12 File Input and Output

Objectives

After you have read and studied this chapter, you should be able to

- Include a **JFileChooser** object in your program to let the user specify a file.
- Write bytes to a file and read them back from the file, using **FileOutputStream** and **FileInputStream.**
- Write values of primitive data types to a file and read them back from the file, using **DataOutput Stream** and **DataInputStream.**
- Write text data to a file and read them back from the file, using **PrintWriter** and **BufferedReader.**
- Read a text file using **Scanner.**
- Write objects to a file and read them back from the file, using **ObjectOutputStream** and **ObjectInputStream.**

Introduction

What is the most important action you should never forget to take while developing programs or writing documents? Saving the data, of course! It's 3 A.M., and you're in the home stretch, applying the finishing touches to the term paper due at 9 A.M. Just as you are ready to select the Print command for the final copy, it happens. The software freezes and it won't respond to your commands anymore. You forgot to turn on the Autosave feature, and you have not saved the data for the last hour. There's nothing you can do but reboot the computer.

Data not saved will be lost, and if we ever want to work on the data again, we must save the data to a file. We call the action of saving, or writing, data to a file ***file output*** and the action of reading data from a file ***file input***. A program we develop must support some form of file input and output capabilities for it to have practical uses. Suppose we develop a program that keeps track of bicycles owned by the dorm students. The program will allow the user to add, delete, and modify the bicycle information. If the program does not support the file input and output features, every time the program is started, the user must reenter the data.

file output and input

In this chapter, we will introduce the classes from the java.io and javax.swing packages that are used for file input and output operations. Also, we will show how the two helper classes from Chapters 8 and 9—Dorm and FileManager—that provided the file input and output support are implemented.

12.1 File and JFileChooser Objects

In this section we introduce two key objects for reading data from or writing data to a file. We use the term *file access* to refer to both read and write operations. If we need to be precise, we write *read access* or *write access*. (We use the terms *save* and *write* interchangeably to refer to file output, but we do not say *save access*.) Suppose we want to read the contents of a file sample.data. Before we can start reading data from this file, we must first create a **File** object (from the java.io package) and associate it to the file. We do so by calling a File constructor:

File

```
File inFile = new File("sample.data");
```

The argument to the constructor designates the name of the file to access. The system assumes the file is located in the ***current directory***. For the following examples, we assume the directory structure shown in Figure 12.1, with Ch12 being the current directory. When you run a program whose source file is located in directory X, then the current directory is X. Please refer to Java compiler manuals for other options for designating the current directory.

current directory

It is also possible to open a file that is stored in a directory other than the current directory by providing a path name and a filename. Assuming there's a file xyz.data in the JavaPrograms directory, we can open it by executing

```
File inFile = new File("C:\\JavaPrograms", "xyz.data");
```

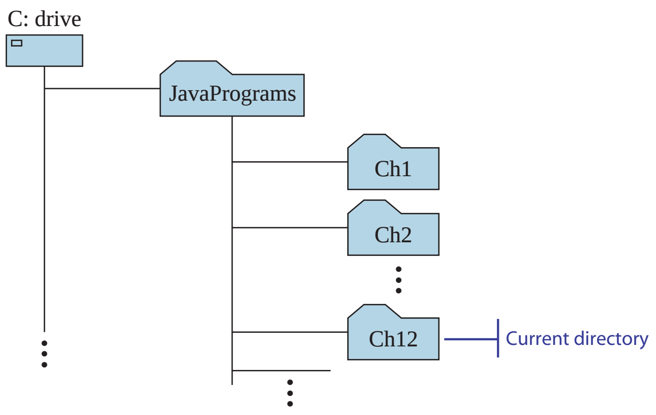


Figure 12.1 Directory structure used for the examples in this section. We assume the Windows environment.

