` to shorten the declarations, since these statements can get quite long!

What about if you need to pass a stream to a method? That's fine, but the stream should be closed in the method that created it.

```
public void printData(InputStream is) throws IOException {
   int b;
   while ((b = is.read()) != -1) {
      System.out.print(b);
   }
}

public void readFile(String fileName) throws IOException {
   try (var fis = new FileInputStream(fileName)) {
      printData(fis);
   }
}
```

In this example, the stream is created and closed in the `readFile()` method, with the `printData()` processing the contents.

CLOSING WRAPPED STREAMS

When working with a wrapped stream, you only need to use `close()` on the topmost object. Doing so will close the underlying streams. The following example is valid and will result in three separate `close()` method calls but is unnecessary:

```
try (var fis = new FileOutputStream("zoo-banner.txt"); // Unnecessary
     var bis = new BufferedOutputStream(fis);
     var ois = new ObjectOutputStream(bis)) {
   ois.writeObject("Hello");
}
```

Instead, we can rely on the `ObjectOutputStream` to close the `BufferedOutputStream` and `FileOutputStream`. The following will call only one `close()` method instead of three:

```
try (var ois = new ObjectOutputStream(
      new BufferedOutputStream(
         new FileOutputStream("zoo-banner.txt")))) {
   ois.writeObject("Hello");
}
```

Manipulating Input Streams

All input stream classes include the following methods to manipulate the order in which data is read from a stream:

```
// InputStream and Reader
public boolean <b>markSupported()</b>
public void void mark(int readLimit)
public void reset() throws IOException
public long skip(long n) throws IOException
```

The `mark()` and `reset()` methods return a stream to an earlier position. Before calling either of these methods, you should call the `markSupported()` method, which returns `true` only if `mark()` is supported. The `skip()` method is pretty simple; it basically reads data from the stream and discards the contents.

NOTE

Not all input stream classes support `mark()` and `reset()`. Make sure to call `markSupported()` on the stream before calling these methods or an exception will be thrown at runtime.

---

mark()* and *reset()

Assume that we have an `InputStream` instance whose next values are `LION`. Consider the following code snippet:

```
public void readData(InputStream is) throws IOException {
   System.out.print((char) is.read());     // L
   if (is.markSupported()) {
      is.mark(100);  // Marks up to 100 bytes
      System.out.print((char) is.read());  // I
      System.out.print((char) is.read());  // O
      is.reset();    // Resets stream to position before I
   }
   System.out.print((char) is.read());    // I
   System.out.print((char) is.read());    // O
   System.out.print((char) is.read());    // N
}
```

The code snippet will output `LIOION` if `mark()` is supported, and `LION` otherwise. It's a good practice to organize your `read()` operations so that the stream ends up at the same position regardless of whether `mark()` is supported.

What about the value of `100` we passed to the `mark()` method? This value is called the `readLimit`. It instructs the stream that we expect to call `reset()` after at most `100` bytes. If our program calls `reset()` after reading more than `100` bytes from calling `mark(100)`, then it may throw an exception, depending on the stream class.

NOTE

In actuality, `mark()` and `reset()` are not really putting the data back into the stream but storing the data in a temporary buffer in memory to be read again. Therefore, you should not call the `mark()` operation with too large a value, as this could take up a lot of memory.

---

skip()

Assume that we have an `InputStream` instance whose next values are `TIGERS`. Consider the following code snippet:

```
System.out.print ((char)is.read()); // T
is.skip(2);  // Skips I and G
is.read();   // Reads E but doesn't output it
System.out.print((char)is.read());  // R
System.out.print((char)is.read());  // S
```

This code prints `TRS` at runtime. We skipped two characters, `I` and `G`. We also read `E` but didn't store it anywhere, so it behaved like calling `skip(1)`.

The return parameter of `skip()` tells us how many values were actually skipped. For example, if we are near the end of the stream and call `skip(1000)`, the return value might be `20`, indicating the end of the stream was reached after `20` values were skipped. Using the return value of `skip()` is important if you need to keep track of where you are in a stream and how many bytes have been processed.

Flushing Output Streams

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. In many operating systems, the data may be cached in memory, with a write occurring only after a temporary cache is filled or after some amount of time has passed.

If the data is cached in memory and the application terminates unexpectedly, the data would be lost, because it was never written to the file system. To address this, all output stream classes provide a `flush()` method, which requests that all accumulated data be written immediately to disk.

```
// OutputStream and Writer
public void flush() throws IOException
```

In the following sample, 1,000 characters are written to a file stream. The calls to `flush()` ensure that data is sent to the hard drive at least once every 100 characters. The JVM or operating system is free to send the data more frequently.

```
try (var fos = new FileOutputStream(fileName)) {
   for(int i=0; i<1000; i++) {
      fos.write('a');
      if(i % 100 == 0) {
         fos.flush();
      }
   }
}
```

The `flush()` method helps reduce the amount of data lost if the application terminates unexpectedly. 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.

You also do not need to call the `flush()` method when you have finished writing data, since the `close()` method will automatically do this.

Reviewing Common I/O Stream Methods

Table 19.4 reviews the common stream methods you should know for this chapter. For the `read()` and `write()` methods that take primitive arrays, the method parameter type depends on the stream type. Byte streams ending in `InputStream` / `OutputStream` use `byte[]`, while character streams ending in `Reader` / `Writer` use `char[]`.

TABLE 19.4 Common I/O stream methods

Stream Class	Method Name	Description
All streams	`void close()`	Closes stream and releases resources
All input streams	`int read()`	Reads a single byte or returns `-1` if no bytes were available
`InputStream`	`int read(byte[] b)`	Reads values into a buffer. Returns number of bytes read
`Reader`	`int read(char[] c)`	
`InputStream`	`int read(byte[] b, int offset, int length)`	Reads up to `length` values into a buffer starting from position `offset`. Returns number of bytes read
`Reader`	`int read(char[] c, int offset, int length)`	
All output streams	`void write(int)`	Writes a single byte
`OutputStream`	`void write(byte[] b)`	Writes an array of values into the stream
`Writer`	`void write(char[] c)`	

Stream Class	Method Name	Description
`OutputStream`	`void write(byte[] b, int offset, int length)`	Writes `length` values from an array into the stream, starting with an `offset` index
`Writer`	`void write(char[] c, int offset, int length)`	
All input streams	`boolean markSupported()`	Returns `true` if the stream class supports `mark()`
All input streams	`mark(int readLimit)`	Marks the current position in the stream
All input streams	`void reset()`	Attempts to reset the stream to the `mark()` position
All input streams	`long skip(long n)`	Reads and discards a specified number of characters
All output streams	`void flush()`	Flushes buffered data through the stream

Remember that input and output streams can refer to both byte and character streams throughout this chapter.

Working with I/O Stream Classes

Now that we've reviewed the types of streams and their properties, it's time to jump in and work with concrete I/O stream classes. Some of the techniques for accessing streams may seem a bit new to you, but as you will see, they are similar among different stream classes.

NOTE

The I/O stream classes include numerous overloaded constructors and methods. Hundreds in fact. Don't panic! In this section, we present the most common constructors and methods that you should be familiar with for the exam.

Reading and Writing Binary Data

The first stream classes that we are going to discuss in detail are the most basic file stream classes, `FileInputStream` and `FileOutputStream`. They are used to read bytes from a file or write bytes to a file, respectively. These classes connect to a file using the following constructors:

```
public FileInputStream(File file) throws FileNotFoundException
public FileInputStream(String name) throws FileNotFoundException

public FileOutputStream(File file) throws FileNotFoundException
public FileOutputStream(String name) throws FileNotFoundException
```

NOTE

If you need to append to an existing file, there's a constructor for that. The `FileOutputStream` class includes overloaded constructors that take a `boolean` append flag. When set to `true`, the output stream will append to the end of a file if it already exists. This is useful for writing to the end of log files, for example.

