Java I/O: Part I 20

Chapter Topics

- Understanding the concept of I/O streams to read data from an input source and write data to an output destination
- Understanding the hierarchy of byte streams, as represented by the `InputStream` and `OutputStream` abstract classes, to handle data as bytes
- Reading and writing bytes using file streams, as represented by the `FileInputStream` and `FileOutputStream` classes
- Reading and writing binary files using the `DataInputStream` and `DataOutputStream` classes
- Understanding the hierarchy of character streams, as represented by the `Reader` and `Writer` abstract classes
- Using character encodings, including Unicode and UTF-8, with the `InputStreamReader` and `OutputStreamWriter` classes
- Reading and writing text files using the `FileReader` and `PrintWriter` classes
- Using buffered character streams for efficient reading and writing of characters, as represented by the `BufferedReader` and `Buffered-Writer` classes
- Understanding the standard input, output, and error streams, as represented by the fields `System.in`, `System.out`, and `System.err`, respectively
- Using the `Console` class that allows character-based input and output from a console
- Performing object serialization/de-serialization—that is, writing and reading binary representation of objects, using the `ObjectOutputStream` and `ObjectInputStream` classes
- Applying selective serialization, exploiting customized serialization, using class versioning, and understanding how the inheritance relationship can affect serialization

Java SE 17 Developer Exam Objectives	
[9.1] Read and write console and file data using I/O Streams	§20.1, p. 1233 *to* §20.4, p. 1256

[9.2] Serialize and de-serialize Java objects	§20.5, p. 1261

Java SE 11 Developer Exam Objectives	
[9.1] Read and write console and file data using I/O Streams	§20.1, p. 1233 *to* §20.4, p. 1256
[9.2] Implement serialization and deserialization techniques on Java objects	§20.5, p. 1261

The `java.io` package provides an extensive library of classes for dealing with input and output of data. Although data can be read from various sources and written to various destinations, the emphasis in this chapter is on using the standard I/O API provided by the `java.io` package for reading and writing various kinds of data to *files*: bytes, characters, binary files, text files, buffered I/O, and object serialization.

The `java.io` package also provides a general interface to interact with the file system of the host platform, but we will use the newer and enhanced I/O API in the `java.nio.file` package that provides access to files, file attributes, and file systems (**Chapter 21**, **p. 1285**).

In this chapter, we assume a basic understanding of a *hierarchical file system* in which files and directories are referenced by a *string* that specifies the *path* of an entity in the file system. The idiosyncrasies of file systems on different platforms are explored in **Chapter 21**, **p. 1285**.

It is important to note that the term *stream* in this chapter refers to *I/O streams*. Streams that provide functional-style operations on sequences of elements are discussed in **Chapter 16**, **p. 879**.

20.1 Input and Output

Java provides *I/O streams* as a general mechanism for dealing with data input and output. I/O streams implement *sequential processing* of data. An *input stream* allows an application to read a sequence of data, and an *output stream* allows an application to write a sequence of data. An *input stream* acts as a *source* of data, and an *output*

stream acts as a *destination* of data. The following entities can act as both input and output streams:

- A file—which is the focus in this chapter
- An array of bytes or characters
- A network connection

There are two categories of I/O streams:

- *Byte streams* that process *bytes* as a unit of data
- *Character streams* that process *characters* as a unit of data

A *low-level I/O stream* operates directly on the data source (e.g., a file or an array of bytes), and processes the data primarily as *bytes*.

A *high-level I/O stream* is *chained* to an underlying stream, and provides additional capabilities for processing data managed by the underlying stream—for example, processing bytes from the underlying stream as Java primitive values or objects. In other words, a high-level I/O stream acts as a *wrapper* for the underlying stream.

In the rest of this chapter we primarily explore how to use I/O streams of the standard I/O API provided by the `java.io` package to read and write various kinds of data that is stored in files.

20.2 Byte Streams: Input Streams and Output Streams

The abstract classes `InputStream` and `OutputStream` in the `java.io` package are the root of the inheritance hierarchies for handling the reading and writing of data as *bytes* (**Figure 20.1**). Their subclasses, implementing different kinds of input and output (I/O) streams, override the following methods from the `InputStream` and `OutputStream` classes to customize the reading and writing of bytes, respectively:

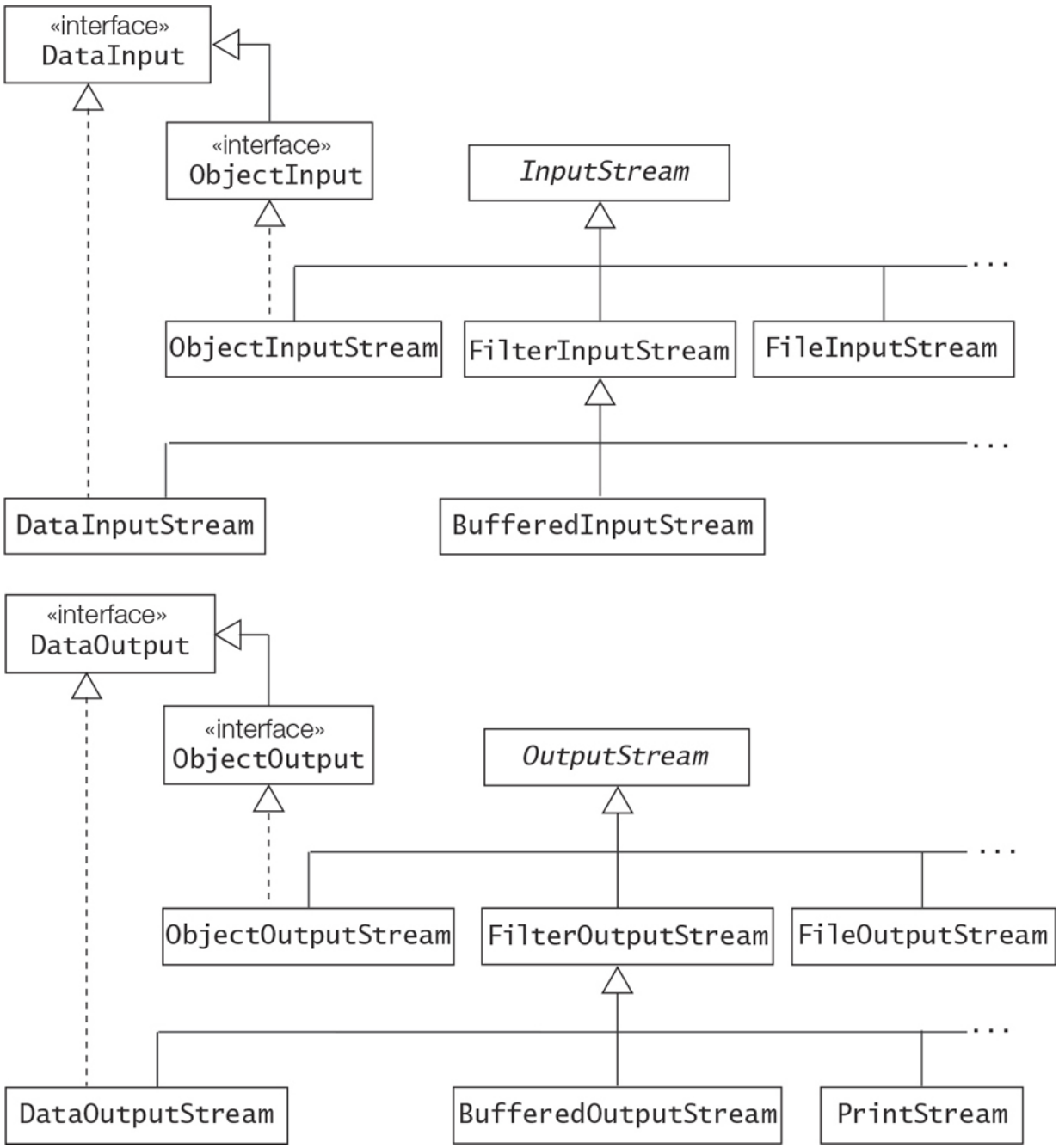


Figure 20.1 *Partial Byte Stream Inheritance Hierarchies in the* `java.io` *Package*

From the InputStream abstract class:

Click here to view code image

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

Note that the first `read()` method reads a *byte*, but returns an `int` value. The byte read resides in the eight least significant bits of the `int` value, while the remaining bits in the `int` value are zeroed out. The `read()` methods return the value –1 when the end of the input stream is reached.

Click here to view code image

```
long transferTo(OutputStream out) throws IOException
```

Reads bytes from this input stream and writes to the specified output stream in the order they are read. The I/O streams are *not closed* after the operation (see below).

From the `OutputStream` abstract class:

Click here to view code image

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

The first `write()` method takes an `int` as an argument, but truncates it to the eight least significant bits before writing it out as a byte to the output stream.

Click here to view code image

```
void close() throws IOException        Both InputStream and OutputStream
void flush() throws IOException        Only for OutputStream
```

A I/O stream should be *closed* when no longer needed, to free system resources. Closing an output stream automatically *flushes* the output stream, meaning that any data in its internal buffer is written out.

Since byte streams also implement the `AutoCloseable` interface, they can be declared in a `try`-with-resources statement (§7.7, p. 407) that will ensure they are properly closed after use at runtime.

An output stream can also be manually flushed by calling the second method.

Read and write operations on streams are *blocking* operations—that is, a call to a read or write method does not return before a byte has been read or written.

Many methods in the classes contained in the `java.io` package throw the checked `IOException`. A calling method must therefore either catch the exception explicitly, or specify it in a `throws` clause.

Table 20.1 and Table 20.2 give an overview of selected byte streams. Usually an output stream has a corresponding input stream of the same type.

Table 20.1 *Selected Input Streams*

`FileInputStream`	Data is read as bytes from a file. The file acting as the input stream can be specified by a `File` object, a `FileDescriptor`, or a `String` file name.
`FilterInputStream`	The superclass of all *input filter streams*. An input filter stream must be chained to an underlying input stream.
`DataInputStream`	A filter stream that allows the *binary representation of Java primitive values* to be read from an underlying input stream. The underlying input stream must be specified.
`ObjectInputStream`	A filter stream that allows *binary representations of Java objects and Java primitive values* to be read from a specified input stream.

Table 20.2 *Selected Output Streams*

`FileOutputStream`	Data is written as *bytes* to a file. The file acting as the output stream can be specified by a `File` object, a `FileDescriptor`, or a `String` file name.
`FilterOutputStream`	The superclass of all *output filter streams*. An output filter stream must be chained to an underlying output stream.
`PrintStream`	A filter output stream that converts a *text representation of Java objects and Java primitive values* to bytes before writing them to an underlying output stream, which must be specified. This is the type of `System.out` and `System.err` (p. 1255).

	However, the `PrintWriter` class is recommended when writing characters rather than bytes (p. 1247).
`DataOutputStream`	A filter stream that allows the *binary representation of Java primitive values* to be written to an underlying output stream. The underlying output stream must be specified.
`ObjectOutputStream`	A filter stream that allows the *binary representation of Java objects and Java primitive values* to be written to a specified underlying output stream.

File Streams

The subclasses `FileInputStream` and `FileOutputStream` represent low-level streams that define byte input and output streams that are connected to files. Data can only be read or written as a sequence of *bytes*. Such file streams are typically used for handling image data.

A `FileInputStream` for reading bytes can be created using the following constructor:

Click here to view code image

```
FileInputStream(String name) throws FileNotFoundException
```

The file designated by the file name is assigned to a new file input stream.

If the file does not exist, a `FileNotFoundException` is thrown. If it exists, it is set to be read from the beginning. A `SecurityException` is thrown if the file does not have read access.

A `FileOutputStream` for writing bytes can be created using the following constructor:

Click here to view code image

```
FileOutputStream(String name) throws FileNotFoundException
```

```
FileOutputStream(String name, boolean append) throws FileNotFoundException
```

The file designated by the file name is assigned to a new file output stream.

If the file does not exist, it is created. If it exists, its contents are reset, unless the appropriate constructor is used to indicate that output should be appended to the file. A `SecurityException` is thrown if the file does not have write access or it cannot be created. A `FileNotFoundException` is thrown if it is not possible to open the file for any other reasons.

The `FileInputStream` class provides an implementation for the `read()` methods in its superclass `InputStream`. Similarly, the `FileOutputStream` class provides an implementation for the `write()` methods in its superclass `OutputStream`.

Example 20.1 demonstrates using a buffer to read bytes from and write bytes to file streams. The input and the output file names are specified on the command line. The streams are created at (1) and (2).

The bytes are read into a buffer by the `read()` method that returns the number of bytes read. The same number of bytes from the buffer are written to the output file by the `write()` method, regardless of whether the buffer is full or not after every read operation.

The end of file is reached when the `read()` method returns the value `-1`. The code at (3a) using a buffer can be replaced by a call to the `transferTo()` method at (3b) to do the same operation. The streams are closed by the `try`-with-resources statement. Note that most of the code consists of a `try`-with-resources statement with `catch` clauses to handle the various exceptions.

Example 20.1 *Copying a File Using a Byte Buffer*

Click here to view code image

```
/* Copy a file using a byte buffer.
   Command syntax: java CopyFile <from_file> <to_file> */
import java.io.*;

class CopyFile {
  public static void main(String[] args) {

    try (// Assign the files:
```

```
          FileInputStream fromFile = new FileInputStream(args[0]);       // (1)
          FileOutputStream toFile = new FileOutputStream(args[1]))  {    // (2)

        // Copy bytes using buffer:                                      // (3a)
        byte[] buffer = new byte[1024];
        int length = 0;
        while((length = fromFile.read(buffer)) != -1) {
          toFile.write(buffer, 0, length);
        }

        // Transfer bytes:
  //    fromFile.transferTo(toFile);                                     // (3b)

      } catch(ArrayIndexOutOfBoundsException e) {
        System.err.println("Usage: java CopyFile <from_file> <to_file>");
      } catch(FileNotFoundException e) {
        System.err.println("File could not be copied: " + e);
      } catch(IOException e) {
        System.err.println("I/O error.");
      }
    }
  }
```

I/O Filter Streams

An *I/O filter stream* is a high-level I/O stream that provides additional functionality to an underlying stream to which it is chained. The data from the underlying stream is manipulated in some way by the filter stream. The `FilterInputStream` and `FilterOutputStream` classes, together with their subclasses, define input and output filter streams. The subclasses `BufferedInputStream` and `BufferedOutputStream` implement filter streams that buffer input from and output to the underlying stream, respectively. The subclasses `DataInputStream` and `DataOutputStream` implement filter streams that allow binary representation of Java primitive values to be read and written, respectively, from and to an underlying stream.

