```
            // 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:

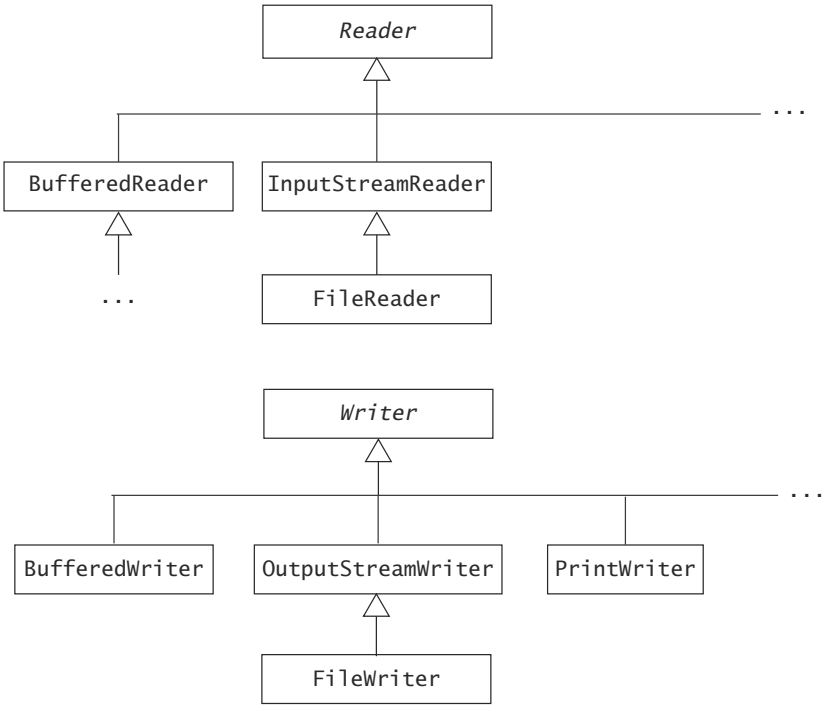
```
true
A
127
-32768
2147483647
-9223372036854775808
3.4028235E38
3.141592653589793
End of input stream.
```

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

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



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.

```
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"`.

```
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.

Table 20.4 *Selected Readers (Continued)*

Reader	Description
`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.

Readers use the following methods for reading Unicode characters:

```
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.

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

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

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

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

```
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.

```
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.

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.

Table 20.5 *Selected Writers (Continued)*

Writers	Description
`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 `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.

Writers use the following methods for writing Unicode characters:

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

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

```
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.

```
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*

```
/* 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:

```
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.

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

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

Writing Text Representation of Primitive Values and Objects

In addition to overriding the `write()` methods from its super class `Writer`, the `PrintWriter` 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).

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).

```
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)):

1. Create a `FileOutputStream`:

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

2. Create an `OutputStreamWriter` which is chained to the `FileOutputStream`:

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

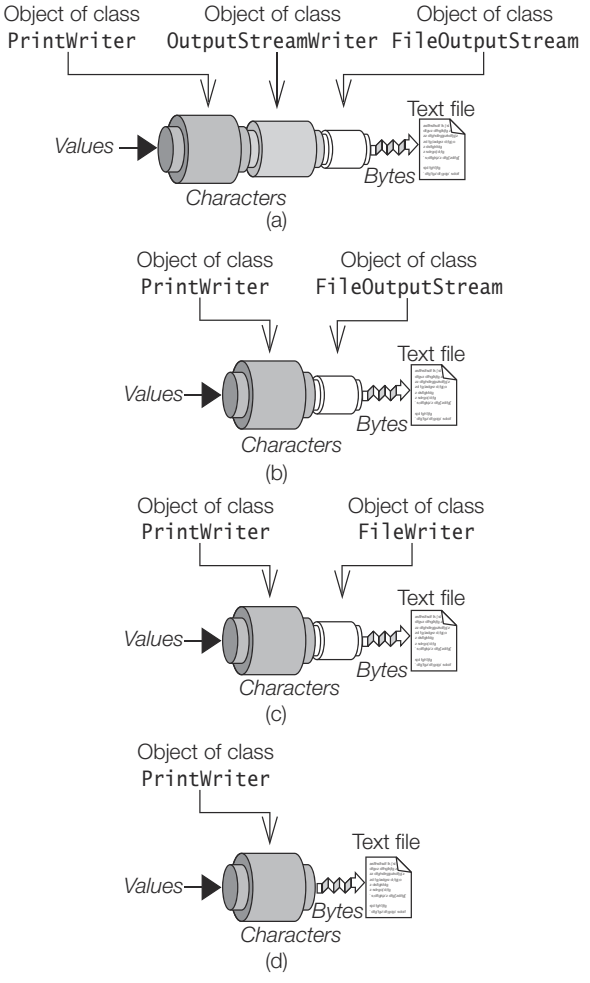
   The `OutputStreamWriter` uses the default character encoding for writing the characters to the file.

3. Create a `PrintWriter` which is chained to the `OutputStreamWriter`:

   ```
   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.