The following code uses `FileInputStream` and `FileOutputStream` to copy a file. It's nearly the same as our previous `copyStream()` method, except that it operates specifically on files.

```
void copyFile(File src, File dest) throws IOException {
   try (var in = new FileInputStream(src);
         var out = new FileOutputStream(dest)) {
      int b;
      while ((b = in.read()) != -1) {
         out.write(b);
      }
   }
}
```

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, since the `append` flag was not sent. The `copy()` method copies one byte at a time until it reads a value of `-1`.

Buffering Binary Data

While our `copyFile()` method is valid, it tends to perform poorly on large files. As discussed earlier, that's because there is a cost associated with each round-trip to the file system. We can easily enhance our implementation using `BufferedInputStream` and `BufferedOutputStream`. As high-level streams, these classes include constructors that take other streams as input.

```
public BufferedInputStream(InputStream in)

public BufferedOutputStream(OutputStream out)
```

WHY USE THE *BUFFERED* CLASSES?

Since the read/write methods that use `byte[]` exist in `InputStream` / `OutputStream`, why use the `Buffered` classes at all? In particular, we could have rewritten our earlier `copyFile()` method to use `byte[]` without introducing the `Buffered` classes. Put simply, the `Buffered` classes contain a number of performance improvements for managing data in memory.

For example, the `BufferedInputStream` class is capable of retrieving and storing in memory more data than you might request with a single `read(byte[])` call. For successive calls to the `read(byte[])` method with a small `byte` array, using the `Buffered` classes would be faster in a wide variety of situations, since the data can be returned directly from memory without going to the file system.

The following shows how to apply these streams:

```
void copyFileWithBuffer(File src, File dest) throws IOException {
   try (var in = new BufferedInputStream(
            new FileInputStream(src));
         var out = new BufferedOutputStream(
            new FileOutputStream(dest))) {
      var buffer = new byte[1024];
      int lengthRead;
      while ((lengthRead = in.read(buffer))> 0) {
         out.write(buffer, 0, lengthRead);
         out.flush();
      }
   }
}
```

Instead of reading the data one byte at a time, we read and write up to `1024` bytes at a time. 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. We also added a `flush()`

command at the end of the loop to ensure data is written to disk between each iteration.

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, then the last read will be only 30 bytes. If we had ignored this return value and instead wrote 1,024 bytes, then 994 bytes from the previous loop would be written to the end of the file.

Real World Scenario

CHOOSING A BUFFER SIZE

Given the way computers organize data, it is often appropriate to choose a buffer size that is a power of 2, such as 1,024. Performance tuning often involves determining what buffer size is most appropriate for your application.

What buffer size should you use? Any buffer size that is a power of 2 from 1,024 to 65,536 is a good choice in practice. Keep in mind, the biggest performance gain you'll see is from moving from nonbuffered access to buffered access. Once you are using a buffered stream, you're less likely to see a huge performance difference between a buffer size of 1,024 and 2,048, for example.

Reading and Writing Character Data

The `FileReader` and `FileWriter` classes, along with their associated buffer classes, are among the most convenient I/O classes because of their built-in support for text data. They include constructors that take the same input as the binary file classes.

```
public FileReader(File file) throws FileNotFoundException
public FileReader(String name) throws FileNotFoundException
```

```
public FileWriter(File file) throws FileNotFoundException
public FileWriter(String name) throws FileNotFoundException
```

The following is an example of using these classes to copy a text file:

```
void copyTextFile(File src, File dest) throws IOException {
   try (var reader = new FileReader(src);
        var writer = new FileWriter(dest)) {
      int b;
      while ((b = reader.read()) != -1) {
         writer.write(b);
      }
   }
}
```

Wait a second, this looks identical to our `copyFile()` method with byte stream! Since we're copying one character at a time, rather than one byte, it is.

The `FileReader` class doesn't contain any new methods you haven't seen before. The `FileWriter` inherits a method from the `Writer` class that allows it to write `String` values.

```
// Writer
public void write(String str) throws IOException
```

For example, the following is supported in `FileWriter` but not `FileOutputStream`:

```
writer.write("Hello World");
```

We'll see even more enhancements for character streams next.

Buffering Character Data

Like we saw with byte streams, Java includes high-level buffered character streams that improve performance. The constructors take existing `Reader` and `Writer` instances as input.

```
public BufferedReader(Reader in)

public BufferedWriter(Writer out)
```

They add two new methods, `readLine()` and `newLine()`, that are particularly useful when working with `String` values.

```
// BufferedReader
public String readLine() throws IOException

// BufferedWriter
public void newLine() throws IOException
```

Putting it all together, the following shows how to copy a file, one line at a time:

```
void copyTextFileWithBuffer(File src, File dest) throws IOException {
   try (var reader = new BufferedReader(new FileReader(src));
         var writer = new BufferedWriter(new FileWriter(dest))) {
      String s;
      while ((s = reader.readLine()) != null) {
         writer.write(s);
         writer.newLine();
      }
   }
}
```

In this example, each loop iteration corresponds to reading and writing a line of a file. Assuming the length of the lines in the file are reasonably sized, this implementation will perform well.

There are some important distinctions between this method and our earlier `copyFileWithBuffer()` method that worked with byte streams. First, instead of a buffer array, we are using a `String` to store the data read during each loop iteration. By storing the data temporarily as a `String`, we can manipulate it as we would any `String` value. For example, we can call `replaceAll()` or `toUpperCase()` to create new values.

Next, we are checking for the end of the stream with a `null` value instead of `-1`. Finally, we are inserting a `newLine()` on every iteration of the loop. This is because `readLine()` strips out the line break character. Without the call to `newLine()`, the copied file would have all of its line breaks removed.

NOTE

In the next chapter, we'll show you how to use NIO.2 to read the lines of a file in a single command. We'll even show you how to process the lines of a file using the functional programming streams that you worked with in Chapter 15.

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 animal's health wouldn't be particularly useful if it had to be entered every time the program runs!

You can certainly use the I/O stream classes you've learned about so far to store text and binary data, but you still have to figure out how to put the data in the stream and then decode it later. There are various file formats like XML and CSV you can standardize to, but oftentimes you have to build the translation yourself.

Luckily, we can use serialization to solve the problem of how to convert objects to/from a stream. *Serialization* is the process of converting an in-memory object to a byte stream. Likewise, *deserialization* is the process of converting from a byte stream into an object. Serialization often involves writing an object to a stored or transmittable format, while deserialization is the reciprocal process.

Figure 19.3 shows a visual representation of serializing and deserializing a `Giraffe` object to and from a `giraffe.txt` file.

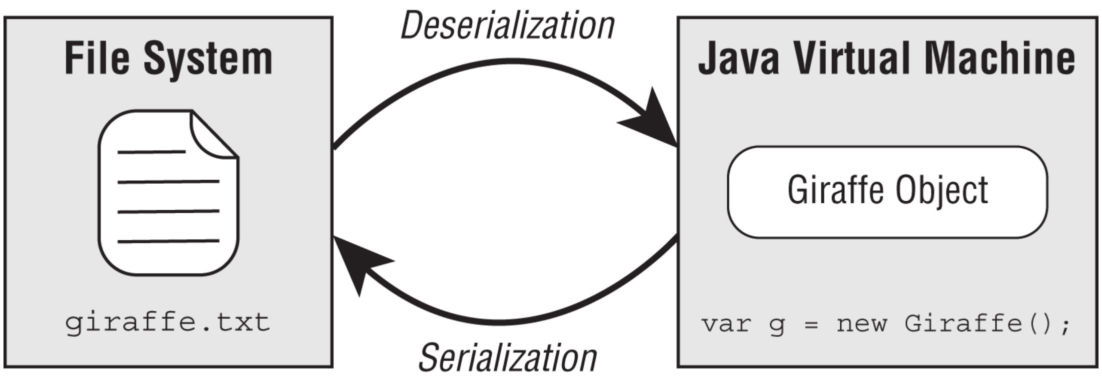

FIGURE 19.3 Serialization process

In this section, we will show you how Java provides built-in mechanisms for serializing and deserializing streams of objects directly to and from disk, respectively.

