- `upperCase`, `lowerCase`, `swapCase`, `capitalize`, and `uncapitalize`: Changes the case
- `countMatches`: Returns the number of the substring occurrences
- `isWhitespace`, `isAsciiPrintable`, `isNumeric`, `isNumericSpace`, `isAlpha`, `isAlphaNumeric`, `isAlphaSpace`, and `isAlphaNumericSpace`: Checks the presence of a certain type of characters
- `isAllLowerCase` and `isAllUpperCase`: Checks the case
- `defaultString`, `defaultIfBlank`, and `defaultIfEmpty`: Returns a default value if `null`
- `rotate`: Rotates characters using a circular shift
- `reverse` and `reverseDelimited`: Reverses characters or delimited groups of characters
- `abbreviate` and `abbreviateMiddle`: Abbreviates a value using an ellipsis or another value
- `difference`: Returns the differences in values
- `getLevenshteinDistance`: Returns the number of changes needed to transform one value into another

As you can see, the `StringUtils` class has a very rich (we have not listed everything) set of methods for string analysis, comparison, and transformation that complements the methods of the `String` class.

I/O streams

Any software system has to receive and produce some kind of data that can be organized as a set of isolated input/output or as a stream of data. A stream can be limited or endless. A program can read from a stream (which is called an **input stream**) or write to a stream (which is called an **output stream**). The Java I/O stream is either byte-based or character-based, meaning that its data is interpreted either as raw bytes or as characters.

The `java.io` package contains classes that support many, but not all, possible data sources. It is built for the most part around input from and to files, network streams, and internal memory buffers. It does not contain many classes necessary for network communication. They belong to `java.net`, `javax.net`, and other packages of a Java networking API. Only after the networking source or destination is established (a network socket, for example) can a program read and write data using the `InputStream` and `OutputStream` classes of the `java.io` package.

The classes of the `java.nio` package have pretty much the same functionality as the classes of `java.io` packages. But, in addition, they can work in *non-blocking* mode, which can substantially increase performance in certain situations. We will talk about non-blocking processing in *Chapter 15*, *Reactive Programming*.

Stream data

Input data has to be binary – expressed in 0s and 1s – at the very least because that is the format a computer can read. Data can be read or written one byte at a time or an array of several bytes at a time. These bytes can remain binary or can be interpreted as characters.

In the first case, they can be read as bytes or byte arrays by the descendants of the `InputStream` and `OutputStream` classes, such as (omitting the package name if the class belongs to the `java.io` package) `ByteArrayInputStream`, `ByteArrayOutputStream`, `FileInputStream`, `FileOutputStream`, `ObjectInputStream`, `ObjectOutputStream`, `javax.sound.sampled.AudioInputStream`, and `org.omg.CORBA.portable.OutputStream`; which one you use depends on the source or destination of the data. The `InputStream` and `OutputStream` classes themselves are abstract and cannot be instantiated.

In the second case, data that can be interpreted as characters is called **text data**, and there are character-oriented reading and writing classes based on `Reader` and `Writer`, which are abstract classes too. Examples of their sub-classes are `CharArrayReader` and `CharArrayWriter`, `InputStreamReader` and `OutputStreamWriter`, `PipedReader` and `PipedWriter`, and `StringReader` and `StringWriter`.

You may have noticed that we listed the classes in pairs. But not every input class has a matching output specialization – for example, there are the `PrintStream` and `PrintWriter` classes that support output to a printing device, but there is no corresponding input partner, not by name at least. However, there is a `java.util.Scanner` class that parses input text in a known format.

There is also a set of buffer-equipped classes that help to improve performance by reading or writing a bigger chunk of data at a time, especially in cases when access to a source or destination takes a long time.

In the rest of this section, we will review classes of the `java.io` package and some popular related classes from other packages.

The InputStream class and its subclasses

In the Java Class Library, the `InputStream` abstract class has the following direct implementations: `ByteArrayInputStream`, `FileInputStream`, `ObjectInputStream`, `PipedInputStream`, `SequenceInputStream`, `FilterInputStream`, and `javax.sound.sampled.AudioInputStream`.

All of them can be used as they are or override the following methods of the `InputStream` class:

- `int available()`: Returns the number of bytes available for reading
- `void close()`: Closes the stream and releases the resources
- `void mark(int readlimit)`: Marks a position in the stream and defines how many bytes can be read
- `boolean markSupported()`: Returns `true` if the marking is supported
- `static InputStream nullInputStream()`: Creates an empty stream
- `abstract int read()`: Reads the next byte in the stream
- `int read(byte[] b)`: Reads data from the stream into the `b` buffer
- `int read(byte[] b, int off, int len)`: Reads `len` or fewer bytes from the stream into the `b` buffer
- `byte[] readAllBytes()`: Reads all the remaining bytes from the stream
- `int readNBytes(byte[] b, int off, int len)`: Reads `len` or fewer bytes into the `b` buffer at the `off` offset
- `byte[] readNBytes(int len)`: Reads `len` or fewer bytes into the `b` buffer
- `void reset()`: Resets the reading location to the position where the `mark()` method was last called
- `long skip(long n)`: Skips `n` or fewer bytes of the stream; returns the actual number of bytes skipped
- `long transferTo(OutputStream out)`: Reads from the input stream and writes to the provided output stream byte by byte; returns the actual number of bytes transferred

`abstract int read()` is the only method that has to be implemented, but most of the descendants of this class override many of the other methods too.

