Java 6 Input Output

Input output (IO) is one of the most common operations performed by a computer program. Examples of IO operations are.

- Create and delete files
- Read from and write to a file or network socket.
- Serialize (or save) objects to persistent storage and retrieve saved objects.

Java provides the **java.io** package that contains types you can use to perform IO operations. Many failed IO operations may throw a **java.io.IOException**. They may also throw a **java.lang.SecurityException** if the failure is related to the lack of permission to perform a certain function.

Learning Java IO programming by iterating the members of **java.io** may not be the best approach, considering there are 12 interfaces, 50 classes, plus 16 exception classes. This chapter therefore presents topics based on functionality and select the most important members of the **java.io** package.

The **java.io.File** class is the first topic in this chapter. It provides methods for creating and deleting files and directories, checking the existence of a file, and so on.

However, the **File** class does not provide functionality to read and write a file's content. For this, you need a stream. Streams, which are discussed in the section "The Concept of Input/Output Streams," act like water pipes that facilitate the transmission of data. There are four types of streams: **InputStream**, **OutputStream**, **Reader**, and **Writer**. For better performance, there are also classes that wrap these streams and buffer the data being read or written. The names of these classes start with **Buffered**

and the classes are **BufferedInputStream**, **BufferedOutputStream**, **BufferedReader**, and **BufferedWriter**.

Reading from and writing to a stream dictate that you do so sequentially, which means to read the second unit of data, you must read the first one first. The **java.io** package provides the **RandomAccessFile** for non-sequential operations. This class is the subsequent topic of discussion.

This chapter concludes with object serialization and deserialization, using the **ObjectInputStream** and **ObjectOutputStream** classes.

The File Class

A **File** object represents a file or directory pathname, and not a physical file or a directory. Therefore, the physical file/directory that a **File** object references does not need to exist. The main advantage of **File** is that it provides a system-independent way of representing a pathname. For example, in Unix/Linux you use a forward slash (/) to separate a directory from a subdirectory or a file. The **myNotes.txt** file in the **tmp** directory can be written as **/tmp/myNotes.txt**. Windows, on the other hand, uses a backslash (****). Therefore, **C:\temp\myNotes.txt** specifies the **myNotes.txt** file in **C:\temp**. When writing Java code to manipulate files and directories, it would be tedious if you had to deal with different separators for different operating systems. Fortunately, the **File** class addresses this issue. For one, you can use its **separator** static field that returns a **String** to separate a directory and a subdirectory or a directory from a file. The value of **separator** depends on the operating system. In Unix/Linux, **separator** returns the string "/". In Windows, it returns "\". The **charSeparator** static field is similar to separator, but returns a **char**.

For instance, if you have a path to a directory named **parent** and a file called **filename**, you can join them in a system-independent way by using this code:

```
parent + File.separator + filename
```

The result will be the correct path to the physical file, regardless the operating system your application is running on.

File Constructors

The **File** class provides several constructors. The simplest one has the following signature:

```
public File(java.lang.String pathname)
```

where *pathname* is either an absolute or relative path name. If *pathname* is null, a **java.lang.NullPointerException** is thrown. For example, you can pass an absolute path to a file or directory like this:

```
File file1 = new File("C:\\temp\\myNote.txt"); // in Windows
File file2 = new File("/tmp/myNote.txt");      // in Linux/Unix
```

Note that in Windows you use the backslash character as the file separator and since it is also the escape character in Java, you need two backslash characters when constructing a **File** in Windows. Using a forward slash as a file separator will also work in Windows, is commonly used and, in my opinion, is less awkward:

```
File file1 = new File("C:/temp/myNote.txt"); // in Windows
```

If you pass a relative path name to the constructor, the path is taken to be relative to the directory from which you run your application. For example, if you invoke the **java** program from **C:\workDir**, the variable **file3** below references a file or directory named **music** under **C:\workDir**.

```
File file3 = new File("music");
```

If you have a file located under a certain directory, you could use this constructor to refer to the file, concatenating the directory and the file using **File.separator**:

```
// userSelectedDir is a directory selected by the user at runtime
// and filename is a String containing the name of the file.
File myFile = new File(userSelectedDir + File.separator +
       filename);
```

However, it is shorter to use the second constructor whose signature is as follows.

```
public File(java.lang.String parent, java.lang.String child)
```

where *parent* is an absolute or relative path to a directory and *child* is the path to a file or a subdirectory. If *child* is an absolute path, then it will be converted into a relative pathname in a system-dependent way. If *parent* is an empty string, then *child* will be converted into an abstract pathname and resolved against a system-dependent default directory.

For example, the following code references a **data** directory under **userSelectedDir**.

```
File myFile = new File(userSelectedDir, data);
```

If *parent* is **null**, it is the same as passing child to the single-argument constructor: **File(*child*)**.

The third constructor is similar to the second one, except that the parent is a **File** object instead of a **String**:

```
public File(File parent, String.java.lang child)
```

If *parent* is **null**, it's the same as invoking the single-argument constructor.

The last constructor of **File** accepts a URI.

```
public File(java.net.URI uri)
```

You use this to create a **File** object by converting the given **file:** URI into an abstract pathname.

File Methods

The following are the more important methods of **File**.

`public boolean canRead()`
: Tests if the application can read the file referenced by this **File** object.

`public boolean canWrite()`
: Tests if the application can write to the file referenced by this **File** object.

`public boolean createNewFile() throws IOException`
: Creates a new empty file in the location and using the name denoted by this File object.

`public boolean delete()`
: Deletes the file or directory referenced by this **File** object.

`public boolean makeDir()`
: Creates the directory named by this **File** object.

`public boolean isFile()`
: Tests if this **File** object references a file.

`public boolean isDirectory()`
: Tests if this **File** object references a directory.