Applying the *Serializable* Interface

To serialize an object using the I/O API, the object must implement the `java.io.Serializable` interface. The `Serializable` interface is a marker interface, similar to the marker annotations you learned about in Chapter 13, "Annotations." By marker interface, it means the interface does not have any methods. Any class can implement the `Serializable` interface since there are no required methods to implement.

NOTE

Since `Serializable` is a marker interface with no `abstract` members, why not just apply it to every class? Generally speaking, you should only mark data-oriented classes serializable. Process-oriented classes, such as the I/O streams discussed in this chapter, or the `Thread` instances you learned about in Chapter 18, “Concurrency,” are often poor candidates for serialization, as the internal state of those classes tends to be ephemeral or short-lived.

The purpose of using the `Serializable` interface is to inform any process attempting to serialize the object that you have taken the proper steps to make the object serializable. All Java primitives and many of the built-in Java classes that you have worked with throughout this book are `Serializable`. For example, this class can be serialized:

```
import java.io.Serializable;
public class Gorilla implements Serializable {
   private static final long serialVersionUID = 1L;
   private String name;
   private int age;
   private Boolean friendly;
   private transient String favoriteFood;

   // Constructors/Getters/Setters/toString() omitted
}
```

In this example, the `Gorilla` class contains three instance members (`name` , `age` , `friendly`) that will be saved to a stream if the class is serialized. Note that since `Serializable` is not part of the `java.lang` package, it must be imported or referenced with the package name.

What about the `favoriteFood` field that is marked `transient` ? Any field that is marked `transient` will not be saved to a stream when the class is serialized. We'll discuss that in more detail next.

Real World Scenario

MAINTAINING A *SERIALVERSIONUID*

It's a good practice to declare a `static serialVersionUID` variable in every class that implements `Serializable`. The version is stored with each object as part of serialization. Then, every time the class structure changes, this value is updated or incremented.

Perhaps our `Gorilla` class receives a new instance member `Double banana`, or maybe the `age` field is renamed. The idea is a class could have been serialized with an older version of the class and deserialized with a newer version of the class.

The `serialVersionUID` helps inform the JVM that the stored data may not match the new class definition. If an older version of the class is encountered during deserialization, a `java.io.InvalidClassException` may be thrown. Alternatively, some APIs support converting data between versions.

Marking Data *transient*

Oftentimes, the `transient` modifier is used for sensitive data of the class, like a `password`. You'll learn more about this topic in Chapter 22, "Security." There are other objects it does not make sense to serialize, like the state of an in-memory `Thread`. If the object is part of a serializable object, we just mark it `transient` to ignore these select instance members.

What happens to data marked `transient` on deserialization? It reverts to its default Java values, such as `0.0` for `double`, or `null` for an object. We'll see examples of this shortly when we present the object stream classes.

NOTE

Marking `static` fields `transient` has little effect on serialization. Other than the `serialVersionUID`, only the instance members of a class are serialized.

Ensuring a Class Is *Serializable*

Since `Serializable` is a marker interface, you might think there are no rules to using it. Not quite! Any process attempting to serialize an object will throw a `NotSerializableException` if the class does not implement the `Serializable` interface properly.

How to Make a Class Serializable

- The class must be marked `Serializable`.
- Every instance member of the class is serializable, marked `transient`, or has a `null` value at the time of serialization.

Be careful with the second rule. For a class to be serializable, we must apply the second rule recursively. Do you see why the following `Cat` class is not serializable?

```
public class Cat implements Serializable {
   private Tail tail = new Tail();
}


public class Tail implements Serializable {
   private Fur fur = new Fur();
}

public class Fur {}
```

`Cat` contains an instance of `Tail`, and both of those classes are marked `Serializable`, so no problems there. Unfortunately, `Tail` contains an instance of `Fur` that is not marked `Serializable`.

Either of the following changes fixes the problem and allows `Cat` to be serialized:

```
public class Tail implements Serializable {
   private transient Fur fur = new Fur();
}

public class Fur implements Serializable {}
```

We could also make our `tail` or `fur` instance members `null`, although this would make `Cat` serializable only for particular instances, rather than all instances.

Storing Data with *ObjectOutputStream* and *ObjectInputStream*

The `ObjectInputStream` class is used to deserialize an object from a stream, while the `ObjectOutputStream` is used to serialize an object to a stream. They are high-level streams that operate on existing streams.

```
public ObjectInputStream(InputStream in) throws IOException

public ObjectOutputStream(OutputStream out) throws IOException
```

While both of these classes contain a number of methods for built-in data types like primitives, the two methods you need to know for the exam are the ones related to working with objects.

```
// ObjectInputStream
public Object readObject() throws IOException, ClassNotFoundException

// ObjectOutputStream
public void writeObject(Object obj) throws IOException
```

We now provide a sample method that serializes a `List` of `Gorilla` objects to a file.

```
void saveToFile(List<Gorilla> gorillas, File dataFile)
      throws IOException {
   try (var out = new ObjectOutputStream(
           new BufferedOutputStream(
              new FileOutputStream(dataFile)))) {
      for (Gorilla gorilla : gorillas)
         out.writeObject(gorilla);
   }
}
```

Pretty easy, right? Notice we start with a file stream, wrap it in a buffered stream to improve performance, and then wrap that with an object stream. Serializing the data is as simple as passing it to `writeObject()`.

Once the data is stored in a file, we can deserialize it using the following method:

```
List<Gorilla> readFromFile(File dataFile) throws IOException,
      ClassNotFoundException {
   var gorillas = new ArrayList<Gorilla>();
   try (var in = new ObjectInputStream(
           new BufferedInputStream(
              new FileInputStream(dataFile)))) {
      while (true) {
         var object = in.readObject();
         if (object instanceof Gorilla)
            gorillas.add((Gorilla) object);
      }
   } catch (EOFException e) {
      // File end reached
   }
   return gorillas;
}
```

Ah, not as simple as our save method, was it? When calling `readObject()`, `null` and `-1` do not have any special meaning, as someone might have serialized objects with those values. Unlike our earlier techniques for reading methods from an input stream, we need to use an infinite loop to process the data, which throws an `EOFException` when the end of the stream is reached.

NOTE

If your program happens to know the number of objects in the stream, then you can call `readObject()` a fixed number of times, rather than using an infinite loop.

Since the return type of `readObject()` is `Object`, we need an explicit cast to obtain access to our `Gorilla` properties. Notice that `readObject()` declares a checked `ClassNotFoundException` since the class might not be available on deserialization.

The following code snippet shows how to call the serialization methods:

```
var gorillas = new ArrayList<Gorilla>();
gorillas.add(new Gorilla("Grodd", 5, false));
gorillas.add(new Gorilla("Ishmael", 8, true));
File dataFile = new File("gorilla.data");

saveToFile(gorillas, dataFile);
var gorillasFromDisk = readFromFile(dataFile);
System.out.print(gorillasFromDisk);
```

Assuming the `toString()` method was properly overridden in the `Gorilla` class, this prints the following at runtime:

```
[[name=Grodd, age=5, friendly=false],
 [name=Ishmael, age=8, friendly=true]]
```

NOTE

`ObjectInputStream` inherits an `available()` method from `InputStream` that you might think can be used to check for the end of the stream rather than throwing an `EOFException`. Unfortunately, this only tells you the number of blocks that can be read without blocking another thread. In other words, it can return `0` even if there are more bytes to be read.