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



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

1. Create a `FileOutputStream`:

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

2. Create a `PrintWriter` which is chained to the `FileOutputStream`:

   ```
   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`:

   ```
   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`:

```
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:

```
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`:

```
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:

```
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)):

1. Create a `FileInputStream`:

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

2. Create an `InputStreamReader` which is chained to the `FileInputStream`:

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

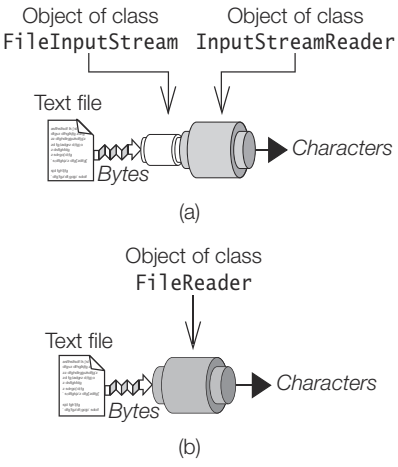
The `InputStreamReader` uses the default character encoding for reading the characters from the file.

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

1. Create a `FileReader`:

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

Figure 20.5 *Setting Up Readers to Read Characters*



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`:

```
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:

```
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:

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

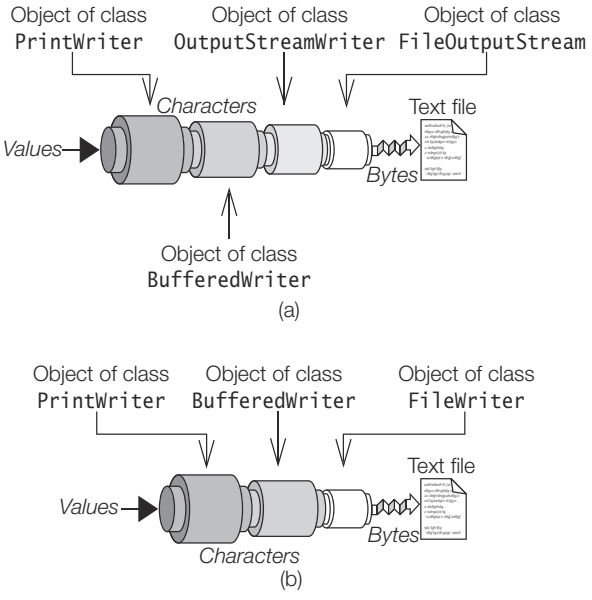
The default buffer size is used, unless the buffer size is explicitly specified.

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)):

```
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);
```

Figure 20.6 *Buffered Writers*



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

```
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 `BufferedWriter` 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:

```
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.

> `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`.

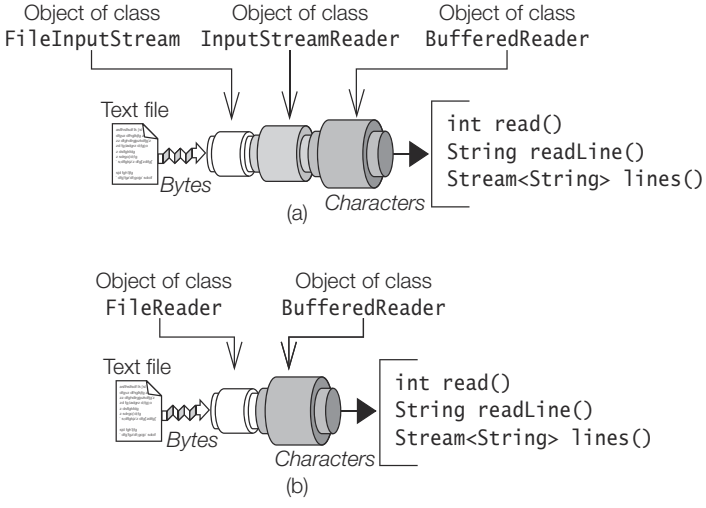
The following code creates a `BufferedReader` that can be used to read text lines from a file (Figure 20.7(b)):

```
// 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.

Figure 20.7 *Buffered Readers*



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*

```
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`
`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*