This style of designating the path name is for the Windows platform. The actual path name we want to specify is

```
C:\JavaPrograms
```

but the backslash character is an escape character. So to specify the backslash character itself, we must use double backslashes. For the UNIX platform, we use the forward slash for a delimiter, for example,

```
"/JavaPrograms"
```

For the Mac platform, we also use a forward slash; for example, if the name of a hard disk is `MacHD`, then we write

```
"/MacHD/JavaPrograms"
```

To maintain the consistency across the platforms, the forward slash character is allowed for the Windows platform also, such as in

```
"C:/JavaPrograms/Ch12"
```

The path name could be absolute or relative to the current directory. The absolute path name is the full path name beginning with the disk drive name, for example,

```
"C:/JavaPrograms/Ch12"
```

The relative path name is relative to the current directory. For example, if the current directory is `Ch12`, then the relative path name

```
"../Ch12"
```

is equivalent to the full path name

```
"C:/JavaPrograms/Ch12"
```

where the two dots (. .) in the string mean "one directory above."

We can check if a File object is associated correctly to an existing file by calling its exists method:

```
if (inFile.exists( )) {
   // inFile is associated correctly to an existing file

} else {
   // inFile is not associated to any existing file
}
```

When a valid association is established, we say *the file is opened;* a file must be opened before we can do any input and output to the file.

Things to Remember

A file must be opened before we can execute any file access operations.

A File object can also be associated to a directory. For example, suppose we are interested in listing the content of directory Ch12. We can first create a File object and associate it to the directory. After the association is made, we can list the contents of the directory by calling the object's list method:

```
File   directory  = new File("C:/JavaPrograms/Ch12");
String filename[] = directory.list();

for (int i = 0; i < filename.length; i++) {
   System.out.println(filename[i]);
}
```

We check whether a File object is associated to a file or a directory by calling its boolean method isFile. The following code will print out I am a directory:

```
File file = new File("C:/JavaPrograms/Ch12");

if (file.isFile()) {
   System.out.println("I am a file");

} else {
   System.out.println("I am a directory");
}
```

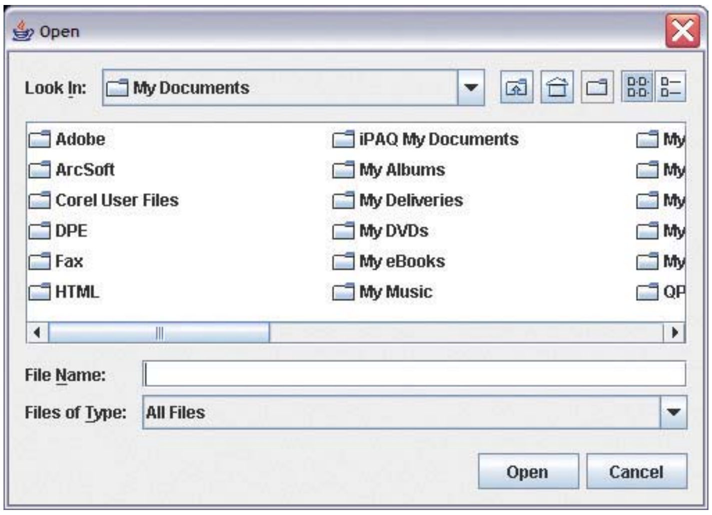


Figure 12.2 A sample **JFileChooser** object displayed with the **showOpenDialog** method. The dialog title and the okay button are labeled **Open.**

JFileChooser

We can use a javax.swing.JFileChooser object to let the user select a file. The following statement displays an open file dialog, such as the one shown in Figure 12.2 (the actual listing depends on the machine on which the program is executed):

```
JFileChooser chooser = new JFileChooser( );
...
chooser.showOpenDialog(null);
```

The null argument to the showOpenDialog indicates that there's no parent frame window, and the dialog is displayed at the center of the screen. We pass a frame window object if we want to position the file dialog at the center of the frame. A frame window is discussed in Chapter 14. In this chapter, we use a null for the argument.

