

Java I/O fundamentals

Exam objectives covered in this chapter	What you need to know
[7.1] Read and write data from the console	How to access the unique console related to a JVM How to read from and write data to the console
[7.2] Use streams to read from and write to files by using classes in the <code>java.io</code> package including <code>BufferedReader</code> , <code>BufferedWriter</code> , <code>File</code> , <code>FileReader</code> , <code>FileWriter</code> , <code>DataInputStream</code> , <code>DataOutputStream</code> , <code>ObjectOutputStream</code> , <code>ObjectInputStream</code> , and <code>PrintWriter</code>	The definitions of byte I/O streams and character streams, and their similarities and differences How to read and write raw bytes, primitive data, strings, and objects to files by using one class or a combination of classes

The Java I/O API is powerful and flexible. It enables you to read and write multiple types of data: raw bytes, characters, and even objects. You can read data from multiple and diverse data sources such as files, network sockets, and memory arrays, and write them to multiple data destinations. Java I/O also provides the flexibility of reading or writing buffered and unbuffered data. You can also chain multiple input and output sources. You can use the same methods to read from an input resource such as a file, a console, or a network connection.

The Java I/O API is huge and could be a textbook on its own. Coverage in this chapter is limited to the Java I/O exam topics for the OCP Java SE 7 Programmer II exam. This chapter assumes no prior knowledge of Java I/O.



NOTE Introduction to I/O concepts is covered in warm-up section 7.1. If you're good with the basics of I/O, please skip this section and move to section 7.2.

This chapter doesn't cover the new file I/O (NIO.2). NIO.2 is covered in the next chapter. This chapter covers

- Introduction to Java I/O
- Types of streams: byte streams and character streams
- Buffering data for faster reading and writing
- Reading and writing bytes, primitives, and objects using data streams and object streams
- Chaining I/O streams
- Writing formatted data to character streams
- Reading data from and writing it to the console

Let's start with the basics of Java I/O in the next section.

7.1 *Introducing Java I/O: WARM-UP*

Java I/O lets you read your files, data, photos, and videos from multiple sources and write them to several destinations. You can use it to prepare a formatted report that you can send to a printer. You can copy your files, create directory structures, delete them, and do much more.

Java I/O includes a lot of classes and can be intimidating to work with. You must get the hang of its basics, so you can categorize and work with a group of classes and methods. Java abstracts all its data sources and data destinations as *streams*.

7.1.1 *Understanding streams*

A question that I'm asked quite often is: What is a stream—the data, data source, or data destination? And it's a fair question. A stream is a sequence of data. The Java I/O stream is an abstraction of a data source *or* a data destination. It represents an object that can produce data or receive data. An input stream is used to read data from a data source. An output stream is used to write data to a data destination.

Just as a stream of water represents a continuous flow of water, a Java stream produces or consumes a continuous flow of data. I/O streams are used to move data from a data source to a Java program, and from a Java program to a data destination.

- An *input stream* enables you to *read* data from a data source *to* a Java application.
- An *output stream* enables you to *write* data from a Java application to a data destination.

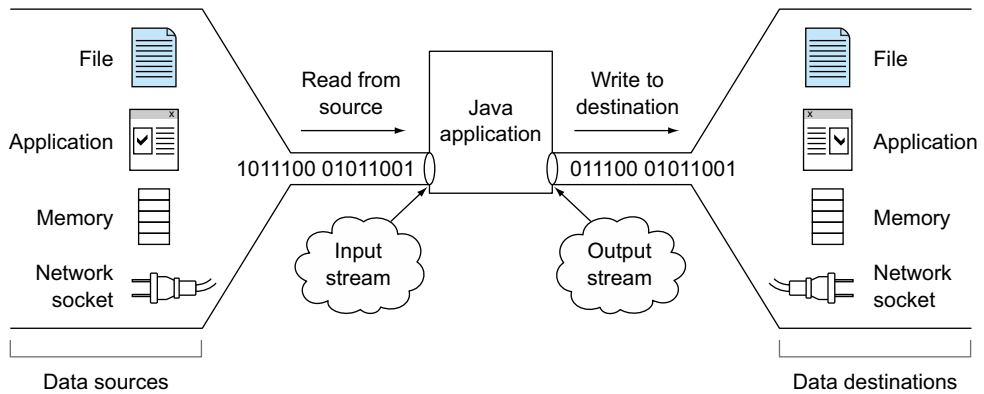


Figure 7.1 Flavors of I/O streams: input streams and output streams. An input stream enables a Java program to read data from a data source. An output stream enables a Java program to write data to a data destination.

A Java program can read from multiple data sources and destinations, like a file on your storage device, another application, an array in memory, a network connection, and others. Figure 7.1 shows the flavors of I/O streams—input streams and output streams—and how they’re connected to data sources and destinations.



NOTE The exam limits the use of streams to reading from and writing to files. So the rest of the chapter covers reading and writing of data in multiple formats from and to only *files*.

The exam covers multiple classes that can read and write bytes, characters, and objects. Before we dive into the details of working with these classes, the next section includes a quick overview of the main types of I/O streams: byte streams and character streams.

7.1.2 Understanding multiple flavors of data

Java I/O uses separate classes to read and write different types of data. The data is broadly divided into byte data and character data. The *byte streams* read and write byte data. To read and write characters and text data, Java I/O uses *readers* and *writers*, referred to as *character streams*. Byte streams can be used to read and write practically all kinds of data including zip files, image or video files, text or PDF files, and others. Character streams are used to read and write character data. Figure 7.2 shows these streams.

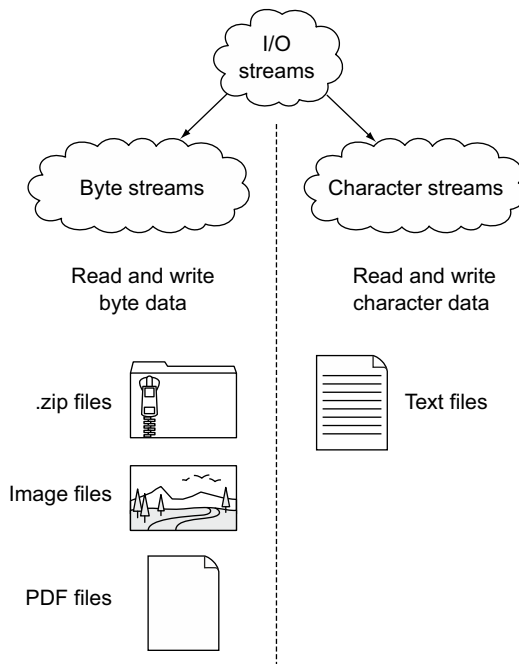


Figure 7.2 Main categories of Java I/O streams: byte streams and character streams

Creating files and reading or writing byte data from them or to them

Though you can read and write practically all byte data by using byte streams, you might need the assistance of specific classes to interpret the read data or written data in a particular format. For example, to create a PDF file, interpret its data, and write data to it, you might need to use an API by Adobe or a third party.

Going back to the basics, a computer system reads and writes all data as binary data (0s and 1s, referred to as *bits*) from all sources and to all destinations. A byte stream reads and write *bytes* (8 bits) to and from data sources. All the other I/O classes build on the byte streams, adding more functionality. For example, character streams take into account Unicode characters and the user's charset. Instead of reading or writing 8-bit data from or to a file, character streams define methods to read or write 16-bit data. But behind the scenes they use byte streams to get their work done. Figure 7.3 shows the input and output streams for reading and writing byte and character data. As mentioned earlier, the exam covers writing data to and reading data from only files. A Java application uses `FileOutputStream` to write bytes to a file and `FileInputStream` to read bytes from a file. Similarly, it uses `FileReader` to read Unicode text from a file and `FileWriter` to write Unicode text to a file.

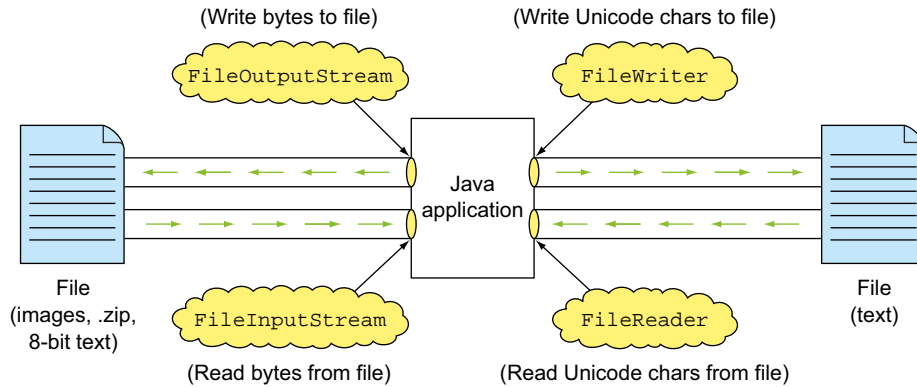


Figure 7.3 A Java application uses `FileInputStream` and `FileOutputStream` to read and write byte data from and to files, respectively. It uses `FileReader` and `FileWriter` to read and write Unicode text from and to files.

Invoking the I/O methods to read and write (very small) basic units of data (byte or character) isn't efficient. *Buffered streams* add functionality to other streams by buffering data. Buffered streams read from a *buffer* (also referred to as memory or an internal array), and the native API is called when it's empty. Similarly, they write data to a buffer and flush it to the underlying output stream when it's full. A buffered stream buffers input/output from another input/output stream. Figure 7.4 shows

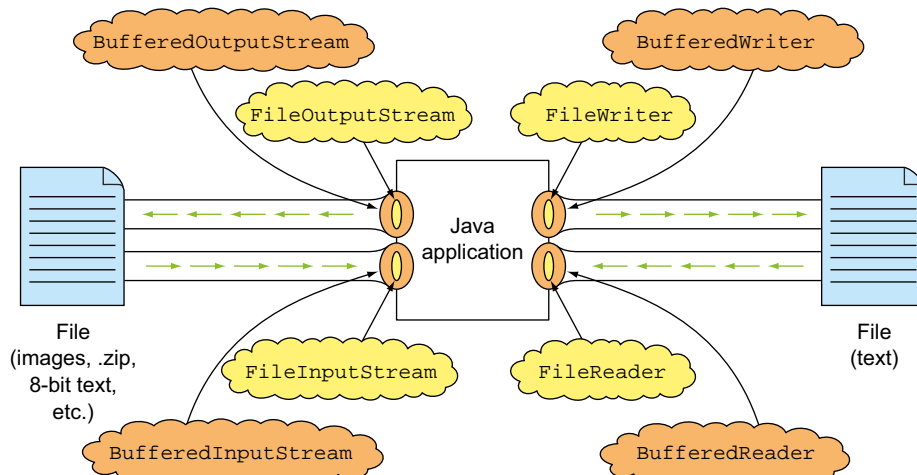


Figure 7.4 A `BufferedInputStream` reads and buffers data from an `InputStream` like `FileInputStream`. `BufferedOutputStream` buffers data before writing it to an `OutputStream` like `FileOutputStream`. Similarly, a `BufferedReader` buffers input from a `Reader` like `FileReader`. A `BufferedWriter` buffers data before sending it off to a `Writer` like `FileWriter`.

how `BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, and `BufferedWriter` buffer data from byte and character streams.

As shown in figure 7.4, a Java application communicates with an object of a buffered stream (like `BufferedInputStream`), which interacts with the underlying `InputStream` (like `FileInputStream`). Notice how figure 7.4 shows that the buffered stream encloses within it an object of another stream. This pictorial representation will be handy when you instantiate buffered streams in the next sections.



EXAM TIP To instantiate a `BufferedInputStream`, you need to pass it an instance of another `InputStream`, like `FileInputStream`. Constructor wrapping (passing instances to instantiate instances) can become complex in Java I/O.

All these buffered classes are specialized *filter* classes for the underlying stream. They modify the way their underlying streams behave, by buffering data. The filter classes such as `BufferedInputStream` and `BufferedOutputStream` are *decorator* classes. They add functionality to the existing base classes.