Understanding the Deserialization Creation Process

For the exam, you need to understand how a deserialized object is created. When you deserialize an object, *the constructor of the serialized class, along with any instance initializers, is not called when the object is created*. Java will call the no-arg constructor of the first nonserializable parent class it can find in the class hierarchy. In our `Gorilla` example, this would just be the no-arg constructor of `Object`.

As we stated earlier, any `static` or `transient` fields are ignored. Values that are not provided will be given their default Java value, such as `null` for `String`, or `0` for `int` values.

Let's take a look at a new `Chimpanzee` class. This time we do list the constructors to illustrate that none of them is used on deserialization.

```
import java.io.Serializable;
public class Chimpanzee implements Serializable {
   private static final long serialVersionUID = 2L;
   private transient String name;
   private transient int age = 10;
   private static char type = 'C';
   { this.age = 14; }

   public Chimpanzee() {
      this.name = "Unknown";
      this.age = 12;
```

```
      this.type = 'Q';
   }

   public Chimpanzee(String name, int age, char type) {
      this.name = name;
      this.age = age;
      this.type = type;
   }

   // Getters/Setters/toString() omitted
}
```

Assuming we rewrite our previous serialization and deserialization methods to process a `Chimpanzee` object instead of a `Gorilla` object, what do you think the following prints?

```
var chimpanzees = new ArrayList<Chimpanzee>();
chimpanzees.add(new Chimpanzee("Ham", 2, 'A'));
chimpanzees.add(new Chimpanzee("Enos", 4, 'B'));
File dataFile = new File("chimpanzee.data");

saveToFile(chimpanzees, dataFile);
var chimpanzeesFromDisk = readFromFile(dataFile);
System.out.println(chimpanzeesFromDisk);
```

Think about it. Go on, we'll wait.

Ready for the answer? Well, for starters, none of the instance members would be serialized to a file. The `name` and `age` variables are both marked `transient` , while the `type` variable is `static` . We purposely accessed the `type` variable using `this` to see whether you were paying attention.

Upon deserialization, none of the constructors in `Chimpanzee` is called. Even the no-arg constructor that sets the values [`name=Unknown,age=12,type=Q`] is ignored. The instance initializer that sets `age` to `14` is also not executed.

In this case, the `name` variable is initialized to `null` since that's the default value for `String` in Java. Likewise, the `age` variable is initialized to `0`. The program prints the following, assuming the `toString()` method is implemented:

```
[[name=null,age=0,type=B],
 [name=null,age=0,type=B]]
```

What about the `type` variable? Since it's `static`, it will actually display whatever value was set last. If the data is serialized and deserialized within the same execution, then it will display `B`, since that was the last `Chimpanzee` we created. On the other hand, if the program performs the deserialization and print on startup, then it will print `C`, since that is the value the class is initialized with.

For the exam, make sure you understand that the constructor and any instance initializations defined in the serialized class are ignored during the deserialization process. Java only calls the constructor of the first non-serializable parent class in the class hierarchy. In Chapter 22, we will go even deeper into serialization and show you how to write methods to customize the serialization process.

Real World Scenario

OTHER SERIALIZATION APIS

In this chapter, we focus on serialization using the I/O streams, such as `ObjectInputStream` and `ObjectOutputStream`. While not part of the exam, you should be aware there are many other (often more popular) APIs for serializing Java objects. For example, there are APIs to serialize data to JSON or encrypted data files.

While these APIs might not use I/O stream classes, many make use of the built-in `Serializable` interface and `transient` modifier. Some of these APIs also include annotations to customize the serialization and deserialization of objects, such as what to do when values are missing or need to be translated.

Printing Data

`PrintStream` and `PrintWriter` are high-level output print streams classes that are useful for writing text data to a stream. We cover these classes together, because they include many of the same methods. Just remember that one operates on an `OutputStream` and the other a `Writer`.

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.

The print stream classes include the following constructors:

```
public PrintStream(OutputStream out)

public PrintWriter(Writer out)
```

For convenience, these classes also include constructors that automatically wrap the print stream around a low-level file stream class, such as `FileOutputStream` and `FileWriter`.

```
public PrintStream(File file) throws FileNotFoundException
public PrintStream(String fileName) throws FileNotFoundException

public PrintWriter(File file) throws FileNotFoundException
public PrintWriter(String fileName) throws FileNotFoundException
```

Furthermore, the `PrintWriter` class even has a constructor that takes an `OutputStream` as input. This is one of the few exceptions in which we can mix a byte and character stream.

```
public PrintWriter(OutputStream out)
```

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`. We'll be covering all three of these objects in the next part of this chapter on user interactions.

Besides the inherited `write()` methods, the print stream classes include numerous methods for writing data including `print()`, `println()`, and `format()`. Unlike the majority of the other streams we've covered, the methods in the print stream classes do not throw any checked exceptions. If they did, you would have been required to catch a checked exception anytime you called `System.out.print()`! The stream classes do provide a method, `checkError()`, that can be used to check for an error after a write.

When working with `String` data, you should use a `Writer`, so our examples in this part of the chapter use `PrintWriter`. Just be aware that many of these examples can be easily rewritten to use a `PrintStream`.

print()

The most basic of the print-based methods is `print()`. The print stream classes include numerous overloaded versions of `print()`, which take everything from primitives and `String` values, to objects. Under the covers, these methods often just perform `String.valueOf()` on the argument and call the underlying stream's `write()` method to add it to the stream. For example, the following sets of `print` / `write` code are equivalent:

```
try (PrintWriter out = new PrintWriter("zoo.log")) {
   out.write(String.valueOf(5));  // Writer method
   out.print(5);                  // PrintWriter method
```

```
      var a = new Chimpanzee();
      out.write(a==null ? "null": a.toString()); // Writer method
      out.print(a);                              // PrintWriter method
   }
```

println()

The next methods available in the `PrintStream` and `PrintWriter` classes are the `println()` methods, which are virtually identical to the `print()` methods, except that they also print a line break after the `String` value is written. These print stream classes also include a no-argument version of `println()`, which just prints a single line break.

The `println()` methods are especially helpful, as the line break character is dependent on the operating system. For example, in some systems a line feed symbol, `\n`, signifies a line break, whereas other systems use a carriage return symbol followed by a line feed symbol, `\r\n`, to signify a line break. Like the `file.separator` property, the `line.separator` value is available from two places, as a Java system property and via a `static` method.

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

format()

In Chapter 16, you learned a lot about formatting messages, dates, and numbers to various locales. Each print stream class includes a `format()` method, which includes an overloaded version that takes a `Locale`.

```
// PrintStream
public PrintStream format(String format, Object args…)
public PrintStream format(Locale loc, String format, Object args…)

// PrintWriter
```

```
public PrintWriter format(String format, Object args…)
public PrintWriter format(Locale loc, String format, Object args…)
```

NOTE

For convenience (as well as to make C developers feel more at home), Java includes `printf()` methods, which function identically to the `format()` methods. The only thing you need to know about these methods is that they are interchangeable with `format()`.

The method parameters are used to construct a formatted `String` in a single method call, rather than via a lot of format and concatenation operations. They return a reference to the instance they are called on so that operations can be chained together.