When we create an instance of JFileChooser by passing no arguments to the constructor, as in this example, it will list the content of the user's home directory. For the Windows platform, the user's home directory by default is the My Documents folder. We can set the file chooser to list the contents of a desired directory when it first appears on the screen. We can do this in two ways. The first is to pass the path name of the directory as a String argument to the constructor. For example, if we want to start the listing from the C:/JavaPrograms/Ch12 directory, then we write

```
JFileChooser chooser
              = new JFileChooser("C:/JavaPrograms/Ch12");
. . .
chooser.showOpenDialog(null);
```

The second way is to use the setCurrentDirectory method as follows:

```
File startDir = new File("C:/JavaPrograms/Ch12");
chooser.setCurrentDirectory(startDir);
...
chooser.showOpenDialog(null);
```

Notice that we have to pass a File object, not a String, to the setCurrentDirectory method.

Instead of designating a fixed directory as in this example, we may wish to begin the listing from the current directory. Since the current directory is different when the program is executed from a different directory, we need a general approach. We can achieve the generality by writing

```
String current = System.getProperty("user.dir");
JFileChooser chooser
                 = new JFileChooser(current);
...
```

or equivalently

```
String current = System.getProperty("user.dir");
JFileChooser chooser
                 = new JFileChooser( );
chooser.setCurrentDirectory(new File(current));
...
```

The content of current is the path name to the current directory.

To check whether the user has clicked on the Open or Cancel button, we test the return value from the showOpenDialog method.

```
int status = chooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION) {
   System.out.println("Open is clicked");
} else { //== JFileChooser.CANCEL_OPTION
   System.out.println("Cancel is clicked");
}
```

Once we determine the Open button is clicked, we can retrieve the selected file as

```
File selectedFile;
selectedFile = chooser.getSelectedFile();
```

and the current directory of the selected file as

```
File currentDirectory;
currentDirectory = chooser.getCurrentDirectory();
```

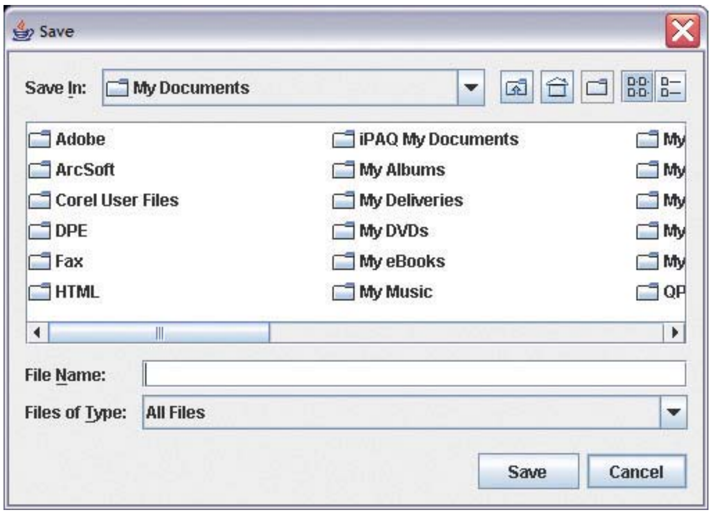


Figure 12.3 A sample **JFileChooser** object displayed with the **showCloseDialog** method. The dialog title and the okay button are labeled **Save.**

To find out the name and the full path name of a selected file, we can use the getName and getAbsolutePath methods of the File class.

```
File file = chooser.getSelectedFile();

System.out.println("Selected File: " +
                            file.getName());
System.out.println("Full path:     " +
                            file.getAbsolutePath());
```

To display a JFileChooser with the Save button, we write

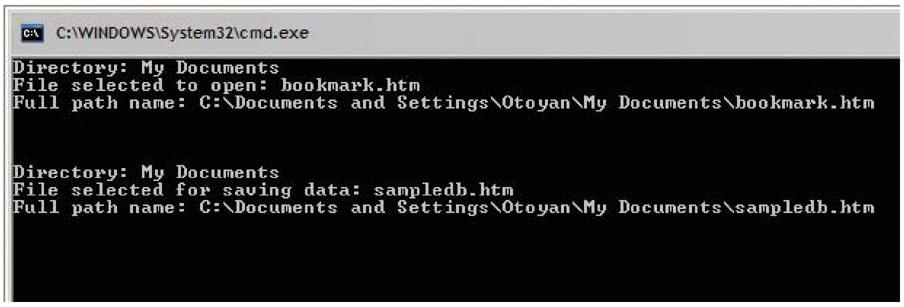
```
chooser.showSaveDialog(null);
```

which results in a dialog shown in Figure 12.3 (the actual listing depends on the machine on which the program is executed).