Other `InputStream` and `OutputStream` subclasses you need to know for the exam are classes `DataInputStream` and `DataOutputStream`. These classes define methods for reading and writing primitive values and strings. For example, a method to read int values, say `readInt()`, will read 4 bytes from an input stream and return an int value. Similarly, a method to write an int value, say `writeInt()`, will write 4 bytes of data. Figure 7.5 shows how you can read and write primitive values and strings from a file by using `DataInputStream` and `DataOutputStream`. As I mentioned previously, because the exam covers reading from and writing to files only, the images show the data source and destination as files.

You can also read and write objects using object streams. Only objects that support serialization (must implement marker interface `Serializable`) can be read from and

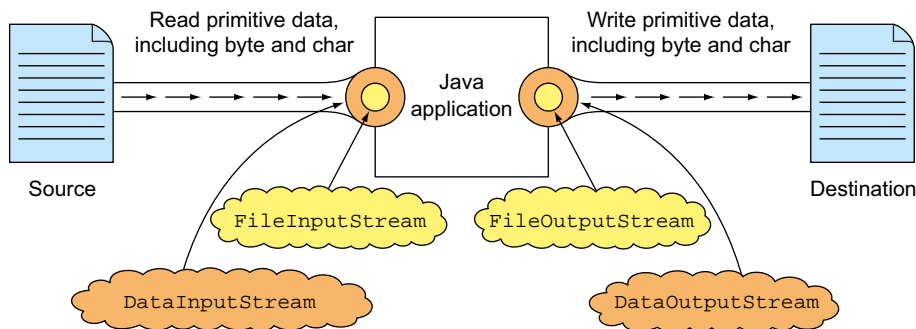


Figure 7.5 A `DataInputStream` can read primitive values (including byte and char) and strings from a file by using `FileInputStream`. A `DataOutputStream` can write primitive values and strings to a file using `FileOutputStream`.

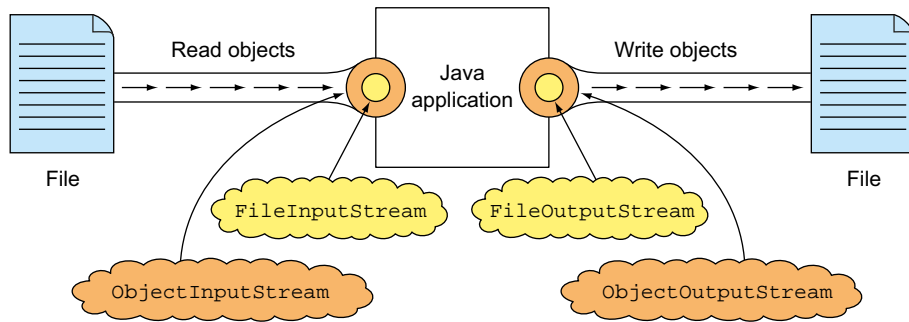


Figure 7.6 An `ObjectInputStream` reads an object from a file by using a `FileInputStream`. An `ObjectOutputStream` writes an object to a file by using a `FileOutputStream`.

written to a data source. The object stream classes are `ObjectInputStream` and `ObjectOutputStream`. Figure 7.6 shows this arrangement.

To read and write data to a physical file, you need instances of `java.io.File`. All classes that read from or write to a file either accept an instance of `File` or instantiate one themselves using a path and filename. Before using streams to read data from and write data to files, let's work with `File` in the next section.



NOTE Java version 7 has introduced a new interface that offers the existing functionality of class `File`, addresses its existing issues, and offers additional functionality: `java.nio.file.Path`. Because `File` is on the exam, we'll continue to use class `File` for all examples in this chapter. `Path` is covered in chapter 8.

7.2 Working with class `java.io.File`



[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

To read from and write to files by using `java.io` classes, you'll need to work with class `File`.

`File` is an abstract representation of a path to a file or a directory. It can be *absolute* or *relative*. An absolute path can be resolved without any further information. A relative path is interpreted from another path. The name of the class `File` is rather misleading because it might *not* be associated with an actual file (or directory) on a system. You can create objects of class `File` to store information about the files or directories on your system. You can use an object of class `File` to create a new file or directory,

delete it, or inquire about or modify its attributes. You can read the contents of a file or write to it by using byte and character I/O classes.



EXAM TIP A `File` instance can refer to either a file or a directory. But it might not be necessarily associated with an actual file or directory.

To read from or write to a file, you'll need to instantiate `File` objects, which can be used by I/O classes such as `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter`. Because a `File` instance can refer to either a file or a directory, you might want to verify that you aren't writing your valuable data to a directory. Let's work with an example to instantiate `File` objects and check whether they represent files or directories. The first step is to examine the `File` constructors:

```
File(String pathname)
File(File parent, String child)
File(String parent, String child)
```

Creates new `File` instance by converting given pathname string into abstract pathname

Creates new `File` instance from parent abstract pathname and child pathname string

Creates new `File` instance from parent pathname string and child pathname string

The next section includes an example to instantiate `File` objects (using the preceding `File` constructors) and check whether they're files or directories.

7.2.1 Instantiating and querying `File` instances

Say you're given the directory in the following listing (`C:\temp`, with its child files and directories). Let's count its (first-level) subdirectories and files. As shown in the following listing, the code should print one directory and two files.

Listing 7.1 Working with `File` instances

```
import java.io.*;
public class CountDirFiles {
    public static void main(String... args) {
        countDirFiles(new File("c:\\temp"));
    }
    public static void countDirFiles(File dir) {
        if (dir.isDirectory()) {
            int fileCount = 0;
            int dirCount = 0;
            String[] list = dir.list();
            File item = null;
            for (String listItem : list) {
                item = new File(dir, listItem);
                if (item.isFile())
                    ++fileCount;
                else if (item.isDirectory())
                    ++dirCount;
            }
        }
    }
}
```

Creates new `File` object

Creates `File` instance by converting given pathname string into abstract pathname

Creates new `File` instance from parent abstract pathname and child pathname string

Creates new `File` instance from parent pathname string and child pathname string

If a directory, proceed

Retrieves files and subdirectories

Iterates files and subdirectories

Increments `fileCount`, for files

Increments `dirCount`, for directories

```

        System.out.println ("File(s):"+ fileCount);
        System.out.println ("Dir(s):"+ dirCount);
    }
    else {
        throw new IllegalArgumentException("Not a directory");
    }
}
}

```

Throws exception if method argument isn't directory

The output of this code is as follows:

```

File(s):2
Dir(s):1

```

In this example, I created an object of class `File` and passed it to method `countDirFiles()`. This method checks whether the `File` object represents a directory, using `dir.isDirectory()`. If `dir.isDirectory()` returns `true`, the method retrieves a list of the names of its child files and directories, using method `list()`. Note that the method returns a list of the names as an array of `String`. To determine whether an individual array item is a file or directory, we need to first create a `File` instance using directory `dir` and the item itself. This is accomplished by using the following line of code:

```
item = new File(dir, listItem);
```

You can query a `File` instance to determine whether it represents a file or a directory, and increment the variables used to store the respective count, as follows:

```

if (item.isFile())
    ++fileCount;
else if (item.isDirectory())
    ++dirCount;

```



EXAM TIP You can create a `File` instance that represents a nonexistent file on your file system. And you can even invoke methods like `isFile()` without getting an exception.

Revisit the following line of code from listing 7.1:

```
item = new File(dir, listItem)
```

This constructor creates a new file instance from a parent abstract pathname and a child pathname string, using the following constructor defined in class `File`:

```
File(File parent, String child)
```

What happens when you replace `new File(dir, listItem)` from listing 7.1 with this constructor: `new File(listItem)`? The code now prints 0 for both the file and directory count.

Why? If you create a `File` object, *without* specifying its *complete path*, it assumes that the file that you're referring to exists in the current directory (the one that contains

your .class file). The variable `listItem` contains just the filename, *without* its complete path. In the absence of the complete path name, the variable `item` refers to a *nonexisting file on your system* (because it's looking in the wrong directory). Because the `item` variable *doesn't* represent a file or directory on your system, methods `isFile()` and `isDirectory()` return `false`, and neither of the variable values, `dirCount` or `fileCount`, is modified.



EXAM TIP The objects of class `File` are immutable; the pathname represented by a `File` object can't be changed.

7.2.2 Creating new files and directories on your physical device

Creating an object of class `File` *won't* create a real file or directory on your system. To create a new file on your system, use method `createNewFile()`, and to create new directories, use methods `mkdir()` or `mkdirs()`, as listed in table 7.1.

Table 7.1 Methods used to create new physical files and directories

File method	Exception thrown	Description (from Java API documentation)
<code>boolean createNewFile()</code>	<code>IOException</code>	Atomically creates a new, empty file named by this abstract pathname if and only if a file with this name doesn't yet exist.
<code>boolean mkdir()</code>	<code>SecurityException</code>	Creates the directory named by this abstract pathname.
<code>boolean mkdirs()</code>	<code>SecurityException</code>	Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails, it may have succeeded in creating some of the necessary parent directories.

Here's an example that creates new files and directories:

```
import java.io.*;
public class CreateFileAndDirs {
    public static void main(String... args) throws IOException {
        File dirs = new File("\\usr\\code\\java");
        System.out.println(dirs.mkdirs());
        File file = new File(dirs, "MyText.txt");
        if (!file.exists())
            System.out.println(file.createNewFile());
    }
}
```

Try to create file →

← Creates multiple directories in root directory

← If file doesn't exist

This code creates a directory tree `usr\code\java` and a file (`MyText.txt`) on our system:



But the results might vary on your system (you might not have the access permission to create files or directories). Be careful when you create objects of class `File` on your system. Even though the following code creates an object of class `File`, with a weird name, it compiles successfully:

```
File f = new File ("*&^%$.yu");
```

The preceding code doesn't throw any exception because it *won't* create a file on the physical storage on your system. It creates only an object of class `File`, which as mentioned before, might not necessarily represent a file or directory. Because creation of files and directories are system-specific operations, creation of a file with the name `*&^%$.yu` might succeed on one system, but fail on another. For example, on a system running Windows, calling method `createNewFile()` on this object will throw the following exception at runtime:

```
Exception in thread "main" java.io.IOException: The filename, directory
name, or volume label syntax is incorrect
    at java.io.WinNTFileSystem.createFileExclusively(Native Method)
    at java.io.File.createNewFile(File.java:883)
```

With the basic knowledge of using a `File` object, let's deep-dive into how to read and write the contents of a file by using byte streams.

7.3 Using byte stream I/O



[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Byte streams work with reading and writing byte data. They're broadly categorized as input streams and output streams. All input streams extend the base abstract class `java.io.InputStream`, and all output streams extend the base abstract class `java.io.OutputStream`. Let's start with input streams.

7.3.1 Input streams

Class `java.io.InputStream` is an abstract base class for all the input streams in Java. It's extended by all the classes that need to read bytes (for example, image data) from multiple data sources. Let's start with the most important method of this class, `read()`, used to read data from a source. The class `InputStream` defines multiple overloaded versions of method `read()`, which can be used to read a single byte of data as `int`, or multiple bytes into a byte array:

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

Reads the next,
single byte of data

Reads multiple bytes of
data into a byte array

But you wouldn't use these methods yourself. You'd use method `read()` by more-specific classes that extend the abstract class `InputStream`. For example, class `FileInputStream` extends `InputStream` and overrides its `read()` method for you to use.



EXAM TIP Watch out for the use of method `read()` from class `InputStream`. It returns the next byte of data, or `-1` if the end of the stream is reached. It doesn't throw an `EOFException`.

Method `close()` is another important method of class `InputStream`. Calling `close()` on a stream releases the system resources associated with it.

Because `InputStream` is an *abstract* class, you *can't* create an instance to read from a data source or destination. You need one of its subclasses to do the work for you. So why do you need to bother with so much theory about it?

Why do you need to bother with so much theory on `InputStream`?

On the exam, you'll be tested on whether you can call a particular method on an object. For example, you might be tested on whether you can call the method `read()` or `close()` on an object of, say, `FileInputStream`, a subclass of `InputStream`. Because methods `read()` and `close()` are defined in class `InputStream`, they can be called on objects of any derived classes of `InputStream`.

These might not be straightforward questions. You need to get the hang of the basics to answer them correctly.

Table 7.2 contains a list of the main methods in class `java.io.InputStream` (from the Java API documentation). Don't worry about the details; we'll cover the relevant details in the subsequent sections.