`public boolean exists()`
: Tests if the file or directory denoted by this **File** object exists.

`public File[] listFiles()`
: If this **File** object denotes a directory, this method returns an array of **File** objects referencing the subdirectories and files in the directory. Otherwise, returns **null**.

`public long getTotalSpace()`
: Returns the size, in bytes, of the partition referenced by this **File** object.

`public long getFreeSpace()`
: Returns the amount of free space, in bytes, in the partition referenced by this **File** object.

`public long getUsableSpace()`
: Returns the number of bytes available to this virtual machine on the partition referenced by this **File** object. The difference between **getUsableSpace** and **getFreeSpace** is that the former takes into account restrictions imposed by the operating system, such as write permissions. The latter does not.

The Concept of Input/Output Streams

Java IO streams can be likened to water pipes. Just like water pipes connect city houses to a water reservoir, a Java stream connects Java code to a “data reservoir.” In Java terminology, this “data reservoir” is called a sink and could be a file, a network socket, or memory. The good thing about streams is you employ a uniform way to transport data from and to different sinks, hence simplifying your code. You just need to construct the correct stream. For example, if the sink is a file you need a file stream.

Depending on the data direction, there are two types of streams, input stream and output stream. You use an input stream to read from a sink and an output stream to write to a sink. Because data can be classified into binary data and characters (human readable data), there are also two types of input streams and two types of output streams. These streams are represented by the following four abstract classes in the **java.io** package.

- **Reader**. A stream to read characters from a sink.
- **Writer**. A stream to write characters to a sink.
- **InputStream**. A stream to read binary data from a sink.
- **OutputStream**. A stream to write binary data to a sink.

As mentioned before, the benefit of streams is they define methods for data reading and writing that can be used regardless the data source or destination. To connect to a particular sink, you simply need to construct the correct implementation class. For example, the **Reader** class defines method for reading characters from a sink. If the sink is a file, you instantiate the **FileReader** class. In a similar token, the **FileWriter** class is the **Writer** class implementation that writes to a file, **FileInputStream** is used to read binary data from a file, and **FileOutputStream** represents a stream to write binary data to a file. A typical sequence of operations when working with a stream is as follows:

1. Create a stream. The resulting object is already open, there is no **open** method to call.
2. Perform reading/writing operations.
3. Close the stream by calling its **close** method.

These classes will be discussed in clear detail in the following sections.

Reading Binary Data

You use an **InputStream** to read binary data from a sink. **InputStream** is an abstract class with a number of implementation classes, as shown in Figure 13.1.

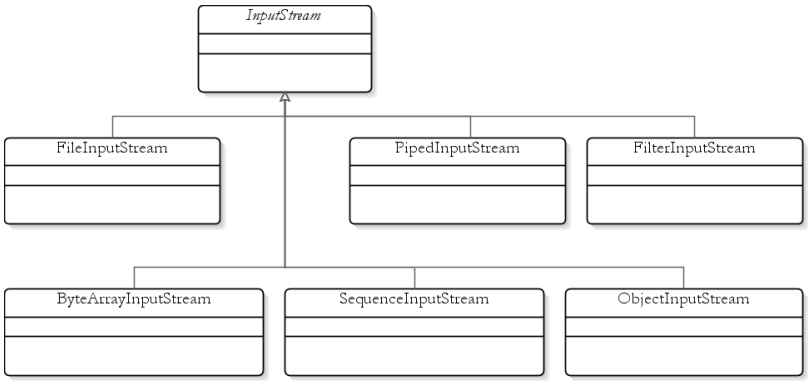

Figure 13.1: The hierarchy of InputStream

This section will discuss two child classes of **InputStream**, **FileInputStream** and **BufferedInputStream**. **FileInputStream** enables easy reading from a file and **BufferedInputStream** provides data buffering that improves performance. The **ObjectInputStream** class is used in object serialization and is discussed in the section, "Object Serialization" later in this chapter.

InputStream

At the core of the **InputStream** class are three **read** method overloads.

```
public int read()
public int read(byte[] data)
public int read(byte[] data, int offset, int length)
```

An **InputStream** employs an internal pointer that points to the starting position of the data to be read. Each of the **read** method overloads returns the number of bytes read or -1 if no data was read into the **InputStream**. This happens when the internal pointer has reached the end of file.

The no-argument **read** method is the easiest to use. It reads the next single byte from this **InputStream** and returns an **int**, which you can then cast to **byte**. Using this method to read a file, you use a **while** block that keeps looping until the **read** method returns -1:

```
int i = inputStream.read();
```

```
while (i != -1) {
    byte b = (byte) I;
    // do something with b
}
```

For speedier reading, you should use the second and third **read** method overloads, which require you to pass a byte array. The data will then be stored in this array. The size of array is a matter of compromise. If you assign a big number, the read operation will be faster because more bytes are read each time. However, this means allocating more memory space for the array. In practice, the array size should start from 1000 and up.

What if there are fewer bytes available than the size of the array? The **read** method overloads return the number of bytes read, so you always know which elements of your array contain valid data. For example, if you use an array of 1,000 bytes to read an **InputStream** and there are 1,500 bytes to read, you will need to invoke the **read** method twice. The first invocation gives you 1,000 bytes, the second 500 bytes.

You can choose to read fewer bytes than the array size using the three-argument **read** method overload:

```
public int read(byte[] data, int offset, int length)
```

This method overload reads *length* bytes into the byte array. The value of *offset* determines the position of the first byte read in the array.

In addition to the **read** methods, there are also these methods:

`public int available() throws IOException`
: This method returns the number of bytes that can be read (or skipped over) without blocking.

`public long skip(long n) throws IOException`
: Skips over the specified number of bytes from this **InputStream**. The actual number of bytes skipped is returned and this may be smaller than the prescribed number.