The following Ch12TestJFileChooser class summarizes the methods of JFile-Chooser and File classes. Note this sample program does not perform actual file input or output.

```
/*
    Chapter 12 Sample Program: Illustrate the use of the
                               JFileChooser and File classes.

    File: Ch12TestJFileChooser.java
*/
```

```
import java.io.*;
import javax.swing.*;

class Ch12TestJFileChooser {
    public static void main (String[] args) {

        JFileChooser chooser;
        File         file, directory;
        int          status;

        chooser = new JFileChooser( );

        status = chooser.showOpenDialog(null);

        if (status == JFileChooser.APPROVE_OPTION) {
            file       = chooser.getSelectedFile();
            directory  = chooser.getCurrentDirectory();

            System.out.println("Directory: " +
                                directory.getName());

            System.out.println("File selected to open: " +
                                file.getName());

            System.out.println("Full path name: " +
                                file.getAbsolutePath());

        } else {
            System.out.println("Open File dialog canceled");
        }

        System.out.println("\n\n");

        status = chooser.showSaveDialog(null);

        if (status == JFileChooser.APPROVE_OPTION) {
            file       = chooser.getSelectedFile();
            directory  = chooser.getCurrentDirectory();

            System.out.println("Directory: " +
                                 directory.getName());

            System.out.println("File selected for saving data: " +
                                 file.getName());

            System.out.println("Full path name: " +
                                file.getAbsolutePath());
        } else {
            System.out.println("Save File dialog canceled");
        }
    }
}
```



```
Directory: My Documents
File selected to open: bookmark.htm
Full path name: C:\Documents and Settings\Otoyan\My Documents\bookmark.htm


Directory: My Documents
File selected for saving data: sampledb.htm
Full path name: C:\Documents and Settings\Otoyan\My Documents\sampledb.htm
```

Figure 12.4 A sample output from running the **Ch12TestJFileChooser** program once.

Figure 12.4 shows a sample output of running the program once.

There is actually no distinction between the Open and Save dialogs created, respectively, by showOpenDialog and showCloseDialog other than the difference in the button label and the dialog title. In fact, they are really a shorthand for calling the showDialog method. Using the showDialog method, we can specify the button label and the dialog title. For example, this code will produce a JFileChooser dialog with the text Compile as its title and label for the okay button:

```
JFileChooser chooser = new JFileChooser();
chooser.showDialog(null, "Compile");
```

file filter

We can use a *file filter* to remove unwanted files from the list. Let's say we want to apply a filter so only the directories and the Java source files (those with the .java extension) are listed in the file chooser. To do so, we must define a subclass of the javax.swing.filechooser.FileFilter class and provide the accept and getDescription methods. The prototypes of these methods are

```
public boolean accept(File file)
public String getDescription( )
```

The accept method returns true if the parameter file is a file to be included in the list. The getDescription method returns a text that will be displayed as one of the entries for the "Files of Type:" drop-down list. Here's how the filter subclass is defined:

```
/*
    Chapter 12 Sample Program: Illustrate how to filter only
                               Java source files
                               for listing in JFileChooser

    File: JavaFilter.java
*/

import java.io.File;
import javax.swing.filechooser.*;
```

Notice that we are stating one class in the package explicitly, instead of using the more common form of **import java.io.*;** to avoid naming conflict. The **java.io** package has the interface named **FileFilter.**

```
class JavaFilter extends FileFilter {
```