Table 7.2 List of methods in class `java.io.InputStream`

Method name	Return type	Description
<code>close()</code>	<code>void</code>	Closes this input stream and releases any system resources associated with the stream.
<code>abstract read()</code>	<code>int</code>	Reads the next byte of data from the input stream.
<code>read(byte[] b)</code>	<code>int</code>	Reads a number of bytes from the input stream and stores them into the buffer array <code>b</code> .
<code>read(byte[] b, int off, int len)</code>	<code>int</code>	Reads up to <code>len</code> bytes of data from the input stream into an array of bytes.

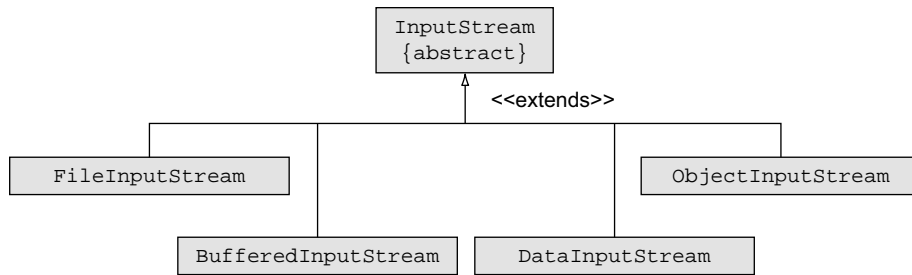


Figure 7.7 Abstract class `InputStream` and its subclasses (the ones that are on the exam)

Figure 7.7 shows the classes that extend the abstract class `java.io.InputStream`. In the next sections, you'll see how to use all these classes. Don't panic: because they share a *lot* of similarity, they'll be simple to work with.



EXAM TIP All the classes that include `InputStream` in their name—`FileInputStream`, `ObjectInputStream`, `BufferedInputStream`, and `DataInputStream`—extend abstract class `InputStream`, directly or indirectly.

Apart from image files, you can also read character data by using byte streams. But you aren't encouraged to do so because the Java API defines well-defined classes for character I/O. On the exam, you'll see questions on valid I/O code and recommended I/O code. Though using a byte stream to read (and write) characters is valid code, it's not a recommended practice.

Let's move on to this class's counterpart for writing byte data to sources: class `java.io.OutputStream`.

7.3.2 Output streams

Class `java.io.OutputStream` is also an abstract class. It's the base class for all the *output* streams in Java. It's extended by all the classes that need to write bytes (for example, image data) to multiple data destinations. The most important method of this class is `write()`, which can be used to write a single byte of data or multiple bytes from a byte array to a data destination:

```

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

Writes single byte

Writes multiple bytes
from a byte array

Methods `close()` and `flush()` are other important methods of class `OutputStream`. Often data isn't written directly to the output stream but buffered for an efficient management of resources. If you want to write data to the output stream right away without waiting for the buffer to be full, call `flush()`. Method `close()` is used to release system resources being used by this stream.

You'd usually work with method `write()` defined by more specific classes that extend `OutputStream`. For example, class `FileOutputStream` extends `OutputStream` and overrides its `write()` method.



EXAM TIP Class `OutputStream` defines methods `write()`, `flush()`, and `close()`. So these are valid methods that can be called on any objects of classes that extend class `OutputStream`.

Table 7.3 contains a list of the methods in class `java.io.OutputStream`. I'll cover the important methods in the subsequent sections.

Table 7.3 Methods in class `java.io.OutputStream`

Method name	Return type	Description
<code>close()</code>	<code>void</code>	Closes this output stream and releases any system resources associated with this stream.
<code>flush()</code>	<code>void</code>	Flushes this output stream and forces any buffered output bytes to be written out.
<code>write(byte[] b)</code>	<code>void</code>	Writes <code>b.length</code> bytes from the specified byte array to this output stream.
<code>write(byte[] b, int off, int len)</code>	<code>void</code>	Writes <code>len</code> bytes from the specified byte array, starting at offset <code>off</code> to this output stream.
<code>abstract write(int b)</code>	<code>void</code>	Writes the specified byte to this output stream.

Figure 7.8 shows abstract class `java.io.OutputStream` and its subclasses.



EXAM TIP All the classes that include `OutputStream` in their name—`FileOutputStream`, `ObjectOutputStream`, `BufferedOutputStream`, and `DataOutputStream`—extend abstract class `OutputStream`, directly or indirectly. Let's start reading from and writing to files with byte streams.

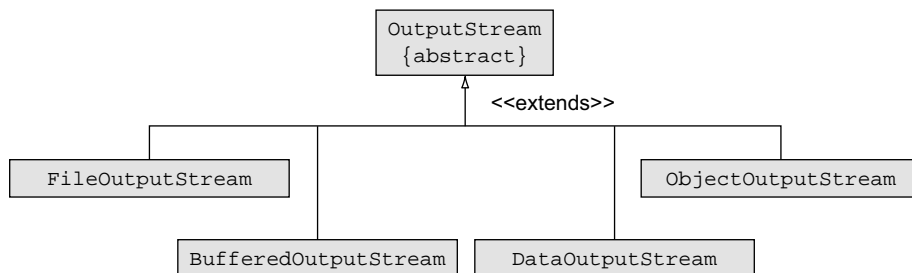


Figure 7.8 Abstract class `OutputStream` and its subclasses covered by this exam

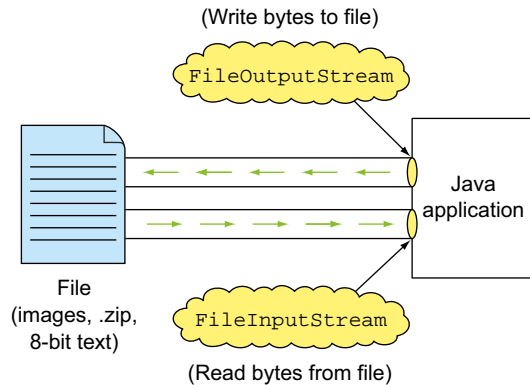


Figure 7.9 A Java application uses `FileInputStream` and `FileOutputStream` to read and write byte data from and to files.

7.3.3 File I/O with byte streams

Bytes can represent *any* type of data, *including* character data. Data in any file type—Portable Document Format (PDF), zip, Microsoft Word, and others—can be read and written as byte data. To read and write raw bytes from and to a file, use `FileInputStream` and `FileOutputStream`, as shown in figure 7.9.

Before you work with an example of reading from and writing to files, take note of the constructors of `FileInputStream` and `FileOutputStream`. You can instantiate `FileInputStream` by passing it the name of a file as a `File` instance or as a string value. The constructor opens a connection to a file and might throw a `FileNotFoundException`, if the specified file can't be found:

```
FileInputStream(File file) throws FileNotFoundException {}
FileInputStream(String name) throws FileNotFoundException {}
```



EXAM TIP `FileInputStream` is instantiated by passing it a `File` or `String` instance. It can't be instantiated by passing it another `InputStream`. The above-mentioned constructors of class `FileInputStream` throw a checked exception, `FileNotFoundException`, which must be handled accordingly.

Instantiation of `FileOutputStream` creates a stream to write to a file specified either as a `File` instance or a string value. You can also pass a boolean value specifying whether to append to the existing file contents. Here are the overloaded constructors of class `FileOutputStream`:

```
FileOutputStream(File file) throws FileNotFoundException
FileOutputStream(File file, boolean append) throws FileNotFoundException
FileOutputStream(String name) throws FileNotFoundException
FileOutputStream(String nm, boolean append) throws FileNotFoundException
```



EXAM TIP The above-mentioned constructors of `FileOutputStream` throw a `FileNotFoundException`, a checked exception. Also, during its instantiation, you can specify whether to append data to an underlying file or override its contents.

Let's work with an example of reading a PDF file and writing it to another file. A PDF file *never* contains *just* character data. Even if a PDF file contains only characters, it also contains formatting information, so as a whole, a PDF file can't be considered character data. You can also use the PDF format for an image file, if you have any doubts.



EXAM TIP Even though `FileInputStream` and `FileOutputStream` can be used to write character data, you shouldn't use them to do so. Unlike byte I/O streams, character streams take into account a user's *charset* and work with Unicode characters, which are better suited for character data.

To read and write to separate PDF files, you need objects of `java.io.FileInputStream` and `java.io.FileOutputStream` that can connect to these input and output PDF files. You need to call at least one method on `FileInputStream` to read data, and one method on `FileOutputStream` to write data. Previously, I mentioned that `InputStream` and `OutputStream` define methods `read()` and `write()` to read and write a single byte of data, respectively. Let's work with these methods in the following listing.

Listing 7.2 Using `FileInputStream` and `FileOutputStream` to read and write bytes

```
import java.io.*;

public class ReadWriteBytesUsingFiles {
    public static void main(String[] args) throws IOException {
        try (
            FileInputStream in = new FileInputStream("Sample.pdf");
            FileOutputStream out = new FileOutputStream("Sample2.pdf");
        ) {
            int data;

            while ((data = in.read()) != -1) {
                out.write(data);
            }
        }
    }
}
```

Instantiates `FileOutputStream` ②

Instantiates `FileInputStream` ①

③ Declares variable to store a single byte of data

④ Loops until end of stream is reached (no more bytes can be read)

⑤ Writes byte data to destination file



EXAM TIP Are you wondering why you need to create a variable of type `int` to read byte data from a file in the preceding code? When a stream exhausts itself and no data can be read from it, method `read()` returns `-1`, which can't be stored by a variable of type `byte`.

The code at ① and ② instantiates `FileInputStream` and `FileOutputStream`. The constructors used in this code accept the filenames as `String` objects. You can also pass the constructors' instances of class `File` or `String`. At ③, you declare a variable of type `int` to store the byte data you read from the file. At ④, you call `in.read()`, which reads and returns a single byte from the underlying file `Sample.pdf`. When no more data can be read from the input file, method `read()` returns `-1`, and this is when

the while loop ends its execution. At ❸, you write a single byte of data to another file by using `out`, an instance of `FileOutputStream`.



NOTE Copying a file's content might not copy its attributes. To copy a file, it's advisable to use methods such as `copy` from class `java.nio.file.Files`.

I/O operations that require reading and writing of a single byte from and to a file are a costly affair. To optimize these operations, you can use a byte array:

```
import java.io.*; public class ReadWriteBytesUsingFiles2 {
    public static void main(String[] args) throws IOException {
        try (
            FileInputStream in = new FileInputStream(
                new File("Sample.pdf"));
            FileOutputStream out = new FileOutputStream("Sample2.pdf");
        ) {
            int data;
            byte[] byteArr = new byte[1024];
            while ((data = in.read(byteArr)) != -1) {
                out.write(byteArr, 0, data);
            }
        }
    }
}
```

Creates byte array of size 1024

read(byteArr) reads data into array byteArr

Writes only read bytes to output stream

Programmers are often confused about the correct use of method `read()` that accepts a byte array. Unlike `read()`, `read(byte[])` *doesn't* return the read bytes. It returns the *count of bytes* read, or `-1` if no more data can be read. The actual data is read in the byte array that is passed to it as a method parameter.

So do you think it makes a difference whether I write the complete byte array to the output stream or write the count of the bytes that were read from the input device? Let me test you on this concept in the next “Twist in the Tale” exercise.

Twist in the Tale 7.1

Let's modify some of the code used in the previous example. Select the correct options for the following code.