As an example, the following two `format()` calls print the same text:

```
String name = "Lindsey";
int orderId = 5;

// Both print: Hello Lindsey, order 5 is ready
System.out.format("Hello "+name+", order "+orderId+" is ready");
System.out.format("Hello %s, order %d is ready", name, orderId);
```

In the second `format()` operation, the parameters are inserted and formatted via symbols in the order that they are provided in the vararg. Table 19.5 lists the ones you should know for the exam.

TABLE 19.5 Common print stream `format()` symbols

Symbol	Description
`%s`	Applies to any type, commonly `String` values
`%d`	Applies to integer values like `int` and `long`
`%f`	Applies to floating-point values like `float` and `double`
`%n`	Inserts a line break using the system-dependent line separator

The following example uses all four symbols from Table 19.5:

```
String name = "James";
double score = 90.25;
int total = 100;
System.out.format("%s:%n   Score: %f out of %d", name, score, total);
```

This prints the following:

```
James:
   Score: 90.250000 out of 100
```

Mixing data types may cause exceptions at runtime. For example, the following throws an exception because a floating-point number is used when an integer value is expected:

```
System.out.format("Food: %d tons", 2.0); // IllegalFormatConversionExceptio
```

USING *FORMAT()* WITH *FLAGS*

Besides supporting symbols, Java also supports optional flags between the `%` and the symbol character. In the previous example, the floating-point number was printed as `90.250000`. By default, `%f` displays exactly six digits past the decimal. If you want to display only one digit after the decimal, you could use `%.1f` instead of `%f`. The `format()` method relies on rounding, rather than truncating when shortening numbers. For example, `90.250000` will be displayed as `90.3` (not `90.2`) when passed to `format()` with `%.1f`.

The `format()` method also supports two additional features. You can specify the total length of output by using a number before the decimal symbol. By default, the method will fill the empty space with blank spaces. You can also fill the empty space with zeros, by placing a single zero before the decimal symbol. The following examples use brackets, `[]`, to show the start/end of the formatted value:

```
var pi = 3.14159265359;
System.out.format("[%f]",pi);      // [3.141593]
System.out.format("[%12.8f]",pi);  // [  3.14159265]
System.out.format("[%012f]",pi);   // [00003.141593]
System.out.format("[%12.2f]",pi);  // [        3.14]
System.out.format("[%.3f]",pi);    // [3.142]
```

The `format()` method supports a lot of other symbols and flags. You don't need to know any of them for the exam beyond what we've discussed already.

Sample *PrintWriter* Program

Let's put it altogether. The following sample code shows the `PrintWriter` class in action:

```
File source = new File("zoo.log");
try (var out = new PrintWriter(
```

```
      new BufferedWriter(new FileWriter(source)))) {
      out.print("Today's weather is: ");
      out.println("Sunny");
      out.print("Today's temperature at the zoo is: ");
      out.print(1 / 3.0);
      out.println('C');
      out.format("It has rained %5.2f inches this year %d", 10.2, 2021);
      out.println();
      out.printf("It may rain %s more inches this year", 1.2);
   }
```

After the program runs, `zoo.log` contains the following:

```
Today's weather is: Sunny
Today's temperature at the zoo is: 0.3333333333333333C
It has rained 10.20 inches this year 2021
It may rain 1.2 more inches this year
```

You should pay close attention to the line breaks in the sample. For example, we called `println()` after our `format()`, since `format()` does not automatically insert a line break after the text. One of the most common bugs with printing data in practice is failing to account for line breaks properly.

Review of Stream Classes

We conclude our discussion of stream classes with Figure 19.4.

This diagram shows all of the I/O stream classes that you should be familiar with for the exam, with the exception of the filter streams. `FilterInputStream` and `FilterOutputStream` are high-level superclasses that filter or transform data. They are rarely used directly.

INPUTSTREAMREADER AND *OUTPUTSTREAMWRITER*

Most of the time, you can't wrap byte and character streams with each other, although as we mentioned, there are exceptions. The `InputStreamReader` class wraps an `InputStream` with a `Reader`, while the `OutputStreamWriter` class wraps an `OutputStream` with a `Writer`.

```
try (Reader r = new InputStreamReader(System.in);
     Writer w = new OutputStreamWriter(System.out)) {
}
```

These classes are incredibly convenient and are also unique in that they are the only I/O stream classes to have both `InputStream` / `OutputStream` and `Reader` / `Writer` in their name.

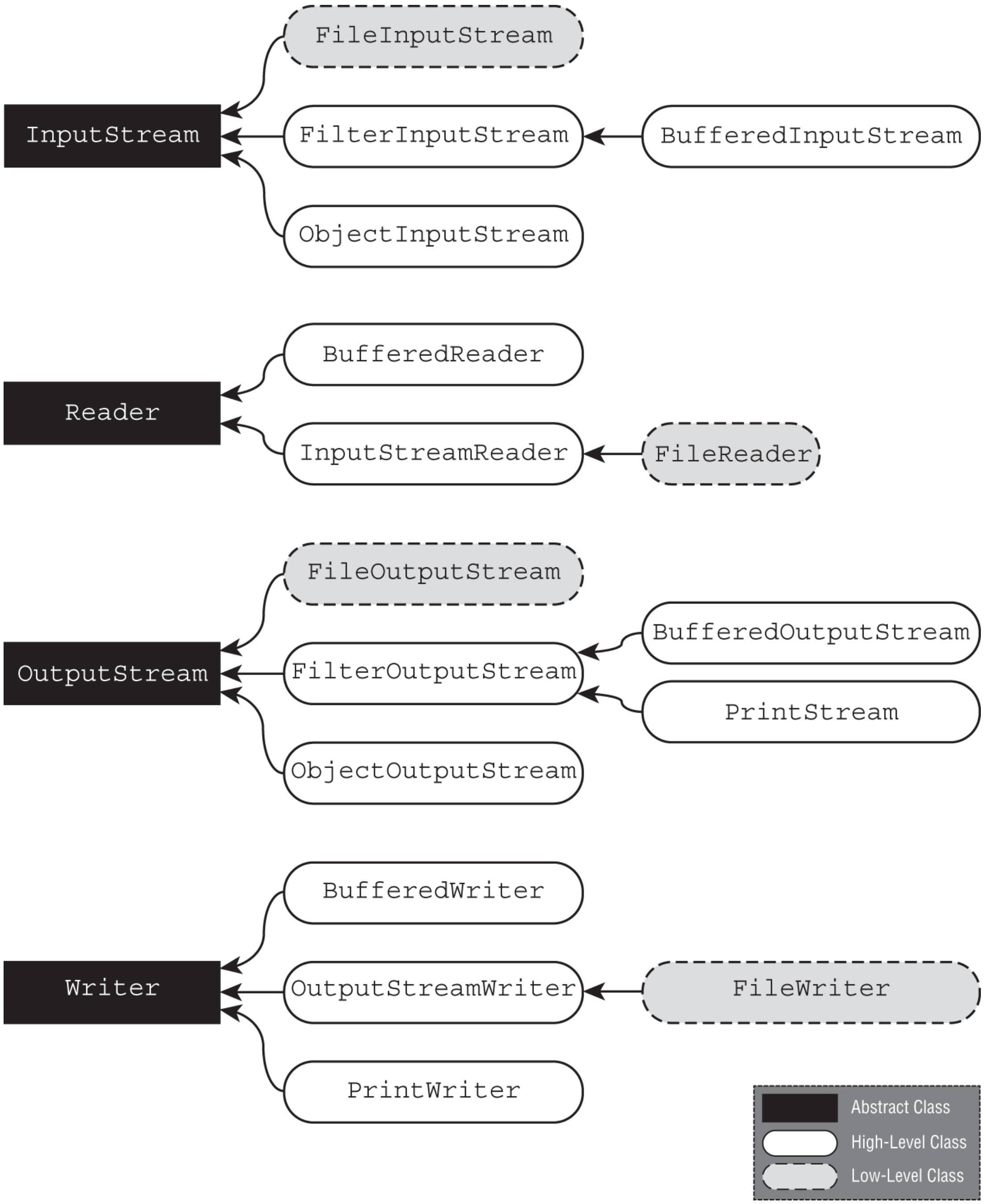

FIGURE 19.4 Diagram of I/O stream classes

Interacting with Users

The `java.io` API includes numerous classes for interacting with the user. For example, you might want to write an application that asks a user to log in and prints a success message. This section contains numerous techniques for handling and responding to user input.