`public void mark(int readLimit)`
: Remember the current position of the internal pointer in this **InputStream**. Calling reset afterwards will return the pointer to the marked position. The *readLimit* argument specifies the number of bytes to be read before the mark position get invalidated.

`public void reset()`

Repositions the internal pointer in this **InputStream** to the marked position. Its signature is as follows.

`public void close()`

Closes this **InputStream**. You should always call this method when you are done with this **InputStream** to release resources.

You will see an example of **InputStream** in the next section, "FileInputStream."

FileInputStream

The **FileInputStream** class is a subclass of **InputStream** and allows you to read binary data sequentially from a file. Its constructors allow you to pass either a **File** object or a file path. Here are the constructors.

```
public FileInputStream(String path)
public FileInputStream(File file)
```

As an example, the code in Listing 13.1 shows the **FileInputStreamTest** class that contains the **compareFiles** method. This method uses **FileInputStream** and methods from the **InputStream** class to compare files.

Listing 13.1: The compareFiles method that uses FileInputStream

```
package app13;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;

public class FileInputStreamTest {
    public boolean compareFiles (String filePath1,
            String filePath2) {
        boolean areFilesIdentical = true;
        File file1 = new File(filePath1);
        File file2 = new File(filePath2);
        if (!file1.exists() || !file2.exists()) {
            System.out.println("One or both files do not exist");
            return false;
        }
        System.out.println("length:" + file1.length());
```

```
        if (file1.length()!= file2.length()) {
            System.out.println("lengths not equal");
            return false;
        }
        try {
            FileInputStream fis1 = new FileInputStream(file1);
            FileInputStream fis2 = new FileInputStream(file2);
            int i1 = fis1.read();
            int i2 = fis2.read();
            while (i1 != -1) {
                if (i1 != i2) {
                    areFilesIdentical = false;
                    break;
                }
                i1 = fis1.read();
                i2 = fis2.read();
            }
            fis1.close();
            fis2.close();
        } catch (IOException e) {
            System.out.println("IO exception");
            areFilesIdentical = false;
        }
        return areFilesIdentical;
    }

    public static void main(String[] args) {
        FileInputStreamTest test = new FileInputStreamTest();
        test.compareFiles("c: \\line2.bmp","c: \\line3.bmp");
    }
}
```

The **compareFiles** method returns the **boolean areFiledIdentical**, which is **true** only if the compared files are identical. The brain of the method is this block.

```
int i1 = fis1.read();
int i2 = fis2.read();
while (i1 != -1) {
    if (i1 != i2) {
        areFilesIdentical = false;
        break;
    }
    i1 = fis1.read();
    i2 = fis2.read();
}
```

It reads the next byte from the first **FileInputStream** to **i1** and the second **FileInputStream** to **i2** and compares **i1** with **i2**. It continues reading until **i1** and **i2** are different, in which case it will break from the **while** loop.

BufferedInputStream

For better performance you should wrap your **InputStream** with a **BufferedInputStream**. **BufferedInputStream** has two constructors that accept an **InputStream**:

```
public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int bufferSize)
```

For example, the following code wraps a **FileInputStream** with a **BufferedInputStream**.

```
FileInputStream fis = new FileInputStream(aFile);
BufferedInputStream bis = new BufferedInputStream(fis);
```

Then, instead of calling the methods on *fis*, work with the **BufferedInputStream** *bis*.

Writing Binary Data

The **OutputStream** abstract class represents a stream for writing binary data to a sink. Its child classes are shown in Figure 13.2.

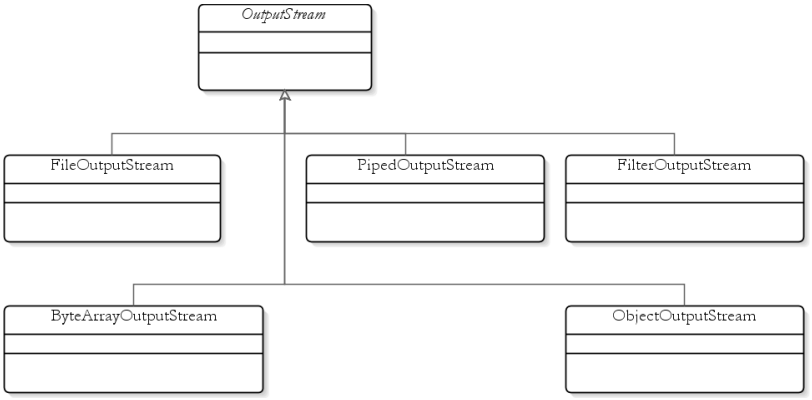

Figure 13.2: The implementation classes of OutputStream

This section discusses two implementation classes: **FileOutputStream** and **BufferedOutputStream**. **FileOutputStream** provides a convenient way to write to a file and **BufferedOutputStream** provides better performance. The **ObjectOutputStream** class plays an important role in object serialization and is discussed in the section, "Object Serialization" later in this chapter.

OutputStream

The OutputStream class defines three **write** method overloads, which are mirrors of the **read** method overloads in **InputStream**:

```
public void write(int b)
public void write(byte[] data)
public void write(byte[] data, int offset, int length)
```

The first overload writes the lowest 8 bits of the integer *b* to this **OutputStream**. The second writes the content of a byte array to this **OutputStream**. The third overload writes *length* bytes of the data starting at offset *offset*.

In addition, there are also the no-argument **close** and **flush** methods. **close** closes the **OutputStream** and **flush** forces any buffered content to be written out to the sink.

We'll look at an example in the next section, "FileOutputStream."