```
import java.io.*;

class Twist {
    public static void main(String[] args) throws IOException {
        try (
            FileInputStream in = new FileInputStream("Twist.java");
            FileOutputStream out = new FileOutputStream("Copy.java");
        )
```

```

    {
        int data;
        byte[] byteArr = new byte[2048];
        while ((data = in.read(byteArr)) != -1) {
            out.write(byteArr);
        }
    }
}

```

- a Class Twist executes successfully and creates its own copy with the name Copy.java.
- b Class Twist executes successfully, doesn't throw any exceptions, but fails to create its own identical copy.
- c Class Copy.java fails to compile.
- d Class Twist doesn't compile.

Watch out! Class `FileOutputStream` defines method `write()` that accepts an `int` parameter. But when you use this method to write an `int` value, it writes only 1 byte to the output stream. An `int` data type uses 4 bytes, or 32 bits. `write(int)` writes the 8 low-order bits of the argument passed to it. The 24 high-order bits are ignored. Here's an example that uses binary literals with underscores:

```

class FileStreamsAlwaysReadWriteBytes {
    public static void main(String args[]) throws Exception {
        try {
            OutputStream os = new FileOutputStream("temp.txt");
            InputStream is = new FileInputStream("temp.txt");
        } {
            int i999 = 0b00000000_00000000_00000011_11100111;
            int i20 = 0b00000000_00000000_00000000_00010100;
            os.write(i999);
            os.write(i20);
            System.out.println(i999 + ":" + is.read());
            System.out.println(i20 + ":" + is.read());
        }
    }
}

```

Writes 1 byte
(8 lower bits) to
underlying file;
999 is written
as 231

Writes 20 because
20 can be coded
in 8 lower bits.

Prints 20:20 Prints 999:231



EXAM TIP Method `write(int)` in class `OutputStream` writes a byte to the underlying output stream. If you write an `int` value by using this method, only the 8 low-order bits are written to the output stream; the rest are ignored.

In the next section, you'll see how buffering data reads and writes speeds up the I/O processes.

7.3.4 Buffered I/O with byte streams

Imagine you need to transfer a couple of boxes from one room of your home to another. Would you need less time if you transferred one box at a time, or if you loaded a couple of them in a container and moved the container? Of course, the latter would need less time. Similarly, buffering stores data in memory before sending a read or write request to the underlying I/O devices. Buffering *drastically* reduces the time required for performing reading and writing I/O operations.

To buffer data with byte streams, you need classes `BufferedInputStream` and `BufferedOutputStream`. You can instantiate a `BufferedInputStream` by passing it an `InputStream` instance. A `BufferedOutputStream` can be instantiated by passing it an `OutputStream` instance. You can also specify a buffer size or use the default size. Here are their constructors:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int size)
```



EXAM TIP The exam might test you on how to instantiate buffered streams correctly. To instantiate `BufferedInputStream`, you must pass it an object of `InputStream`. To instantiate `BufferedOutputStream`, you must pass it an object of `OutputStream`.

Let's compare reading and writing a PDF file of considerable size with and without using buffered streams. To start, let's read and write the PDF file for the Java 7 Language specification (`jls7.pdf`):

```
import java.io.*;
import java.util.Date;
public class NonBufferedBytesReadWrite {
    public static void main(String[] args) throws IOException {
        try (
            FileInputStream in = new FileInputStream("jls7.pdf");
            FileOutputStream out = new FileOutputStream("jls7-copy.pdf");
        ) {
            long start = System.currentTimeMillis();

            int data;
            while ((data = in.read()) != -1) {
                out.write(data);
            }

            long end = System.currentTimeMillis();
            System.out.println("Milliseconds elapsed : " + (end-start));
        }
    }
}
```

Size of
jls7.pdf is
2.96 MB

Reads, writes
1 byte at a time

Outputs milliseconds
used to copy jls7.pdf to
jls-copy.pdf.

Let's see how the same I/O process performs using buffered streams:

```
import java.io.*;
import java.util.Date;
public class BufferedReadWriteBytes{
    public static void main(String[] args) throws IOException {
        try (
            FileInputStream in = new FileInputStream("jls7.pdf");
            FileOutputStream out = new FileOutputStream("jls7-copy.pdf");
            BufferedInputStream bis = new BufferedInputStream(in);
            BufferedOutputStream bos = new BufferedOutputStream(out);
        )
        {
            long start = System.currentTimeMillis();

            Reads and buffers data → int data;
            Buffers and writes data → while ((data = bis.read()) != -1) {
                bos.write(data);
            }

            long end = System.currentTimeMillis();
            System.out.println("Milliseconds elapsed : " + (end-start));
        }
    }
}
```

Creates BufferedInputStream, passing it object of FileInputStream.

Creates BufferedOutputStream, passing it object of FileOutputStream.

Outputs milliseconds used to copy jls7.pdf to jls-copy.pdf.

As exhibited by the preceding examples, buffered I/O operations are way faster than nonbuffered I/O operations.

You can use `FileInputStream` and `FileOutputStream` to read and write *only* byte data from and to an underlying file. These classes (`FileInputStream` and `FileOutputStream`) don't define methods to work with any other specific primitive data types or objects, which is what you might need most of the time. In the next section, you'll see how to read and write primitive values and strings by using `DataInputStream` and `DataOutputStream`.

7.3.5 *Primitive values and strings I/O with byte streams*

Data input and output streams let you read and write primitive values and strings from and to an underlying I/O stream in a machine-independent way. Data written with `DataOutputStream` can be read by `DataInputStream`. You can instantiate a `DataInputStream` by passing it an `InputStream` instance. A `DataOutputStream` can be constructed by passing it an `OutputStream` instance. Here are their constructors:

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

Here's an example of reading and writing `char`, `int`, `double`, and `boolean` primitive values and strings using data input and output streams:

```
import java.io.*;
public class ReadWritePrimitiveData {
```

```

public static void main(String... args) throws IOException {
    try {
        FileOutputStream fos = new FileOutputStream(
            new File("myData.data"));
        DataOutputStream dos = new DataOutputStream(fos);

        FileInputStream fis = new FileInputStream("myData.data");
        DataInputStream dis = new DataInputStream(fis);

        dos.writeChars("OS");
        dos.writeInt(999);
        dos.writeDouble(45.8);
        dos.writeBoolean(true);
        dos.writeUTF("Will score 100%");

        System.out.println(dis.readChar());
        System.out.println(dis.readChar());
        System.out.println(dis.readInt());
        System.out.println(dis.readDouble());
        System.out.println(dis.readBoolean());
        System.out.println(dis.readUTF());

        //System.out.println(dis.readBoolean());
    }
}

```

Instantiates DataOutputStream by passing instance of FileOutputStream

Instantiates DataInputStream by passing instance of FileInputStream

Writes primitive data and Unicode String to output stream

Reads second char

Reads first char

Reads int value

Reads double value

Reads Boolean value

Reads Unicode string value

If uncommented, this line throws EOFException at runtime.

The contents of file `myData.data` are shown as a screenshot in figure 7.10. As you can see, this file doesn't store its data in human-readable form. It can be reconstructed using `DataInputStream`.

Any read operation by `DataInputStream` past the end of the file will throw an `EOFException`. The data should be read in the same order as written by `DataOutputStream` for the corresponding primitive data. Different data types may be read in a different manner and occupy a different number of bytes (e.g., `int` and `double` uses 8 bytes). If the data being read is not in the same order as written, you'll get unexpected values. For example, if you read a `double` as an `int`, you'll get unexpected values instead, you'll get unexpected values.

```
import java.io.*;
public class ReadWritePrimitiveData1 {
    public static void main(String... args) throws IOException {
        try {
            FileOutputStream fos = new FileOutputStream(
                new File("myData.data"));
            DataOutputStream dos = new DataOutputStream(fos);
```

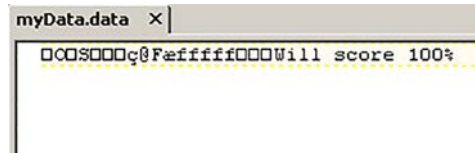


Figure 7.10 Snapshot of contents of file `myData.data`

```

        FileInputStream fis = new FileInputStream("myData.data");
        DataInputStream dis = new DataInputStream(fis);
    } {
        dos.writeDouble(45.8);
        System.out.println(dis.readInt());
        System.out.println(dis.readInt());
    }
}

```

Writes double value
Reads int value
Reads another int value

The output of the preceding code is:

```

1078388326
1717986918

```

In the preceding code, bytes from a double value could be interpreted as int values, though incorrectly. What happens if you try to read bytes from a char value as int values?

```

import java.io.*;
public class ReadWritePrimitiveData1 {
    public static void main(String... args) throws IOException {
        try {
            FileOutputStream fos = new FileOutputStream(
                new File("myData.data"));
            DataOutputStream dos = new DataOutputStream(fos);

            FileInputStream fis = new FileInputStream("myData.data");
            DataInputStream dis = new DataInputStream(fis);
        } {
            dos.writeBoolean(true);
            dos.writeChar('A');
            dos.writeInt(99);
            System.out.println(dis.readInt());
        }
    }
}

```

Writes 1 byte of data (Boolean = 1 byte of data)

Writes 2 bytes of data (char = 2 bytes)

Writes 4 bytes of data (int = 4 bytes)

Reads 4 bytes of data; prints 16793856; interprets first 4 bytes as int value.



EXAM TIP If a mismatch occurs in the type of data written by `DataOutputStream` and the type of data read by `DataInputStream`, you might not get a runtime exception. Because data streams read and write bytes, the read operation constructs the requested data from the available bytes, though incorrectly.

In the next section, you'll read and write the state of your objects to a file.

7.3.6 Object I/O with byte streams: reading and writing objects

To read and write objects, you can use *object streams*. An `ObjectOutputStream` can write primitive values and objects to an `OutputStream`, which can be read by an `ObjectInputStream`. To write objects to a file, use a file for the stream. Objects of a class that implements the `java.io.Serializable` interface can be written to a stream. Serialization is making a deep copy of the object so the whole object graph is written—

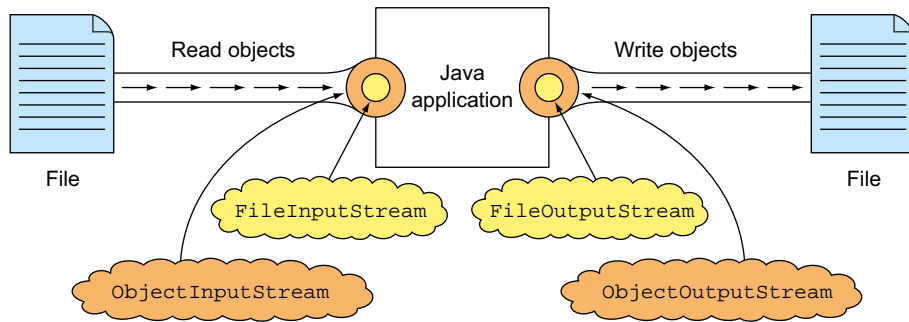


Figure 7.11 `ObjectOutputStream` writes an object to a file by using `FileOutputStream`. `ObjectInputStream` can reconstruct saved objects back from a file by using `FileInputStream`.

that is, all instance variables, and if some instance variables refer to some other objects, all the instance variables of the referred objects will be too (only if they implement `Serializable`), and so on. The default serialization process doesn't write the values of static or transient variables of an object. The reading process, or de-serialization, builds the complete object, including the dependencies.

You can use classes `ObjectInputStream` and `ObjectOutputStream` to read and write objects *and* primitive values. Figure 7.11 shows this arrangement.

You can instantiate these classes by passing them objects of `InputStream` or `OutputStream`. Here are their constructors that read from or write to a specified `InputStream` or `OutputStream`:

```
public ObjectInputStream(InputStream in)
public ObjectOutputStream(OutputStream out)
```



EXAM TIP You can use `ObjectOutputStream` and `ObjectInputStream` to read and write all serializable objects *and* primitive values.

IMPLEMENT `SERIALIZABLE` TO READ AND WRITE OBJECTS FROM AND TO A FILE

Let's start by writing an object of class `Car` to a file. The following code declares a bare-bones class `Car` and writes its object to a file, `object.data`, by using `ObjectOutputStream` and `FileOutputStream`:

```
import java.io.*;
class Car {}
class WriteObject{
    public static void main(String args[]) throws IOException{
        try (
            FileOutputStream out = new FileOutputStream("object.data");
            ObjectOutputStream oos = new ObjectOutputStream(out);
        ) {
            Car c = new Car();
            oos.writeObject(c);
        }
    }
}
```

```

        oos.flush();
    }
}

```

Though the preceding code compiles successfully, it throws a `java.io.NotSerializableException` at runtime. What went wrong? Class `Car` should implement the `Serializable` interface so that it can be written to and read from a file. Here's the modified code:

```

import java.io.*;
class Car implements Serializable {}
class ReadWriteObject{
    public static void main(String args[]) throws Exception{
        try (
            FileOutputStream out = new FileOutputStream("object.data");
            ObjectOutputStream oos = new ObjectOutputStream(out);
            FileInputStream in = new FileInputStream("object.data");
            ObjectInputStream ois = new ObjectInputStream(in);
        ) {
            Car c = new Car();
            oos.writeObject(c);
            oos.flush();
            Car c2 = (Car)ois.readObject();
            System.out.println(c2);
        }
    }
}

```

Car implements `Serializable` interface.

