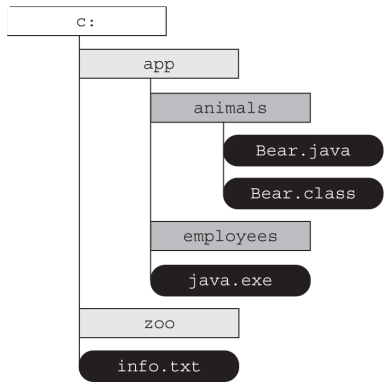
Chapter 19 - I/O

Understanding Files and Directories

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

a *path* is a `String` representation of a file or directory within a file system.



Introducing the *File* Class

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

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

Creating a File Object

A `File` object often is initialized with a `String` containing either an absolute or relative path to the file or directory within the file system.

For convenience, Java offers two options to retrieve the local separator character: a system property and a `static` variable defined in the `File` class. Both of the following examples will output the separator character for the current environment:

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

```

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

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

There are three `File` constructors you should know for the exam.

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

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

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

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

TABLE 19.1 Commonly used java.io.File methods

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

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

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

```

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

```
File Exists: false
```

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

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

```
Last Modified: 1606860000000

```

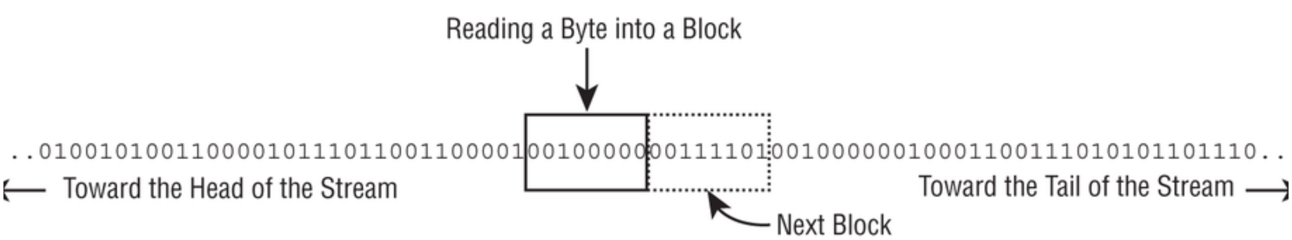
Finally, if the path provided pointed to a valid directory, such as `c:\data`, it would output something similar to the following:

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

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

Understanding I/O Stream Fundamentals

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



Byte Streams vs. Character Streams

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

Differences between Byte and Character Streams

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

The byte streams are primarily used to work with binary data, such as an image or executable file, while character streams are used to work with text files. Since the byte stream classes can write all types of binary data, including strings, it follows that the character stream classes aren't strictly necessary. There are advantages, though, to using the character stream classes, as they are specifically focused on managing character and string data. For example, you can use a `Writer` class to output a `String` value to a file without necessarily having to worry about the underlying character encoding of the file.

The *character encoding* determines how characters are encoded and stored in bytes in a stream and later read back or decoded as characters. Although this may sound simple, Java supports a wide variety of character encodings, ranging from ones that may use one byte for Latin characters, `UTF-8` and `ASCII` for example, to using two or more bytes per character, such as `UTF-16`

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

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

```
    Charset utf16Charset = Charset.forName("UTF-16");
```

Input vs. Output Streams

Most `InputStream` stream classes have a corresponding `OutputStream` class, and vice versa.
It follows, then, that most `Reader` classes have a corresponding `Writer` class.
There are exceptions to this rule. For the exam, you should know that `PrintWriter` has no accompanying `PrintReader` class. Likewise, the `PrintStream` is an `OutputStream` that has no corresponding `InputStream` class. It also does not have `Output` in its name. We will discuss these classes later this chapter.

Low-Level vs. High-Level Streams

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

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

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

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

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

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

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

Stream Base Classes

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

Decoding I/O Class Names

Review of java.io Class Name Properties

- A class with the word `InputStream` or `OutputStream` in its name is used for reading or writing binary or byte data, respectively.
- A class with the word `Reader` or `Writer` in its name is used for reading or writing character or string data, respectively.
- Most, but not all, input classes have a corresponding output class.
- A low-level stream connects directly with the source of the data.
- A high-level stream is built on top of another stream using wrapping.

- A class with `Buffered` in its name reads or writes data in groups of bytes or characters and often improves performance in sequential file systems.
- With a few exceptions, you only wrap a stream with another stream if they share the same abstract parent.