Data members

```
    private static final String JAVA  = "java";
    private static final char   DOT   = '.';
```

accept

```
    //accepts only directories and
    //files with .java extension only
    public boolean accept(File f) {

        if (f.isDirectory()) {
            return true;
        }

        if (extension(f).equalsIgnoreCase(JAVA)) {
            return true;
        } else {
            return false;
        }
    }
```

getDescription

```
    //description of the filtered files
    public String getDescription( ) {
        return "Java source files (.java)";
    }
```

extension

```
    //extracts the extension from the filename
    private String extension(File f) {

        String filename = f.getName();
        int    loc      = filename.lastIndexOf(DOT);

        if (loc > 0 && loc < filename.length() - 1) {
            //make sure the dot is not
            //at the first or the last character position
            return filename.substring(loc+1);
        } else {
            return "";
        }
    }
}
```

With the filter class Java Filter in place, we can set a file chooser to list only directories and Java source files by writing

```
JFileChooser chooser = new JFileChooser( );

chooser.setFileFilter(new JavaFilter(());

int status = chooser.showOpenDialog(null);
```

Quick CHECK

1. This question is specific to the Windows platform. Suppose you want to open a file prog1.java inside the directory C:\JavaProjects\Ch11\Step4. What is the actual String value you pass in the constructor for the File class?
2. What is wrong with the following statement?

```
JFileChooser chooser
   = new JFileChooser("Run");
   chooser.showDialog(null);
```

3. Which method of the JFileChooser class do you use to get the filename of the selected file? What is returned from the method if the Cancel button is clicked?

12.2 Low-Level File I/O

Once a file is opened by properly associating a File object to it, the actual file access can commence. In this section, we will introduce basic objects for file operations. To actually read data from or write data to a file, we must create one of the Java stream objects and attach it to the file. A *stream* is simply a sequence of data items, usually 8 bits per item. Java has two types of streams: an input stream and an output stream. An input stream has a *source* from which the data items come, and an output stream has a *destination* to which the data items go. To read data items from a file, we attach one of the Java input stream objects to the file. Similarly, to write data items to a file, we attach one of the Java output stream objects to the file.

stream

source

destination

Java comes with a large number of stream objects for file access operations. We will cover only those that are straightforward and easy to learn for beginners. We will study two of them in this section—FileOutputStream and FileInputStream. These two objects provide low-level file access operations. In Section 12.3 we will study other stream objects.

FileOutput-Stream

Let's first study how to write data values to a file by using **FileOutputStream**. Using a FileOutputStream object, we can output only a sequence of bytes, that is, values of data type byte. In this example, we will output an array of bytes to a file named sample1.data. First we create a File object:

```
File outFile = new File("sample1.data");
```

Then we associate a new FileOutputStream object to outFile:

```
FileOutputStream outStream
                       = new FileOutputStream(outFile);
```

Now we are ready for output. Consider the following byte array:

```
byte[] byteArray = {10, 20, 30, 40, 50, 60, 70, 80};
```

We write the whole byte array at once to the file by executing

```
outStream.write(byteArray);
```

Notice that we are not dealing with the File object directly, but with outStream. It is also possible to write array elements individually, for example,

```
//output the first and fifth bytes
outStream.write(byteArray[0]);
outStream.write(byteArray[4]);
```

After the values are written to the file, we must close the stream:

```
outStream.close();
```

data caching

data buffer

If the stream object is not closed, then some data may get lost due to data caching. Because of the physical characteristics of secondary memory such as hard disks, the actual process of saving data to a file is a very time-consuming operation, whether you are saving 1 or 100 bytes. So instead of saving bytes individually, we save them in a block of, say, 500 bytes to reduce the overall time it takes to save the whole data. The operation of saving data as a block is called *data caching*. To carry out data caching, a part of memory is reserved as a *data buffer* or *cache*, which is used as a temporary holding place. A typical size for a data buffer is anywhere from 1 KB to 2 KB. Data are first written to a buffer, and when the buffer becomes full, the data in the buffer are actually written to a file. If there are any remaining data in the buffer and the file is not closed, then those data will be lost. Therefore, to avoid losing any data, it is important to close the file at the end of the operations.