Reading and Writing Binary Values

The `java.io` package contains the two interfaces `DataInput` and `DataOutput`, which streams can implement to allow reading and writing of *binary representation of Java primitive values* (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`). The methods for writing binary representations of Java primitive values are named `write`*X*, where *X* is any Java primitive data type. The methods for reading binary representations of Java primitive values are similarly named `read`*X*. **Table 20.3** gives an overview of the `read`*X*`()` and `write`*X*`()` methods found in these two in-

terfaces. A file containing *binary values* (i.e., binary representation of Java primitive values) is usually called a *binary file*.

Table 20.3 *The* `DataInput` *and* `DataOutput` *Interfaces*

Type	Methods in the `DataInput` interface	Methods in the `DataOutput` interface
`boolean`	`readBoolean()`	`writeBoolean(boolean b)`
`char`	`readChar()`	`writeChar(int c)`
`byte`	`readByte()`	`writeByte(int b)`
`short`	`readShort()`	`writeShort(int s)`
`int`	`readInt()`	`writeInt(int i)`
`long`	`readLong()`	`writeLong(long l)`
`float`	`readFloat()`	`writeFloat(float f)`
`double`	`readDouble()`	`writeDouble(double d)`
`String`	`readLine()`	`writeChars(String str)`
`String`	`readUTF()`	`writeUTF(String str)`

Note the methods provided for reading and writing strings. However, the recommended practice for reading and writing characters is to use *character streams*, called *readers* and *writers*, which are discussed in §20.3.

The filter streams `DataOutputStream` and `DataInputStream` implement the `DataOutput` and `DataInput` interfaces, respectively, and can be used to read and write binary representation of Java primitive values from and to an underlying stream. Both the `write`*X*`()` and `read`*X*`()` methods throw an `IOException` in the event of an I/O error. In particular, the `read`*X*`()` methods throw an `EOFException` (a subclass of `IOException`) if the input stream does not contain the correct number

of bytes to read. Bytes can also be skipped from a `DataInput` stream, using the `skipBytes(int n)` method which skips `n` bytes.

Click here to view code image

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

These constructors can be used to set up filter streams from an underlying stream for reading and writing Java primitive values, respectively.

Writing Binary Values to a File

To write the binary representation of Java primitive values to a *binary file*, the following procedure can be used, which is also depicted in **Figure 20.2**.

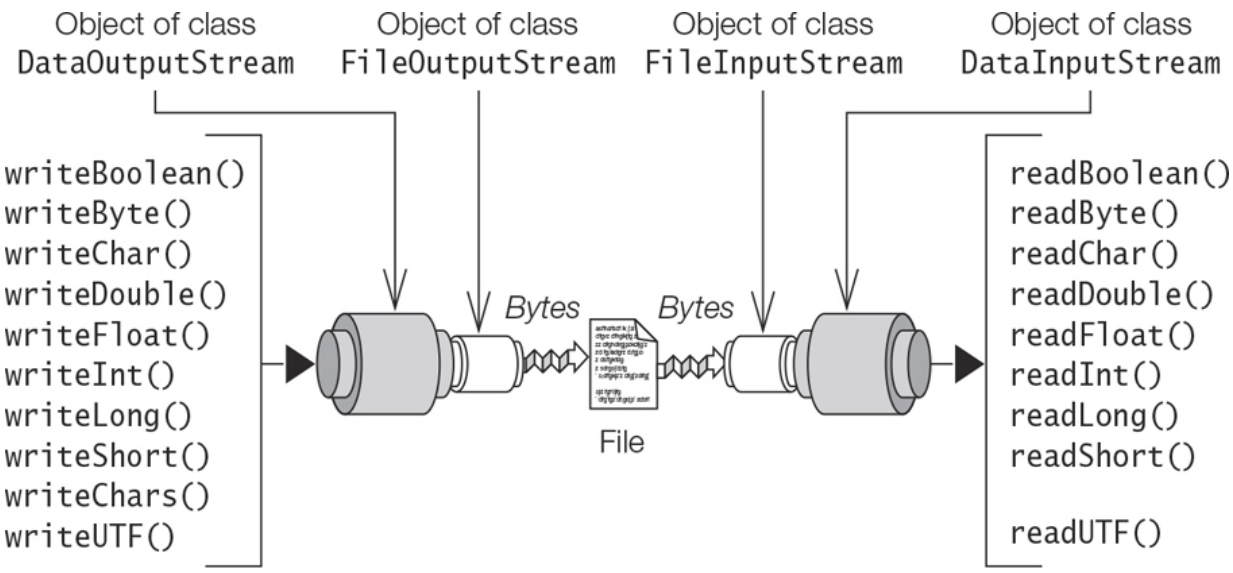


Figure 20.2 *Stream Chaining for Reading and Writing Binary Values to a File*

1. Use a `try`-with-resources statement for declaring and creating the necessary streams, which guarantees closing of the filter stream and any underlying stream.
2. Create a `FileOutputStream`:
   Click here to view code image

   ```
   FileOutputStream outputFile = new FileOutputStream("primitives.data");
   ```

3. Create a `DataOutputStream` which is chained to the `FileOutputStream`:
   Click here to view code image

```
DataOutputStream outputStream = new DataOutputStream(outputFile);
```

4. Write Java primitive values using relevant `write`*X*`()` methods:
   Note that in the case of `char`, `byte`, and `short` data types, the `int` argument to the `write`*X*`()` method is converted to the corresponding type, before it is written (see **Table 20.3**).

See also the numbered lines in **Example 20.2** corresponding to the steps above.

Reading Binary Values from a File

To read the binary representation of Java primitive values from a *binary file*, the following procedure can be used, which is also depicted in **Figure 20.2**.

1. Use a `try`-with-resources statement for declaring and creating the necessary streams, which guarantees closing of the filter stream and any underlying stream.
2. Create a `FileInputStream`:
   Click here to view code image

```
FileInputStream inputFile = new FileInputStream("primitives.data");
```

3. Create a `DataInputStream` which is chained to the `FileInputStream`:
   Click here to view code image

```
DataInputStream inputStream = new DataInputStream(inputFile);
```

4. Read the (exact number of) Java primitive values in the *same order* they were written out to the file, using relevant `read`*X*`()` methods. Not doing so will unleash the wrath of the `IOException`.

See also the numbered lines in **Example 20.2** corresponding to the steps above. **Example 20.2** uses both procedures described above: first to write and then to read some Java primitive values to and from a file. It also checks to see if the end of the stream has been reached, signaled by an `EOFException`. The values are also written to the standard output stream.

Example 20.2 *Reading and Writing Binary Values*

Click here to view code image

```
import java.io.*;
```

```
public class BinaryValuesIO {
  public static void main(String[] args) throws IOException {

    // Write binary values to a file:
    try(                                                              // (1)
        // Create a FileOutputStream.                                 (2)
        FileOutputStream outputFile = new FileOutputStream("primitives.data");
        // Create a DataOutputStream which is chained to the FileOutputStream.(3)
        DataOutputStream outputStream = new DataOutputStream(outputFile)) {

      // Write Java primitive values in binary representation:        (4)
      outputStream.writeBoolean(true);
      outputStream.writeChar('A');               // int written as Unicode char
      outputStream.writeByte(Byte.MAX_VALUE);    // int written as 8-bits byte
      outputStream.writeShort(Short.MIN_VALUE);  // int written as 16-bits short
      outputStream.writeInt(Integer.MAX_VALUE);
      outputStream.writeLong(Long.MIN_VALUE);
      outputStream.writeFloat(Float.MAX_VALUE);
      outputStream.writeDouble(Math.PI);
    }

    // Read binary values from a file:
    try (                                                             // (1)
        // Create a FileInputStream.                                  (2)
        FileInputStream inputFile = new FileInputStream("primitives.data");

        // Create a DataInputStream which is chained to the FileInputStream.  (3)
        DataInputStream inputStream = new DataInputStream(inputFile)) {

      // Read the binary representation of Java primitive values
      // in the same order they were written out:                     (4)
      System.out.println(inputStream.readBoolean());
      System.out.println(inputStream.readChar());
      System.out.println(inputStream.readByte());
      System.out.println(inputStream.readShort());
      System.out.println(inputStream.readInt());
      System.out.println(inputStream.readLong());
      System.out.println(inputStream.readFloat());
      System.out.println(inputStream.readDouble());

      // Check for end of stream:
      int value = inputStream.readByte();
      System.out.println("More input: " + value);
    } catch (FileNotFoundException fnf) {
      System.out.println("File not found.");
    } catch (EOFException eof) {
      System.out.println("End of input stream.");
    }
```

```
    }
}
```

Output from the program:

```
true
A
127
-32768
2147483647
-9223372036854775808
3.4028235E38
3.141592653589793
End of input stream.
```

20.3 Character Streams: Readers and Writers

A *character encoding* is a scheme for representing characters. Java programs represent values of the `char` type internally in the 16-bit Unicode character encoding, but the host platform might use another character encoding to represent and store characters externally. For example, the ASCII (American Standard Code for Information Interchange) character encoding is widely used to represent characters on many platforms. However, it is only one small subset of the Unicode standard.

The abstract classes `Reader` and `Writer` are the roots of the inheritance hierarchies for streams that read and write *Unicode characters* using a specific character encoding (**Figure 20.3**). A *reader* is an input character stream that implements the `Readable` interface and reads a sequence of Unicode characters, and a *writer* is an output character stream that implements the `Writer` interface and writes a sequence of Unicode characters. Character encodings (usually called *charsets*) are used by readers and writers to convert between external bytes and internal Unicode characters. The same character encoding that was used to write the characters must be used to read those characters. The `java.nio.charset.Charset` class represents charsets. Kindly refer to the `Charset` class API documentation for more details.

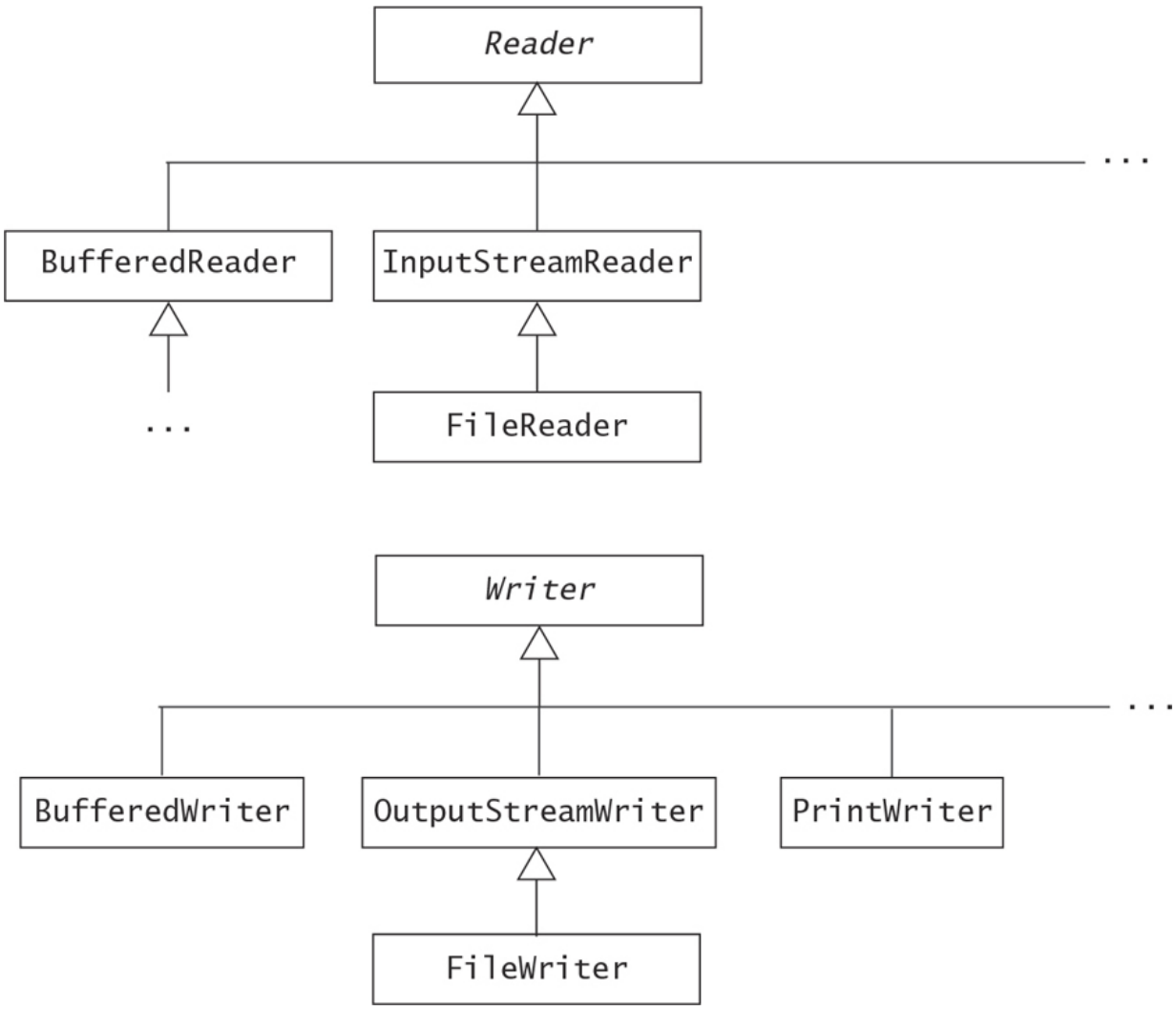


Figure 20.3 *Selected Character Streams in the* `java.io` *Package*

Click here to view code image

```
static Charset forName(String charsetName)
```

Returns a charset object for the named charset. Selected common charset names are `"UTF-8"`, `"UTF-16"`, `"US-ASCII"`, and `"ISO-8859-1"`.

Click here to view code image

```
static Charset defaultCharset()
```

Returns the default charset of this Java virtual machine.

Table 20.4 and Table 20.5 give an overview of some selected character streams found in the `java.io` package.

Table 20.4 *Selected Readers*

Reader	Description
`BufferedReader`	A reader is a high-level input stream that buffers the characters read from an underlying stream. The underlying stream must be specified and an optional buffer size can be given.
`InputStreamReader`	Characters are read from a byte input stream which must be specified. The default character encoding is used if no character encoding is explicitly specified in the constructor. This class provides the bridge from byte streams to character streams.
`FileReader`	Characters are read from a file, using the default character encoding, unless an encoding is explicitly specified in the constructor. The file can be specified by a `String` file name. It automatically creates a `FileInputStream` that is associated with the file.