TABLE 19.2 The java.io abstract stream base classes

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

TABLE 19.3 The java.io concrete stream classes

Class Name	Low/High Level	Description
`FileInputStream`	Low	Reads file data as bytes
`FileOutputStream`	Low	Writes file data as bytes
`FileReader`	Low	Reads file data as characters
`FileWriter`	Low	Writes file data as characters
`BufferedInputStream`	High	Reads byte data from an existing `InputStream` in a buffered manner, which improves efficiency and performance
`BufferedOutputStream`	High	Writes byte data to an existing `OutputStream` in a buffered manner, which improves efficiency and performance
`BufferedReader`	High	Reads character data from an existing `Reader` in a buffered manner, which improves efficiency and performance
`BufferedWriter`	High	Writes character data to an existing `Writer` in a buffered manner, which improves efficiency and performance
`ObjectInputStream`	High	Deserializes primitive Java data types and graphs of Java objects from an existing `InputStream`
`ObjectOutputStream`	High	Serializes primitive Java data types and graphs of Java objects to an existing `OutputStream`
`PrintStream`	High	Writes formatted representations of Java objects to a binary stream
`PrintWriter`	High	Writes formatted representations of Java objects to a character stream

Common I/O Stream Operations

Reading and Writing Data

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

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

```

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

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

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

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

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

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

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

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

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

```

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

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

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

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

Manipulating Input Streams

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

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

```

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

mark() and *reset()*

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

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

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

skip()

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

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

The return parameter of `skip()` tells us how many values were actually skipped. For example, if we are near the end of the stream and call `skip(1000)`, the return value might be `20`, indicating the end of the stream was reached after `20` values were skipped.

Flushing Output Streams

When data is written to an output stream, the underlying operating system does not guarantee that the data will make it to the file system immediately. In many operating systems, the data may be cached in memory, with a write occurring only after a temporary cache is filled or after some amount of time has passed.

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

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

Reviewing Common I/O Stream Methods

TABLE 19.4 Common I/O stream methods

Stream Class	Method Name	Description
All streams	`void close()`	Closes stream and releases resources

All input streams	`int read()`	Reads a single byte or returns `-1` if no bytes were available
`InputStream`	`int read(byte[] b)`	Reads values into a buffer. Returns number of bytes read
`Reader`	`int read(char[] c)`	
`InputStream`	`int read(byte[] b, int offset, int length)`	Reads up to `length` values into a buffer starting from position `offset`. Returns number of bytes read
`Reader`	`int read(char[] c, int offset, int length)`	
All output streams	`void write(int)`	Writes a single byte
`OutputStream`	`void write(byte[] b)`	Writes an array of values into the stream
`Writer`	`void write(char[] c)`	
`OutputStream`	`void write(byte[] b, int offset, int length)`	Writes `length` values from an array into the stream, starting with an `offset` index
`Writer`	`void write(char[] c, int offset, int length)`	
All input streams	`boolean markSupported()`	Returns `true` if the stream class supports `mark()`
All input streams	`mark(int readLimit)`	Marks the current position in the stream
All input streams	`void reset()`	Attempts to reset the stream to the `mark()` position
All input streams	`long skip(long n)`	Reads and discards a specified number of characters
All output streams	`void flush()`	Flushes buffered data through the stream

Reading and Writing Binary Data

`FileInputStream` and `FileOutputStream` are used to read bytes from a file or write bytes to a file, respectively. These classes connect to a file using the following constructors:

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

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

Buffering Binary Data

the `Buffered` classes contain a number of performance improvements for managing data in memory.

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

Reading and Writing Character Data

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

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

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

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

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

```

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

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

Buffering Character Data

Like we saw with byte streams, Java includes high-level buffered character streams that improve performance.

```
public BufferedReader(Reader in)

public BufferedWriter(Writer out)