ByteArrayInputStream

The `ByteArrayInputStream` class allows reading a byte array as an input stream. It has the following two constructors that create an object of the class and define the buffer used to read the input stream of bytes:

- `ByteArrayInputStream(byte[] buffer)`
- `ByteArrayInputStream(byte[] buffer, int offset, int length)`

The second of the constructors allows you to set, in addition to the buffer, the offset and the length of the buffer too. Let's look at an example and see how this class can be used. We will assume there is a source of the `byte[]` array with data:

```
byte[] bytesSource(){
    return new byte[]{42, 43, 44};
}
```

Then, we can write the following:

```
byte[] buffer = bytesSource();
try(ByteArrayInputStream bais = new 
ByteArrayInputStream(buffer)){
    int data = bais.read();
    while(data != -1) {
        System.out.print(data + " ");   //prints: 42 43 44
        data = bais.read();
    }
} catch (Exception ex){
    ex.printStackTrace();
}
```

The `bytesSource()` method produces the array of bytes that fills the buffer that is passed into the constructor of the `ByteArrayInputStream` class as a parameter. The resulting stream is then read byte by byte using the `read()` method until the end of the stream is reached (and the `read()` method returns `-1`). Each new byte is printed out (without a line feed and with white space after it, so all the read bytes are displayed in one line separated by the white space).

The preceding code is usually expressed in a more compact form, as follows:

```
byte[] buffer = bytesSource();
try(ByteArrayInputStream bais = new
ByteArrayInputStream(buffer)){
    int data;
    while ((data = bais.read()) != -1) {
        System.out.print(data + " ");   //prints: 42 43 44
    }
} catch (Exception ex){
    ex.printStackTrace();
}
```

Instead of just printing the bytes, they can be processed in any other manner necessary, including interpreting them as characters, such as the following:

```
byte[] buffer = bytesSource();
try(ByteArrayInputStream bais =
                             new ByteArrayInputStream(buffer)){
  int data;
  while ((data = bais.read()) != -1) {
      System.out.print(((char)data) + " ");   //prints: * + ,
  }
} catch (Exception ex){
    ex.printStackTrace();
}
```

But, in such a case, it is better to use one of the `Reader` classes that are specialized for character processing. We will talk about them in the *Reader and writer classes and their subclasses* section.

FileInputStream

The `FileInputStream` class gets data from a file in a filesystem – the raw bytes of an image, for example. It has the following three constructors:

- `FileInputStream(File file)`
- `FileInputStream(String name)`
- `FileInputStream(FileDescriptor fdObj)`

Each constructor opens the file specified as the parameter. The first constructor accepts the `File` object; the second, the path to the file in the filesystem; and the third, the file descriptor object that represents an existing connection to an actual file in the filesystem. Let's look at the following example:

```
String file =  classLoader.getResource("hello.txt").getFile();
try(FileInputStream fis = new FileInputStream(file)){
    int data;
    while ((data = fis.read()) != -1) {
        System.out.print(((char)data) + " ");
                                           //prints: H e l l o !
    }
} catch (Exception ex){
    ex.printStackTrace();
}
```

In the `src/main/resources` folder, we have created the `hello.txt` file that has only one line in it – `Hello!`. The output of the preceding example looks as follows:

```
H e l l o !
```

After reading bytes from the `hello.txt` file, we decided, for demo purposes, to cast each `byte` to `char` so that you can see that the code does read from the specified file, but the `FileReader` class is a better choice for text file processing (we will discuss it shortly). Without the cast, the result would be the following:

```
System.out.print((data) + " ");
                                 //prints: 72 101 108 108 111 33
```

By the way, because the `src/main/resources` folder is placed by the IDE (using Maven) on the classpath, a file placed in it can also be accessed via a class loader that creates a stream using its own `InputStream` implementation:

```
try(InputStream is = InputOutputStream.class.
getResourceAsStream("/hello.txt")){
    int data;
    while ((data = is.read()) != -1) {
        System.out.print((data) + " ");
             //prints: 72 101 108 108 111 33
    }
} catch (Exception ex){
    ex.printStackTrace();
}
```

The `InputOutputStream` class in the preceding example is not a class from some library. It is just the main class we used to run the example. The `InputOutputStream.class.getResourceAsStream()` construct allows you to use the same classloader that has loaded the `InputOutputStream` class for the purpose of finding a file on the classpath and creating a stream that contains its content. In the *File management* section, we will present other ways of reading a file too.

ObjectInputStream

The set of methods of the `ObjectInputStream` class is much bigger than the set of methods of any other `InputStream` implementation. The reason for that is that it is built around reading the values of the object fields that can be of various types. In order for `ObjectInputStream` to be able to construct an object from the input stream of data, the object has to be *deserializable*, which means it has to be *serializable* in the first place – that is, to be convertible into a byte stream. Usually, it is done for the purpose of transporting objects over a network. At the destination, the serialized objects are deserialized, and the values of the original objects are restored.