FileOutputStream

The **FileOutputStream** class is a subclass of **OutputStream**. You use **FileOutputStream** to write binary data to a file. The most important thing to note is its constructors. They allow you to construct a **FileOutputStream** object by passing a string containing a path name or a **File** object. You can also specify whether you want to append the output to an existing file.

Here are the signatures of some of its constructors.

```
public FileOutputStream(String path)
```

```
public FileOutputStream(String path, boolean append)
public FileOutputStream(File file)
public FileOutputStream(File file, boolean append)
```

With the first and third constructors, if a file by the specified name already exists, the file will be overwritten. To append to an existing file, pass **true** to the second or fourth constructor.

As an example, Listing 13.2 shows the **FileOutputStreamTest** class that contains the **copyFile** method.

Listing 13.2: The FileOutputStreamTest class

```
package app13;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileOutputStreamTest {
    public void copyFiles(String originPath, String
       destinationPath)
            throws IOException {
        File originFile = new File(originPath);
        File destinationFile = new File(destinationPath);
        if (!originFile.exists() || destinationFile.exists()) {
            throw new IOException(
                    "Origin file must exist and " +
                    "Destination file must not exist");
        }
        try {
            byte[] readData = new byte[1024];
            FileInputStream fis = new FileInputStream(originFile);
            FileOutputStream fos =
                    new FileOutputStream(destinationFile);
            int i = fis.read(readData);
            while (i != -1) {
                fos.write(readData, 0, i);
                i = fis.read(readData);
            }
            fis.close();
            fos.close();
        } catch (IOException e) {
            throw e;
        }
```

```
    }

    public static void main(String[] args) {
        FileOutputStreamTest test = new FileOutputStreamTest();
        try {
            test.copyFiles("c:\\temp\\line1.bmp",
       "c:\\temp\\line3.bmp");
            System.out.println("Copied Successfully");
        } catch (IOException e) {
        }
    }
}
```

This part of the **copyFile** method does the work.

```
byte[] readData = new byte[1024];
FileInputStream fis = new FileInputStream(originFile);
FileOutputStream fos = new FileOutputStream(destinationFile);
int i = fis.read(readData);
while (i != -1) {
    fos.write(readData, 0, i);
    i = fis.read(readData);
}
fis.close();
fos.close();
```

The **readData byte** array is used to store the data read from the **FileInputStream**. The number of bytes read is assigned to **i**. The code then calls the write method on the **FileOutputStream** object, passing the byte array and **i** as the third argument.

```
fos.write(readData, 0, i);
```

BufferedOutputStream

You should always wrap your **OutputStream** with a **BufferedOutputStream** for better performance. **BufferedOutputStream** has two constructors that accept an **OutputStream**.

```
public BufferedOutputStream(OutputStream out)
public BufferedOutputStream(OutputStream out, int bufferSize)
```

The first constructor uses the default buffer size, the second lets you decide. For example, you'll get better performance if you wrap a **FileOutputStream** like this:

```
FileOutputStream fos = new FileOutputStream(aFile);
BufferedOutputStream bos = new BufferedOutputStream(fos);
```

Writing Text (Characters)

The abstract class **Writer** defines a stream used for writing characters. Figure 13.3 shows the implementation classes of **Writer**.

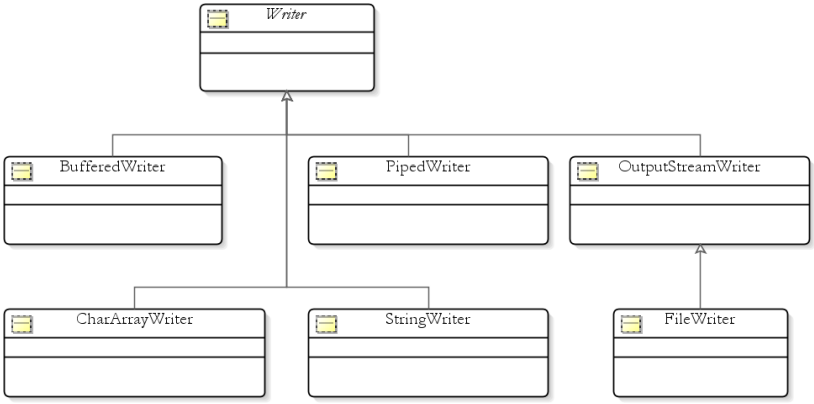

Figure 13.3: The subclasses of Writer

OutputStreamWriter facilitates the translation of characters into byte streams using a given character set. The character set guarantees that any Unicode characters you write to this **OutputStreamWriter** will be translated into the correct byte representation. **FileWriter** is a child class of **OutputStreamWriter** that provides a convenient way to write characters to a file. However, **FileWriter** is not without flaws. When using **FileWriter** you are forced to output characters using the computer's encoding, which means characters outside the current character set will not be translated correctly into bytes. A better alternative to **FileWriter** is **PrintWriter**.

This section discusses the **BufferedWriter** class that buffers characters written to this **Writer** for better performance.

Writer

This class is similar to the **OutputStream** class, except that **Writer** deals with characters instead of bytes. Like **OutputStream**, the **Writer** class has three **write** method overloads:

```
public void write(int b)
public void write(char[] text)
public void write(char[] text, int offset, int length)
```

However, when working with text or characters, you ordinarily use strings. Therefore, there are two other overloads of the **write** method that accept a **String** object.

```
public void write(String text)
public void write(String text, int offset, int length)
```

The last **write** method overload allows you to pass a String and write part of the **String** to this **Writer**.

You will learn to use several implementations of **Writer** in the following subsections.

OutputStreamWriter

An **OutputStreamWriter** is a bridge from character streams to byte streams: Characters written to an **OutputStreamWriter** are encoded into bytes using a specified character set. The latter is an important element of **OutputStreamWriter** because it enables the correct translations of Unicode characters into byte representation.