```

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

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

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

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

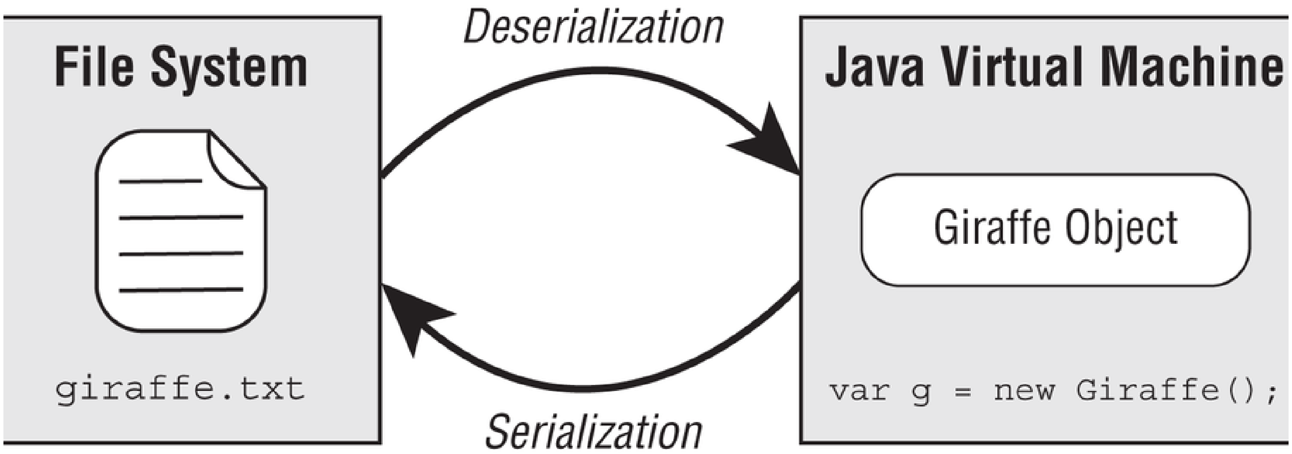
```
void copyTextFileWithBuffer(File src, File dest) throws IOException {
   try (var reader = new BufferedReader(new FileReader(src));
        var writer = new BufferedWriter(new FileWriter(dest))) {
      String s;
      while ((s = reader.readLine()) != null) {
         writer.write(s);
         writer.newLine();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

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

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

Serializing Data

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



Applying the *Serializable* Interface

To serialize an object using the I/O API, the object must implement the `java.io.Serializable` interface. The `Serializable` interface is a marker interface, similar to the marker annotations you learned about in Chapter 13, "Annotations."

The purpose of using the `Serializable` interface is to inform any process attempting to serialize the object that you have taken the proper steps to make the object serializable.

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

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

Any field that is marked `transient` will not be saved to a stream when the class is serialized.

Marking Data *transient*

Oftentimes, the `transient` modifier is used for sensitive data of the class, like a `password`. There are other objects it does not make sense to serialize, like the state of an in-memory `Thread`.

What happens to data marked `transient` on deserialization? It reverts to its default Java values, such as `0.0` for `double`, or `null` for an object.

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

Ensuring a Class Is *Serializable*

How to Make a Class Serializable

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

Storing Data with *ObjectOutputStream* and *ObjectInputStream*

The `ObjectInputStream` class is used to deserialize an object from a stream, while the `ObjectOutputStream` is used to serialize an object to a stream.

```
public ObjectInputStream(InputStream in) throws IOException

```

```
public ObjectOutputStream(OutputStream out) throws IOException
```

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

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

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

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

`ObjectInputStream` inherits an `available()` method from `InputStream` that you might think can be used to check for the end of the stream rather than throwing an `EOFException`.

nderstanding the Deserialization Creation Process

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

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

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

   public Chimpanzee() {
      this.name = "Unknown";
      this.age = 12;
      this.type = 'Q';
   }

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

   // Getters/Setters/toString() omitted
}

```

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

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

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

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

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

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

Printing Data

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

The print stream classes include the following constructors:

```
public PrintStream(OutputStream out)

public PrintWriter(Writer out)
```

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

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

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

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

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

Just be aware that many of these examples can be easily rewritten to use a `PrintStream`.

print()

The most basic of the print-based methods is `print()`. The print stream classes include numerous overloaded versions of `print()`, which take everything from primitives and `String` values, to objects. Under the covers, these methods often just perform

`String.valueOf()` on the argument and call the underlying stream's `write()` method to add it to the stream. For example, the following sets of `print` / `write` code are equivalent:

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

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

println()

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

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

format()

Each print stream class includes a `format()` method, which includes an overloaded version that takes a `Locale`.