Printing Data to the User

Java includes two `PrintStream` instances for providing information to the user: `System.out` and `System.err`. While `System.out` should be old hat to you, `System.err` might be new to you. The syntax for calling and using `System.err` is the same as `System.out` but is used to report errors to the user in a separate stream from the regular output information.

```
try (var in = new FileInputStream("zoo.txt")) {
   System.out.println("Found file!");
} catch (FileNotFoundException e) {
   System.err.println("File not found!");
}
```

How do they differ in practice? In part, that depends on what is executing the program. For example, if you are running from a command prompt, they will likely print text in the same format. On the other hand, if you are working in an integrated development environment (IDE), they might print the `System.err` text in a different color. Finally, if the code is being run on a server, the `System.err` stream might write to a different log file.

Real World Scenario

USING LOGGING APIS

While `System.out` and `System.err` are incredibly useful for debugging stand-alone or simple applications, they are rarely used in professional software development. Most applications rely on a logging service or API.

While there are many logging APIs available, they tend to share a number of similar attributes. First, you create a `static` logging object in each class. Then, you log a message with an appropriate logging level: `debug()`, `info()`, `warn()`, or `error()`. The `debug()` and `info()` methods are useful as they allow developers to log things that aren't errors but may be useful.

The log levels can be enabled as needed at runtime. For example, a server might only output `warn()` and `error()` to keep the logs clean and easy to read. If an administrator notices a lot of errors, then they might enable `debug()` or `info()` logging to help isolate the problem.

Finally, loggers can be enabled for specific classes or packages. While you may be interested in a `debug()` message for a class you write, you are probably not interested in seeing `debug()` messages for every third-party library you are using.

Reading Input as a Stream

The `System.in` returns an `InputStream` and is used to retrieve text input from the user. It is commonly wrapped with a `BufferedReader` via an `InputStreamReader` to use the `readLine()` method.

```
var reader = new BufferedReader(new InputStreamReader(System.in));
String userInput = reader.readLine();
System.out.println("You entered: " + userInput);
```

When executed, this application first fetches text from the user until the user presses the Enter key. It then outputs the text the user entered to the screen.

Closing *System* Streams

You might have noticed that we never created or closed `System.out`, `System.err`, and `System.in` when we used them. In fact, these are the only I/O streams in the entire chapter that we did not use a try-with-resources block on!

Because these are `static` objects, the `System` streams are shared by the entire application. The JVM creates and opens them for us. They can be used in a try-with-resources statement or by calling `close()`, although *closing them is not recommended*. Closing the `System` streams makes them permanently unavailable for all threads in the remainder of the program.

What do you think the following code snippet prints?

```
try (var out = System.out) {}
System.out.println("Hello");
```

Nothing. It prints nothing. Remember, the methods of `PrintStream` do not throw any checked exceptions and rely on the `checkError()` to report errors, so they fail silently.

What about this example?

```
try (var err = System.err) {}
System.err.println("Hello");
```

This one also prints nothing. Like `System.out`, `System.err` is a `PrintStream`. Even if it did throw an exception, though, we'd have a hard time seeing it since our stream for reporting errors is closed! Closing

`System.err` is a particularly bad idea, since the stack traces from all exceptions will be hidden.

Finally, what do you think this code snippet does?

```
var reader = new BufferedReader(new InputStreamReader(System.in));
try (reader) {}
String data = reader.readLine();  // IOException
```

It prints an exception at runtime. Unlike the `PrintStream` class, most `InputStream` implementations will throw an exception if you try to operate on a closed stream.

Acquiring Input with *Console*

The `java.io.Console` class is specifically designed to handle user interactions. After all, `System.in` and `System.out` are just raw streams, whereas `Console` is a class with numerous methods centered around user input.

The `Console` class is a singleton because it is accessible only from a factory method and only one instance of it is created by the JVM. For example, if you come across code on the exam such as the following, it does not compile, since the constructors are all `private`:

```
Console c = new Console();  // DOES NOT COMPILE
```

The following snippet shows how to obtain a `Console` and use it to retrieve user input:

```
Console console = System.console();
if (console != null) {
   String userInput = console.readLine();
   console.writer().println("You entered: " + userInput);
} else {
```

```
    System.err.println("Console not available");
}
```

NOTE

The `Console` object may not be available, depending on where the code is being called. If it is not available, then `System.console()` returns `null`. It is imperative that you check for a `null` value before attempting to use a `Console` object!

This program first retrieves an instance of the `Console` and verifies that it is available, outputting a message to `System.err` if it is not. If it is available, then it retrieves a line of input from the user and prints the result. As you might have noticed, this example is equivalent to our earlier example of reading user input with `System.in` and `System.out`.

reader()* and *writer()

The `Console` class includes access to two streams for reading and writing data.

```
public Reader reader()
public PrintWriter writer()
```

Accessing these classes is analogous to calling `System.in` and `System.out` directly, although they use character streams rather than byte streams. In this manner, they are more appropriate for handling text data.

format()

For printing data with a `Console`, you can skip calling the `writer().format()` and output the data directly to the stream in a single call.

```
public Console format(String format, Object… args)
```

The `format()` method behaves the same as the `format()` method on the stream classes, formatting and printing a `String` while applying various arguments. They are so alike, in fact, that there's even an equivalent `Console printf()` method that does the same thing as `format()`. We don't want our former C developers to have to learn a new method name!

The following sample code prints information to the user:

```
Console console = System.console();
if (console == null) {
   throw new RuntimeException("Console not available");
} else {
   console.writer().println("Welcome to Our Zoo!");
   console.format("It has %d animals and employs %d people", 391, 25);
   console.writer().println();
   console.printf("The zoo spans %5.1f acres", 128.91);
}
```

Assuming the `Console` is available at runtime, it prints the following:

```
Welcome to Our Zoo!
It has 391 animals and employs 25 people
The zoo spans 128.9 acres.
```

USING *CONSOLE* WITH A *LOCALE*

Unlike the print stream classes, `Console` does not include an overloaded `format()` method that takes a `Locale` instance. Instead, `Console` relies on the system locale. Of course, you could always use a specific `Locale` by retrieving the `Writer` object and passing your own `Locale` instance, such as in the following example:

```
Console console = System.console();
console.writer().format(new Locale("fr", "CA"), "Hello World");
```

---

readLine() and *readPassword()*

The `Console` class includes four methods for retrieving regular text data from the user.

```
public String readLine()
public String readLine(String fmt, Object… args)

public char[] readPassword()
public char[] readPassword(String fmt, Object… args)
```

Like using `System.in` with a `BufferedReader`, the `Console` `readLine()` method reads input until the user presses the Enter key. The overloaded version of `readLine()` displays a formatted message prompt prior to requesting input.

The `readPassword()` methods are similar to the `readLine()` method with two important differences.

- The text the user types is not echoed back and displayed on the screen as they are typing.
- The data is returned as a `char[]` instead of a `String`.

The first feature improves security by not showing the password on the screen if someone happens to be sitting next to you. The second feature involves preventing passwords from entering the `String` pool and will be discussed in Chapter 22.