Table 20.5 *Selected Writers*

Writers	Description
`BufferedWriter`	A writer is a high-level output stream that buffers the characters before writing them to an underlying stream. The underlying stream must be specified, and an optional buffer size can be specified.
`OutputStreamWriter`	Characters are written to a byte output stream that must be specified. The default character encoding is used if no explicit character encoding is specified in the constructor. This class provides the bridge from character streams to byte streams.
`FileWriter`	Characters are written to a file, using the default character encoding, unless an encoding is explicitly specified in the constructor. The file can be specified by a `String` file name. It automatically creates a

Writers	Description
	`FileOutputStream` that is associated with the file. A `boolean` parameter can be specified to indicate whether the file should be overwritten or appended with new content.
`PrintWriter`	A print writer is a high-level output stream that allows *text representation of Java objects and Java primitive values* to be written to an underlying output stream or writer. The underlying output stream or writer must be specified. An explicit encoding can be specified in the constructor, and also whether the print writer should do automatic line flushing.

Readers use the following methods for reading Unicode characters:

Click here to view code image

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

Note that the `read()` methods read the character as an `int` in the range 0 to 65,535 (0x0000–0xFFFF).

The first method returns the character as an `int` value. The last two methods store the characters in the specified array and return the number of characters read. The value `-1` is returned if the end of the stream has been reached.

Click here to view code image

```
long skip(long n) throws IOException
```

A reader can skip over characters using the `skip()` method.

Click here to view code image

```
void close() throws IOException
```

Like byte streams, a character stream should be closed when no longer needed in order to free system resources.

Click here to view code image

```
boolean ready() throws IOException
```

When called, this method returns `true` if the next read operation is guaranteed not to block. Returning `false` does *not* guarantee that the next read operation will block.

Click here to view code image

```
long transferTo(Writer out) throws IOException
```

Reads all characters from this reader and writes the characters to the specified writer in the order they are read. The I/O streams are not *closed* after the operation.

Writers use the following methods for writing Unicode characters:

Click here to view code image

```
void write(int c) throws IOException
```

The `write()` method takes an `int` as an argument, but writes only the least significant 16 bits.

Click here to view code image

```
void write(char[] cbuf) throws IOException
void write(String str) throws IOException
void write(char[] cbuf, int off, int length) throws IOException
void write(String str, int off, int length) throws IOException
```

Write the characters from an array of characters or a string.

Click here to view code image

```
void close() throws IOException
void flush() throws IOException
```

Like byte streams, a character stream should be closed when no longer needed in order to free system resources. Closing a character output stream automatically *flushes* the stream. A character output stream can also be manually flushed.

Like byte streams, many methods of the character stream classes throw a checked `IOException` that a calling method must either catch explicitly or specify in a `throws` clause. They also implement the `AutoCloseable` interface, and can thus be declared in a `try`-with-resources statement (§7.7, p. 407) that will ensure they are automatically closed after use at runtime.

Analogous to **Example 20.1** that demonstrates usage of a byte buffer for writing and reading bytes to and from file streams, **Example 20.3** demonstrates using a character buffer for writing and reading characters to and from file streams. Later in this section, we will use buffered readers (p. 1251) and buffered writers (p. 1250) for reading and writing characters from files, respectively.

Example 20.3 *Copying a File Using a Character Buffer*

Click here to view code image

```
/* Copy a file using a character buffer.
   Command syntax: java CopyCharacterFile <from_file> <to_file> */
import java.io.*;

class CopyCharacterFile {
  public static void main(String[] args) {

    try (// Assign the files:
        FileReader fromFile = new FileReader(args[0]);                // (1)
        FileWriter toFile = new FileWriter(args[1]))  {               // (2)

      // Copy characters using buffer:                                // (3a)
      char[] buffer = new char[1024];
      int length = 0;
      while((length = fromFile.read(buffer)) != -1) {
        toFile.write(buffer, 0, length);
      }

      // Transfer characters:
//    fromFile.transferTo(toFile);                                    // (3b)

    } catch(ArrayIndexOutOfBoundsException e) {
      System.err.println("Usage: java CopyCharacterFile <from_file> <to_file>");
```

```
    } catch(FileNotFoundException e) {
      System.err.println("File could not be copied: " + e);
    } catch(IOException e) {
      System.err.println("I/O error.");
    }
  }
}
```

Print Writers

The capabilities of the `OutputStreamWriter` and the `InputStreamReader` classes are limited, as they primarily write and read characters.

In order to write a *text representation* of Java primitive values and objects, a `PrintWriter` should be chained to either a writer, or a byte output stream, or accept a `String` file name, using one of the following constructors:

Click here to view code image

```
PrintWriter(Writer out)
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(String fileName) throws FileNotFoundException
PrintWriter(String fileName, Charset charset)
            throws FileNotFoundException
PrintWriter(String fileName, String charsetName)
            throws FileNotFoundException, UnsupportedEncodingException
```

The boolean `autoFlush` argument specifies whether the `PrintWriter` should do automatic line flushing.

When the underlying writer is specified, the character encoding supplied by the underlying writer is used. However, an `OutputStream` has no notion of any character encoding, so the necessary intermediate `OutputStreamWriter` is automatically created, which will convert characters into bytes, using the default character encoding.

```
boolean checkError()
protected void clearError()
```

The first method flushes the output stream if it's not closed and checks its error state.

The second method clears the error state of this output stream.

Writing Text Representation of Primitive Values and Objects

In addition to overriding the `write()` methods from its super class `Writer`, the `Print-Writer` class provides methods for writing text representation of Java primitive values and of objects (see **Table 20.6**). The `println()` methods write the text representation of their argument to the underlying stream, and then append a *line separator*. The `println()` methods use the correct platform-dependent line separator. For example, on Unix-based platforms the line separator is `'\n'` (newline), while on Windows-based platforms it is `"\r\n"` (carriage return + newline) and on Mac-based platforms it is `'\r'` (carriage return).

Table 20.6 *Print Methods of the* `PrintWriter` *Class*

The `print()` methods	The `println()` methods
_ print(boolean b) print(char c) print(int i) print(long l) print(float f) print(double d) print(char[] s) print(String s) print(Object obj)	println() println(boolean b) println(char c) println(int i) println(long l) println(float f) println(double d) println(char[] ca) println(String str) println(Object obj)

The print methods create a text representation of an object by calling the `toString()` method on the object. The print methods do not throw any `IOException`. Instead, the `checkError()` method of the `PrintWriter` class must be called to check for errors.

Writing Formatted Values

Although formatting of values is covered extensively in **Chapter 18**, **p. 1095**, here we mention the support for formatting values provided by I/O streams. The `PrintWriter` class provides the `format()` methods and the `printf()` convenient

methods to write *formatted* values. The `printf()` methods are functionally equivalent to the `format()` methods. As the methods return a `PrintWriter`, calls to these methods can be chained.

The `printf()` and the `format()` methods for printing formatted values are also provided by the `PrintStream` and the `Console` classes (p. 1256). The `format()` method is also provided by the `String` class (§8.4, p. 457). We assume familiarity with printing formatted values on the standard output stream by calling the `printf()` method on the `System.out` field which is an object of the `PrintStream` class (§1.9, p. 24).

Click here to view code image

```
PrintWriter format(String format, Object... args)
PrintWriter format(Locale loc, String format, Object... args)
PrintWriter printf(String format, Object... args)
PrintWriter printf(Locale loc, String format, Object... args)
```

The `String` parameter `format` specifies how formatting will be done. It contains *format specifiers* that determine how each subsequent value in the variable arity parameter `args` will be formatted and printed. The resulting string from the formatting will be written to the current writer.

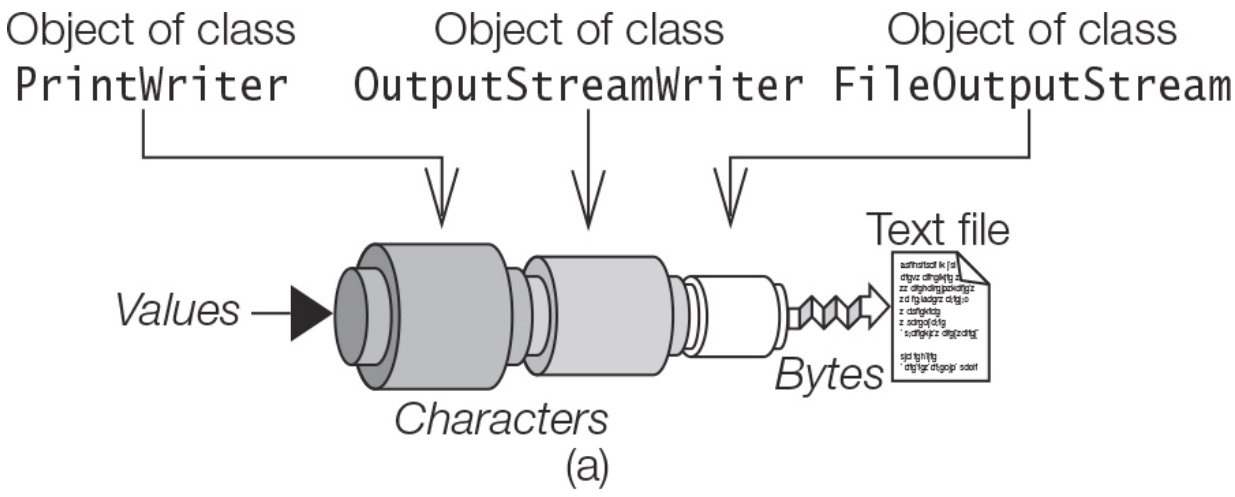
If the locale is specified, it is taken into consideration to format the `args`.

Any error in the format string will result in a runtime exception.

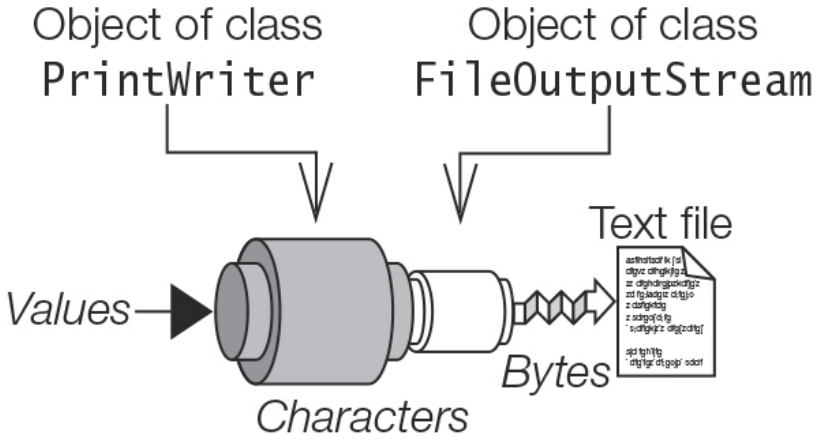
Writing Text Files

When writing text representation of values to a file using the default character encoding, any one of the following four procedures for setting up a `PrintWriter` can be used.

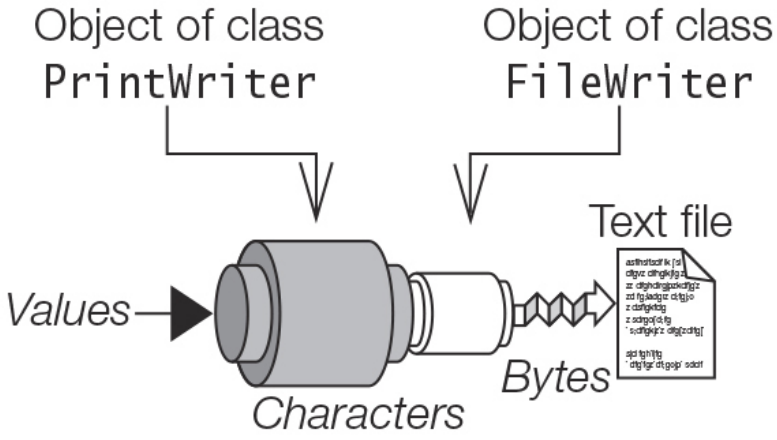
Setting up a `PrintWriter` based on an `OutputStreamWriter` which is chained to a `FileOutputStream` (Figure 20.4(a)):


Object of class
PrintWriter
Object of class
OutputStreamWriter
Object of class
FileOutputStream
Text file
Values
Bytes
Characters

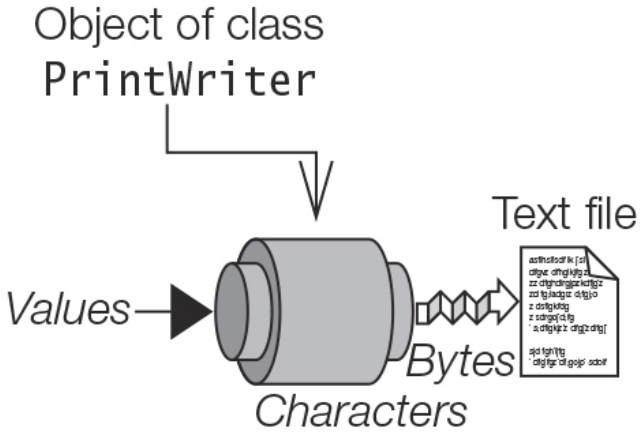
(a)


Object of class
PrintWriter
Object of class
FileOutputStream
Text file
Values
Bytes
Characters

(b)


Object of class
PrintWriter
Object of class
FileWriter
Text file
Values
Bytes
Characters

(c)


Object of class
PrintWriter
Text file
Values
Bytes
Characters

(d)

Figure 20.4 *Setting Up a* `PrintWriter` *to Write to a File*

1. Create a `FileOutputStream`:
   Click here to view code image

```
FileOutputStream outputFile = new FileOutputStream("info.txt");
```

2. Create an `OutputStreamWriter` which is chained to the `FileOutputStream`:
   Click here to view code image

```
OutputStreamWriter outputStream = new OutputStreamWriter(outputFile);
```

   The `OutputStreamWriter` uses the default character encoding for writing the characters to the file.
3. Create a `PrintWriter` which is chained to the `OutputStreamWriter`:
   Click here to view code image

```
PrintWriter printWriter1 = new PrintWriter(outputStream, true);
```

   The value `true` for the second parameter in the constructor will result in the output buffer being flushed by the `println()` and `printf()` methods.

Setting up a `PrintWriter` based on a `FileOutputStream` (**Figure 20.4(b)**):

1. Create a `FileOutputStream`:
   Click here to view code image

```
FileOutputStream outputFile = new FileOutputStream("info.txt");
```

2. Create a `PrintWriter` which is chained to the `FileOutputStream`:
   Click here to view code image

```
PrintWriter printWriter2 = new PrintWriter(outputFile, true);
```

   The intermediate `OutputStreamWriter` to convert the characters using the default encoding is automatically supplied. The output buffer will also perform automatic line flushing.

Setting up a `PrintWriter` based on a `FileWriter` (**Figure 20.4(c)**):

1. Create a `FileWriter` which is a subclass of `OutputStreamWriter`:
   Click here to view code image

   ```
   FileWriter fileWriter = new FileWriter("info.txt");
   ```

   This is equivalent to having an `OutputStreamWriter` chained to a `FileOutputStream` for writing the characters to the file, as shown in **Figure 20.4(a)**.
2. Create a `PrintWriter` which is chained to the `FileWriter`:
   Click here to view code image

   ```
   PrintWriter printWriter3 = new PrintWriter(fileWriter, true);
   ```

   The output buffer will be flushed by the `println()` and `printf()` methods.