Note

The **System.getProperty("file.encoding")** method returns the default encoding of your computer.

The **OutputStreamWriter** class has four constructors:

```
public OutputStreamWriter(OutputStream out)
public OutputStreamWriter(OutputStream out,
  java.nio.charset.Charset cs)
public OutputStreamWriter(OutputStream out,
  java.nio.charset.CharsetEncoder enc)
```

```
public OutputStreamWriter(OutputStream out, String encoding)
```

All the constructors accept an **OutputStream**, to which bytes resulting from the translation of characters written to this **OutputStreamWriter** will be written. Therefore, if you want to write to a file, you can pass a **FileOutputStream** to the constructor.

The first constructor creates an instance that uses the default encoding, but the others allow you to pass the encoding that will be used when translating character streams into byte streams. The **OutputStreamWriter** class adds the **getEncoding** method that will return the name of the encoding used in this **OutputStreamWriter** as a **String**.

The **java.nio.charset.Charset** class, an argument to the second constructor, is not discussed in this book, so I will show an example that uses the last constructor. This example is given in Listing 13.3..

Listing 13.3: Using OutputStreamWriter

```
package app13;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;

public class OutputStreamWriterTest {
    public static void main(String[] args) {
        try {
            char[] chars = new char[2];
            chars[0] = '\u4F60'; // representing ?
            chars[1] = '\u597D'; // representing ?;
            String encoding = "GB18030";
            File textFile = new File("C:\\temp\\myFile.txt");
            OutputStreamWriter writer = new OutputStreamWriter(
                    new FileOutputStream(textFile), encoding);
            writer.write(chars);
            writer.close();
        } catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}
```

The code in Listing 13.3 creates an **OutputStreamWriter** based on a **FileOutputStream** that writes to **C:\temp\myFile.txt** on Windows. Therefore, if you are using Linux/Unix you need to change the value of **textFile**. The use of an absolute path is intentional since most readers find it easier to find if they want to open the file. The **OutputStreamWriter** uses the GB18030 encoding, the encoding the Chinese government requires all its software suppliers to use in their applications. This encoding defines the character set for Simplified Chinese characters. The use of an encoding other than English is good for showing how encoding works.

The code in Listing 13.3 passes two Chinese characters: 你 (represented by the Unicode 4F60) and 好 (Unicode 597D). 你好 means 'How are you?' in Chinese.

When executed, the **OutputStreamWriterTest** class will create the **myFile.txt** file. It is 4 bytes long. You can open it and see the Chinese characters. For the characters to be displayed correctly, you need to have the Chinese font installed in your computer.

FileWriter

FileWriter provides a convenient way of writing characters to a file. Using **FileWriter** is fine as long as you are not trying to write characters belonging to other character sets. In other words, if your computer's default language is English, writing Korean characters using **FileWriter** will not work. Unfortunately, there is no way you can change the encoding of a **FileWriter**.

FileWriter has constructors that allow you to construct an instance from a **File**, a file path, or a **FileDescriptor**. You also have the option to append to an existing file or to create a new file. Here are the constructors.

```
public FileWriter(File file)
public FileWriter(File file, boolean append)
public FileWriter(String path)
public FileWriter(String path, boolean append)
public FileWriter(FileDescriptor fileDescriptor)
```

For example, you can construct a **Writer** that writes to a file easily using **FileWriter**:

```
FileWriter writer = new FileWriter("myFile.txt");
writer.write(chars);
```

PrintWriter

PrintWriter is a better alternative to **OutputStreamWriter** and **FileWriter**. Like **OutputStreamWriter**, **PrintWriter** lets you choose an encoding by passing the encoding information to one of its constructors. Here are some of its constructors:

```
public PrintWriter(File file)
public PrintWriter(File file, String characterSet)
public PrintWriter(String filepath)
public PrintWriter(String filepath, String ccharacterSet)
public PrintWriter(OutputStream out)
public PrintWriter(Writer out)
```

For example, using the first, second, third, and fourth constructors, you can create a **PrintWriter** that writes to a file. The two-argument constructors let you pass the name of the character set to use, so you can write any Unicode characters. In addition, you can construct a **PrintWriter** object by passing an **OutputStream** or another **Writer** object.

Also, it is easier to construct a **PrintWriter** than an **OutputStreamWriter**. For example, the following line of code

```
OutputStreamWriter writer = new OutputStreamWriter(
    new FileOutputStream(filePath), encoding);
```

can be replaced by this shorter one.

```
PrintWriter writer = new PrintWriter(filePath, encoding);
```

PrintWriter is more convenient to work with than **OutputStreamWriter** because the former adds nine **print** method overloads that allow you to output any type of Java primitives and objects. Here are the method overloads:

```
public void print(boolean b)
public void print(char c)
public void print(char[] s)
public void print(double d)
public void print(float f)
```

```
public void print(int i)
public void print(long l)
public void print(Object object)
public void print(String string)
```

There are also nine **println** method overloads, which are the same as the **print** method overloads, except that they print a new line character after printing the argument.

In addition, there are two **format** method overloads that enable you to print according to a print format. This method was covered in Chapter 5, "Core Classes."

Listing 13.4 presents an example of **PrintWriter**.

Listing 13.4: Using PrintWriter

```
package app13;
import java.io.IOException;
import java.io.PrintWriter;

public class PrintWriterTest {
    public static void main(String[] args) {
        try {
            PrintWriter pw = new
                    PrintWriter("c:\\temp\\printWriterOutput.txt");
            pw.println("PrintWriter is easy to use.");
            pw.println(1234);
            pw.close();
        } catch (IOException e) {
        }
    }
}
```

The nice thing about writing using a **PrinterWriter** is, when you open the resulting file, everything is human-readable. The file created by the preceding example says:

```
PrinterWriter is easy to use.
1234
```

BufferedWriter

Always wrap your **Writer** with a **BufferedWriter** for better performance. **BufferedWriter** has the following constructors that allow you to pass a **Writer** object.

```
public BufferedWriter(Writer writer)
public BufferedWriter(Writer writer, in bufferSize)
```

The first constructor creates a **BufferedWriter** with the default buffer size (the documentation does not say how big). The second one lets you choose the buffer size.