Things to Remember

To ensure that all data are saved to a file, close the file at the end of file access operations.

Many of the file operations, such as write and close, throw I/O exceptions, so we need to handle them. For the short sample programs, we use the propagation approach. Here's the complete program:

```
/*
    Chapter 12 Sample Program:
             A test program to save data to a file using FileOutputStream

    File: Ch12TestFileOutputStream.java
*/
import java.io.*;

class Ch12TestFileOutputStream {
   public static void main (String[] args) throws IOException {
```

Needs this clause because the file methods throw I/O exceptions.

```
      //set up file and stream
      File             outFile   = new File("sample1.data");
      FileOutputStream outStream = new FileOutputStream(outFile);

      //data to output
      byte[] byteArray = {10, 20, 30, 40, 50, 60, 70, 80};

      //write data to the stream
      outStream.write(byteArray);

      //output done, so close the stream
      outStream.close();
   }
}
```

Hints, & Tips, Pitfalls

It may seem odd at first to have both **File** and **FileStream** objects to input data from a file. Why not have just a **File** to handle everything? **File** represents a physical file that is a source of data. **Stream** objects represent the mechanism we associate to a file to perform input and output routines. **Stream** objects can also be associated to a nonfile data source such as a serial port. So separating the tasks following the STO principle resulted in more than one class to input data from a file.

Now it's true that we can make a shortcut statement such as

```
fileOutputStream outStream
    = new FileOutputStream("input.txt");
```

where we avoid the explicit creation of a **File** object. But this shortcut does not eliminate the fact that the **Stream** object is associated to a file.

FileInput-Stream

To read the data into a program, we reverse the steps in the output routine. We use the read method of **FileInputStream** to read in an array of bytes. First we create a FileInputStream object:

```
File            inFile    = new File("sample1.data");
FileInputStream inStream  = new FileInputStream(inFile);
```

Then we read the data into an array of bytes:

```
inStream.read(byteArray);
```

Before we call the read method, we must declare and create byteArray:

```
int    filesize   = (int) inFile.length();
byte[] byteArray  = new byte[filesize];
```

We use the length method of the File class to determine the size of the file, which in this case is the number of bytes in the file. We create an array of bytes whose size is the size of the file.

The following program uses FileInputStream to read in the byte array from the file sample1.data.

```
/*
   Chapter 12 Sample Program:
             A test program to read data from a file using
FileInputStream

   File: Ch12TestFileInputStream.java
*/
import java.io.*;

class Ch12TestFileInputStream {
   public static void main (String[] args) throws IOException {

      //set up file and stream
      File             inFile    = new File("sample1.data");
      FileInputStream  inStream  = new FileInputStream(inFile);

      //set up an array to read data in
      int fileSize = (int) inFile.length();
      byte[] byteArray = new byte[fileSize];

      //read data in and display them
      inStream.read(byteArray);
      for (int i = 0; i < fileSize; i++) {
          System.out.println(byteArray[i]);
      }

      //input done, so close the stream
      inStream.close();
   }
}
```

It is possible to output data other than bytes if we can convert (i.e., typecast) them into bytes. For example, we can output character data by typecasting them to bytes.

```
File             outFile   = new File("sample1.data");
FileOutputStream outStream = new FileOutputStream(outFile);

//data to output
byte[] byteArray = {(byte) 'J',
                     (byte) 'a',
                     (byte) 'v',
                     (byte) 'a' };
```

Typecast characters to bytes.

```
//write data to the stream
outStream.write(byteArray);

//output done, so close the stream
outStream.close();
```

To read the data back, we use the read method again. If we need to display the bytes in the original character values, we need to typecast byte to char. Without the typecasting, numerical values would be displayed. The following code illustrates the typecasting of byte to char for display.

```
File            inFile    = new File("sample1.data");
FileInputStream inStream  = new FileInputStream(inFile);

//set up an array to read data in
int     fileSize  = inFile.length();
byte[] byteArray = new byte[fileSize];

//read data in and display them
inStream.read(byteArray);

for (int i = 0; i < fileSize; i++) {

    System.out.println((char) byteArray[i]);
}

//input done, so close the stream
inStream.close();
```

Typecast bytes back to characters.