Setting up a `PrintWriter`, given the file name (**Figure 20.4(d)**):

1. Create a `PrintWriter`, supplying the file name:
   Click here to view code image

   ```
   PrintWriter printWriter4 = new PrintWriter("info.txt");
   ```

   The underlying `OutputStreamWriter` is created to write the characters to the file in the default encoding, as shown in **Figure 20.4(d)**. In this case, there is no automatic flushing by the `println()` and `printf()` methods.

If a specific character encoding is desired for the writer, the third procedure (**Figure 20.4(c)**) can be used, with the encoding being specified for the `FileWriter`:

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileWriter fileWriter = new FileWriter("info.txt", utf8);
PrintWriter printWriter5 = new PrintWriter(fileWriter, true);
```

This writer will use the UTF-8 character encoding to write the characters to the file. Alternatively, we can use a `PrintWriter` constructor that accepts a character encoding:

Click here to view code image

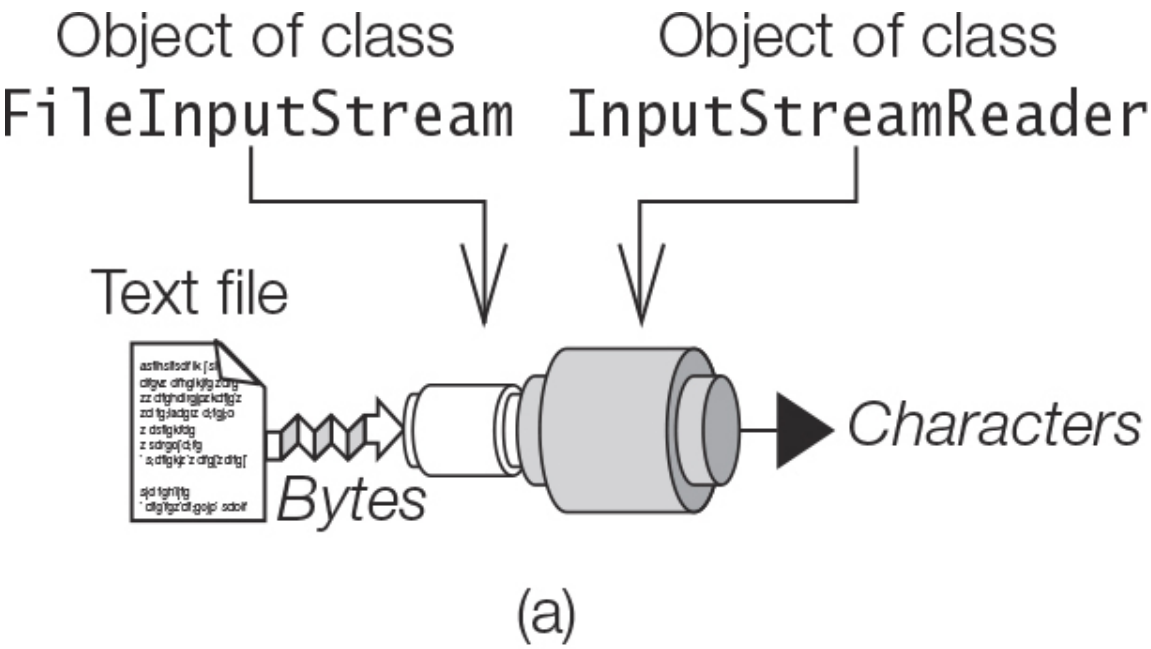
```
Charset utf8 = Charset.forName("UTF-8");
PrintWriter printWriter6 = new PrintWriter("info.txt", utf8);
```

A `BufferedWriter` can also be used to improve the efficiency of writing characters to the underlying stream, as explained later in this section.

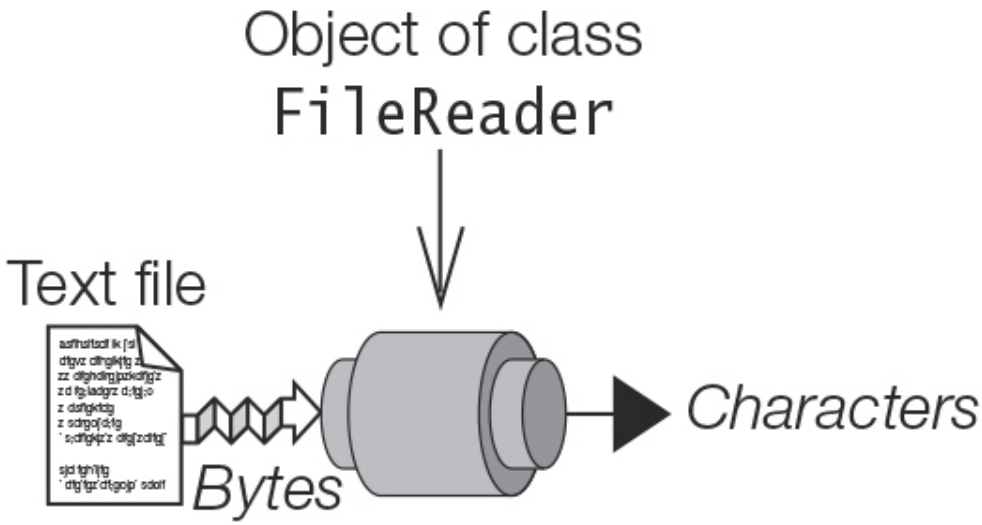
Reading Text Files

When reading *characters* from a file using the default character encoding, the following two procedures for setting up an `InputStreamReader` can be used.

Setting up an `InputStreamReader` which is chained to a `FileInputStream` (**Figure 20.5(a)**):



(a)



(b)

Figure 20.5 *Setting Up Readers to Read Characters*

1. Create a `FileInputStream`:
   Click here to view code image

   ```
   FileInputStream inputFile = new FileInputStream("info.txt");
   ```

2. Create an `InputStreamReader` which is chained to the `FileInputStream`:
   Click here to view code image

   ```
   InputStreamReader reader = new InputStreamReader(inputFile);
   ```

   The `InputStreamReader` uses the default character encoding for reading the characters from the file.

Setting up a `FileReader` which is a subclass of `InputStreamReader` (**Figure 20.5(b)**):

1. Create a `FileReader`:
   Click here to view code image

   ```
   FileReader fileReader = new FileReader("info.txt");
   ```

   This is equivalent to having an `InputStreamReader` chained to a `FileInputStream` for reading the characters from the file, using the default character encoding.

If a specific character encoding is desired for the reader, the first procedure can be used (**Figure 20.5(a)**), with the encoding being specified for the `InputStreamReader`:

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileInputStream inputFile = new FileInputStream("info.txt");
InputStreamReader reader = new InputStreamReader(inputFile, utf8);
```

This reader will use the UTF-8 character encoding to read the characters from the file. Alternatively, we can use one of the `FileReader` constructors that accept a character encoding:

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileReader reader = new FileReader("info.txt", utf8);
```

A `BufferedReader` can also be used to improve the efficiency of reading characters from the underlying stream, as explained later in this section (p. 1251).

Using Buffered Writers

A `BufferedWriter` can be chained to the underlying writer by using one of the following constructors:

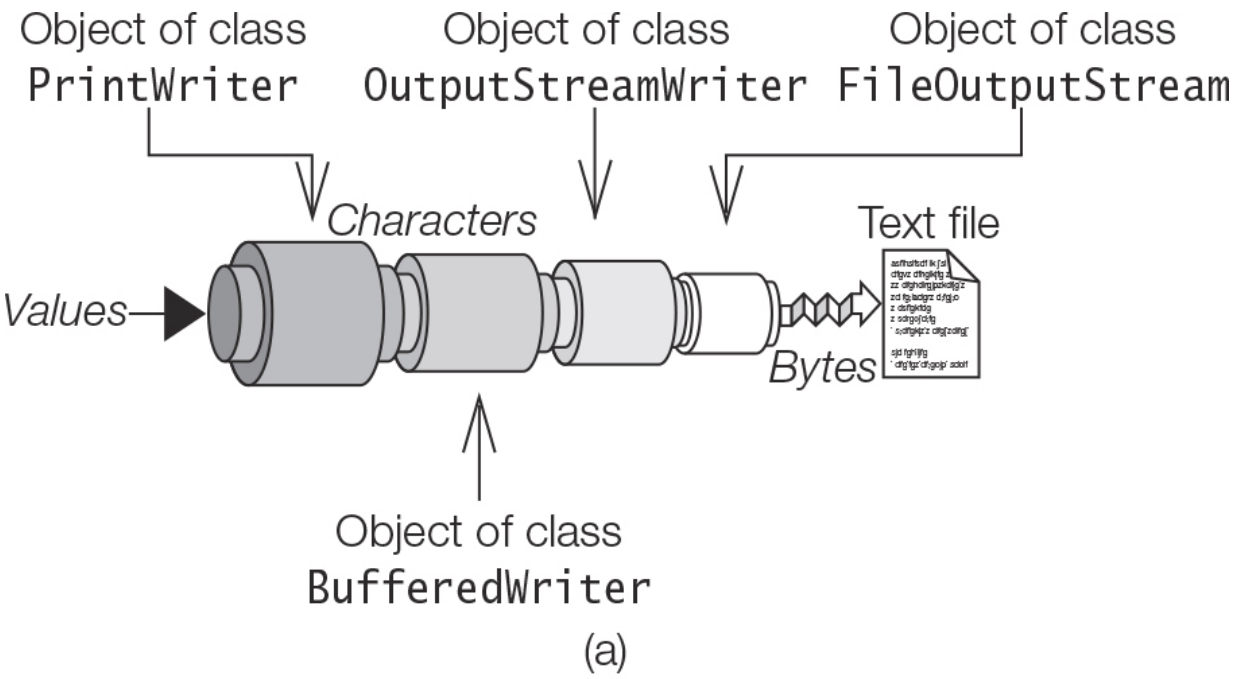
Click here to view code image

```
BufferedWriter(Writer out)
BufferedWriter(Writer out, int size)
```

The default buffer size is used, unless the buffer size is explicitly specified.

Characters, strings, and arrays of characters can be written using the methods for a `Writer`, but these now use buffering to provide efficient writing of characters. In addition, the `BufferedWriter` class provides the method `newLine()` for writing the platform-dependent line separator.

The following code creates a `PrintWriter` whose output is buffered, and the characters are written using the UTF-8 character encoding (**Figure 20.6(a)**):



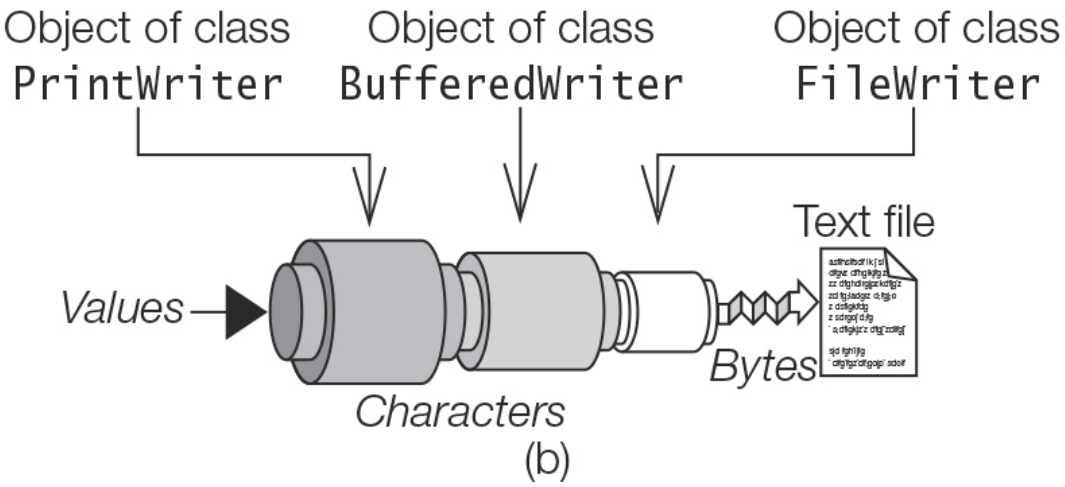


Figure 20.6 *Buffered Writers*

Click here to view code image

```
Charset utf8 = Charset.forName("UTF-8");
FileOutputStream   outputFile      = new FileOutputStream("info.txt");
OutputStreamWriter outputStream    = new OutputStreamWriter(outputFile, utf8);
BufferedWriter     bufferedWriter1 = new BufferedWriter(outputStream);
PrintWriter        printWriter1    = new PrintWriter(bufferedWriter1, true);
```

The following code creates a `PrintWriter` whose output is buffered, and the characters are written using the default character encoding (**Figure 20.6(b)**):

Click here to view code image

```
FileWriter     fileWriter      = new FileWriter("info.txt");
BufferedWriter bufferedWriter2 = new BufferedWriter(fileWriter);
PrintWriter    printWriter2    = new PrintWriter(bufferedWriter2, true);
```

Note that in both cases, the `PrintWriter` is used to write the characters. The `Buffered-Writer` is sandwiched between the `PrintWriter` and the underlying `OutputStreamWriter` (which is the superclass of the `FileWriter` class).

Using Buffered Readers

A `BufferedReader` can be chained to the underlying reader by using one of the following constructors:

Click here to view code image

```
BufferedReader(Reader in)
BufferedReader(Reader in, int size)
```

The default buffer size is used, unless the buffer size is explicitly specified.

In addition to the methods of the `Reader` class, the `BufferedReader` class provides the method `readLine()` to read a line of text from the underlying reader.

Click here to view code image

```
String readLine() throws IOException
```

The `null` value is returned when the end of the stream is reached. The returned string must explicitly be converted to other values.

The `BufferedReader` class also provides the `lines()` method to create a *stream of text lines* with a buffered reader as the data source (§16.4, p. 902).