Therefore, if you are working with a **FileWriter**, you'll get better performance if you wrap it with a **BufferedWriter**:

```
FileWriter fw = new FileWriter(aFile);
BufferedWriter bw = new BufferedWriter(fw);
```

When working with **PrintWriter** (you'll be working mostly with this when you need to output characters to a stream), you cannot wrap it in a similar fashion, such as:

```
PrintWriter pw = new PrintWriter(aFile);
BufferedWriter bw = new BufferedWriter(pw);
```

Because then you would not be able to use the methods of the **PrintWriter**. Instead, wrap the **Writer** that is passed to a **PrintWriter**.

```
FileWriter fw = new FileWriter(aFile);
PrintWriter pw = new PrintWriter(new BufferedWriter(fw));
```

Reading Text (Characters)

You use the **Reader** class to read text (characters, i.e. human readable data). The hierarchy of this class is shown in Figure 13.4.

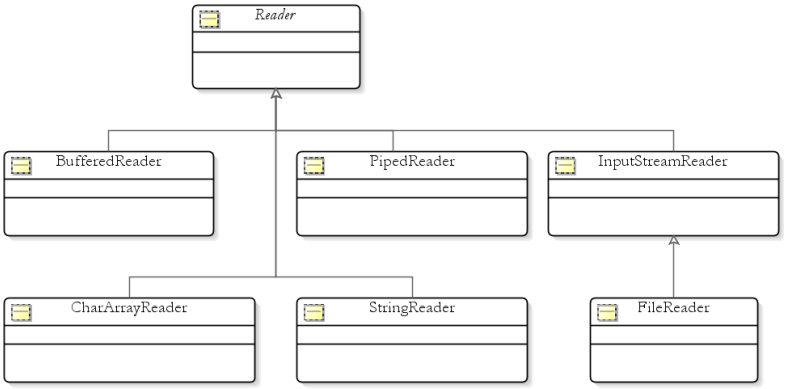

Figure 13.4: Reader and its descendants

This section discusses several child classes of **Reader**.

> **Note**
> The two more popular implementation classes of **Reader** are **InputStreamReader** and **BufferedReader**.

Reader

Reader is an abstract class that represents an input stream for reading characters. It is similar to **InputStream** except that **Reader** objects deal with characters and not bytes. The **Reader** class has three **read** method overloads that are similar to the **read** methods in **InputStream**:

```
public int read()
public int read(char[] data)
public int read(char[] data, int offset, int length)
```

These method overloads allow you to read a single character or multiple characters that will be stored in a char array. Additionally, **Reader** has the fourth read method that enables you to read characters into a **java.nio.CharBuffer**.

```
public int read(java.nio.CharBuffer target)
```

In addition, **Reader** provides the following methods that are similar to those in **InputStream**: **close**, **mark**, **reset**, and **skip**.

InputStreamReader

An **InputStreamReader** reads bytes and translates them into characters using the specified character set. Therefore, **InputStreamReader** is ideal for reading from the output of an **OutputStreamWriter** or a **PrintWriter**. The key is you must know the encoding used when writing the characters to correctly read them back.

The **InputStreamReader** class has four constructors, all of which require you to pass an **InputStream**.

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in,
        java.nio.charset.Charset cs)
public InputStreamReader(InputStream in,
        java.nio.charset.CharsetDecoder, dec)
public InputStreamReader(InputStream in, String charsetName)
```

For instance, to create an **InputStreamReader** that reads from a file, you can pass a **FileInputStream** to its constructor.

```
InputStreamReader reader = new InputStreamReader(
        new FileInputStream(filePath), charSet);
```

Listing 13.5 presents the **InputStreamReaderTest** class that uses a **PrintWriter** to write two Chinese characters and read them back.

Listing 13.5: Using InputStreamReader

```
package app13;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;

public class InputStreamReaderTest {
    public static void main(String[] args) {
        try {
            char[] chars = new char[2];
            chars[0] = '\u4F60'; // representing ?
            chars[1] = '\u597D'; // representing ?;
            String encoding = "GB18030";
```

```
            File textFile = new File("C:\\temp\\myFile.txt");
            PrintWriter writer = new PrintWriter(textFile,
                    encoding);
            writer.write(chars);
            writer.close();

            // read back
            InputStreamReader reader = new InputStreamReader(
                    new FileInputStream(textFile), encoding);
            char[] chars2 = new char[2];
            reader.read(chars2);
            System.out.print(chars2[0]);
            System.out.print(chars2[1]);
            reader.close();
        } catch (IOException e) {
            System.out.println(e.toString());
        }
    }

}
```

FileReader

A subclass of **InputStreamReader**, **FileReader** is a convenient class to read characters from a file. However, like **FileWriter**, it lacks the ability to use an encoding other than the default one. **FileReader** cannot read the characters correctly if the encoding used for writing the file is different than the computer's current encoding.

To construct a **FileReader** object, use one of the following constructors.

```
public FileReader(File file)
public FileReader(FileDescriptor fileDescriptor)
public FileReader(String filePath)
```

BufferedReader

BufferedReader is good for two things:

1. Wraps another **Reader** and provides a buffer that will generally improve performance.
2. Provides a **readLine** method to read a line of text.

The **readLine** method has the following signature:

```
public java.lang.String readLine() throws IOException
```

It returns a line of text from this **Reader** or null if the end of the stream has been reached.

Using **BufferedReader** is very easy. For example, the following lines of code wraps an **InputStreamReader** (which is also a reader) that uses a certain encoding.