Typecasting char to byte or byte to char is simple because ASCII uses 8 bits. But what if we want to perform file I/O on numerical values such as integers and real numbers? It takes more than simple typecasting to output these numerical values to FileOutputStream and read them back from FileInputStream. An integer takes 4 bytes, so we need to break a single integer into 4 bytes and perform file I/O on this 4 bytes. Such a conversion would be too low-level and tedious. Java provides stream objects that allow us to read from or write numerical values to a file without doing any conversions ourselves. We will discuss two of them in Section 12.3.

Quick CHECK

1. What is the method you call at the end of all file I/O operations?
2. What is wrong with the following statements? Assume that outStream is a properly declared and created FileOutputStream object.

```
byte[ ]  byteArray = {(byte) 'H', (byte) 'i'};
...
outStream.print(byteArray);
...
outStream.close( );
```

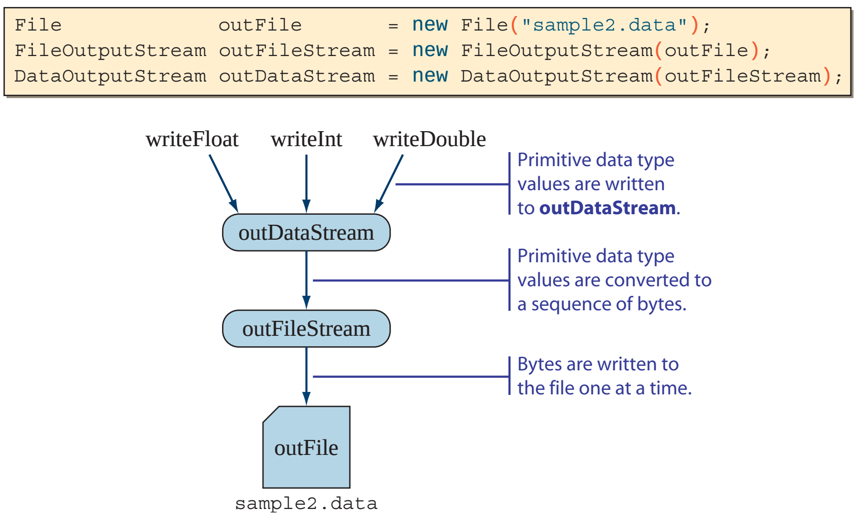
12.3 High-Level File I/O

By using DataOutputStream, we can output Java primitive data type values. A DataOutputStream object will take care of the details of converting the primitive data type values to a sequence of bytes. Let's look at the complete program first. The following program writes out values of various Java primitive data types to a file. The names of the output methods (those preceded with write) should be self-explanatory.

```
/*
    Chapter 12 Sample Program:
                A test program to save data to a file using
                DataOutputStream for high-level I/O.

    File: Ch12TestDataOutputStream.java
*/
import java.io.*;

class Ch12TestDataOutputStream {
   public static void main (String[] args) throws IOException {

      //set up the streams
      File                outFile       = new File("sample2.data");
      FileOutputStream    outFileStream = new FileOutputStream(outFile);
      DataOutputStream    outDataStream = new DataOutputStream
                                            (outFileStream);

      //write values of primitive data types to the stream
      outDataStream.writeInt(987654321);
      outDataStream.writeLong(11111111L);
      outDataStream.writeFloat(22222222F);
      outDataStream.writeDouble(3333333D);
      outDataStream.writeChar('A');
      outDataStream.writeBoolean(true);

      //output done, so close the stream
      outDataStream.close();
   }
}
```

Notice the sequence of statements for creating a DataOutputStream object:

```
File             outFile      = new File("sample2.data");
FileOutputStream outFileStream= new FileOutputStream(outFile);
DataOutputStream outDataStream
                        = new DataOutputStream(outFileStream);
```



```
File             outFile       = new File("sample2.data");
FileOutputStream outFileStream = new FileOutputStream(outFile);
DataOutputStream outDataStream = new DataOutputStream(outFileStream);
```

writeFloat
writeInt
writeDouble
Primitive data type values are written to **outDataStream**.
outDataStream
Primitive data type values are converted to a sequence of bytes.
outFileStream
Bytes are written to the file one at a time.
outFile
sample2.data

Figure 12.5 A diagram showing how the three objects **outFile, outFileStream,** and **outDataStream** are related.