```
Stream<String> lines()
```

Returns a *finite sequential ordered* `Stream` of element type `String`, where the elements are text lines read by this `BufferedReader`.

The following code creates a `BufferedReader` that can be used to read text lines from a file (**Figure 20.7(b)**):

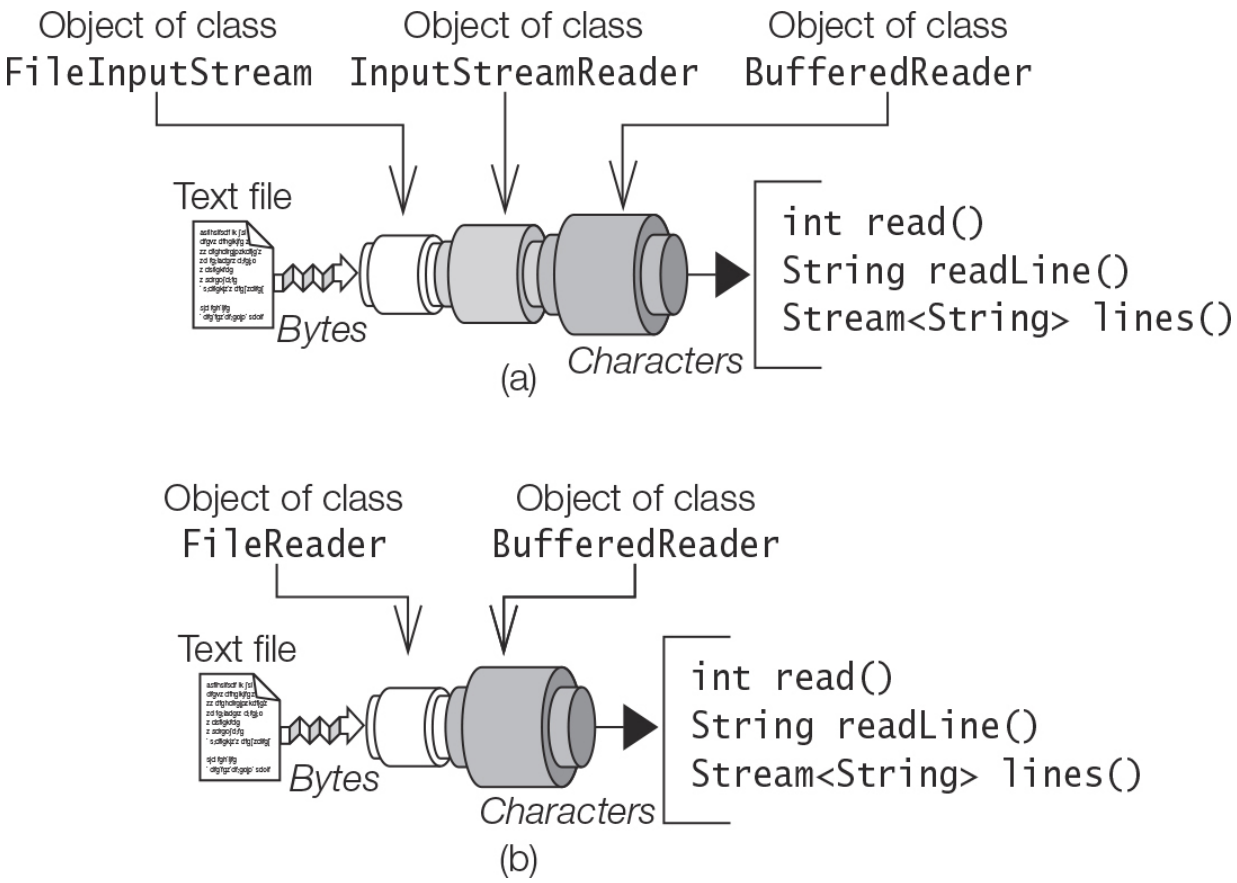


Figure 20.7 *Buffered Readers*

Click here to view code image

```
// Using the UTF-8 character encoding:
Charset utf8 = Charset.forName("UTF-8");
FileReader     fileReader      = new FileReader("lines.txt", utf8);
BufferedReader bufferedReader1 = new BufferedReader(reader)

// Use the default encoding:
FileReader     fileReader      = new FileReader("lines.txt");
BufferedReader bufferedReader2 = new BufferedReader(fileReader);
```

Note that in both cases the `BufferedReader` object is used to read the text lines.

Java primitive values and objects cannot be read directly from their text representation in a file. Characters must be read and converted to the relevant values explicitly. If the text representation of the values is written as *lines of text*, each line can be read and *tokenized* first—that is, grouping characters into *tokens* that meaningfully represent a value. For example, the line `"Potatoes 2.50"` contains two tokens: the item token `"Potatoes"` and the price token `"2.50"`. Once a line is tokenized, the tokens can be parsed to obtain the appropriate type of value. The item token is already of

type `String`, but the price token `"2.50"` needs to be parsed to a `double` value using the `Double.parseDouble()` method. A *scanner* provided by the `java.io.Scanner` class or the `String.split()` method called on each line can be used to tokenize the character input—both of which are beyond the scope of this book.

In contrast to **Example 20.2**, which demonstrated the reading and writing of binary representations of primitive data values, **Example 20.4** illustrates the reading and writing of text representations of values using I/O streams that are readers and writers.

The `CharEncodingDemo` class in **Example 20.4** writes text representations of values using the UTF-8 character encoding specified at (1). It uses a `try`-with-resources statement for handling the closing of I/O streams, as shown at (2) and (4). The `PrintWriter` is buffered (**Figure 20.6(b)**). Its underlying writer uses the specified encoding, as shown at (2). Values are written out with the text representation of one value on each line, as shown at (3). The example uses the same character encoding to read the text file. A `BufferedReader` is created (**Figure 20.7(b)**). Its underlying reader uses the specified encoding, as shown at (4). The text representation of the values is read as one value per line, and parsed accordingly. Each text line in the file is read by calling the `readLine()` method. The characters in the line are explicitly converted to an appropriate type of value, as shown at (5).

The values are printed on the standard output stream, as shown at (6). We check for the end of the stream at (7), which is signaled by the `null` value returned by the `readLine()` method of the `BufferedReader` class. Note the exceptions that are specified in the `throws` clause of the `main()` method.

Although buffering might seem like overkill in this simple example, for efficiency reasons, it should be considered when reading and writing characters from external storage.

It is a useful exercise to modify **Example 20.4** to use the various setups for chaining streams for reading and writing characters, as outlined in this section.

Example 20.4 *Demonstrating Readers and Writers, and Character Encoding*

Click here to view code image

```
import java.io.*;
import java.nio.charset.Charset;
import java.time.LocalDate;

public class CharEncodingDemo {
```

```
  public static void main(String[] args)
         throws FileNotFoundException, IOException, NumberFormatException {

    // UTF-8 character encoding.
    Charset utf8 = Charset.forName("UTF-8");                              // (1)

    try(// Create a BufferedWriter that uses UTF-8 character encoding      (2)
        FileWriter writer = new FileWriter("info.txt", utf8);
        BufferedWriter bufferedWriter1 = new BufferedWriter(writer);
        PrintWriter printWriter = new PrintWriter(bufferedWriter1, true);) {

      System.out.println("Writing using encoding: " + writer.getEncoding());
      // Print some values, one on each line.                               (3)
      printWriter.println(LocalDate.now());
      printWriter.println(Integer.MAX_VALUE);
      printWriter.println(Long.MIN_VALUE);
      printWriter.println(Math.PI);
    }

    try(// Create a BufferedReader that uses UTF-8 character encoding      (4)
        FileReader reader = new FileReader("info.txt", utf8);
        BufferedReader bufferedReader = new BufferedReader(reader);) {

      System.out.println("Reading using encoding: " + reader.getEncoding());
      // Read the character input and parse accordingly.                    (5)
      LocalDate ld = LocalDate.parse(bufferedReader.readLine());
      int iMax = Integer.parseInt(bufferedReader.readLine());
      long lMin = Long.parseLong(bufferedReader.readLine());
      double pi = Double.parseDouble(bufferedReader.readLine());

      // Write the values read on the terminal                              (6)
      System.out.println("Values read:");
      System.out.println(ld);
      System.out.println(iMax);
      System.out.println(lMin);
      System.out.println(pi);

      // Check for end of stream:                                           (7)
      String line = bufferedReader.readLine();
      if (line != null ) {
        System.out.println("More input: " + line);
      } else {
        System.out.println("End of input stream");
      }
    }
  }
}
```

Output from the program:

```
Writing using encoding: UTF8
Reading using encoding: UTF8
Values read:
2021-06-22
2147483647
-9223372036854775808
3.141592653589793
End of input stream
```

The Standard Input, Output, and Error Streams

The *standard output* stream (usually the display) is represented by the `PrintStream` object `System.out`. The *standard input* stream (usually the keyboard) is represented by the `InputStream` object `System.in`. In other words, it is a byte input stream. The *standard error* stream (also usually the display) is represented by `System.err`, which is another object of the `PrintStream` class. The `PrintStream` class offers `print()` methods that act as corresponding `print()` methods from the `PrintWriter` class. The `print()` methods can be used to write output to `System.out` and `System.err`. In other words, both `System.out` and `System.err` act like `PrintWriter`, but in addition they have `write()` methods for writing bytes.

The `System` class provides the methods `setIn(InputStream)`, `setOut(PrintStream)`, and `setErr(PrintStream)` that can be passed an I/O stream to reassign the standard streams.

In order to read *characters* typed by the user, the `Console` class is recommended (p. 1256).

Comparison of Byte Streams and Character Streams

It is instructive to see which byte streams correspond to which character streams. Table 20.7 shows the correspondence between byte and character streams. Note that not all classes have a corresponding counterpart.

Table 20.7 *Correspondence between Selected Byte and Character Streams*

Byte streams	Character streams
`OutputStream`	`Writer`

Byte streams	Character streams
`InputStream`	`Reader`
No counterpart	`OutputStreamWriter`
No counterpart	`InputStreamReader`
`FileOutputStream`	`FileWriter`
`FileInputStream`	`FileReader`
`BufferedOutputStream`	`BufferedWriter`
`BufferedInputStream`	`BufferedReader`
`PrintStream`	`PrintWriter`
`DataOutputStream`	*No counterpart*
`DataInputStream`	*No counterpart*
`ObjectOutputStream`	*No counterpart*
`ObjectInputStream`	*No counterpart*

20.4 The `Console` Class

A *console* is a unique *character-based* device associated with a JVM. Whether a JVM has a console depends on the platform, and also on the manner in which the JVM is invoked. When the JVM is started from a command line, and the standard input and output streams have not been redirected, the console will normally correspond to the keyboard and the display (**Figure 20.8**). In any case, the console will be represented by an instance of the class `java.io.Console`. This `Console` instance is a *singleton*, and can only be obtained by calling the static method `console()` of the `System` class. If there is no console associated with the JVM, the `null` value is returned by this method.

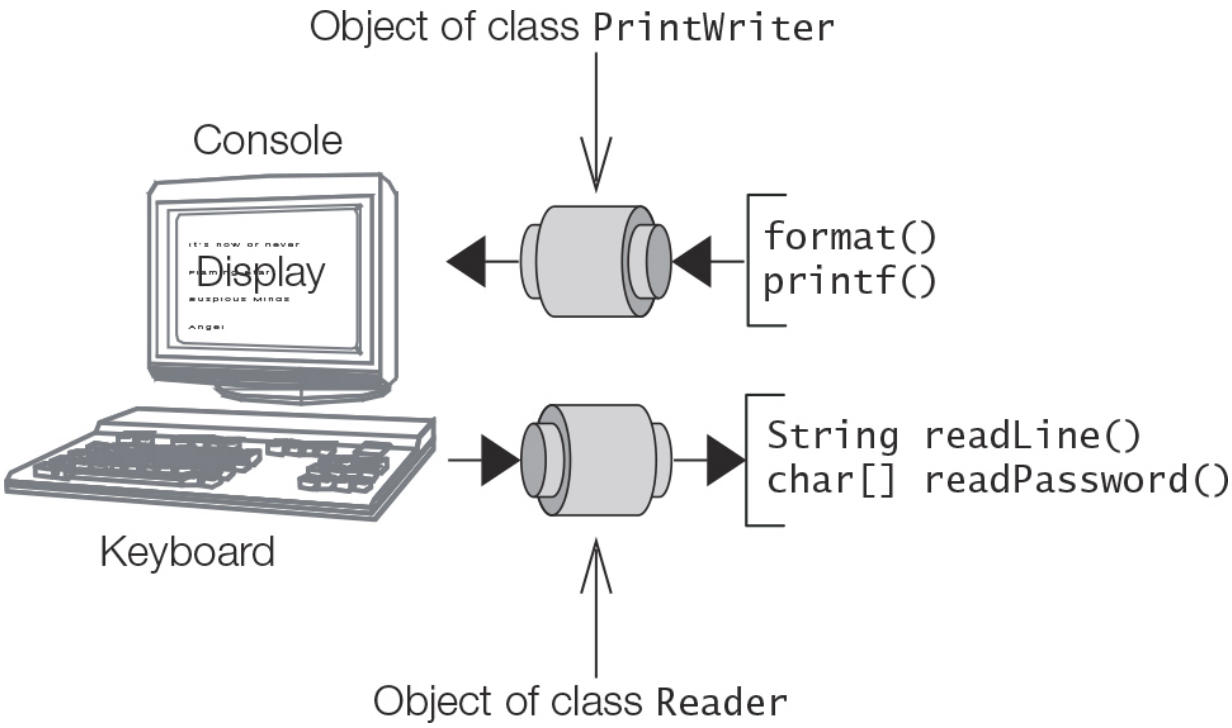


Figure 20.8 *Keyboard and Display as Console*

Click here to view code image

```
// Obtaining the console:
Console console = System.console();
if (console == null) {
  System.err.println("No console available.");
  return;
}
// Continue ...
```

For creating dialogue for console-based applications, the `Console` class provides the following functionality:

- Prompt and read a line of character-based response.
  Click here to view code image

  ```
  String username = console.readLine("Enter the username (%d chars): ", 4);
  ```

  The `readLine()` method first prints the formatted prompt on the console, and then returns the characters typed at the console when the line is terminated by the `ENTER` key.
- Prompt and read passwords without echoing the characters on the console.
  Click here to view code image

  ```
  char[] password;
  do {
  ```

```
        password = console.readPassword("Enter password (min. %d chars): ", 6);
    } while (password.length < 6);
```

The `readPassword()` method first prints the formatted prompt, and returns the password characters typed by the user in an array of `char` when the line is terminated by the `ENTER` key. The password characters are not echoed on the display. Since a password is sensitive data, one recommended practice is to have it stored in memory for only as long as it is necessary and to zero-fill the `char` array as soon as possible in order to overwrite the password characters.

- Print formatted values to the console.
  Similar to the `PrintWriter` and the `PrintStream` classes, the `Console` class also provides the `format()` and the `printf()` methods for printing formatted values, but its methods do *not* allow a locale to be specified.

Note that the console only returns character-based input. For reading other types of values from the standard input stream, the `java.util.Scanner` class can be considered.

The `Console` class provides methods for *formatted prompting* and *reading* from the console, and obtaining the reader associated with it.

Click here to view code image

```
String readLine()
String readLine(String format, Object... args)
```

The first method reads a single line of text from the console. The second method prints a formatted prompt first, then reads a single line of text from the console. The prompt is constructed by formatting the specified `args` according to the specified `format`.

Click here to view code image

```
char[] readPassword()
char[] readPassword(String format, Object... args)
```

The first method reads a password or a password phrase from the console with echoing disabled. The second method does the same, but first prints a formatted prompt.