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

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

TABLE 19.5 Common print stream `format()` symbols

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

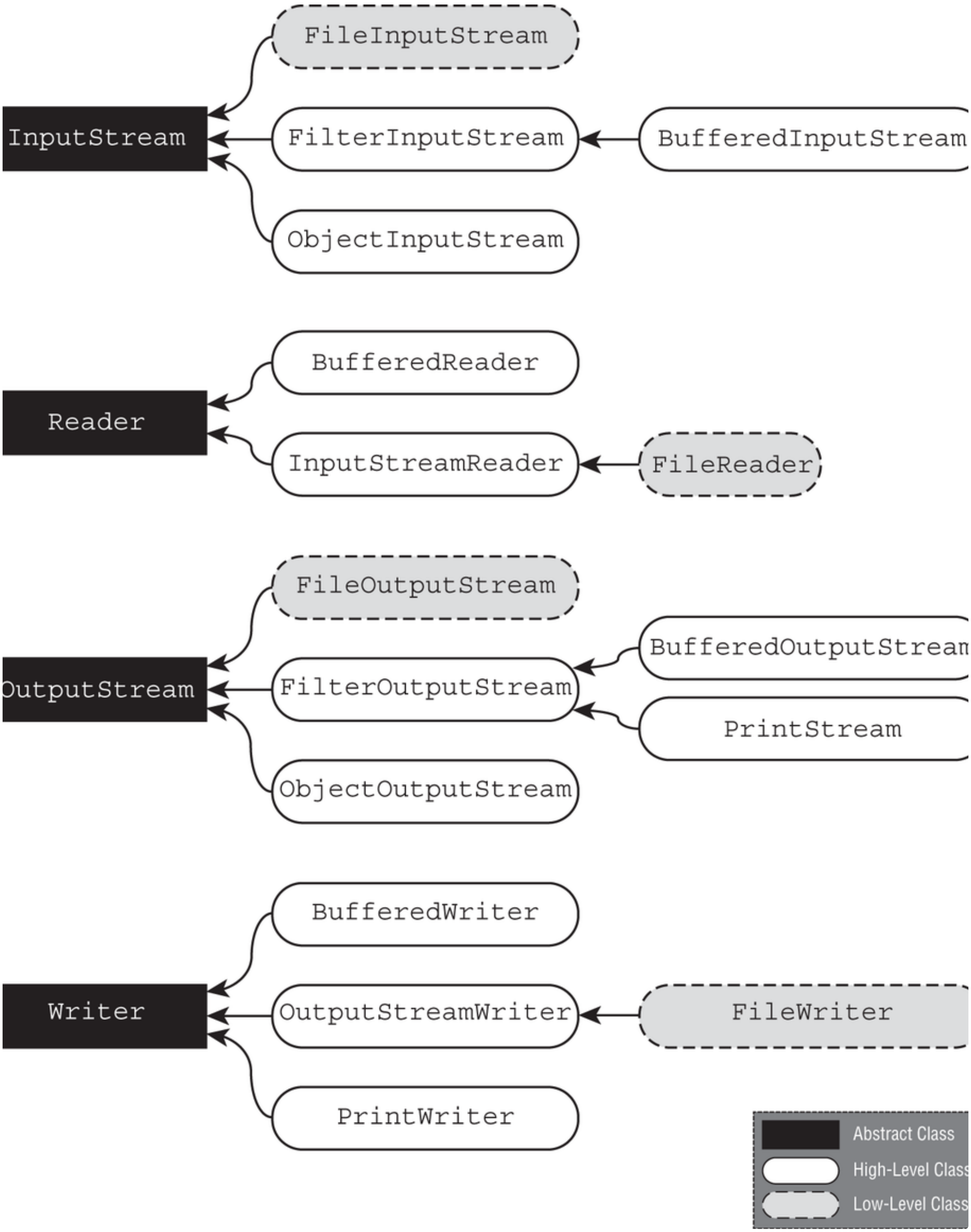
```
String name = "James";
double score = 90.25;
int total = 100;
System.out.format("%s:%n   Score: %f out of %d", name, score, total);
```

Sample *PrintWriter* Program

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

```
File source = new File("zoo.log");
try (var out = new PrintWriter(
   new BufferedWriter(new FileWriter(source)))) {
   out.print("Today's weather is: ");
   out.println("Sunny");
   out.print("Today's temperature at the zoo is: ");
   out.print(1 / 3.0);
```

```
    out.println('C');
    out.format("It has rained %5.2f inches this year %d", 10.2, 2021);
    out.println();
    out.printf("It may rain %s more inches this year", 1.2);
}
```



Printing Data to the User

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

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

Reading Input as a Stream

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

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

```

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

Closing *System* Streams

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

What do you think the following code snippet prints?

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

```

Nothing. It prints nothing. Remember, the methods of `PrintStream` do not throw any checked exceptions and rely on the `checkError()`

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

Acquiring Input with *Console*

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

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

reader() and *writer()*

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

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

format()

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

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

Unlike the print stream classes, `Console` does not include an overloaded `format()` method that takes a `Locale` instance. Instead, `Console` relies on the system locale.

readLine()* and *readPassword()

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

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

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

Like using `System.in` with a `BufferedReader`, the `Console readLine()` method reads input until the user presses the Enter key.

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

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