Writes object `c` to file

`readObject()` returns `Object`, so needs explicit cast to `Car` class.

Apart from declaring to throw an `IOException`, method `readObject()` might also throw a `ClassNotFoundException`, if the JRE fails to retrieve the class information corresponding to the retrieved object.



EXAM TIP To write objects to a file, their classes should implement `java.io.Serializable`, or the code will throw a `java.io.NotSerializableException`.

READ AND WRITE OBJECTS WITH NONSERIALIZABLE PARENT CLASSES

What happens if class `Car` extends another class, say `Vehicle`, which doesn't implement the `Serializable` interface? Would you be able to write `Car` objects to a file? In this case, the variables of the `Vehicle` class are serialized to a file. For example

```

import java.io.*;
class Vehicle {
    String name = "Vehicle";
}
class Car extends Vehicle implements Serializable {
    String model = "Luxury";
}
class ParentNotSerializable{
    public static void main(String args[]) throws Exception{

```

Base class doesn't implement `Serializable`.

Derived class implements `Serializable`.

```

try (
    FileOutputStream out = new FileOutputStream("object.data");
    ObjectOutputStream oos = new ObjectOutputStream(out);
    FileInputStream in = new FileInputStream("object.data");
    ObjectInputStream ois = new ObjectInputStream(in);

    ) {
        Car c = new Car();
        oos.writeObject(c);
        oos.flush();
        Car c2 = (Car)ois.readObject();
        System.out.println(c2.name + ":" + c2.model);
    }
}

```

Prints
Vehicle:Luxury

READ AND WRITE OBJECTS WITH NONSERIALIZABLE DATA MEMBERS

Would you be able to write objects to Car to a file, if any of its object fields doesn't implement the Serializable interface? In this case, the code will throw a `java.io.NotSerializableException` when you attempt to *write* a Car object to a file.

For example

```

import java.io.*;

class Engine {
    String make = "198768";
}

class Car implements Serializable {
    String model = "Luxury";
    Engine engine = new Engine();
}

class DataMemberNotSerializable{
    public static void main(String args[]) throws Exception{
        try (
            FileOutputStream out = new FileOutputStream("object.data");
            ObjectOutputStream oos = new ObjectOutputStream(out);

        ) {
            Car c = new Car();
            oos.writeObject(c);
            oos.flush();
        }
    }
}

```

Engine doesn't
implement Serializable.

Car
implements
Serializable.

Throws
NotSerializableException.



EXAM TIP A class whose object fields don't implement the Serializable interface can't be serialized even though the class itself implements the Serializable interface. An attempt to serialize such object fields will throw a runtime exception.

READ AND WRITE OBJECTS ALONG WITH PRIMITIVE VALUES FROM AND TO A FILE

You can use `ObjectInputStream` and `ObjectOutputStream` to read and write both objects and primitive values from and to a file. The data should be retrieved in the order that it was written. In the following example, class `WritePrimAndObjects` writes

a boolean value and then a Car instance. These values should be retrieved in this order. An attempt to read mismatching data types will result in throwing runtime exception `OptionalDataException`:

Checked exceptions that readObject can throw: `IOException`, `ClassNotFoundException`.

```

class ReadPrimAndObjects{
    public static void main(String args[]) throws IOException,
                                   ClassNotFoundException{
        try (
            FileInputStream in = new FileInputStream("object.data");
            ObjectInputStream ois = new ObjectInputStream(in);
        ) {
            System.out.println(ois.readBoolean());
            Car c = (Car)ois.readObject();
            System.out.println(c.name);
        }
    }
}
class Car implements Serializable{
    String name;
    Car(String value) {
        name = value;
    }
}

```

1 `readObject` returns instance of `Object` and can throw `OptionalDataException`



EXAM TIP Retrieve the data (primitive and objects) in the order it was written using object streams, or it might throw a runtime exception.

The code at **1** reads an object from the underlying stream. Method `readObject()` returns `Object`, which is explicitly casted to class `Car`. For the exam, you should know that this method can throw multiple exceptions:

- `ClassNotFoundException`—Class of a serialized object cannot be found
- `OptionalDataException`—Primitive data was found in the stream instead of objects.
- `IOException`—Any of the usual input-/output-related exceptions

THE TRANSIENT AND STATIC VARIABLES AREN'T WRITTEN TO A FILE

When you write objects to a file using `ObjectOutputStream`, its transient or static variables aren't written to the file. For example

```

import java.io.*;
class Car implements Serializable{
    String name;
    transient String model;
    transient int days;
    static int carCount;
    Car(String value) {
        name = value;
        model = "some value";
        days = 98;
        ++carCount;
    }
}

```

static variable → `carCount`

transient variables | `model`, `days`

Assign value to transient variables | `model = "some value";`, `days = 98;`

```

class ReadWriteCarObjects{
    public static void main(String args[]) throws Exception {
        try {
            FileOutputStream out = new FileOutputStream("object.data");
            ObjectOutputStream oos = new ObjectOutputStream(out);
            FileInputStream in = new FileInputStream("object.data");
            ObjectInputStream ois = new ObjectInputStream(in);
        } {
            Car c = new Car("AAA");
            oos.writeObject(c);
            oos.flush();

            new Car("BBB");

            Car c2 = (Car)ois.readObject();
            System.out.println(c2.name);
            System.out.println(c2.model + ":" + c2.days);
            System.out.println(c2.carCount);
        }
    }
}

```

Prints null:0

Prints 2

In the preceding code, because the value of transient variables `model` and `days` wasn't written to the file, the deserialization process assigns default values to these variables: `null` for objects and `0` for `int` type.



NOTE You can also change the serialization process using methods `defaultReadObject()` and `defaultWriteObject()`. But this is beyond the scope of this exam.

METHODS OF OBJECTINPUTSTREAM AND OBJECTOUTPUTSTREAM

Figure 7.12 shows the methods to read and write byte data, primitive data, objects, and a few miscellaneous methods of classes `ObjectInputStream` and `ObjectOutputStream`.

In this section, we covered how to read and write primitive data, strings, and objects to an underlying file by using *byte* I/O classes. Let's move forward with covering character I/O with readers and writers.

7.4 Using character I/O with readers and writers



[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

`Reader` and `Writer` are abstract base classes for reading and writing Unicode-compliant character data. They don't replace the byte-oriented I/O classes, but supplement them.

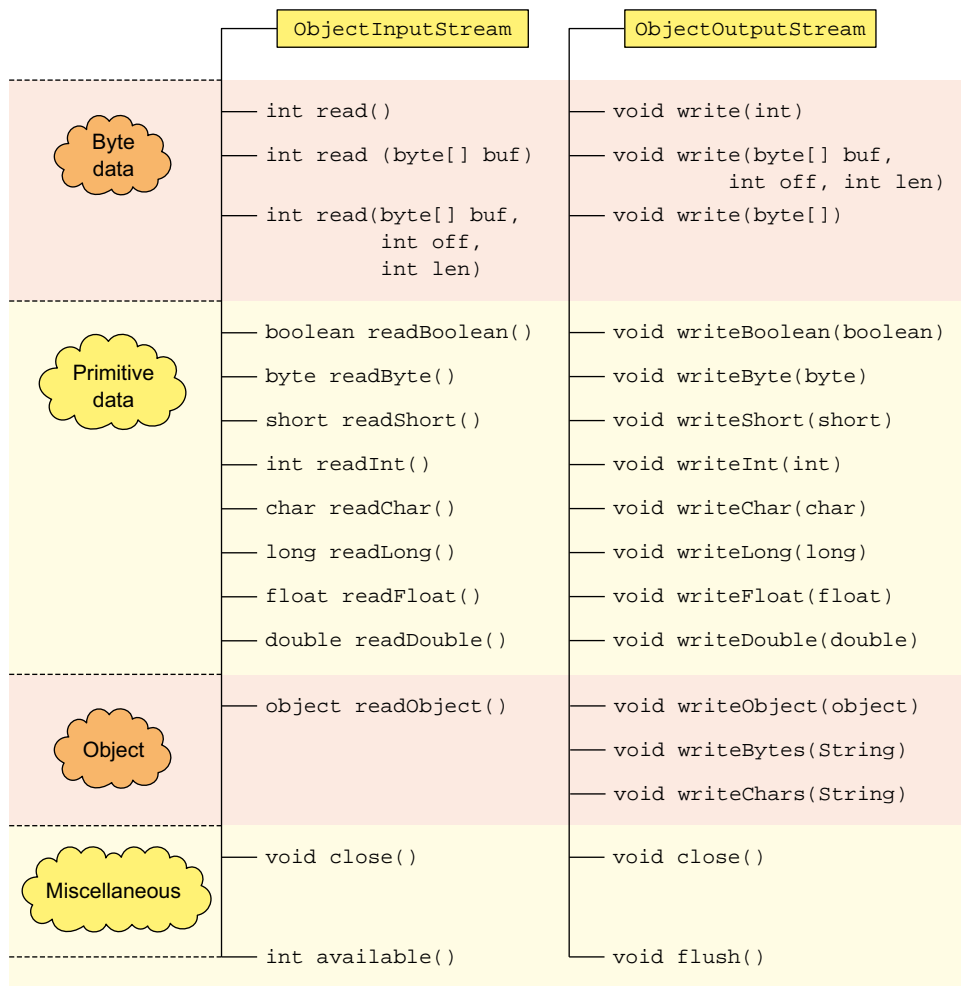


Figure 7.12 Methods of class `ObjectInputStream` and `ObjectOutputStream` to read and write byte data, primitive data, and objects

Classes `Reader` and `Writer` handle 16-bit Unicode well, which isn't supported by the byte-oriented `InputStream` and `OutputStream` classes. Also note that Java's primitive data type `char` stores 16-bit Unicode values. Even though you can use `InputStream` and `OutputStream` to read and write characters, you should use the character-oriented `Reader` and `Writer` classes to read and write character data. Internationalization is possible only by using 16-bit Unicode values. Also `Reader` and `Writer` classes offer faster I/O operations.

7.4.1 Abstract class `java.io.Reader`

Class `Reader` defines overloaded `read()` methods to read character data from an underlying data stream:

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

Reads single character

Reads characters into array

Reads characters into portion of array

Class `Reader` implements `Closeable` (and its parent interface `AutoCloseable`). So `Reader` objects can be declared as resources with a `try-with-resources` statement.



EXAM TIP Compare the overloaded `read()` methods of class `InputStream` with the `read()` methods of class `Reader`. The `read()` methods of `InputStream` accept an array of byte as their method parameter, and the `read()` methods of `Reader` accept an array of `char` as their method parameter.

Because class `Reader` is an abstract class, you won't use it to read data. Instead you'll use one of its concrete subclasses to do the reading for you. Figure 7.13 shows abstract class `Reader` and its subclasses (`BufferedReader` and `FileReader`) that are on the exam.

7.4.2 Abstract class `java.io.Writer`

The abstract class `Writer` defines overloaded `write()` methods to write character data to an underlying data source:

```
void write(char[] cbuf)
abstract void write(char[] cbuf, int off, int len)
void write(int c)
void write(String str)
void write(String str, int off, int len)
```

Writes array of characters

Writes portion of array of characters

Writes single character

Writes string

Writes portion of string

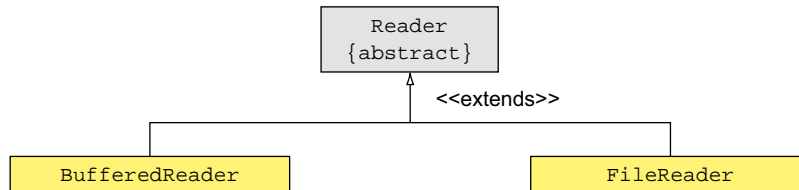


Figure 7.13 Abstract class `Reader` and its subclasses that are on the exam

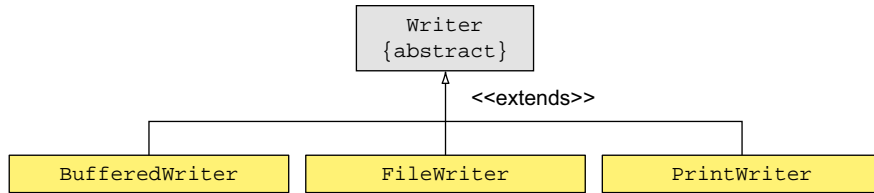


Figure 7.14 Abstract class `Writer` and its subclasses that are on the exam

Class `Writer` implements `Closeable` (and its parent interface `AutoCloseable`). So you can instantiate and use its objects with a `try-with-resources` statement, which results in an implicit call to `Writer`'s method `close()`. `Reader`'s method `close()` is used to close the resources associated with the stream. Method `flush()` is used to write any saved values from a previous write, from a buffer, to the intended destination.