Primitive types and most Java classes, including the `String` class and primitive type wrappers, are serializable. If a class has fields of custom types, they have to be made serializable by implementing `java.io.Serizalizable`. How to do that is outside the scope of this book. For now, we are going to use only the serializable types. Let's look at this class:

```
class SomeClass implements Serializable {
    private int field1 = 42;
    private String field2 = "abc";
}
```

We have to tell the compiler that it is serializable. Otherwise, the compilation will fail. It is done in order to make sure that, before stating that the class is serializable, the programmer either reviewed all the fields and made sure they are serializable or has implemented the methods necessary for the serialization.

Before we can create an input stream and use `ObjectInputStream` for deserialization, we need to serialize the object first. That is why we first use `ObjectOutputStream` and `FileOutputStream` to serialize an object and write it into the `someClass.bin` file. We will talk more about them in the *The OutputStream class and its subclasses* section. Then, we read from the file using `FileInputStream` and deserialize the file content using `ObjectInputStream`:

```
String fileName = "someClass.bin";
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(fileName));
     ObjectInputStream ois = new ObjectInputStream(new
                             FileInputStream(fileName))){
    SomeClass obj = new SomeClass();
    oos.writeObject(obj);
    SomeClass objRead = (SomeClass) ois.readObject();
    System.out.println(objRead.field1);  //prints: 42
    System.out.println(objRead.field2);  //prints: abc
} catch (Exception ex){
    ex.printStackTrace();
}
```

Note that the file has to be created first before the preceding code is run. We will show how it can be done in the *Creating files and directories* section. And, as a reminder, we have used the `try-with-resources` statement because `InputStream` and `OutputStream` both implement the `Closeable` interface.

PipedInputStream

A piped input stream has a very particular specialization; it is used as one of the mechanisms of communication between threads. One thread reads from a `PipedInputStream` object and passes data to another thread that writes data to a `PipedOutputStream` object. Here is an example:

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
```

The `javax.sound.sampled.AudioFormat` class describes audio-format properties such as channels, encoding, frame rate, and similar. The `javax.sound.sampled.TargetDataLine` class has the `open()` method that opens the line with the specified format and the `read()` method that reads audio data from the data line's input buffer.

There is also the `javax.sound.sampled.AudioSystem` class, and its methods handle `AudioInputStream` objects. They can be used for reading from an audio file, a stream, or a URL, and they write to an audio file. They also can be used to convert an audio stream to another audio format.

The OutputStream class and its subclasses

The `OutputStream` class is a peer of the `InputStream` class that writes data instead of reading. It is an abstract class that has the following direct implementations in the **Java Class Library** (**JCL**): `ByteArrayOutputStream`, `FilterOutputStream`, `ObjectOutputStream`, `PipedOutputStream`, and `FileOutputStream`.

The `FileOutputStream` class has the following direct extensions: `BufferedOutputStream`, `CheckedOutputStream`, `DataOutputStream`, `PrintStream`, `javax.crypto.CipherOutputStream`, `java.util.zip.DeflaterOutputStream`, `java.security.DigestOutputStream`, and `java.util.zip.InflaterOutputStream`.

All of them can be used as they are or override the following methods of the `OutputStream` class:

- `void close()`: Closes the stream and releases the resources
- `void flush()`: Forces the remaining bytes to be written out
- `static OutputStream nullOutputStream()`: Creates a new `OutputStream` that writes nothing
- `void write(byte[] b)`: Writes the provided byte array to the output stream
- `void write(byte[] b, int off, int len)`: Writes `len` bytes of the provided byte array, starting at the `off` offset, to the output stream
- `abstract void write(int b)`: Writes the provided byte to the output stream

The only method that has to be implemented is `abstract void write(int b)`, but most of the descendants of the `OutputStream` class override many of the other methods too.

After learning about the input streams in the *The InputStream class and its subclasses* section, all of the `OutputStream` implementations, except the `PrintStream` class, should be intuitively familiar to you. So, we will discuss here only the `PrintStream` class.

PrintStream

The `PrintStream` class adds to another output stream the ability to print data as characters. We have actually used it already many times. The `System` class has an object of the `PrintStream` class set as a `System.out` public static property. This means that every time we print something using `System.out`, we are using the `PrintStream` class:

```
System.out.println("Printing a line");
```

Let's look at another example of the `PrintStream` class usage:

```
String fileName = "output.txt";
try(FileOutputStream  fos = new FileOutputStream(fileName);
    PrintStream ps = new PrintStream(fos)){
    ps.println("Hi there!");
} catch (Exception ex) {
    ex.printStackTrace();
}
```

As you can see, the `PrintStream` class takes the `FileOutputStream` object and prints the characters generated by it. In this case, it prints out all the bytes that `FileOutputStream` writes to the file. By the way, there is no need to create the destination file explicitly. If absent, it will be created automatically inside the `FileOutputStream` constructor. If we open the file after the preceding code is run, we will see one line in it – `"Hi there!"`.

Alternatively, the same result can be achieved using another `PrintStream` constructor that takes the `File` object, as follows:

```
String fileName = "output.txt";
File file = new File(fileName);
try(PrintStream ps = new PrintStream(file)){
    ps.println("Hi there!");
} catch (Exception ex) {
    ex.printStackTrace();
}
```

An even simpler solution can be created using the third variation of the `PrintStream` constructor that takes the filename as a parameter:

```
String fileName = "output.txt";
try(PrintStream ps = new PrintStream(fileName)){
    ps.println("Hi there!");
} catch (Exception ex) {
    ex.printStackTrace();
}
```

The previous two examples are possible because the `PrintStream` constructor uses the `FileOutputStream` class behind the scenes, exactly as we did it in the first example of the `PrintStream` class usage. So, the `PrintStream` class has several constructors just for convenience, but all of them essentially have the same functionality:

- `PrintStream(File file)`
- `PrintStream(File file, String csn)`
- `PrintStream(File file, Charset charset)`
- `PrintStream(String fileName)`
- `PrintStream(String fileName, String csn)`
- `PrintStream(String fileName, Charset charset)`
- `PrintStream(OutputStream out)`
- `PrintStream(OutputStream out, boolean autoFlush)`
- `PrintStream(OutputStream out, boolean autoFlush, String encoding)`
- `PrintStream(OutputStream out, boolean autoFlush, Charset charset)`

Some of the constructors also take a `Charset` instance or just its name (`String csn`), which allows you to apply a different mapping between sequences of 16-bit Unicode code units and sequences of bytes. You can see all available charsets by just printing them out, as shown here:

```
for (String chs : Charset.availableCharsets().keySet()) {
    System.out.println(chs);
}
```

Other constructors take `boolean autoFlush` as a parameter. This parameter indicates (when `true`) that the output buffer should be flushed automatically when an array is written or the symbol end-of-line is encountered.

Once an object of `PrintStream` is created, it provides a variety of methods, as listed here:

- `void print(T value)`: Prints the value of any `T` primitive type passed in without moving to another line
- `void print(Object obj)`: Calls the `toString()` method on the passed in object and prints the result without moving to another line; does not generate `NullPointerException` in case the passed-in object is `null` and prints `null` instead
- `void println(T value)`: Prints the value of any `T` primitive type passed in and moves to another line
- `void println(Object obj)`: Calls the `toString()` method on the passed-in object, prints the result, and moves to another line; does not generate `NullPointerException` in case the passed-in object is `null` and prints `null` instead
- `void println()`: Just moves to another line
- `PrintStream printf(String format, Object... values)`: Substitutes the placeholders in the provided `format` string with the provided `values` and writes the result into the stream
- `PrintStream printf(Locale l, String format, Object... args)`: The same as the preceding method but applies localization using the provided `Local` object; if the provided `Local` object is `null`, no localization is applied, and this method behaves exactly like the preceding one
- `PrintStream format(String format, Object... args)` and `PrintStream format(Locale l, String format, Object... args)`: Behaves the same way as `PrintStream printf(String format, Object... values)` and `PrintStream printf(Locale l, String format, Object... args)` (already described in the list), such as the following:

  ```
  System.out.printf("Hi, %s!%n", "dear reader");
                                 //prints: Hi, dear reader!
  System.out.format("Hi, %s!%n", "dear reader");
                                 //prints: Hi, dear reader!
  ```

In the preceding example, (`%`) indicates a formatting rule. The following symbol (`s`) indicates a `String` value. Other possible symbols in this position can be (`d`) (decimal), (`f`) (floating-point), and so on. The symbol (`n`) indicates a new line (the same as the (`\n`) escape character). There are many formatting rules. All of them are described in the documentation for the `java.util.Formatter` class.

- `PrintStream append(char c)`, `PrintStream append(CharSequence c)`, and `PrintStream append(CharSequence c, int start, int end)`: Appends the provided character to the stream, such as the following:

```
System.out.printf("Hi %s", "there").append("!\n");
                                          //prints: Hi there!
System.out.printf("Hi ")
                .append("one there!\n two", 4, 11);
                                          //prints: Hi there!
```

With this, we conclude the discussion of the `OutputStream` subclass and now turn our attention to another class hierarchy – the `Reader` and `Writer` classes and their subclasses from the JCL.

The Reader and Writer classes and their subclasses

As we mentioned several times already, the `Reader` and `Writer` classes are very similar in their function to the `InputStream` and `OutputStream` classes but specialize in processing texts. They interpret stream bytes as characters and have their own independent `InputStream` and `OutputStream` class hierarchy. It is possible to process stream bytes as characters without `Reader` and `Writer` or any of their subclasses. We have seen such examples in the preceding sections describing the `InputStream` and `OutputStream` classes. However, using the `Reader` and `Writer` classes makes text processing simpler and code easier to read.

Reader and its subclasses

The `Reader` class is an abstract class that reads streams as characters. It is an analog to `InputStream` and has the following methods:

- `abstract void close()`: Closes the stream and other used resources
- `void mark(int readAheadLimit)`: Marks the current position in the stream
- `boolean markSupported()`: Returns `true` if the stream supports the `mark()` operation

- `static Reader nullReader()`: Creates an empty `Reader` that reads no characters
- `int read()`: Reads one character
- `int read(char[] buf)`: Reads characters into the provided `buf` array and returns the count of the read characters
- `abstract int read(char[] buf, int off, int len)`: Reads the `len` characters into an array starting from the `off` index
- `int read(CharBuffer target)`: Attempts to read characters into the provided `target` buffer
- `boolean ready()`: Returns `true` when the stream is ready to be read
- `void reset()`: Resets the mark; however, not all streams support this operation, with some supporting it but not supporting a mark being set in the first place
- `long skip(long n)`: Attempts to skip the `n` characters; returns the count of skipped characters
- `long transferTo(Writer out)`: Reads all characters from this reader and writes the characters to the provided `Writer` object

As you can see, the only methods that need to be implemented are the two abstract `read()` and `close()` methods. Nevertheless, many children of this class override other methods too, sometimes for better performance or different functionality. The `Reader` subclasses in the JCL are `CharArrayReader`, `InputStreamReader`, `PipedReader`, `StringReader`, `BufferedReader`, and `FilterReader`. The `BufferedReader` class has a `LineNumberReader` subclass, and the `FilterReader` class has a `PushbackReader` subclass.

Writer and its subclasses

The abstract `Writer` class writes to character streams. It is an analog to `OutputStream` and has the following methods:

- `Writer append(char c)`: Appends the provided character to the stream
- `Writer append(CharSequence c)`: Appends the provided character sequence to the stream
- `Writer append(CharSequence c, int start, int end)`: Appends a subsequence of the provided character sequence to the stream
- `abstract void close()`: Flushes and closes the stream and related system resources

- `abstract void flush()`: Flushes the stream
- `static Writer nullWriter()`: Creates a new `Writer` object that discards all characters
- `void write(char[] c)`: Writes an array of `c` characters
- `abstract void write(char[] c, int off, int len)`: Writes the `len` elements of an array of `c` characters, starting from the `off` index
- `void write(int c)`: Writes one character
- `void write(String str)`: Writes the provided string
- `void write(String str, int off, int len)`: Writes a substring of the `len` length from the provided `str` string, starting from the `off` index

As you can see, the three abstract methods, `write(char[], int, int)`, `flush()`, and `close()`, must be implemented by the children of this class. They also typically override other methods too.

The `Writer` subclasses in the JCL are `CharArrayWriter`, `OutputStreamWriter`, `PipedWriter`, `StringWriter`, `BufferedWriter`, `FilterWriter`, and `PrintWriter`. The `OutputStreamWriter` class has a `FileWriter` subclass.

Other classes of the java.io package

Other classes of the `java.io` package include the following:

- `Console`: Allows interaction with the character-based console device, associated with the current Java Virtual Machine instance
- `StreamTokenizer`: Takes an input stream and parses it into `tokens`
- `ObjectStreamClass`: The serialization's descriptor for classes
- `ObjectStreamField`: A description of a serializable field from a serializable class
- `RandomAccessFile`: Allows random access for reading from and writing to a file, but its discussion is outside the scope of this book
- `File`: Allows creating and managing files and directories; described in the *File management* section

Console

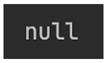
There are several ways to create and run a **Java Virtual Machine** (**JVM**) instance that executes an application. If the JVM is started from a command line, a console window is automatically opened. It allows you to type on the display from the keyboard; however, the JVM can be started by a background process too. In such a case, a console is not created.

To check programmatically whether a console exists, you can invoke the `System.console()` static method. If no console device is available, then an invocation of that method will return `null`. Otherwise, it will return an object of the `Console` class that allows interaction with the console device and the application user.

Let's create the following `ConsoleDemo` class:

```
package com.packt.learnjava.ch05_stringsIoStreams;
import java.io.Console;
public class ConsoleDemo {
    public static void main(String... args)  {
        Console console = System.console();
        System.out.println(console);
    }
}
```

If we run it from the IDE, as we usually do, the result will be as follows:



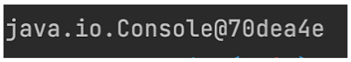
```
null
```

That is because the JVM was not launched from the command line. In order to do it, let's compile our application and create a `.jar` file by executing the `mvn clean package` Maven command in the root directory of the project. (We assume that you have Maven installed on your computer.) It will delete the `target` folder, then recreate it, compile all the `.java` files to the corresponding `.class` files in the `target` folder, and then archive them in a `.jar` file, `learnjava-1.0-SNAPSHOT.jar`.

Now, we can launch the `ConsoleDemo` application from the same project root directory using the following command:

```
java -cp ./target/examples-1.0-SNAPSHOT.jar
          com.packt.learnjava.ch05_stringsIoStreams.ConsoleDemo
```

The preceding `-cp` command option depicts a classpath, so in our case, we tell the JVM to look for the classes in the `.jar` file in the folder target. The command is shown in two lines because the page width cannot accommodate it. But if you want to run it, make sure you do it as one line. The result will be as follows:



```
java.io.Console@70dea4e
```

This tells us that we have the `Console` class object now. Let's see what we can do with it. The class has the following methods:

- `String readLine()`: Waits until the user hits the *Enter* key and reads the line of text from the console
- `String readLine(String format, Object... args)`: Displays a prompt (the message produced after the provided format had the placeholders substituted with the provided arguments), waits until the user hits the *Enter* key, and reads the line of text from the console; if no arguments (`args`) are provided, it displays the format as the prompt
- `char[] readPassword()`: Performs the same function as the `readLine()` function but without echoing the typed characters
- `char[] readPassword(String format, Object... args)`: Performs the same function as `readLine(String format, Object... args)` but without echoing the typed characters

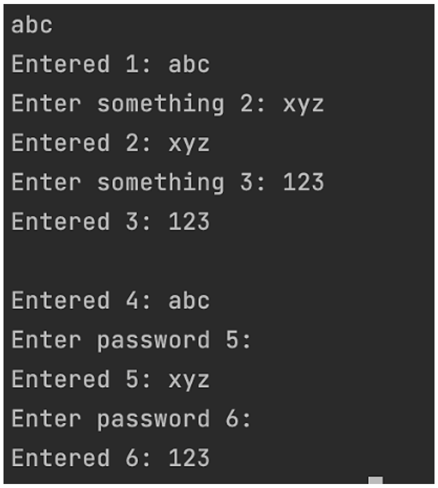
To run each of the following code sections individually, you need to comment out the `console1()` call in the `main` method and uncomment `console2()` or `console3()`, recompile using `mvn package`, and then rerun the `java` command shown previously.

Let's demonstrate the preceding methods with the following example (the `console2()` method):