Reviewing *Console* Methods

The last code sample we present asks the user a series of questions and prints results based on this information using many of various methods we learned in this section:

```
Console console = System.console();
if (console == null) {
   throw new RuntimeException("Console not available");
} else {
   String name = console.readLine("Please enter your name: ");
   console.writer().format("Hi %s", name);
   console.writer().println();

   console.format("What is your address? ");
   String address = console.readLine();

   char[] password = console.readPassword("Enter a password "
      + "between %d and %d characters: ", 5, 10);
   char[] verify = console.readPassword("Enter the password again: ");
   console.printf("Passwords "
      + (Arrays.equals(password, verify) ? "match" : "do not match"));
}
```

Assuming a `Console` is available, the output should resemble the following:

```
Please enter your name: Max
Hi Max
What is your address? Spoonerville
Enter a password between 5 and 10 digits:
Enter the password again:
Passwords match
```

Summary

This chapter is all about using classes in the `java.io` package. We started off showing you how to operate on files and directories using the `java.io.File` class.

We then introduced I/O streams and explained how they are used to read or write large quantities of data. While there are a lot of I/O streams, they differ on some key points.

- Byte vs. character streams
- Input vs. output streams
- Low-level vs. high-level streams

Oftentimes, the name of the I/O stream can tell you a lot about what it does.

We visited many of the I/O stream classes that you will need to know for the exam in increasing order of complexity. A common practice is to start with a low-level resource or file stream and wrap it in a buffered stream to improve performance. You can also apply a high-level stream to manipulate the data, such as an object or print stream. We described what it means to be serializable in Java, and we showed you how to use the object stream classes to persist objects directly to and from disk.

We concluded the chapter by showing you how to read input data from the user, using both the system stream objects and the `Console` class. The `Console` class has many useful features, such as built-in support for passwords and formatting.

Exam Essentials

- **Understand files, directories, and streams.** Files are records that store data within a persistent storage device, such as a hard disk drive, that is available after the application has finished executing. Files are organized within a file system in directories, which in turn may contain other directories. The root directory is the topmost directory in a file system.

- **Be able to use the *java.io.File* class.** A `java.io.File` instance can be created by passing a path `String` to the `File` constructor. The `File` class includes a number of instance methods for retrieving information about both files and directories. It also includes methods to create/delete files and directories, as well as retrieve a list of files within the directory.
- **Distinguish between byte and character streams.** Streams are either byte streams or character streams. Byte streams operate on binary data and have names that end with `Stream`, while character streams operate on text data and have names that end in `Reader` or `Writer`.
- **Distinguish between input and output streams.** Operating on a stream involves either receiving or sending data. The `InputStream` and `Reader` classes are the topmost abstract classes that receive data, while the `OutputStream` and `Writer` classes are the topmost abstract classes that send data. All I/O output streams covered in this chapter have corresponding input streams, with the exception of `PrintStream` and `PrintWriter`. `PrintStream` is also unique in that it is the only `OutputStream` without the word `Output` in its name.
- **Distinguish between low-level and high-level streams.** A low-level stream is one that operates directly on the underlying resource, such as a file or network connection. A high-level stream is one that operates on a low-level or other high-level stream to filter data, convert data, or improve performance.
- **Be able to perform common stream operations.** All streams include a `close()` method, which can be invoked automatically with a try-with-resources statement. Input streams include methods to manipulate the stream including `mark()`, `reset()`, and `skip()`. Remember to call `markSupported()` before using `mark()` and `reset()`, as some streams do not support this operation. Output streams include a `flush()` method to force any buffered data to the underlying resource.
- **Be able to recognize and know how to use various stream classes.** Besides the four top-level abstract classes, you should be familiar with

the file, buffered, print, and object stream classes. You should also know how to wrap a stream with another stream appropriately.

- **Understand how to use Java serialization.** A class is considered serializable if it implements the `java.io.Serializable` interface and contains instance members that are either serializable or marked `transient`. All Java primitives and the `String` class are serializable. The `ObjectInputStream` and `ObjectOutputStream` classes can be used to read and write a `Serializable` object from and to a stream, respectively.
- **Be able to interact with the user.** Be able to interact with the user using the system streams (`System.out`, `System.err`, and `System.in`) as well as the `Console` class. The `Console` class includes special methods for formatting data and retrieving complex input such as passwords.

Review Questions

The answers to the chapter review questions can be found in the Appendix.

1. Which class would be best to use to read a binary file into a Java object?
   A. `ObjectWriter`
   B. `ObjectOutputStream`
   C. `BufferedStream`
   D. `ObjectReader`
   E. `FileReader`
   F. `ObjectInputStream`
   G. None of the above
2. Which of the following are methods available on instances of the `java.io.File` class? (Choose all that apply.)
   A. `mv()`
   B. `createDirectory()`
   C. `mkdirs()`
   D. `move()`
   E. `renameTo()`

F. `copy()`

G. `mkdir()`

3. What is the value of `name` after the instance of `Eagle` created in the `main()` method is serialized and then deserialized?

```
import java.io.Serializable;
class Bird {
   protected transient String name;
   public void setName(String name) { this.name = name; }
   public String getName() { return name; }
   public Bird() {
      this.name = "Matt";
   }
}
public class Eagle extends Bird implements Serializable {
   { this.name = "Olivia"; }
   public Eagle() {
      this.name = "Bridget";
   }
   public static void main(String[] args) {
      var e = new Eagle();
      e.name = "Adeline";
   }
}
```

A. `Adeline`

B. `Matt`

C. `Olivia`

D. `Bridget`

E. `null`

F. The code does not compile.

G. The code compiles but throws an exception at runtime.

4. Which classes will allow the following to compile? (Choose all that apply.)

```
var is = new BufferedInputStream(new FileInputStream("z.txt"));
InputStream wrapper = new __________(is);
try (wrapper) {}
```

A. `BufferedInputStream`

B. `FileInputStream`

C. `BufferedWriter`

D. `ObjectInputStream`

E. `ObjectOutputStream`

F. `BufferedReader`

G. None of the above, as the first line does not compile.

5. Which of the following are true? (Choose all that apply.)

A. `System.console()` will throw an `IOException` if a `Console` is not available.

B. `System.console()` will return `null` if a `Console` is not available.

C. A new `Console` object is created every time `System.console()` is called.

D. `Console` can be used only for writing output, not reading input.

E. `Console` includes a `format()` method to write data to the console's output stream.

F. `Console` includes a `println()` method to write data to the console's output stream.

6. Which statements about closing I/O streams are correct? (Choose all that apply.)

A. `InputStream` and `Reader` instances are the only I/O streams that should be closed after use.

B. `OutputStream` and `Writer` instances are the only I/O streams that should be closed after use.

C. `InputStream` / `OutputStream` and `Reader` / `Writer` all should be closed after use.

D. A traditional `try` statement can be used to close an I/O stream.

E. A try-with-resources can be used to close an I/O stream.

F. None of the above.

7. Assume that `in` is a valid stream whose next bytes are `XYZABC`. What is the result of calling the following method on the stream, using a `count` value of `3`?

```
public static String pullBytes(InputStream in, int count)
      throws IOException {
```

```
        in.mark(count);
        var sb = new StringBuilder();
        for(int i=0; i<count; i++)
           sb.append((char)in.read());
        in.reset();
        in.skip(1);
        sb.append((char)in.read());
        return sb.toString();
     }
```

A. It will return a `String` value of `XYZ`.
B. It will return a `String` value of `XYZA`.
C. It will return a `String` value of `XYZX`.
D. It will return a `String` value of `XYZY`.
E. The code does not compile.
F. The code compiles but throws an exception at runtime.
G. The result cannot be determined with the information given.