```
InputStreamReader inputStreamReader = new InputStreamReader(new
        FileInputStream(textFile), encoding  );
BufferedReader bufferedReader = new
        BufferedReader(inputStreamReader);
```

The resulting **BufferedReader** still supports the encoding of the underlying **InputStreamReader**, but at the same time it supports buffering and provides the **readLine** method.

As another example, this snippet reads a text file and displays it line by line.

```
FileReader fr = new FileReader(aFile);
BufferedReader br = new BufferedReader(fr);
String line = br.readLine();
while (line != null) {
    System.out.println(line);
    line = br.readLine();
}
```

Also, prior to Java 5, you used a **BufferedReader** to read user input to the console. Listing 13.6 shows the **getUserInput** method that can take user input on the console.

Listing 13.6: The getUserInput method

```
public static String getUserInput() {
    BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
    try {
        return br.readLine();
    } catch (IOException ioe) {
    }
    return null;
```

```
}
```

You can do this because **System.in** is of type **java.io.InputStream**.

> **Note**
> The **java.util.Scanner** class can be used to easily read user input. See Chapter 5, “Core Classes” for more detail.

Logging with PrintStream

By now you must be familiar with the print method of **System.out**. You use it especially for displaying messages and this helps you debug your code. However, by default **System.out** sends the message to the console, and this is not always preferable. For instance, if the amount of data displayed exceeds a certain lines, previous messages are no longer visible. Also, you might want to process the messages further, such as sending the messages via email.

The **PrintStream** class is an indirect subclass of **OutpuStream**. It has seven constructors:

```
public PrintStream(File file)
public PrintStream(File file, String characterSet)
public PrintStream(OutputStream out)
public PrintStream(OutputStream out, boolean autoFlush)
public PrintStream(OutputStream out, boolean autoFlush,
        String encoding)
public PrintStream(String filePath)
public PrintStream(String filePath, String characterSet)
```

You can construct a **PrintStream** by passing a **File**, an **OutputStream**, or a path to a file. Some of its constructors let you choose a character set to use.

PrintStream is very similar to **PrintWriter**. For example, both have nine **print** method overloads. Also, **PrintStream** has the **format** method similar to the **format** method of the String class. See Chapter 5, “Core Classes” for more information.

System.out is of type **java.io.PrintStream**. The **System** object lets you replace the default **PrintStream** by using the **setOut** method. Listing 13.7 presents an example that redirect the output of **System.out** to a file.

Listing 13.7: Redirecting System.out to a file

```
package app13;
import java.io.File;
import java.io.IOException;
import java.io.PrintStream;

public class PrintStreamTest {
    public static void main(String[] args) {
        File file = new File("C:\\temp\\debug.txt");
        try {
            PrintStream ps = new PrintStream(file);
            System.setOut(ps);
        } catch (IOException e) {
        }
        System.out.println("To File");
    }
}
```

Note

You can also replace the default **in** and **out** in the **System** object by using **setIn** and **setErr** methods.

RandomAccessFile

Using a stream to access a file dictates that the file is accessed sequentially, e.g. the first character must be read before the second, etc. Streams are ideal when the data comes in a sequential fashion, for example if the medium is a tape (long time ago when the disk had not been invented) or a network socket. Streams are good for most of your applications, however sometimes you need to access a file randomly and using a stream would not be fast enough. For example, you may want to change the 1000^{th} byte of a file without having to read the first 999 bytes. For random access like this, **RandomAccessFile** is ideal to work with.

RandomAccessFile derives directly from **java.lang.Object** and can perform both read and write operations. When opening a file using a **RandomAccessFile**, you can choose whether to open it read-only or read-write. **RandomAccessFile** has two constructors:

```
public RandomAccessFile(File file, String mode)
        throws FileNotFoundException
public RandomAccessFile(String filePath, String mode)
        throws FileNotFoundException
```

The value of **mode** can be one of these:

- "r". Open for reading only.
- "rw". Open for reading and writing. If the file does not already exist, **RandomAccessFile** creates the file.
- "rws". Open for reading and writing and require that every update to the file's content and metadata be written synchronously.
- "rwd". Open for reading and writing and require that every update to the file's content (but not metadata) be written synchronously.

A **RandomAccessFile** employs an internal pointer that points to the next byte to read. When first created, a **RandomAccessFile** points to the first byte. You can change the pointer's position by invoking the **seek** method. Its signature is as follows.

```
public void seek(long position) throws IOException
```

This pointer is zero-based which means the first byte is indicated by index 0. You can pass a number greater than the file size without throwing an exception, but this will not change the size of the file.

In addition to **seek**, the **skipBytes** method moves the pointer by the specified number of bytes. Here is its signature.