```
Console console = System.console();

System.out.print("Enter something 1: ");
String line = console.readLine();
System.out.println("Entered 1: " + line);
line = console.readLine("Enter something 2: ");
System.out.println("Entered 2: " + line);
line = console.readLine("Enter some%s", "thing 3: ");
System.out.println("Entered 3: " + line);
System.out.print("Enter password: ");
char[] password = console.readPassword();
System.out.println("Entered 4: " + new String(password));
password = console.readPassword("Enter password 5: ");
System.out.println("Entered 5: " + new String(password));
password = console.readPassword("Enter pass%s", "word 6: ");
System.out.println("Entered 6: " + new String(password));
```

The result of the preceding example is as follows:



```
abc
Entered 1: abc
Enter something 2: xyz
Entered 2: xyz
Enter something 3: 123
Entered 3: 123

Entered 4: abc
Enter password 5:
Entered 5: xyz
Enter password 6:
Entered 6: 123
```

Some IDEs cannot run these examples and throw `NullPointerException`. If that is the case, run the console-related examples from the command line, as described previously. Don't forget to run the `maven package` command every time you change code.

Another group of `Console` class methods can be used in conjunction with the previously demonstrated methods:

- `Console format(String format, Object... args)`: Substitutes the placeholders in the provided `format` string with the provided `args` values and displays the result
- `Console printf(String format, Object... args)`: Behaves the same way as the `format()` method

As an example, look at the following line:

```
String line = console.format("Enter some%s", "thing:").
readLine();
```

It produces the same result as this line:

```
String line = console.readLine("Enter some%s", "thing:");
```

And finally, the last three methods of the `Console` class are as follows:

- `PrintWriter writer()`: Creates a `PrintWriter` object associated with this console that can be used for producing an output stream of characters
- `Reader reader()`: Creates a `Reader` object associated with this console that can be used for reading the input as a stream of characters
- `void flush()`: Flushes the console and forces any buffered output to be written immediately

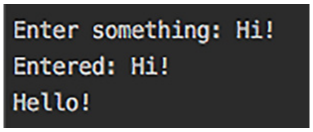
Here is an example of their usage (the `console3()` method):

```
try (Reader reader = console.reader()){
    char[] chars = new char[10];
    System.out.print("Enter something: ");
    reader.read(chars);
    System.out.print("Entered: " + new String(chars));
} catch (IOException e) {
    e.printStackTrace();
}

PrintWriter out = console.writer();
out.println("Hello!");

console.flush();
```

The result of the preceding code looks as follows:



```
Enter something: Hi!
Entered: Hi!
Hello!
```

`Reader` and `PrintWriter` can also be used to create other `Input` and `Output` streams that we have been talking about in this section.

As you can see, using the wealth of the preceding methods allows you to fine-tune the text interpretation.

ObjectStreamClass and ObjectStreamField

The `ObjectStreamClass` and `ObjectStreamField` classes provide access to the serialized data of a class loaded in the JVM. The `ObjectStreamClass` object can be found/created using one of the following lookup methods:

- `static ObjectStreamClass lookup(Class cl)`: Finds the descriptor of a serializable class
- `static ObjectStreamClass lookupAny(Class cl)`: Finds the descriptor for any class, whether serializable or not

After `ObjectStreamClass` is found and the class is serializable (implementing the `Serializable` interface), it can be used to access the `ObjectStreamField` objects, each containing information about one serialized field. If the class is not serializable, there is no `ObjectStreamField` object associated with any of the fields.

Let's look at an example. Here is the method that displays information obtained from the `ObjectStreamClass` and `ObjectStreamField` objects:

```
void printInfo(ObjectStreamClass osc) {
    System.out.println(osc.forClass());
    System.out.println("Class name: " + osc.getName());
    System.out.println("SerialVersionUID: " +
                               osc.getSerialVersionUID());
    ObjectStreamField[] fields = osc.getFields();
    System.out.println("Serialized fields:");
    for (ObjectStreamField osf : fields) {
        System.out.println(osf.getName() + ": ");
        System.out.println("\t" + osf.getType());
        System.out.println("\t" + osf.getTypeCode());
        System.out.println("\t" + osf.getTypeString());
    }
}
```

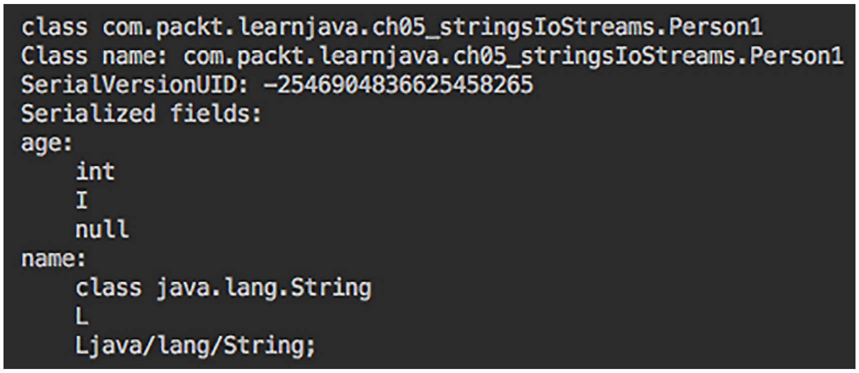
To demonstrate how it works, we will create a serializable `Person1` class:

```
package com.packt.learnjava.ch05_stringsIoStreams;
import java.io.Serializable;
public class Person1 implements Serializable {
    private int age;
    private String name;
    public Person1(int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

We did not add methods because only the object state is serializable, not the methods. Now, let's run the following code:

```
ObjectStreamClass osc1 =
        ObjectStreamClass.lookup(Person1.class);
printInfo(osc1);
```

The result will be as follows:



```
class com.packt.learnjava.ch05_stringsIoStreams.Person1
Class name: com.packt.learnjava.ch05_stringsIoStreams.Person1
SerialVersionUID: -2546904836625458265
Serialized fields:
age:
    int
    I
    null
name:
    class java.lang.String
    L
    Ljava/lang/String;
```

As you can see, there is information about the class name and all field names and types. There are also two other methods that can be called using the `ObjectStreamField` object:

- `boolean isPrimitive()`: Returns `true` if this field has a primitive type
- `boolean isUnshared()`: Returns `true` if this field is unshared (private or accessible only from the same package)

Now, let's create a non-serializable `Person2` class:

```
package com.packt.learnjava.ch05_stringsIoStreams;
public class Person2 {
    private int age;
    private String name;
    public Person2(int age, String name) {
        this.age = age;
        this.name = name;
    }
}
```

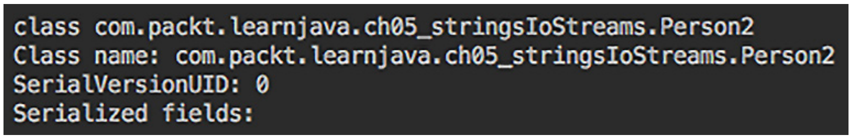
This time, we will run the code that only looks up the class, as follows:

```
ObjectStreamClass osc2 =
             ObjectStreamClass.lookup(Person2.class);
System.out.println("osc2: " + osc2);    //prints: null
```

As expected, the non-serializable object was not found using the `lookup()` method. In order to find a non-serializable object, we need to use the `lookupAny()` method:

```
ObjectStreamClass osc3 =
           ObjectStreamClass.lookupAny(Person2.class);
printInfo(osc3);
```

If we run the preceding example, the result will be as follows:



```
class com.packt.learnjava.ch05_stringsIoStreams.Person2
Class name: com.packt.learnjava.ch05_stringsIoStreams.Person2
SerialVersionUID: 0
Serialized fields:
```

From a non-serializable object, we were able to extract information about the class but not about the fields.

The java.util.Scanner class

The `java.util.Scanner` class is typically used to read input from a keyboard but can also read text from any object that implements the `Readable` interface (this interface only has the `int read(CharBuffer buffer)` method). It breaks the input value by a delimiter (white space is a default delimiter) into tokens that are processed using different methods.

File management

We have already used some methods for finding, creating, reading, and writing files using the JCL classes. We had to do it in order to support a demo code of input/output streams. In this section, we are going to talk about file management using the JCL in more detail.

The `File` class from the `java.io` package represents the underlying filesystem. An object of the `File` class can be created with one of the following constructors:

- `File(String pathname)`: Creates a new `File` instance based on the provided pathname
- `File(String parent, String child)`: Creates a new `File` instance based on the provided parent pathname and a child pathname
- `File(File parent, String child)`: Creates a new `File` instance based on the provided parent `File` object and a child pathname
- `File(URI uri)`: Creates a new `File` instance based on the provided `URI` object that represents the pathname

We will now see some examples of the constructors' usage while talking about creating and deleting files.

Creating and deleting files and directories

To create a file or directory in the filesystem, you need first to construct a new `File` object using one of the constructors listed in the *File management* section. For example, assuming that the filename is `FileName.txt`, the `File` object can be created as `new File("FileName.txt")`. If the file has to be created inside a directory, then either a path has to be added in front of the filename (when it is passed into the constructor) or one of the other three constructors has to be used, such as the following (see the `createFile2()` method in the `Files` class):

```
String path = "demo1" + File.separator +
                            "demo2" + File.separator;
String fileName = "FileName.txt";
File f = new File(path + fileName);
```

Note the usage of `File.separator` instead of the slash symbols, (/) or (\). That is because `File.separator` returns the platform-specific slash symbol. Here is an example of another `File` constructor usage:

```
String path = "demo1" + File.separator +
                            "demo2" + File.separator;
String fileName = "FileName.txt";
File f = new File(path, fileName);
```