```
Reader reader()
```

This retrieves the unique `Reader` object associated with this console.

The `Console` class provides the following methods for *writing* formatted strings to the console, and obtaining the writer associated with it:

Click here to view code image

```
Console format(String format, Object... args)
Console printf(String format, Object... args)
```

Write a formatted string to this console's output stream using the specified format string and arguments, according to the default locale. See the `PrintWriter` class with analogous methods (p. 1245).

```
PrintWriter writer()
```

Retrieves the unique `PrintWriter` object associated with this console.

```
void flush()
```

Flushes the console and forces any buffered output to be written immediately.

Example 20.5 illustrates using the `Console` class to change a password. The example illustrates the capability of the `Console` class, and in no way should be construed to provide the ultimate secure implementation to change a password.

The console is obtained at (1). The code at (2) implements the procedure for changing the password. The user is asked to submit the new password, and then asked to confirm it. Note that the password characters are not echoed. The respective `char` arrays returned with this input are compared for equality by the static method `equals()` in the `java.util.Arrays` class, which compares two arrays.

Example 20.5 *Changing Passwords*

Click here to view code image

```
import java.io.Console;
import java.io.IOException;
import java.util.Arrays;

/** Class to change the password of a user */
public class ChangePassword {
  public static void main (String[] args) throws IOException  {

    // Obtain the console:                                          (1)
    Console console = System.console();
    if (console == null) {
      System.err.println("No console available.");
      return;
    }

    // Changing the password:                                       (2)
    boolean noMatch = false;
    do {
      // Read the new password and its confirmation:
      char[] newPasswordSelected
          = console.readPassword("Enter your new password: ");
      char[] newPasswordConfirmed
          = console.readPassword("Confirm your new password: ");

      // Compare the supplied passwords:
      noMatch = newPasswordSelected.length == 0  ||
                newPasswordConfirmed.length == 0 ||
                !Arrays.equals(newPasswordSelected, newPasswordConfirmed);
      if (noMatch) {
        console.format("Passwords don't match. Please try again.%n");
      } else {
        // Necessary code to change the password.
        console.format("Password changed.");
      }
    } while (noMatch);
  }
}
```

Running the program:

```
>java ChangePassword
Enter your new password:
Confirm your new password:
Password changed.
```

Review Questions

20.1 Given the following code, under which circumstance will the method return `false`?

Click here to view code image

```
public static boolean test(InputStream is) throws IOException {
  int value = is.read();
  return value >= 0;
}
```

Select the one correct answer.

a. A character of more than 8 bits was read from the input stream.

b. An I/O error occurred.

c. This method will never return `false`.

d. The end of the stream was reached in the input stream.

20.2 How can we programmatically check whether a call to a `print()` method of the `PrintWriter` class was successful or not?

Select the one correct answer.

a. Check if the return value from the call is `-1`.

b. Check if the return value from the call is `null`.

c. Catch the `IOException` that is thrown when an I/O error occurs.

d. Call the `checkError()` method of the `PrintWriter` class immediately after the `print()` method call returns to see if an `IOException` was thrown.

20.3 Given the following program:

Click here to view code image

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

```
  public static void main(String[] args) {
    try (FileInputStream fis = new FileInputStream("seq.txt");
        InputStreamReader isr = new InputStreamReader(fis);) {
      int i = isr.read();
      while (i != -1) {
        System.out.print((char)i + "|");
        i = isr.read();
      }
    } catch (FileNotFoundException fnf) {
      System.out.println("File not found");
    } catch (EOFException eofe) {
      System.out.println("End of stream");
    } catch (IOException ioe) {
      System.out.println("Input error");
    }
  }
}
```

Assume that the file `seq.txt` exists in the current directory, has the required access permissions, and contains the string `"Hello"`.

Which statement about the program is true?

Select the one correct answer.

a. The program will fail to compile because a certain unchecked exception is not caught.

b. The program will compile and print `H|e|l|l|o|Input error`.

c. The program will compile and print `H|e|l|l|o|End of stream`.

d. The program will compile, print `H|e|l|l|o|`, and then terminate normally.

e. The program will compile, print `H|e|l|l|o|`, and then block in order to read from the file.

f. The program will compile, print `H|e|l|l|o|`, and terminate because of an un-caught exception.

20.4 Given the following program:

Click here to view code image

```
import java.io.*;

public class NoEndings {
  public static void main(String[] args) {
    try (FileReader fr = new FileReader("greetings.txt");
         BufferedReader br = new BufferedReader(fr)) {
      System.out.print(br.readLine() + "|");
      System.out.print(br.readLine() + "|");
      System.out.print(br.readLine() + "|");
    } catch (EOFException eofe) {
      System.out.println("End of stream");
    } catch (IOException ioe) {
      System.out.println("Input error");
    }
  }
}
```

Assume that the file `greeting.txt` exists in the current directory, has the required access permissions, and contains the following two text lines:

```
Hello
Howdy
```

Which statement is true about the program?

Select the one correct answer.

a. The program will fail to compile because the `FileNotFoundException` is not caught.

b. The program will compile, print `Hello|Howdy|null|`, and then terminate normally.

c. The program will compile and print `Hello|Howdy|Input error`.

d. The program will compile and print `Hello|Howdy|End of stream`.

e. The program will compile, print `Hello|Howdy|`, and then block in order to read from the file.

f. The program will compile, print `Hello|Howdy|`, and terminate because of an uncaught exception.

20.5 Object Serialization

Object serialization allows the state of an object to be transformed into a sequence of bytes that can be converted back into a copy of the object (called *deserialization*). After deserialization, the object has the same state as it had when it was serialized, barring any data members that were not serializable. This mechanism is generally known as *persistence*—the serialized result of an object can be stored in a repository from which it can be later retrieved.

Java provides the object serialization facility through the `ObjectInput` and `Object-Output` interfaces, which allow the writing and reading of objects to and from I/O streams. These two interfaces extend the `DataInput` and `DataOutput` interfaces, respectively (**Figure 20.1**, **p. 1235**).

The `ObjectOutputStream` class and the `ObjectInputStream` class implement the `Object-Output` interface and the `ObjectInput` interface, respectively, providing methods to write and read binary representation of both objects as well as Java primitive values. **Figure 20.9** gives an overview of how these classes can be chained to underlying streams and some selected methods they provide. The figure does not show the methods inherited from the abstract `OutputStream` and `InputStream` superclasses.

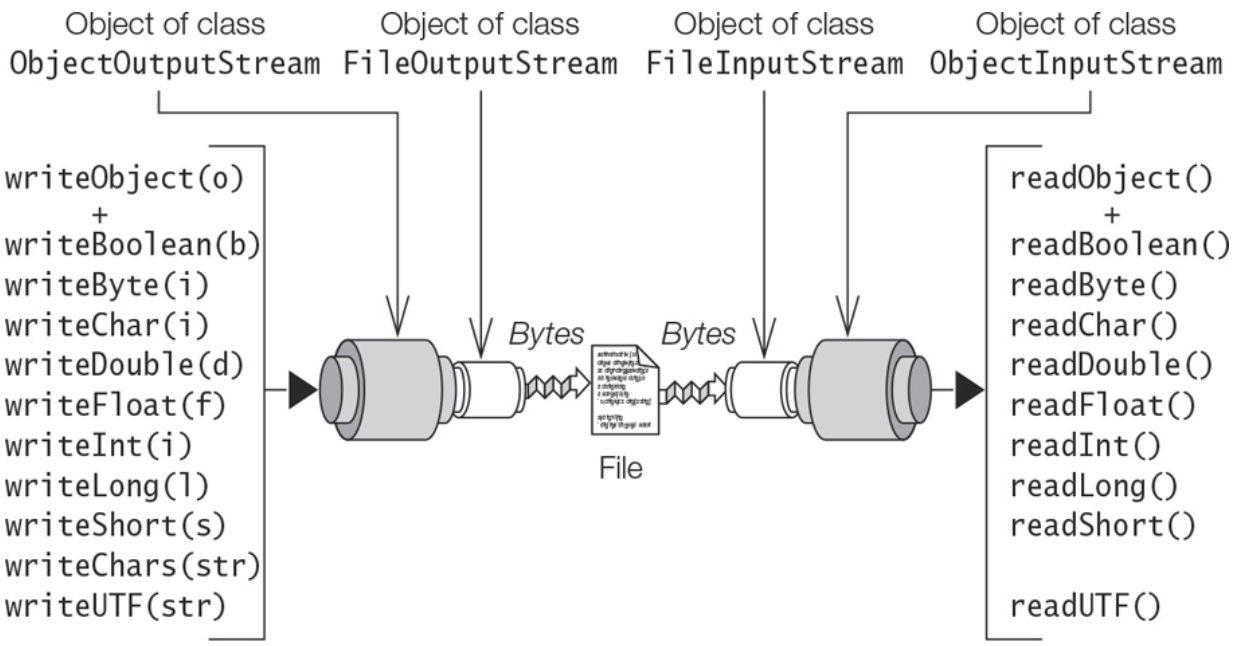


Figure 20.9 *Object Stream Chaining*

The read and write methods in the two classes can throw an `IOException`, and the read methods will throw an `EOFException` if an attempt is made to read past the end of the stream.

The `ObjectOutputStream` Class

The class `ObjectOutputStream` can write objects to any stream that is a subclass of the `OutputStream`—for example, to a file or a network connection (socket). An `Object-OutputStream` must be chained to an `OutputStream` using the following constructor:

Click here to view code image

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

For example, in order to store objects in a file and thus provide persistent storage for objects, an `ObjectOutputStream` can be chained to a `FileOutputStream`:

Click here to view code image

```
FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
ObjectOutputStream outputStream = new ObjectOutputStream(outputFile);
```

Objects can be written to the stream using the `writeObject()` method of the `Object-OutputStream` class:

Click here to view code image

```
final void writeObject(Object obj) throws IOException
```

The `writeObject()` method can be used to write *any* object to a stream, including strings and arrays, as long as the object implements the `java.io.Serializable` interface, which is a *marker interface* with no methods. The `String` class, the primitive wrapper classes, and all array types implement the `Serializable` interface. A serializable object can be any compound object containing references to other objects, and all constituent objects that are serializable are serialized recursively when the compound object is written out. This is true even if there are cyclic references between the objects. Each object is written out only once during serialization. The following information is included when an object is serialized:

- The class information needed to reconstruct the object

- The values of all serializable non-transient and non-static members, including those that are inherited

A checked exception of the type `java.io.NotSerializableException` is thrown if a non-serializable object is encountered during the serialization process. Note also that objects of subclasses that extend a serializable class are always serializable.

The `ObjectInputStream` Class

An `ObjectInputStream` is used to restore (*deserialize*) objects that have previously been serialized using an `ObjectOutputStream`. An `ObjectInputStream` must be chained to an `InputStream`, using the following constructor:

Click here to view code image

```
ObjectInputStream(InputStream in) throws IOException
```

For example, in order to restore objects from a file, an `ObjectInputStream` can be chained to a `FileInputStream`:

Click here to view code image

```
FileInputStream inputFile = new FileInputStream("obj-storage.dat");
ObjectInputStream inputStream = new ObjectInputStream(inputFile);
```

The `readObject()` method of the `ObjectInputStream` class is used to read the serialized state of an object from the stream:

Click here to view code image

```
final Object readObject() throws ClassNotFoundException, IOException
```

Note that the reference type of the returned object is `Object`, regardless of the actual type of the retrieved object, and can be cast to the desired type. Objects and values must be read in the same order as when they were serialized.

Serializable, non-transient data members of an object, including those data members that are inherited, are restored to the values they had at the time of serialization. For compound objects containing references to other objects, the constituent objects are read to re-create the whole object structure. In order to deserialize objects, the appropriate classes must be available at runtime. Note that new objects are created during deserialization, so that no existing objects are overwritten.

The class `ObjectSerializationDemo` in **Example 20.6** serializes some objects in the `writeData()` method at (1), and then deserializes them in the `readData()` method at (2). The `readData()` method also writes the data to the standard output stream.

The `writeData()` method at (1) writes the following values to the output stream: an array of strings (`strArray`), a `long` value (`num`), an array of `int` values (`intArray`), a `String` object (`commonStr`) which is shared with the array `strArray` of strings, and an instance (`oneCD`) of the record class `CD` whose component fields are all serializable.

Duplication is automatically avoided when the same object is serialized several times. The shared `String` object (`commonStr`) is actually only serialized once. Note that the array elements and the characters in a `String` object are not written out explicitly one by one. It is enough to pass the object reference in the `writeObject()` method call. The method also recursively goes through the array of strings, `strArray` , serializing each `String` object in the array. The current state of the `oneCD` instance is also serialized.

The method `readData()` at (2) deserializes the data in the order in which it was written. An explicit cast is needed to convert the reference of a deserialized object to a subtype. Applying the right cast is of course the responsibility of the application. Note that new objects are created by the `readObject()` method, and that an object created during the deserialization process has the same state as the object that was serialized.

Efficient Record Serialization

Example 20.6 shows an example of serializing and deserializing an instance of a record class (**§5.14**, **p. 299**). It is worth noting that both processes on records are very efficient as the state components of a record entirely describe the state values to serialize, and as the canonical constructor of a record class is always used to create the complete state of a record, the canonical constructor is also always used during deserialization. By design, selective and customized serialization, discussed later in this section, are not allowed for records.

Example 20.6 *Object Serialization*

Click here to view code image

```
import java.io.Serializable;
import java.time.Year;
/** A record class that represents a CD. */
public record CD(String artist, String title, int noOfTracks,
                 Year year, Genre genre) implements Serializable {
  public enum Genre implements Serializable {POP, JAZZ, OTHER}
}
```

Click here to view code image

```
//Reading and Writing Objects
import java.io.*;
import java.time.Year;
import java.util.Arrays;

public class ObjectSerializationDemo {
  void writeData() {                                    // (1)
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("obj-storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Write data:
      String[] strArray = {"Seven", "Eight", "Six"};
      long num = 2014;
      int[] intArray = {1, 3, 1949};
      String commonStr = strArray[2];                   // "Six"
      CD oneCD = new CD("Jaav", "Java Jive", 8, Year.of(2017), CD.Genre.POP);
      outputStream.writeObject(strArray);
      outputStream.writeLong(num);
      outputStream.writeObject(intArray);
      outputStream.writeObject(commonStr);
      outputStream.writeObject(oneCD);

    } catch (FileNotFoundException e) {
      System.err.println("File not found: " + e);
    } catch (IOException e) {
      System.err.println("Write error: " + e);
    }
  }

  void readData() {                                     // (2)
    try (// Set up the input stream:
```

```
          FileInputStream inputFile = new FileInputStream("obj-storage.dat");
          ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