```
public int skipBytes(int offset)
```

If skipping *offset* number of bytes will pass the end of file, the internal pointer will only move to as much as the end of file. The **skipBytes** method returns the actual number of bytes skipped.

For reading from a file, **RandomAccessFile** provides a number of methods, each for reading a different data type:

```
public boolean readBoolean() throws IOException
public byte readByte() throws IOException
public char readChar() throws IOException
public double readDouble() throws IOException
public int readInt() throws IOException
public long readLong() throws IOException
```

```
public short readShort() throws IOException
```

In addition, **RandomAccessFile** can also behave like a **Reader** because it provides the **readLine** method that reads a line of text:

```
public String readLine()
```

For faster reading, **RandomAccessFile** provides the **readFully** method overloads, that read data to a byte array:

```
public void readFully(byte[] data)
public void readFully(byte[] data, int offset, int length)
```

For writing, **RandomAccessFile** provides methods that mirror the methods for reading, such as **writeBoolean**, **writeByte**, etc. Plus, there are two **write** method overloads to write the content of a byte array:

```
public void write(byte[] data)
public void write(byte[] data, int offset, int length)
```

RandomAccessFile is suitable for accessing a file that has a fixed structure randomly. For instance, you might use **RandomAccessFile** as a simple database to store fixed-length elements of data.

As an example, the code in Listing 13.8 employs **RandomAccessFile** to store **int**s and changes the value of the third **int**. An **int** takes 4 bytes, therefore the fourth **int** is pointed by the index 3-1 * 4 (3-1 because it's zero-based)

Listing 13.8: Using RandomAccessFile

```
package app13;
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileTest {
    public static void main(String[] args) {
        try {
            RandomAccessFile raf = new RandomAccessFile(
                    "c:/temp/RAFsample.txt", "rw");
            raf.writeInt(10);
            raf.writeInt(43);
            raf.writeInt(88);
            raf.writeInt(455);
```

```
            // change the 3rd integer from 88 to 99
            raf.seek((3 - 1) * 4);
            raf.writeInt(99);
            raf.seek(0); // go to the first integer
            int i = raf.readInt();
            while (i != -1) {
                System.out.println(i);
                i = raf.readInt();
            }
            raf.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Object Serialization

You sometimes need to persist objects in memory into permanent storage so that the states of the objects can be retained and later retrieved. Java supports this through object serialization. To serialize objects, i.e. to save objects to permanent storage, you use an **ObjectOutputStream**. To deserialize objects, namely to retrieved saved objects, use **ObjectInputStream**. **ObjectOutputStream** is a subclass of **OutputStream** and **ObjectInputStream** is derived from **InputStream**.

The **ObjectOutputStream** class has one public constructor:

```
public ObjectOutputStream(OutputStream out)
```

Therefore, to serialize objects to a file, you can pass a **FileOutputStream** to the constructor. Once you have an **ObjectOutputStream**, you can serialize objects or primitives or the combination of both. The **ObjectOutputStream** class provides the **write*XXX*** methods for each individual type, where ***XXX*** denotes a type. Here is the list of the **write*XXX*** methods.

```
public void writeBoolean(boolean value)
public void writeByte(int value)
public void writeBytes(String value)
public void writeChar(int value)
public void writeChars(String value)
public void writeDouble(double value)
```

```
public void writeFloat(float value)
public void writeInt(int value)
public void writeLong(long value)
public void writeShort(short value)
public void writeObject(java.lang.Object value)
```

For objects to be serializable their classes must implement the **java.io.Serializable** interface. This interface has no method and is a marker interface. A marker interface is one that tells the JVM that an instance of an implementing class belongs to a certain type.

If a serialized object contains other objects, the contained objects' classes must also implement **Serializable** for the contained objects to be serializable.

The **ObjectInputStream** class has a public constructor:

```
public ObjectInputStream(InputStream in)
```

Therefore, to deserialize from a file, you can pass a **FileInputStream** to the constructor. The **ObjectInputStream** class has methods that are the opposites of the **write*XXX*** methods in **ObjectOutputStream**. They are as follows:

```
public boolean readBoolean()
public byte readByte()
public char readChar()
public double readDouble()
public float readFloat()
public int readInt()
public long readLong()
public short readShort()
public java.lang.Object readObject()
```

One important thing to note: object serialization is based on a last in first out method. When deserializing multiple primitives/objects, the objects that were serialized first must be deserialized last.

Listing 13.10 shows a class that serializes an **int** and a **Customer** object. Note that the **Customer** class, given in Listing 13.9, implements **Serializable**.

Listing 13.9: The Customer class

```
package app13;
import java.io.Serializable;

public class Customer implements Serializable {
    public int id;
    public String name;
    public String address;
    public Customer (int id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }
}
```

Listing 13.10: Object serialization example

```
package app13;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class ObjectSerializationTest {

    public static void main(String[] args) {
        // Serialize
        try {
            Customer customer = new Customer(1, "Joe Blog",
                    "12 West Cost");
            FileOutputStream fos = new FileOutputStream(
                    "c:\\temp\\objectOutput");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            // write first object
            oos.writeObject(customer);
            // write second object
            oos.writeObject("Customer Info");
            oos.close();
            fos.close();
        } catch (FileNotFoundException e) {
            System.out.print("FileNotFound");
        } catch (IOException e) {
            System.out.print("IOException");
        }

        // Deserialize
```

```
        try {
            FileInputStream fis =
                    new FileInputStream("c:\\temp\\objectOutput");
            ObjectInputStream ois = new ObjectInputStream(fis);
            // read first object
            Customer customer2 = (Customer) ois.readObject();
            System.out.println("First Object: ");
            System.out.println(customer2.id);
            System.out.println(customer2.name);
            System.out.println(customer2.address);

            // read second object
            System.out.println();
            System.out.println("Second object: ");
            String info = (String) ois.readObject();
            System.out.println(info);
            ois.close();
            fis.close();
        } catch (ClassNotFoundException ex) {
            System.out.print("ClassNotFound " + ex.getMessage());
        } catch (IOException ex2) {
            System.out.print("IOException " + ex2.getMessage());
        }
    }
}
```

Summary

Input output operations are supported through the members of the **java.io** package. You can read and write data through streams and data is classified into binary data and text. In addition, Java support object serialization through the **Serializable** interface and the **ObjectInputStream** and **ObjectOutputStream** classes.

Questions

1. What is a stream?
2. Name four abstract classes that represent streams in the **java.io** package.

3. What is object serialization?
4. What is the requirement for a class to be serializable?