EXAM TIP With the overloaded `write()` methods of class `Writer`, you can write a single character or multiple characters, stored in `char` arrays or `String`, to a data source.

Because class `Writer` is an abstract class, you won't use it to write data. Instead you'll use one of its concrete subclasses to do the writing for you. Figure 7.14 shows abstract class `Writer` and its subclasses (`BufferedWriter`, `FileWriter`, and `PrintWriter`) that are on the exam.

Let's read and write files by using classes `FileReader` and `FileWriter`.

7.4.3 *File I/O with character streams*

`FileReader` and `FileWriter` are convenience classes for reading and writing character data from files. Unless specified, the instances of these classes assume the default character set of the operating system. You can instantiate a `FileReader` by passing it the name of a file as a string value or as a `File` instance. Following are the overloaded constructors of class `FileReader`:

```
FileReader(File file)
FileReader(String fileName)
```

You can instantiate `FileWriter` by passing it the name of a file as a string value or as a `File` instance. You also have the option of specifying whether you want to override the existing content of a file or append new content to it, by passing a boolean value to the constructor. Here are the overloaded constructors of class `FileWriter`:

```
FileWriter(File file)
FileWriter(File file, boolean append)
FileWriter(String fileName)
FileWriter(String fileName, boolean append)
```

The following listing shows an example of reading and writing characters from a file by using `FileReader` and `FileWriter`.

Listing 7.3 Reading and writing characters from files

```
import java.io.*;

public class ReadChars{
    public static void main(String[] args) throws IOException {
        try (
            FileReader input = new FileReader("ReadChars.java");
            FileWriter output = new FileWriter("CopyReadChars.java");
        )
        {
            int data;

            while ((data = input.read()) != -1) {
                output.write(data);
            }
        }
    }
}
```

Read from
ReadChars.java until no
more characters found

Write character to
CopyReadChars.java

The preceding code is similar to the code shown in listing 7.2, which uses `FileInputStream` and `FileOutputStream` to read and write bytes from files. But it uses `FileReader` to read characters from a source and `FileWriter` to write it to a destination.

Data buffering helps produce efficient and faster I/O operations. Buffered I/O with character streams is covered in the next section.

7.4.4 Buffered I/O with character streams

In the real world, nonbuffered data read and write operations are rare, because reading characters one at a time from a file is a costly affair. Whenever you call method `read()` on `FileReader`, it makes a read request to the underlying character stream. This slows the performance of your application.

To buffer data with character streams, you need classes `BufferedReader` and `BufferedWriter`. You can instantiate a `BufferedReader` by passing it a `Reader` instance. A `BufferedWriter` can be instantiated by passing it a `Writer` instance. You can also specify a buffer size or use the default size. Here are their constructors:

```
public BufferedReader(Reader in)
public BufferedReader(Reader in, int sz)
public BufferedWriter(Writer out)
public BufferedWriter(Writer out, int sz)
```



EXAM TIP The exam might test you on how to instantiate buffered character streams correctly. To instantiate `BufferedReader`, you must pass it an object of `Reader`. To instantiate `BufferedWriter`, you must pass it an object of `Writer`.

Class `BufferedReaderWriter` copies characters from `BufferedReaderWriter.java` to `Copy.java` using `BufferedReader` and `BufferedWriter`:

```
import java.io.*;
public class BufferedReaderWriter{
    public static void main(String... args){
        try (
            FileReader fr = new FileReader("BufferedReaderWriter.java");
            BufferedReader br = new BufferedReader(fr);
            FileWriter fw = new FileWriter("Copy.java");
            BufferedWriter bw = new BufferedWriter(fw);
        ){
            String line = null;
            while ((line = br.readLine()) != null) {
                bw.write(line);
                bw.newLine();
            }
        } catch (IOException ioe) {
            System.out.println (ioe);
        }
    }
}
```

Reads single line

Write characters to BufferedWriter

`readLine()` and `close()` can throw `IOException`; `Reader` and `Writer` instantiation can throw `FileNotFoundException`

The preceding code instantiates `BufferedReader` by using a `FileReader` instance and `BufferedWriter` by using a `FileWriter` instance. Instead of using method `read()`, it uses method `readLine()` to read a single line from the character stream. Method `readLine()` doesn't include line-feed (`\n`) and carriage-return (`\r`) characters. So you use `bw.newLine()` to insert a new line in the output file.

Class `BufferedReader` also defines methods `read()` and `read(char[])` to read single characters of data from `Reader`. But these methods are implemented differently than the same methods of class `FileReader`. Class `BufferedReader` buffers data on the first read, and the subsequent request to the `read()` methods returns data from the buffer. But this isn't the case with class `FileReader`.

In the next section, you'll work with `PrintWriter`, which can be used to write all primitive values and strings to an underlying `File`, `OutputStream`, or `Writer` object.

7.4.5 **Data streams with character streams: using `PrintWriter` to write to a file**

Class `PrintWriter` can be used to print (write) formatted representations of objects to a file. This class implements all the print methods found in class `PrintStream`. This essentially means that you can use all the overloaded print methods that you've been using (via the class variable `System.out`) to write data to a file, a `PrintWriter` instance. Here's an example:

```
public class WriteToFileUsingPrintWriter{
    public static void main(String... args){
```

```

try {
    FileWriter fw = new FileWriter("file1.txt", true);
    PrintWriter pw = new PrintWriter(fw);

    pw.write(97);
    pw.write("String");
    pw.write("PartialString", 0, 4);
    pw.write(new char[]{'c','h','a','r'});
    pw.write(new char[]{'c','h','a','r'}, 1, 1);

    pw.print(true);
    pw.print('a');
    pw.print(12.45f);
    pw.print(41.87);
    pw.print(39865L);
    pw.print(pw);
    pw.print(new Integer(10));

    pw.println(true);
    pw.println('a');
    pw.println(12.45f);
    pw.println(41.87);
    pw.println(39865L);
    pw.println(pw);
    pw.println(new Integer(10));

    pw.close();
}
catch (IOException ioe) {
    System.out.println(ioe);
}
}

```

**Instantiates
PrintWriter**

**Overloaded write()
methods**

**Overloaded print()
methods**

**Overloaded println()
methods**

**Flush and
close stream**

The preceding example creates a `PrintWriter` instance by using a `FileWriter` instance. It then uses the overloaded version of methods `write()`, `print()`, and `println()` to write to the underlying file. The overloaded versions of methods `print()` and `println()` are convenient methods to print (or write) data of primitive types and objects. A `PrintWriter` instance can be created by passing it the name of a file as a string value or as a `File` instance with or without specifying an explicit character set to use. These are created without automatic line flushing. You can also instantiate `PrintWriter` by passing it a `Writer` instance and a boolean value specifying auto-flushing. Here's the list of its constructors:

```

PrintWriter(File file)
PrintWriter(File file, String charset)
PrintWriter(String fileName)
PrintWriter(String fileName, String charset)
PrintWriter(Writer out, boolean autoFlush)

```

Table 7.4 shows the commonly used methods for class `PrintWriter` and the valid method arguments.

Table 7.4 Commonly used methods with their valid arguments

Method	Valid method arguments
<code>write</code>	<code>int/ char[]/ String</code>
<code>print</code>	<code>boolean/ int/ char[]/ char/ long/ float/ double/ Object/ String</code>
<code>println</code>	<code>boolean/ int/ char[]/ char/ long/ float/ double/ Object/ String</code>
<code>format</code>	<code>(Locale locale, String format, Object... args)</code>
<code>format</code>	<code>(String format, Object... args)</code>
<code>printf</code>	<code>(Locale locale, String format, Object... args)</code>
<code>printf</code>	<code>(String format, Object... args)</code>



NOTE Methods `format()` and `printf()`, listed in table 7.4, accept a format string and arguments. These methods are covered in detail in chapters 5 and 12.

It's easy to confuse how the constructors of the I/O classes can be chained. The next section will help you understand the concept.

7.4.6 Constructor chaining with I/O classes

Most programmers get cold feet when they think about constructor chaining in I/O classes. It's quite intimidating. Do you think the following line of code instantiates a `BufferedReader` using `FileReader` and `File` instances correctly?

```
BufferedReader br = new BufferedReader(new FileReader(new File("ab.txt")));
```

Let's evaluate the preceding line of code. The only way to construct an object of class `BufferedReader` is by passing to it an object of class `Reader`. Because class `FileReader` passes the ISA `Reader` test, `FileReader` is a valid argument to the constructor of class `BufferedReader`. Now, class `FileReader` can accept an argument of either `String` or `File`, and class `File` accepts an argument of type `String`. So the preceding line of code is valid.

Figure 7.15 shows the inheritance tree of the I/O classes you need to know for the exam, together with their constructors. It should help answer all questions related to chaining of constructors in file I/O.

With a firm understanding of working with byte and character streams to read and write all types of data to a file, let's move on to reading and writing data from a console.

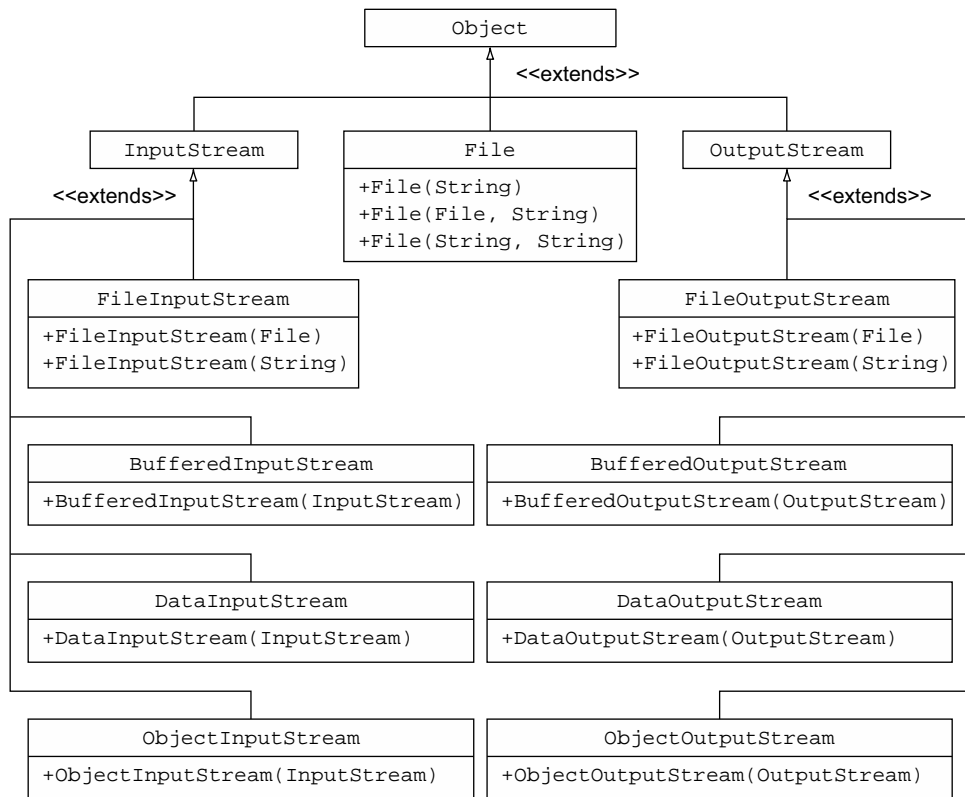


Figure 7.15 Hierarchy of I/O classes, including constructors and their method parameters

7.5 Working with the console



[7.1] Read and write data from the console

Class `java.io.Console` defines methods to access the character-based console device, associated with the current Java virtual machine. It's interesting to note that you *may* or *may not* be able to access the console associated with a JVM, depending on the underlying platform and how the JVM was started. If you invoke a JVM from the command line without redirecting the standard input and output streams, you'll be able to access its console, which will typically be connected to the keyboard and display from which the virtual machine was launched. You may *not* be able to access the console associated with a JVM, if it started automatically as a result of execution of some other program.