Yet another constructor can be used as follows:

```
String path = "demo1" + File.separator +
                            "demo2" + File.separator;
String fileName = "FileName.txt";
File f = new File(new File(path), fileName);
```

However, if you prefer or have to use a **Universal Resource Identifier** (**URI**), you can construct a `File` object like this:

```
String path = "demo1" + File.separator +
                            "demo2" + File.separator;
String fileName = "FileName.txt";
URI uri = new File(path + fileName).toURI();
File f = new File(uri);
```

Then, one of the following methods has to be invoked on the newly created `File` object:

- `boolean createNewFile()`: If a file with this name does not yet exist, creates a new file and returns `true`; otherwise, returns `false`
- `static File createTempFile(String prefix, String suffix)`: Creates a file in the temporary-file directory
- `static File createTempFile(String prefix, String suffix, File directory)`: Creates the directory; the provided prefix and suffix are used to generate the directory name

If the file you would like to create has to be placed inside a directory that does not exist yet, one of the following methods has to be used first, invoked on the `File` object that represents the filesystem path to the file:

- `boolean mkdir()`: Creates the directory with the provided name
- `boolean mkdirs()`: Creates the directory with the provided name, including any necessary but nonexistent parent directories

Before we look at a code example, we need to explain how the `delete()` method works:

- `boolean delete()`: Deletes the file or empty directory, which means you can delete the file but not all of the directories, as follows:

  ```
  String path = "demo1" + File.separator +
                              "demo2" + File.separator;
  String fileName = "FileName.txt";
  File f = new File(path + fileName);
  f.delete();
  ```

Let's look at how to overcome this limitation in the following example:

```
String path = "demo1" + File.separator +
                            "demo2" + File.separator;
String fileName = "FileName.txt";
File f = new File(path + fileName);
try {
    new File(path).mkdirs();
    f.createNewFile();
    f.delete();
    path = StringUtils
            .substringBeforeLast(path, File.separator);
    while (new File(path).delete()) {
        path = StringUtils
             .substringBeforeLast(path, File.separator);
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

This example creates and deletes a file and all related directories. Notice our usage of the `org.apache.commons.lang3.StringUtils` class, which we discussed in the *String utilities* section. It allowed us to remove from the path the just-deleted directory and to continue doing it until all the nested directories are deleted, and the top-level directory is deleted last.

Listing files and directories

The following methods can be used for listing directories and the files in them:

- `String[] list()`: Returns the names of the files and directories in the directory
- `File[] listFiles()`: Returns the `File` objects that represent the files and directories in the directory
- `static File[] listRoots()`: Lists the available filesystem roots

In order to demonstrate the preceding methods, let's assume we have created the directories and two files in them, as follows:

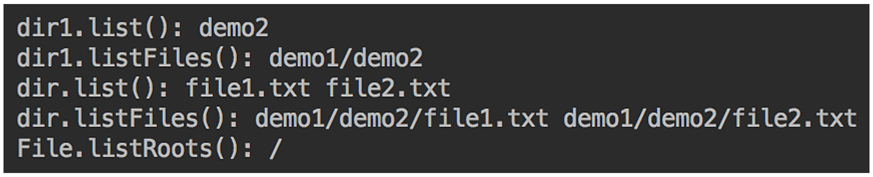
```
String path1 = "demo1" + File.separator;
String path2 = "demo2" + File.separator;
String path = path1 + path2;
File f1 = new File(path + "file1.txt");
File f2 = new File(path + "file2.txt");
File dir1 = new File(path1);
File dir = new File(path);
dir.mkdirs();
f1.createNewFile();
f2.createNewFile();
```

After that, we should be able to run the following code:

```
System.out.print("\ndir1.list(): ");
for(String d: dir1.list()){
    System.out.print(d + " ");
}
System.out.print("\ndir1.listFiles(): ");
for(File f: dir1.listFiles()){
    System.out.print(f + " ");
}
```

```
System.out.print("\ndir.list(): ");
for(String d: dir.list()){
    System.out.print(d + " ");
}
System.out.print("\ndir.listFiles(): ");
for(File f: dir.listFiles()){
    System.out.print(f + " ");
}
System.out.print("\nFile.listRoots(): ");
for(File f: File.listRoots()){
    System.out.print(f + " ");
}
```

The result should be as follows:



```
dir1.list(): demo2
dir1.listFiles(): demo1/demo2
dir.list(): file1.txt file2.txt
dir.listFiles(): demo1/demo2/file1.txt demo1/demo2/file2.txt
File.listRoots(): /
```

The demonstrated methods can be enhanced by adding the following filters to them so that they will list only the files and directories that match the filter:

- `String[] list(FilenameFilter filter)`
- `File[] listFiles(FileFilter filter)`
- `File[] listFiles(FilenameFilter filter)`

However, a discussion of the file filters is outside the scope of this book.

Apache Commons' FileUtils and IOUtils utilities

A popular companion of JCL is the Apache Commons project (`https://commons.apache.org`), which provides many libraries that complement the JCL functionality. The classes of the `org.apache.commons.io` package are contained in the following root package and sub-packages:

- The `org.apache.commons.io` root package contains utility classes with static methods for common tasks, such as the popular `FileUtils` and `IOUtils` classes, described in the *FileUtils class* and *Class IOUtils class* sections respectively.