        // Read the data:
        String[] strArray = (String[]) inputStream.readObject();
        long num = inputStream.readLong();
        int[] intArray = (int[]) inputStream.readObject();
        String commonStr = (String) inputStream.readObject();
        CD oneCD = (CD) inputStream.readObject();

        // Write data to the standard output stream:
        System.out.println(Arrays.toString(strArray));
        System.out.println(num);
        System.out.println(Arrays.toString(intArray));
        System.out.println(commonStr);
        System.out.println(oneCD);

      } catch (FileNotFoundException e) {
        System.err.println("File not found: " + e);
      } catch (EOFException e) {
        System.err.println("End of stream: " + e);
      } catch (IOException e) {
        System.err.println("Read error: " + e);
      } catch (ClassNotFoundException e) {
        System.err.println("Class not found: " + e);
      }
    }

    public static void main(String[] args) {
      ObjectSerializationDemo demo = new ObjectSerializationDemo();
      demo.writeData();
      demo.readData();
    }
}
```

Output from the program:

Click here to view code image

```
[Seven, Eight, Six]
2014
[1, 3, 1949]
Six
CD[artist=Jaav, title=Java Jive, noOfTracks=8, year=2017, genre=POP]
```

Selective Serialization

As noted earlier, static fields are not serialized, as these are not part of the state of an object. An instance field of an object can be omitted from being serialized by specifying the `transient` modifier in the declaration of the field—typically used for sensitive data in a field. Selective serialization discussed here is *not* applicable to record classes.

Example 20.7 illustrates some salient aspects of serialization. The setup comprises the classes `Wheel` and `Unicycle`, and their client class `SerialClient`. The class `Unicycle` has a field of type `Wheel`, and the class `Wheel` has a field of type `int`. The class `Unicycle` is a compound object with a `Wheel` object as a constituent object. The class `Serial-Client` serializes and deserializes a unicycle in the `try`-with-resources statements at (4) and (5), respectively. The state of the objects is printed to the standard output stream before serialization, and so is the state of the object created by deserialization.

Both the Compound Object and Its Constituents Are Serializable

If we run the program with the following declarations for the `Wheel` and the `Unicycle` classes, where a compound object of the serializable class `Unicycle` uses an object of the serializable class `Wheel` as a constituent object:

Click here to view code image

```
class Wheel implements Serializable {                                  // (1a)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                               // (2)
  private Wheel wheel;                                                 // (3a)
  ...
}
```

we get the following output, showing that both serialization and deserialization were successful:

Click here to view code image

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

A compound object with its constituent objects is often referred to as an *object graph*. Serializing a compound object serializes its complete object graph—that is, the compound object and its constituent objects are recursively serialized.

Example 20.7 *Non-Serializable Objects*

Click here to view code image

```
import java.io.Serializable;

// public class Wheel implements Serializable {                    // (1a)
public class Wheel {                                                // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

Click here to view code image

```
import java.io.Serializable;

public class Unicycle implements Serializable {                     // (2)
  private Wheel wheel;                                              // (3a)
//transient private Wheel wheel;                                    // (3b)

  public Unicycle (Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }
}
```

Click here to view code image

```
import java.io.*;

public class SerialClient {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {

    try (// Set up the output stream:                               // (4)
```

```
          FileOutputStream outputFile = new FileOutputStream("storage.dat");
          ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

        // Write the data:
        Wheel wheel = new Wheel(65);
        Unicycle uc = new Unicycle(wheel);
        System.out.println("Before writing: " + uc);
        outputStream.writeObject(uc);
      }

      try (// Set up the input streams:                                  // (5)
          FileInputStream inputFile = new FileInputStream("storage.dat");
          ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

        // Read data.
        Unicycle uc = (Unicycle) inputStream.readObject();

        // Write data on standard output stream.
        System.out.println("After reading: " + uc);
      }
    }
  }
```

Transient Fields Are Not Serializable

If we declare the `wheel` field of the `Unicycle` class in **Example 20.7** to be `transient` for the sake of demonstrating how such fields are handled at serialization, (3b):

Click here to view code image

```
class Wheel implements Serializable {                                  // (1a)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                               // (2)
  transient private Wheel wheel;                                       // (3b)
  ...
}
```

we get the following output, showing that the `wheel` field of the `Unicycle` object was not serialized:

Click here to view code image

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with null
```

Serializable Compound Object with Non-Serializable Constituents

In order for a compound object to be serialized, its constituent objects must be serializable; otherwise, serialization will not succeed. If the class `Wheel` in **Example 20.7** is *not* serializable, (1b):

Click here to view code image

```
class Wheel {                                                    // (1b)
  private int wheelSize;
  ...
}

class Unicycle implements Serializable {                         // (2)
  private Wheel wheel;                                           // (3a)
  ...
}
```

we get the following output when we run the program—that is, a `Unicycle` object *cannot* be serialized because its constituent `Wheel` object is not serializable:

Click here to view code image

```
>java SerialClient
Before writing: Unicycle with wheel size: 65
Exception in thread "main" java.io.NotSerializableException: Wheel
  ...
  at SerialClient.main(SerialClient.java:20)
```

Customizing Object Serialization

As we have seen, the class of the object must implement the `Serializable` interface if we want the object to be serialized. If this object is a compound object, then all its constituent objects must also be serializable, and so on.

It is not always possible for a client to declare that a class is `Serializable`. It might be declared `final`, and therefore not extendable. The client might not have access to the code, or extending this class with a serializable subclass might not be an option. Java provides a customizable solution for serializing objects in such cases.

Customized serialization discussed here is *not* applicable to record classes.

The basic idea behind the scheme is to use default serialization as much as possible, and to provide *hooks* in the code for the serialization mechanism to call specific methods to deal with objects or values that should not or cannot be serialized by the default methods of the object streams.

Customizing serialization is illustrated in **Example 20.8**, using the `Wheel` and `Unicycle` classes from **Example 20.7**. The serializable class `Unicycle` would like to use the `Wheel` class, but this class is not serializable. If the `wheel` field in the `Unicycle` class is declared to be `transient`, it will be ignored by the default serialization procedure. This is not a viable option, as the unicycle will be missing the wheel size when a serialized unicycle is deserialized, as was illustrated in **Example 20.7**.

Any serializable object has the option of customizing its own serialization if it implements the following pair of methods:

Click here to view code image

```
private void writeObject(ObjectOutputStream) throws IOException;
private void readObject(ObjectInputStream)
                    throws IOException, ClassNotFoundException;
```

These methods are *not* part of any interface. Although private, these methods can be called by the JVM. The first method above is called on the object when its serialization starts. The serialization procedure uses the reference value of the object to be serialized that is passed in the call to the `ObjectOutputStream.writeObject()` method, which in turn calls the first method above on this object. The second method above is called on the object created when the deserialization procedure is initiated by the call to the `ObjectInputStream.readObject()` method.

Customizing serialization for objects of the class `Unicycle` in **Example 20.8** is achieved by the private methods at (3c) and (3d). Note that the field `wheel` is declared `transient` at (3b) and excluded by the normal serialization process.

In the private method `writeObject()` at (3c) in **Example 20.8**, the pertinent lines of code are the following:

Click here to view code image

```
oos.defaultWriteObject();                // Method in the ObjectOutputStream class
oos.writeInt(wheel.getWheelSize());      // Method in the ObjectOutputStream class
```

The call to the `defaultWriteObject()` method of the `ObjectOutputStream` does what its name implies: normal serialization of the current object. The second line of code does the customization: It writes the binary `int` value of the wheel size to the `ObjectOutputStream`. The code for customization can be called both before and after the call to the `defaultWriteObject()` method, as long as the same order is used during deserialization.

In the private method `readObject()` at (3d), the pertinent lines of code are the following:

Click here to view code image

```
ois.defaultReadObject();                 // Method in the ObjectInputStream class
int wheelSize = ois.readInt();           // Method in the ObjectInputStream class
this.wheel = new Wheel(wheelSize);
```

The call to the `defaultReadObject()` method of the `ObjectInputStream` does what its name implies: normal deserialization of the current object. The second line of code reads the binary `int` value of the wheel size from the `ObjectInputStream`. The third line of code creates a `Wheel` object, passes this value in the constructor call, and assigns its reference value to the `wheel` field of the current object. Again, code for customization can be called both before and after the call to the `defaultReadObject()` method, as long as it is in correspondence with the customization code in the `writeObject()` method.

The client class `SerialClient` in **Example 20.8** is the same as the one in **Example 20.7**. The output from the program confirms that the object state prior to serialization is identical to the object state after deserialization.

Example 20.8 *Customized Serialization*

Click here to view code image

```
public class Wheel {                                                    // (1b)
  private int wheelSize;

  public Wheel(int ws) { wheelSize = ws; }

  public int getWheelSize() { return wheelSize; }
```

```
  @Override
  public String toString() { return "wheel size: " + wheelSize; }
}
```

Click here to view code image

```
import java.io.*;

public class Unicycle implements Serializable {                    // (2)
  transient private Wheel wheel;                                   // (3b)

  public Unicycle(Wheel wheel) { this.wheel = wheel; }

  @Override
  public String toString() { return "Unicycle with " + wheel; }

  private void writeObject(ObjectOutputStream oos) {               // (3c)
    try {
      oos.defaultWriteObject();
      oos.writeInt(wheel.getWheelSize());
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  private void readObject(ObjectInputStream ois) {                 // (3d)
    try {
      ois.defaultReadObject();
      int wheelSize = ois.readInt();
      this.wheel = new Wheel(wheelSize);
    } catch (IOException e) {
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      e.printStackTrace();
    }
  }
}
```

Click here to view code image

```
public class SerialClient { // Same as in Example 20.7 }
```

Output from the program:

Click here to view code image

```
Before writing: Unicycle with wheel size: 65
After reading: Unicycle with wheel size: 65
```

Serialization and Inheritance

The inheritance hierarchy of an object also determines what its state will be after it is deserialized. An object will have the same state at deserialization as it had at the time it was serialized if *all* its superclasses are also serializable. This is because the normal object creation and initialization procedure using constructors is *not* run during deserialization.

However, if any superclass of an object is *not* serializable, then the normal creation procedure using *no-argument* or *default* constructors *is* run, starting at the first non-serializable superclass, all the way up to the `Object` class. This means that the state at deserialization might not be the same as at the time the object was serialized because superconstructors run during deserialization may have initialized the state of the object. If the non-serializable superclass does not provide a non-argument constructor or the default constructor, a `java.io.InvalidClassException` is thrown during deserialization.

Example 20.9 illustrates how inheritance affects serialization. The `Student` class is a subclass of the `Person` class. Whether the superclass `Person` is serializable or not has implications for serializing objects of the `Student` subclass, in particular, when their byte representation is deserialized.

Superclass Is Serializable

If the superclass is serializable, then any object of a subclass is also serializable. In **Example 20.9**, the code at (4) in the class `SerialInheritance` serializes a `Student` object:

Click here to view code image

```
Student student = new Student("Pendu", 1007);
System.out.println("Before writing: " + student);
outputStream.writeObject(student);
```

The corresponding code for deserialization of the streamed object is at (5) in the class `SerialInheritance`:

Click here to view code image

```
Student student = (Student) inputStream.readObject();
System.out.println("After reading: " + student);
```

We get the following output from the program in **Example 20.9** when it is run with (1a) and (3a) in the `Person` class and the `Student` class, respectively—that is, when the superclass is serializable and so is the subclass, by virtue of inheritance.

The results show that the object state prior to serialization is identical to the object state after deserialization. In this case, no superclass constructors were run during deserialization.

Click here to view code image

```
Before writing: Student state(Pendu, 1007)
After reading: Student state(Pendu, 1007)
```

Superclass Is Not Serializable

However, the result of deserialization is not the same when the superclass `Person` is not serializable, but the subclass `Student` is. We get the following output from the program in **Example 20.9** when it is run with (1b) and (3b) in the `Person` class and the `Student` class, respectively—that is, when only the subclass is serializable, but not the superclass. The output shows that the object state prior to serialization is not identical to the object state after deserialization.

Click here to view code image

```
Before writing: Student state(Pendu, 1007)
No-argument constructor executed.
After reading: Student state(null, 1007)
```

During deserialization, the *zero-argument* constructor of the `Person` superclass at (2) is run. As we can see from the declaration of the `Person` class in **Example 20.9**, this zero-argument constructor does not initialize the `name` field, which remains initialized with the default value for reference types (`null`).

If the superclass `Person` does not provide the no-argument constructor or the default constructor, as in the declaration below, the call to the `readObject` () method to perform deserialization throws an `InvalidClassException` .

Click here to view code image

```
public class Person  {                                          // (1b)
  private String name;

  public Person(String name) { this.name = name; }

  public String getName() { return name; }
}
```

Output from the program (*edited to fit on the page*):

Click here to view code image

```
Before writing: Student state(Pendu, 1007)
Exception in thread "main" java.io.InvalidClassException:
        Student; no valid constructor
        ...
        at SerialInheritance.main(SerialInheritance.java:28)
```

The upshot of serializing objects of subclasses is that the superclass should be serializable, unless there are compelling reasons for why it is not. And if the superclass is not serializable, it should at least provide either the default constructor or the no-argument constructor to avoid an exception during deserialization.

Although a superclass might be serializable, its subclasses can prevent their objects from being serialized by implementing the `private` method `writeObject(ObjectOutputStream)` that throws a `java.io.NotSerializableException`.

Example 20.9 *Serialization and Inheritance*

Click here to view code image

```
import java.io.Serializable;

// A superclass
public class Person implements Serializable {                  // (1a)
//public class Person  {                                        // (1b)
  private String name;

  public Person() {                                             // (2)
    System.out.println("No-argument constructor executed.");
  }
  public Person(String name) { this.name = name; }
```

```
  public String getName() { return name; }
}
```

Click here to view code image

```
import java.io.Serializable;

public class Student extends Person {                          // (3a)
//public class Student extends Person implements Serializable {  // (3b)

  private long studNum;

  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }

  @Override
  public String toString() {
    return "Student state(" + getName() + ", " + studNum + ")";
  }
}
```

Click here to view code image

```
import java.io.*;

public class SerialInheritance {
  public static void main(String[] args)
      throws IOException, ClassNotFoundException {

    // Serialization:
    try (// Set up the output stream:                                    (4)
        FileOutputStream outputFile = new FileOutputStream("storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Write data:
      Student student = new Student("Pendu", 1007);
      System.out.println("Before writing: " + student);
      outputStream.writeObject(student);
    }

    // Deserialization:
    try (// Set up the input stream:                                     (5)
        FileInputStream inputFile = new FileInputStream("storage.dat");
```

```
          ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

        // Read data.
        Student student = (Student) inputStream.readObject();