NOTE You'll not get access to the console when using IDEs like Eclipse. To execute code that needs to access the console, work with a text editor and command prompt.

Let's work with an example of how to use the class `Console` to interact with a user. Code in the following example outputs a formatted message to a user and accepts input and a password. The `Console` object supports secure password entry by its method `readPassword()`. This method suppresses echoing of the password when a user enters it. Also the return type of this method is (mutable) `char[]` and not `String`:

```
public class InteractWithUserUsingConsole{
    public static void main(String... args){
        Console console = System.console();
        if (console != null) {
            String file = console.readLine("Enter File to delete:");
            console.format("About to delete %s %n", file);

            console.printf("Sure to delete %s(Y/N)?", file);
            String delete = console.readLine();

            if (delete.toUpperCase().trim().equals("Y")) {
                char[] pwd = console.readPassword("Enter Password:");
                if (pwd.length>0)
                    console.format("Deleted %s", file);
                else
                    console.format("No password. Cancelling deletion");
            }
            else
                console.format("Operation cancelled... %nNow exiting.");
        }
        else
            System.out.println("No console");
    }
}
```

1 **Accesses console by calling `System.console()`.**

2 **Prompts user to enter filename to delete.**

3 **Uses `format()` to display text and variable values.**

4 **Outputs formatted data.**

5 **Prompts user to enter password; password values aren't echoed**

The code at ❶ accesses an object of class `Console` by calling `System.console()`. This method returns `null` if no console is associated with the current JVM. You can't create an object of this class yourself. Class `Console` doesn't define a public constructor.

The code at ❷ uses `readLine()` to prompt a user to enter the name of the file to delete. The answer is read by the same method. The code at ❸ and ❹ uses methods `format()` and `printf()` to print formatted data to the user and accept whether the code proceeds with deleting the file. If the user enters `Y` or `y`, the code at ❺ prompts the user to enter the password and reads it using `readPassword()`. If a password is entered, an appropriate message is displayed to the user.



EXAM TIP If no console device is available, `System.console()` returns `null`. A null value signals that either the program was launched in a non-interactive environment or perhaps the underlying operating system doesn't support the console operations.

Table 7.5 lists important methods of class `Console`.

Coverage of class `Console` brings us to the end of this chapter on the basics of file I/O.

Table 7.5 Important methods of class `Console` from the Java API documentation

Constructor	Description
<code>Console format(String fmt, Object... args)</code>	Writes a formatted string to the console's output stream by using the specified format string and arguments.
<code>Console printf(String format, Object... args)</code>	A convenience method to write a formatted string to the console's output stream by using the specified format string and arguments
<code>String readLine()</code>	Reads a single line of text from the console.
<code>String readLine(String fmt, Object... args)</code>	Provides a formatted prompt, and then reads a single line of text from the console.
<code>char[] readPassword()</code>	Reads a password or passphrase from the console with echoing disabled.
<code>char[] readPassword(String fmt, Object... args)</code>	Provides a formatted prompt, and then reads a password or passphrase from the console with echoing disabled.

7.6 Summary

This chapter covers the fundamentals of Java I/O. It starts with an introduction to the streams. The Java I/O stream is an abstraction of a data source or a data destination. An input stream is used to read data from a data source. An output stream is used to write data to a data destination.

To read from and write to files, you'll need to work with class `File`. `File` is an abstract representation of a path to a file or a directory. You can use an object of class `File` to create a new file or directory, delete it, or inquire about or modify its attributes. You can read the contents of a file or write to it by using byte and character I/O classes.

Byte streams work with reading and writing byte data. They're broadly categorized as input streams and output streams. All input streams extend the base abstract class `java.io.InputStream`, and all output streams extend the base abstract class `java.io.OutputStream`.

Class `java.io.InputStream` is an abstract base class for all the input streams in Java. It's extended by all the classes that need to read bytes (for example, image data) from multiple data sources. Class `java.io.OutputStream` is also an abstract class. It's the base class for all the output streams in Java. It's extended by all the classes that need to write bytes (for example, image data) to multiple data destinations.

To read and write any byte data from and to files, you can use `FileInputStream` and `FileOutputStream`. To buffer data with byte streams, use classes `BufferedInputStream` and `BufferedOutputStream`.

Data input and output streams let you read and write primitive values and strings from and to an underlying I/O stream in a machine-independent way. `DataInputStream` and `DataOutputStream` are decorator classes that define methods for reading

and writing primitive values and strings. Data written with `DataOutputStream` can be read by `DataInputStream`.

To read and write objects, you can use object streams. An `ObjectOutputStream` can write primitive values and objects to an `OutputStream`, which can be read by an `ObjectInputStream`. The objects to be written to a file must implement the `java.io.Serializable` interface.

`Reader` and `Writer` are abstract base classes for reading and writing Unicode-compliant character data. `FileReader` and `FileWriter` are convenience classes for reading and writing character data from files. To buffer data with character streams, use `BufferedReader` and `BufferedWriter` classes.

Class `PrintWriter` can be used to print (write) formatted representations of objects to a file. Class `java.io.Console` defines methods to access the character-based console device associated with the current JVM.

REVIEW NOTES

This section lists the main points covered in this chapter.

Working with class `java.io.File`

- `File` is an abstract representation of a path to a file or a directory.
- You can use an object of class `File` to create a new file or directory, delete it, or inquire about or modify its attributes.
- A `File` instance might not be necessarily associated with an actual file or directory.
- `File`'s method `isDirectory()` returns `true` if the path it refers to is a directory.
- `File`'s method `isFile()` returns `true` if the path it refers to is a file.
- For a directory, `File`'s method `list()` returns an array of subdirectories and files.
- You can create a `File` instance that represents a nonexistent file on your file system. And you can even invoke methods like `isFile()` without getting an exception.
- The objects of class `File` are immutable; the pathname represented by a `File` object can't be changed.
- Methods `createNewFile()`, `mkdir()`, and `mkdirs()` can be used to create new files or directories.

Using byte stream I/O

- Class `java.io.InputStream` is an abstract base class for all the input streams.
- Class `InputStream` defines multiple overloaded versions of method `read()`, which can be used to read a single byte of data as `int`, or multiple bytes into a byte array.
- Method `read()` returns the next byte of data, or `-1` if the end of the stream is reached. It doesn't throw an `EOFException`.

- Method `close()` is another important method of class `InputStream`. Calling `close()` on a stream releases the system resources associated with it.
- Class `java.io.OutputStream` is an abstract class. It's the base class for all the output streams in Java.
- The most important method of `OutputStream` class is `write()`, which can be used to write a single byte of data or multiple bytes from a byte array to a data destination.
- Class `OutputStream` also defines methods `write()`, `flush()`, and `close()`. So these are valid methods that can be called on any objects of classes that extend class `OutputStream`.
- All the classes that include `OutputStream` in their name—`FileOutputStream`, `ObjectOutputStream`, `BufferedOutputStream`, and `DataOutputStream`—extend abstract class `OutputStream`, directly or indirectly.
- To read and write raw bytes from and to a file, use `FileInputStream` and `FileOutputStream`.
- `FileInputStream` is instantiated by passing it a `File` instance or string value. It can't be instantiated by passing it another `InputStream`.
- Instantiation of `FileOutputStream` creates a stream to write to a file specified either as a `File` instance or a string value. You can also pass a boolean value specifying whether to append to the existing file contents.
- Copying a file's content might not copy its attributes. To copy a file, it's advisable to use methods such as `copy()` from class `java.nio.file.Files`.
- I/O operations that require reading and writing of a single byte from and to a file are a costly affair. To optimize the operation, you can use a byte array.
- Unlike `read()`, `read(byte[])` doesn't return the read bytes. It returns the count of bytes read, or `-1` if no more data can be read. The actual data is read in the byte array that's passed to it as a method parameter.
- Method `write(int)` in class `OutputStream` writes a byte to the underlying output stream. If you write an `int` value by using this method, only the 8 low-order bits are written to the output stream; the rest are ignored.
- To buffer data with byte streams, you need classes `BufferedInputStream` and `BufferedOutputStream`.
- You can instantiate a `BufferedInputStream` by passing it an `InputStream` instance.
- A `BufferedOutputStream` can be instantiated by passing it an `OutputStream` instance.
- You can specify a buffer size or use the default size for both `BufferedInputStream` and `BufferedOutputStream`.
- To instantiate `BufferedInputStream`, you must pass it an object of `InputStream`. To instantiate `BufferedOutputStream`, you must pass it an object of `OutputStream`.

- You can use `FileInputStream` and `FileOutputStream` to read and write only byte data from and to an underlying file. These classes (`FileInputStream` and `FileOutputStream`) don't define methods to work with any other specific primitive data types or objects.
- Data input and output streams let you read and write primitive values and strings from and to an underlying I/O stream in a machine-independent way. Data written with `DataOutputStream` can be read by `DataInputStream`.
- If a mismatch occurs in the type of data written by `DataOutputStream` and the type of data read by `DataInputStream`, you might not get a runtime exception. Because data streams read and write bytes, the read operation constructs the requested data from the available bytes, though incorrectly.
- An `ObjectOutputStream` can write primitive values and objects to an `OutputStream`, which can be read by an `ObjectInputStream`.
- To write objects to a file, their classes should implement `java.io.Serializable`, or the code will throw a `java.io.NotSerializableException`.
- If a class implements the `Serializable` interface, but its base class doesn't, the class's instance can be serialized.
- A class whose object fields don't implement the `Serializable` interface can't be serialized even though the class itself implements the `Serializable` interface. An attempt to serialize such object fields will throw a runtime exception.
- Retrieve the data (primitive and objects) in the order it was written using object streams, or it might throw a runtime exception.
- When you write objects to a file using `ObjectOutputStream`, its `transient` or `static` variables aren't written to the file.

Using character I/O with readers and writers

- `Reader` and `Writer` are abstract base classes for reading and writing Unicode-compliant character data.
- Classes `Reader` and `Writer` handle 16-bit Unicode well, which isn't supported by the byte-oriented `InputStream` and `OutputStream` classes.
- Abstract class `Reader` defines overloaded `read()` methods to read character data from an underlying data stream.
- Class `Reader` implements `Closeable` (and its parent interface `AutoCloseable`). So `Reader` objects can be declared as resources with a `try-with-resources` statement.
- Compare the overloaded `read()` methods of class `InputStream` with the `read()` methods of class `Reader`. The `read()` methods of `InputStream` accept an array of `byte` as their method parameter, and the `read()` methods of `Reader` accept an array of `char` as their method parameter.
- Abstract class `Writer` defines overloaded `write()` methods to write character data to an underlying data source.

- With the overloaded `write()` methods of class `Writer`, you can write a single character or multiple characters stored in char arrays or string to a data source.
- `FileReader` and `FileWriter` are convenience classes for reading and writing character data from files.
- You can instantiate a `FileReader` by passing it the name of a file as a string value or as a `File` instance.
- You can instantiate a `FileWriter` by passing it the name of a file as a string value or as a `File` instance. You also have the option of specifying whether you want to override the existing content of a file or append new content to it by passing a boolean value to the constructor.
- To buffer data with character streams, you need classes `BufferedReader` and `BufferedWriter`.
- You can instantiate a `BufferedReader` by passing it a `Reader` instance.
- You can instantiate a `BufferedWriter` by passing it a `Writer` instance.
- You can also specify a buffer size or use the default size for both `BufferedReader` and `BufferedWriter`.
- Class `PrintWriter` can be used to print (write) formatted representations of objects to a file. This class implements all the `print()` methods found in class `PrintStream`.
- This essentially means that you can use all the overloaded `print()` methods that you've been using (via the class variable `System.out`) to write data to a file, a `PrintWriter` instance.