8. Which of the following are true statements about serialization in Java? (Choose all that apply.)
   A. Deserialization involves converting data into Java objects.
   B. Serialization involves converting data into Java objects.
   C. All nonthread classes should be marked `Serializable`.
   D. The `Serializable` interface requires implementing `serialize()` and `deserialize()` methods.
   E. `Serializable` is a functional interface.
   F. The `readObject()` method of `ObjectInputStream` may throw a `ClassNotFoundException` even if the return object is not cast to a specific type.
9. Assuming `/` is the root directory within the file system, which of the following are true statements? (Choose all that apply.)
   A. `/home/parrot` is an absolute path.
   B. `/home/parrot` is a directory.
   C. `/home/parrot` is a relative path.
   D. `new File("/home")` will throw an exception if `/home` does not exist.
   E. `new File("/home").delete()` throws an exception if `/home` does not exist.

10. What are the requirements for a class that you want to serialize to a stream? (Choose all that apply.)
   A. The class must be marked `final`.
   B. The class must extend the `Serializable` class.
   C. The class must declare a `static serialVersionUID` variable.
   D. All `static` members of the class must be marked `transient`.
   E. The class must implement the `Serializable` interface.
   F. All instance members of the class must be serializable or marked `transient`.
11. Given a directory `/storage` full of multiple files and directories, what is the result of executing the `deleteTree("/storage")` method on it?

```
public static void deleteTree(File file) {
   if(!file.isFile())                       // f1
      for(File entry: file.listFiles())  // f2
         deleteTree(entry);
   else file.delete();
}
```

   A. It will delete only the empty directories.
   B. It will delete the entire directory tree including the `/storage` directory itself.
   C. It will delete all files within the directory tree.
   D. The code will not compile because of line `f1`.
   E. The code will not compile because of line `f2`.
   F. None of the above
12. What are possible results of executing the following code? (Choose all that apply.)

```
public static void main(String[] args) {
   String line;
   var c = System.console();
   Writer w = c.writer();
   try (w) {
      if ((line = c.readLine("Enter your name: ")) != null)
         w.append(line);
      w.flush();
```

```
        }
    }
```

A. The code runs but nothing is printed.

B. The code prints what was entered by the user.

C. An `ArrayIndexOutOfBoundsException` is thrown.

D. A `NullPointerException` is thrown.

E. None of the above, as the code does not compile

13. Suppose that the absolute path `/weather/winter/snow.dat` represents a file that exists within the file system. Which of the following lines of code creates an object that represents the file? (Choose all that apply.)

A.
```
new File("/weather", "winter", "snow.dat")
```

B.
```
new File("/weather/winter/snow.dat")
```

C.
```
new File("/weather/winter", new File("snow.dat"))
```

D.
```
new File("weather", "/winter/snow.dat")
```

E.
```
new File(new File("/weather/winter"), "snow.dat")
```

F. None of the above

14. Which of the following are built-in streams in Java? (Choose all that apply.)

A. `System.err`

B. `System.error`

C. `System.in`

D. `System.input`

E. `System.out`

F. `System.output`

15. Which of the following are not `java.io` classes? (Choose all that apply.)

    A. `BufferedReader`

    B. `BufferedWriter`

    C. `FileReader`

    D. `FileWriter`

    E. `PrintReader`

    F. `PrintWriter`

16. Assuming `zoo-data.txt` exists and is not empty, what statements about the following method are correct? (Choose all that apply.)

```
private void echo() throws IOException {
   var o = new FileWriter("new-zoo.txt");
   try (var f = new FileReader("zoo-data.txt");
         var b = new BufferedReader(f); o) {
      o.write(b.readLine());
   }
   o.write("");
}
```

    A. When run, the method creates a new file with one line of text in it.

    B. When run, the method creates a new file with two lines of text in it.

    C. When run, the method creates a new file with the same number of lines as the original file.

    D. The method compiles but will produce an exception at runtime.

    E. The method does not compile.

    F. The method uses byte stream classes.

17. Assume `reader` is a valid stream that supports `mark()` and whose next characters are `PEACOCKS`. What is the expected output of the following code snippet?

```
var sb = new StringBuilder();
sb.append((char)reader.read());
reader.mark(10);
for(int i=0; i<2; i++) {
   sb.append((char)reader.read());
   reader.skip(2);
```

```
}
reader.reset();
reader.skip(0);
sb.append((char)reader.read());
System.out.println(sb.toString());
```

A. `PEAE`
B. `PEOA`
C. `PEOE`
D. `PEOS`
E. The code does not compile.
F. The code compiles but throws an exception at runtime.
G. The result cannot be determined with the information given.

18. Suppose that you need to write data that consists of `int`, `double`, `boolean`, and `String` values to a file that maintains the data types of the original data. You also want the data to be performant on large files. Which three `java.io` stream classes can be chained together to best achieve this result? (Choose three.)

A. `FileWriter`
B. `FileOutputStream`
C. `BufferedOutputStream`
D. `ObjectOutputStream`
E. `DirectoryOutputStream`
F. `PrintWriter`
G. `PrintStream`

19. Given the following method, which statements are correct? (Choose all that apply.)

```
public void copyFile(File file1, File file2) throws Exception {
   var reader = new InputStreamReader(
      new FileInputStream(file1));
   try (var writer = new FileWriter(file2)) {
      char[] buffer = new char[10];
      while(reader.read(buffer) != -1) {
         writer.write(buffer);
         // n1
      }
```

```
        }
    }
```

A. The code does not compile because `reader` is not a `Buffered` stream.
B. The code does not compile because `writer` is not a `Buffered` stream.
C. The code compiles and correctly copies the data between some files.
D. The code compiles and correctly copies the data between all files.
E. If we check `file2` on line `n1` within the file system after five iterations of the `while` loop, it may be empty.
F. If we check `file2` on line `n1` within the file system after five iterations, it will contain exactly 50 characters.
G. This method contains a resource leak.

20. Which values when inserted into the blank independently would allow the code to compile? (Choose all that apply.)

```
Console console = System.console();
String color = console.readLine("Favorite color? ");
console.__________("Your favorite color is %s", color);
```

A. `reader().print`
B. `reader().println`
C. `format`
D. `writer().print`
E. `writer().println`
F. None of the above

21. What are some reasons to use a character stream, such as `Reader` / `Writer`, over a byte stream, such as `InputStream` / `OutputStream`? (Choose all that apply.)

A. More convenient code syntax when working with `String` data
B. Improved performance
C. Automatic character encoding
D. Built-in serialization and deserialization

E. Character streams are high-level streams.

F. Multithreading support

22. Which of the following fields will be `null` after an instance of the class created on line 15 is serialized and then deserialized using `ObjectOutputStream` and `ObjectInputStream`? (Choose all that apply.)

```
1:  import java.io.Serializable;
2:  import java.util.List;
3:  public class Zebra implements Serializable {
4:     private transient String name = "George";
5:     private static String birthPlace = "Africa";
6:     private transient Integer age;
7:     List<Zebra> friends = new java.util.ArrayList<>();
8:     private Object stripes = new Object();
9:     { age = 10;}
10:    public Zebra() {
11:       this.name = "Sophia";
12:    }
13:    static Zebra writeAndRead(Zebra z) {
14:       // Implementation omitted
15:    }
16:    public static void main(String[] args) {
17:       var zebra = new Zebra();
18:       zebra = writeAndRead(zebra);
19:    } }
```

A. `name`

B. `stripes`

C. `age`

D. `friends`

E. `birthPlace`

F. The code does not compile.

G. The code compiles but throws an exception at runtime.