        // Write data on standard output stream.
        System.out.println("After reading: " + student);
      }
    }
  }
```

Serialization and Versioning

Class versioning comes into play when we serialize an object with one definition of a class, but deserialize the streamed object with a different class definition. By streamed object, we mean the serialized representation of an object. Between serialization and deserialization of an object, the class definition can change.

Note that at serialization and at deserialization, the definition of the class (i.e., its bytecode file) should be accessible. In the examples so far, the class definition has been the same at both serialization and deserialization. **Example 20.10** illustrates the problem of class definition mismatch at deserialization and the solution provided by Java.

Example 20.10 makes use of the following classes (numbering refers to code lines in the example):

(1) The original version of the serializable class `Item`. It has one field named `price`. An object of this class will be serialized and read using different versions of this class.
(2) A newer version of the class `Item` that has been augmented with a field for the weight of an item. This class will only be used for deserialization of objects that have been serialized with the original version of the class.
(3) The class `Serializer` serializes an object of the original version of the class `Item`.
(4) The class `DeSerializer` deserializes a streamed object of the class `Item`. In the example, deserialization is based on a different version of the `Item` class than the one used at serialization.

There are no surprises if we use the original class `Item` to serialize and deserialize an object of the `Item` class.

Click here to view code image

```
// Original version of the Item class.
public class Item implements Serializable {                          // (1)
  private double price;
//...
}
```

Result:

```
Before writing: Price: 100.00
After reading: Price: 100.00
```

If we deserialize a streamed object of the original class `Item` at (1) based on the byte-code file of the augmented version of the `Item` class at (2), an `InvalidClassException` is thrown at runtime.

Click here to view code image

```
// New version of the Item class.
public class Item implements Serializable {                          // (2)
  private double price;
  private double weight;                                  // Additional field
//...
}
```

Result (*edited to fit on the page*):

Click here to view code image

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = -4194294879924868414,
  local class serialVersionUID = -1186964199368835959
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

The question is, how was the class definition mismatch discovered at runtime? The answer lies in the stack trace of the exception thrown. The local class was incompatible, meaning the class we are using to deserialize is not compatible with the class that was used when the object was serialized. In addition, two long numbers are printed, representing the `serialVersionUID` of the respective class definitions. The first `serialVersionUID` is generated by the serialization process based on the class definition and becomes part of the streamed object. The second `serialVersionUID`

is generated based on the local class definition that is accessible at deserialization. The two are not equal, and deserialization fails.

A serializable class can provide its `serialVersionUID` by declaring it in its class declaration, exactly as shown below, except for the initial value which of course can be different:

Click here to view code image

```
static final long serialVersionUID = 100L;            // Appropriate value.
```

As we saw in the example above, if a serializable class does not provide a `serialVersionUID`, one is implicitly generated. By providing an explicit `serialVersionUID`, it is possible to control what happens at deserialization. As newer versions of the class are created, the `serialVersionUID` can be kept the same until it is deemed that older streamed objects are no longer compatible for deserialization. After the change to the `serialVersionUID`, it will not be possible to deserialize older streamed objects of the class based on newer versions of the class. Although static members of a class are not serialized, the only exception is the value of the `static final long serialVersionUID` field.

In the scenario below, the original version and the newer version of the class `Item` both declare a `serialVersionUID` at (1a) and at (2a), respectively, that has the same value. An `Item` object is serialized using the original version, but deserialized based on the newer version. We see that serialization succeeds, and the `weight` field is initialized to the default value `0.0`. In other words, the object created is of the newer version of the class.

Click here to view code image

```
// Original version of the Item class.
public class Item implements Serializable {    // (1)
  static final long serialVersionUID = 1000L;  // (1a)

  private double price;
//...
}

// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1000L;  // (2a) Same serialVersionUID
  private double price;
  private double weight;                       // Additional field
```

```
//...
}
```

Result:

Click here to view code image

```
Before writing: Price: 100.00
After reading: Price: 100.00, Weight: 0.00
```

However, if we now deserialize the streamed object of the original class having `1000L` as the `serialVersionUID`, based on the newer version of the class having the `serialVersionUID` equal to `1001L`, deserialization fails as we would expect because the `serialVersionUID`s are different.

Click here to view code image

```
// New version of the Item class.
public class Item implements Serializable {    // (2)
  static final long serialVersionUID = 1001L;  // (2b) Different serialVersionUID
  private double price;
  private double weight;
//...
}
```

Result (*edited to fit on the page*):

Click here to view code image

```
Exception in thread "main" java.io.InvalidClassException: Item;
local class incompatible:
  stream classdesc serialVersionUID = 1000,
  local class serialVersionUID = 1001
     ...
     at DeSerializer.main(DeSerializer.java:14)
```

Best practices advocate that serializable classes should use the `serialVersionUID` solution for better control of what happens at deserialization as classes evolve.

Example 20.10 *Class Versioning*

Click here to view code image

```
import java.io.Serializable;

// Original version of the Item class.
public class Item implements Serializable {                              // (1)
//static final long serialVersionUID = 1000L;                            // (1a)

  private double price;

  public Item(double price) {
    this.price = price;
  }

  @Override
  public String toString() {
    return String.format("Price: %.2f%n", this.price);
  }
}
```

Click here to view code image

```
import java.io.Serializable;

// New version of the Item class.
public class Item implements Serializable {                              // (2)
//static final long serialVersionUID = 1000L;                            // (2a)
//static final long serialVersionUID = 1001L;                            // (2b)

  private double price;
  private double weight;
  public Item(double price, double weight) {
    this.price = price;
    this.weight = weight;
  }
  @Override
  public String toString() {
    return String.format("Price: %.2f, Weight: %.2f", this.price, this.weight);
  }
}
```

Click here to view code image

```
// Serializer for objects of class Item.
import java.io.*;

public class Serializer {                                                 // (3)
```

```
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the output stream:
        FileOutputStream outputFile = new FileOutputStream("item_storage.dat");
        ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {

      // Serialize an object of the original class:
      Item item = new Item(100.00);
      System.out.println("Before writing: " + item);
      outputStream.writeObject(item);
    }
  }
}
```

Click here to view code image

```
// Deserializer for objects of class Item.
import java.io.*;

public class DeSerializer{                                          // (4)
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (// Set up the input streams:
        FileInputStream inputFile = new FileInputStream("item_storage.dat");
        ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {

      // Read a serialized object of the Item class.
      Item item = (Item) inputStream.readObject();

      // Write data on standard output stream.
      System.out.println("After reading: " + item);
    }
  }
}
```

Review Questions

20.5 Which of the following best describes the data written by an `ObjectOutputStream`?

Select the one correct answer.

a. Bytes and other Java primitive types

b. Object graphs

c. Object graphs and Java primitive types

d. Single objects

e. Single objects and Java primitive types

20.6 Which of the following statements are true about serialization? Select the four correct answers.

a. All static fields of a class are treated the same way as `transient` instance fields of the class when it comes to serialization.

b. All instance fields of a serializable class must also be serializable or be specified with the `transient` modifier.

c. Any `private` instance field of a class is not serialized.

d. A serializable class must implement the `Serializable` interface.

e. A serializable class must extend the `Serializable` class.

f. Subclasses of a serializable class must explicitly extend the `Serializable` class in order to serialize subclass objects.

g. Subclasses of a serializable class must explicitly implement the `Serializable` interface in order to serialize subclass objects.

h. A serializable class must be specified with the `final` modifier.

i. A `serialVersionUID` is always stored in the streamed object of a serializable class.

20.7 Which of the following statements is true?

Select the one correct answer.

a. All versions of a serializable class must provide a declaration of a `serialVersionUID`.

b. Even if two versions of a serializable class have the same `serialVersionUID`, there is no guarantee that an object serialized based on one version can be deserialized to an object of the other.

c. The `serialVersionUID` of any two unrelated serializable classes must be unique.

d. The `serialVersionUID` of a serializable class must be incremented every time a new version of the class is created.

e. Only a serializable class can declare a `static final` field of type `long` having the name `serialVersionUID`.

20.8 Given the following code:

Click here to view code image

```
public class Person  {
  protected String name;
  public Person() {this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
_____________________________________________________________________
import java.io.Serializable;
public class Student extends Person implements Serializable {
  private long studNum;
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
  public String toString() { return "(" + name + ", " + studNum + ")"; }
}
_____________________________________________________________________
import java.io.*;
public class RQ800_10 {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat");
         ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
      Student stud1 = new Student("Aesop", 100);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {
      Student stud2 = (Student) inputStream.readObject();
      System.out.println(stud2);
    }
```

```
  }
}
```

Which statement about the program is true?

Select the one correct answer.

a. It will fail to compile.

b. It will compile, but it will throw an exception at runtime.

c. It will print `(Aesop, 100)(Aesop, 100)`.

d. It will print `(Aesop, 100)(null, 100)`.

e. It will print `(Aesop, 100)(NoName, 100)`.

20.9 Given the following code:

Click here to view code image

```
import java.io.Serializable;
public class Person implements Serializable {
  protected String name;
  public Person() { this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
_____________________________________________________________________
public class Student extends Person {
  private long studNum;
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
  public String toString() { return "(" + name + ", " + studNum + ")"; }
}
_____________________________________________________________________
import java.io.*;
public class RQ800_30 {
  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat");
         ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
      Student stud1 = new Student("Aesop", 100);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
```

```
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {
      Student stud2 = (Student) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

Which statement about the program is true?

Select the one correct answer.

a. It will fail to compile.

b. It will compile, but it will throw an exception at runtime.

c. It will print `(Aesop, 100)(Aesop, 100)`.

d. It will print `(Aesop, 100)(null, 100)`.

e. It will print `(Aesop, 100)(NoName, 100)`.

20.10 Given the following code:

Click here to view code image

```
public class Person {
  protected transient String name;
  public Person() { this.name = "NoName"; }
  public Person(String name) { this.name = name; }
}
_____________________________________________________________________
public class Student extends Person {
  protected long studNum;
  public Student() { }
  public Student(String name, long studNum) {
    super(name);
    this.studNum = studNum;
  }
}
_____________________________________________________________________
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.Serializable;
```

```
public class GraduateStudent extends Student implements Serializable {
  private int year;
  public GraduateStudent(String name, long studNum, int year) {
    super(name, studNum);
    this.year = year;
  }

  public String toString() {
    return "(" + name + ", " + studNum + ", " + year  + ")";
  }

  private void readObject(ObjectInputStream ois)
      throws IOException, ClassNotFoundException {
    ois.defaultReadObject();
    this.name = "NewName";
    this.studNum = 200;
    this.year = 2;
  }
}
```

```
import java.io.*;
public class RQ800_70 {

  public static void main(String args[])
      throws IOException, ClassNotFoundException {
    try (FileOutputStream outputFile = new FileOutputStream("storage.dat");
         ObjectOutputStream outputStream = new ObjectOutputStream(outputFile)) {
      GraduateStudent stud1 = new GraduateStudent("Aesop", 100, 1);
      System.out.print(stud1);
      outputStream.writeObject(stud1);
    }

    try (FileInputStream inputFile = new FileInputStream("storage.dat");
         ObjectInputStream inputStream = new ObjectInputStream(inputFile)) {
      GraduateStudent stud2 = (GraduateStudent) inputStream.readObject();
      System.out.println(stud2);
    }
  }
}
```

Which statement about the program is true?

Select the one correct answer.

a. It will fail to compile.

b. It will compile, but it will throw an exception at runtime.

c. It will print `(Aesop, 100, 1)(Aesop, 100, 1)`.

d. It will print `(Aesop, 100, 1)(NoName, 0, 1)`.

e. It will print `(Aesop, 100, 1)(NewName, 200, 2)`.

20.11 Given the following code:

Click here to view code image

```
import java.io.Serializable;
public class Product implements Serializable {
  private String name;

  public Product(String name) {
    this.name = name;
    System.out.print("product ");
  }

  @Override
  public String toString() {
    return name;
  }
}
```

```
public class Food extends Product {
  private int calories;

  public Food(String name, int calories) {
    super(name);
    this.calories = calories;
    System.out.print("food ");
  }

  @Override
  public String toString() {
    return super.toString()+" "+calories;
  }
}
```

```
import java.io.*;
public class RQ20 {
  public static void main(String[] args) {

    Product p = new Food("cookie", 300);
```

```
    try(ObjectOutputStream out =
            new ObjectOutputStream(new FileOutputStream("prod.dat"))) {
      out.writeObject(p);
    } catch (Exception ex) {
      System.out.println("error serializing product");
    }
    try(ObjectInputStream in =
            new ObjectInputStream(new FileInputStream("prod.dat"))) {
      p = (Food)in.readObject();
    } catch (Exception ex) {
      System.out.println("error deserializing product");
    }
    System.out.println(p);
  }
}
```

What is the result?

Select the one correct answer.

a. `product food cookie 300`

b. `product food product food cookie 300`

c. `product food cookie 0`

d. `product food product food cookie 0`

e. `product food error serializing product`

f. `product food error deserializing product`

20.12 Given the following code:

Click here to view code image

```
import java.io.*;
public class RQ21 {
  public static void main(String[] args) {
    char[] buffer = new char[4];
    int count = 0;
    try(FileReader in = new FileReader("test1.txt");
        FileWriter out = new FileWriter("test2.txt")) {
      while((count = in.read(buffer)) != -1) {
        out.write(buffer);
      }
```

```
    } catch (Exception ex) {
      System.out.println("error");
    }
  }
}
```

Assume that the `text1.txt` file only contains the line `"abcdefg"`. What is the content of the `text2.txt` file after the program is run?

Select the one correct answer.

a. `abcdefg`

b. `abcd`

c. `abcdefgd`

d. `abcdef`

20.13 Given the following code:

Click here to view code image

```
import java.io.*;
public class Album implements Serializable {
  private static int numberOfTracks = 5;

  private String title;
  private transient int currentTrack;

  public Album(String title, int currentTrack) {
    this.title = title;
    this.currentTrack = currentTrack;
  }

  public void readObject(ObjectInputStream in)
                         throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    currentTrack = 3;
  }

  @Override
  public String toString() {
    return title+" "+numberOfTracks +" "+currentTrack;
  }
}
```

```
_____________________________________________________________________
import java.io.*;
public class RQ22 {
  public static void main(String[] args) {
    Album a = new Album("Songs", 2);
    try (ObjectOutputStream out =
             new ObjectOutputStream(new FileOutputStream("song.dat"))) {
      out.writeObject(a);
    } catch (Exception ex) {
      System.out.println("error serializing product");
    }
    try (ObjectInputStream in =
             new ObjectInputStream(new FileInputStream("song.dat"))) {
      a = (Album)in.readObject();
    } catch (Exception ex) {
      ex.printStackTrace();
    }
    System.out.println(a);
  }
}
```

What is the result?

Select the one correct answer.

a. `Songs 5 3`

b. `Songs 5 2`

c. `Songs 5 0`

d. `Songs 0 3`

e. `Songs 0 2`

f. `Songs 0 0`