Working with the console

- Class `java.io.Console` defines methods to access the character-based console device associated with the current JVM.
- You may or may not be able to access the console associated with a JVM, depending on the underlying platform and how the JVM was started.
- If you invoke a JVM from the command line without redirecting the standard input and output streams, you'll be able to access its console, which will typically be connected to the keyboard and display from which the virtual machine was launched.
- You may not be able to access the console associated with a JVM if it started automatically as a result of the execution of some other program.
- You will not get access to the console when using IDEs like Eclipse.
- You can access an object of class `Console` by calling `System.console()`.
- If no console device is available, `System.console()` returns `null`. A `null` value signals that either the program was launched in a noninteractive environment or perhaps the underlying operating system doesn't support the console operations.
- You can't create an object of `Console` yourself. Class `Console` doesn't define a public constructor.

SAMPLE EXAM QUESTIONS

Q 7-1. What's the output of the following code?

```
import java.io.*;
class Q1 {
    public static void main(String args[]) throws IOException {
        DataOutputStream dos = new DataOutputStream(
            new FileOutputStream("contacts.txt"));
        dos.writeDouble(999.999);
        DataInputStream dis = new DataInputStream(
            new FileInputStream("contacts.txt"));
        System.out.println(dis.read());
        System.out.println(dis.read());
        dis.close();
        dos.close();
    }
}
```

- a** 999.999
-1
- b** 999
999
- c** 999.999
EOFException
- d** None of the above

Q 7-2. Which options are true for the following code?

```
import java.io.*;
class ReadFromConsole {
    public static void main(String args[]) throws IOException {
        Console console = System.console();
        String name = "";
        while (name.trim().equals("")) {
            name = console.readLine("What is your name?\n");
            console.printf(name);
        }
    }
}
```

- a** Class ReadFromConsole can be used to repeatedly prompt a user to enter a name, until the user doesn't enter a value.
- b** console.readLine prints the prompt What is your name? and waits for the user to enter a value.
- c** console.printf(name) prints the value, entered by the user, back to the console.
- d** Class ReadFromConsole can never throw a NullPointerException.

Q 7-3. Select the incorrect statements.

- a `PrintWriter` can be used to write a `String` to the text file `data.txt`.
- b `PrintWriter` can be used to write to `FileOutputStream`.
- c `PrintWriter`'s `format` and `println` methods can write a formatted string by using the specified format string and arguments.
- d None of the methods or constructors of class `PrintWriter` throw I/O exceptions.

Q 7-4. Assuming that a user enters the values *eJava* and *Guru*, when prompted to enter username and password values, what values would be sent to file `login-credentials.txt` when using the following code?

```
class FromConsoleToFile {
    public static void main(String args[]) throws Exception {
        try (PrintWriter pw = new PrintWriter(
            new File("login-credentials.txt"));) {
            Console console = System.console();
            String username = console.readLine("Username:");
            String pwd = console.readPassword("Password:");

            pw.println(username);
            pw.println(pwd);
            pw.flush();
        }
    }
}
```

- a `eJava`
`Guru`
- b `eJava`
`****`
- c `eJava`
`<BLANK LINE>`
- d `eJava`
`String@b6546`
- e Compilation error
- f Runtime exception

Q 7-5. Assuming that the variable `file` refers to a valid `File` object, which of the following options can be used to instantiate a `BufferedInputStream`?

- a `BufferedInputStream bis = new BufferedInputStream(file);`
- b `BufferedInputStream bis = new DataInputStream(new FileInputStream(file));`
- c `BufferedInputStream bis = new BufferedInputStream(new DataInputStream(new FileInputStream(file)));`
- d `BufferedInputStream bis = new BufferedInputStream(new FileInputStream(new DataInputStream(file)));`

Q 7-6. Which option(s) will make the Console object accessible?

- a `Console console = new Console(System.in, System.out);`
- b `Console console = System.console();`
- c `Console console = System.getConsole();`
- d `Console console = System.accessConsole();`
- e `Console console = new Console();`
- f `Console console = System.createConsole();`
- g None of the above

Q 7-7. Given the following code to read and write instances of Phone, which definition of class Phone will persist only its variable model?

```
class ReadWriteObjects {
    public void write(Phone ph, String fileName) throws Exception {
        ObjectOutputStream oos = new ObjectOutputStream(
            new FileOutputStream(fileName));
        oos.writeObject(ph);
        oos.flush();
        oos.close();
    }
    public Phone read(String fileName) throws Exception {
        ObjectInputStream ois = new ObjectInputStream(
            new FileInputStream(fileName));
        Phone ph = (Phone)ois.readObject();
        ois.close();
        return ph;
    }
}
```

- a `class Phone implements Serializable {`
`int sessionId;`
`String model = "EJava";`
`}`
- b `class Phone {`
`transient int sessionId;`
`String model = "EJava";`
`}`
- c `class Phone implements Persistable {`
`transient int sessionId;`
`String model = "EJava";`
`}`
- d `class Phone implements Serializable {`
`transient int sessionId;`
`String model = "EJava";`
`}`

Q 7-8. Which lines of code, when inserted at //INSERT CODE HERE, will enable class Copy to copy contents from file ejava.txt to ejava-copy.txt?

```
class Copy {
    public static void main(String args[]) {
        try {
            InputStream fis = new DataInputStream(new FileInputStream(
                new File("eJava.txt")));
            OutputStream fos = new DataOutputStream(new FileOutputStream(
                new File("eJava-copy.txt")));

            int data = 0;
            while ((data = fis.read()) != -1) {
                fos.write(data);
            }
        }
        //catch (/* INSERT CODE HERE */) {
        }
        finally {
            try {
                if (fos != null) fos.close();
                if (fis != null) fis.close();
            }
            catch (IOException e) {}
        }
    }
}
```

- a catch (IOException e) {
- b catch (FileNotFoundException e) {
- c catch (FileNotFoundException | IOException e) {
- d catch (IOException | FileNotFoundException e) {

Q 7-9. Which option, when inserted at //INSERT CODE HERE, will enable method read() in class ReadFile to read and output its own source file?

```
class ReadFile{
    public void read() throws IOException{
        File f = new File("ReadFile.java");
        FileReader fr = new FileReader(f);
        BufferedReader br = new BufferedReader(fr);
        String line = null;
        //INSERT CODE HERE
        System.out.println(line);
        br.close();
        fr.close();
    }
}
```

- a while ((line = br.readLine()) != null)
- b while ((line = fr.readLine()) != null)

- c while ((line = f.readLine()) != null)
- d while ((line = br.readLine()) != -1)
- e while ((line = fr.readLine()) != -1)
- f while ((line = f.readLine()) != -1)

Q 7-10. Given the following XML, which code options, when inserted at //INSERT CODE HERE, will write this XML (in exactly the same format) to a text file?

```
<emp>
<id>8743</id>
<name>Harry</name>
</emp>
class WriteXML{
    public void writeEmpData() throws IOException{
        File f = new File("empdata.txt");
        PrintWriter pw = null;
        //INSERT CODE HERE
        pw.close();
    }
}
```

- a pw = new PrintWriter(f);
pw.println("<emp>");
pw.write("<id>");
pw.writeInt(8743);
pw.println("</id>");
pw.print("<name>Harry</name>");
pw.print("</emp>");
pw.flush();
- b pw = new PrintWriter(new FileOutputStream(f));
pw.write("<emp>");
pw.write("<id>8743</id>");
pw.write("<name>Harry</name>");
pw.write("</emp>");
pw.flush();
- c pw = new PrintWriter(f);
pw.println("<emp>");
pw.println("<id>8743</id>");
pw.println("<name>Harry</name>");
pw.println("</emp>");
pw.flush();
- d pw = new PrintWriter(new FileOutputStream(f));
pw.println("<emp>");
pw.println("<id>8743</id>");
pw.println("<name>Harry</name>");
pw.println("</emp>");
pw.flush();

ANSWERS TO SAMPLE EXAM QUESTIONS

A 7-1. d

[7.2] Use streams to read from and write to files by using classes in the java.io package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Explanation: `dos.writeDouble(999.999)` writes 8 bytes of data to the underlying stream, and `dis.read()` reads a single byte of data from the underlying stream, interprets it as an integer value, and outputs it. So the code neither prints 999.999 nor throws an `EOFException`.

A 7-2. a, b, c

[7.1] Read and write data from the console

Explanation: Option (d) is incorrect because `System.console()` might return `null`, depending on how the JVM is invoked. A console isn't available to a JVM if it's started using another program or a background process, or if the underlying OS doesn't support it. In such cases, `console.readLine` throws a `NullPointerException`.

A 7-3. d

[7.2] Use streams to read from and write to files by using classes in the java.io package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Explanation: Options (a) and (b) are correct statements. A `PrintWriter` can be used to write to a file, an `OutputStream`, and a `Writer`.

Option (c) is also a correct statement because the `PrintWriter`'s methods `format()` and `printf()` can write a formatted string by using the specified format string and arguments.

Option (d) is an incorrect statement. Some of the constructors of `PrintWriter` may throw I/O exceptions (for example, when a file couldn't be found). But none of the methods of `PrintWriter` throw an exception. You can use `checkError()` to verify if an error has occurred (for example, format conversion has failed).

A 7-4. e

[7.2] Use streams to read from and write to files by using classes in the java.io package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

[7.1] Read and write data from the console

Explanation: The code fails to compile because the return type of method `readPassword()` is `char[]` and not `String`. If the type of the variable `pwd` is changed from

String to `char[]`, the contents of the file `login-credentials.txt` will match as shown in option (a). The overloaded `println()` method in class `PrintWriter` accepts a `char[]` parameter and prints its individual characters to the underlying file, `OutputStream`, or `Writer`.

A 7-5. c

[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Explanation: Options (a), (b), and (d) won't compile. Option (a) is incorrect. To instantiate `BufferedReader`, you must pass to its constructor an instance of `InputStream`. In option (b), you can't instantiate `BufferedInputStream` by invoking a constructor of `DataInputStream`. In option (d), you can't instantiate `FileInputStream` by passing it an instance of `InputStream`. It needs an instance of a `File` or `String`.

A 7-6. b

[7.1] Read and write data from the console

Explanation: Options (a), (c), (d), and (f) are invalid code options. Option (e) is incorrect because `Console`'s constructor is defined as private. They won't compile.

A 7-7. d

[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Explanation: To persist instances of a class, the class must implement the `Serializable` interface. `Serializable` is a marker interface; it doesn't define any methods. Also, to prevent a field from persisting, define it as a transient variable.

Option (a) doesn't mark the field `sessionId` as transient.

Option (b) doesn't implement either `Serializable` or `Externalizable`.

Option (c) implements the `Persistable` interface, which doesn't exist.

A 7-8. a

[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Explanation: Code in class `Copy` can throw two checked exceptions: `FileNotFoundException` and `IOException`. Instantiation of `FileInputStream` and `FileOutputStream` can throw a `FileNotFoundException`. Calling methods `read()` and `write()`

can throw an `IOException`. Because `IOException` is the base class of `FileNotFoundException`, the exception handler for `IOException` will handle `FileNotFoundException` also.

Option (b) won't handle the `IOException` thrown by methods `read()` and `write()`.

Options (c) and (d) are incorrect because alternatives in a multi-catch statement must not pass the IS-A test. The alternatives used in these options pass the IS-A test; `FileNotFoundException` extends `IOException`.

A 7-9. a

[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Explanation: Method `readLine()` isn't defined in classes `File` and `FileReader`. It's defined only in class `BufferedReader`. Also, `readLine()` returns `null` if the end of the stream has been reached.

A 7-10. c, d

[7.2] Use streams to read from and write to files by using classes in the `java.io` package including `BufferedReader`, `BufferedWriter`, `File`, `FileReader`, `FileWriter`, `DataInputStream`, `DataOutputStream`, `ObjectOutputStream`, `ObjectInputStream`, and `PrintWriter`

Explanation: Option (a) won't compile. Class `PrintWriter` doesn't define method `writeInt()`. It defines overloaded `write()` methods with accepted method parameters of type `char[]`, `int`, or `String`.

Option (b) is incorrect because it writes all the given XML in a single line, without inserting line breaks.

Option (c) is correct. The overloaded `println()` method in `PrintWriter` writes data to the underlying stream and then terminates the line.

Option (d) is correct. A `PrintWriter` can be used to write to a byte stream (`OutputStream`) and a character stream (`Writer`).