DataOutput-Stream

The argument to the **DataOutputStream** constructor is a FileOutputStream object. A DataOutputStream object does not get connected to a file directly. The diagram in Figure 12.5 illustrates the relationships established among the three objects. The role of the DataOutputStream object is to provide high-level access to a file by converting a primitive data value to a sequence of bytes, which are then written to a file via a FileOutputStream object.

DataInput-Stream

To read the data back from the file, we reverse the operation. We use three objects: File, FileInputStream, and **DataInputStream**. The following program reads the data saved by the program Ch12TestDataOutputStream.

```
/*
    Chapter 12 Sample Program:
                    A test program to load data from a file using
                    DataInputStream for high-level I/O.

    File: Ch12TestDataInputStream.java
*/
import java.io.*;

class Ch12TestDataInputStream {
   public static void main (String[] args) throws IOException {

      //set up file and stream
      File            inFile       = new File("sample2.data");
      FileInputStream inFileStream = new FileInputStream(inFile);
      DataInputStream inDataStream = new DataInputStream(inFileStream);
```

```
        //read values back from the stream and display them
        System.out.println(inDataStream.readInt());
        System.out.println(inDataStream.readLong());
        System.out.println(inDataStream.readFloat());
        System.out.println(inDataStream.readDouble());
        System.out.println(inDataStream.readChar());
        System.out.println(inDataStream.readBoolean());

        //input done, so close the stream
        inDataStream.close();
    }
}
```

Figure 12.6 shows the relationship among the three objects. Notice that we must read the data back in the precise order. In other words, if we write data in the order of integer, float, and character, then we must read the data back in that order, as illustrated in Figure 12.7. If we don't read the data back in the correct order, the results will be unpredictable.

binary file

Both FileOutputStream and DataOutputStream objects produce a *binary file* in which the contents are stored in the format (called *binary format*) in which they are stored in the main memory. Instead of storing data in binary format, we can store them in ASCII format. With the ASCII format, all data are converted to string data. A file whose contents are stored in ASCII format is called a *text file*. One major

text file

```
File             inFile       = new File("sample2.data");
FileInputStream  inFileStream = new FileInputStream(inFile);
DataInputStream  inDataStream = new DataInputStream(inFileStream);
```

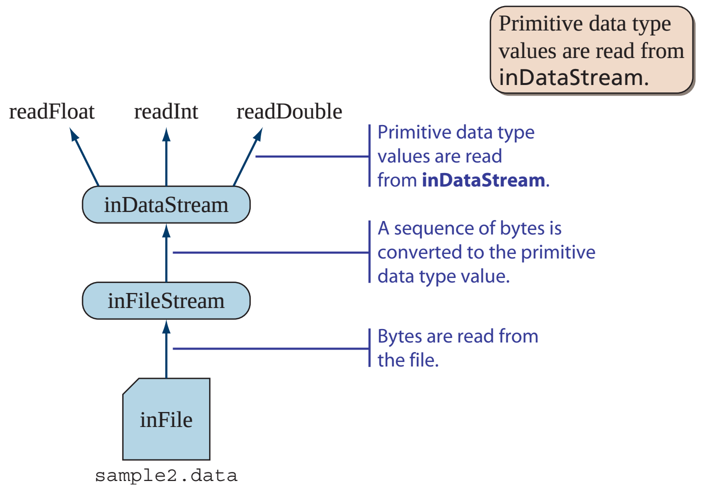

Primitive data type values are read from inDataStream.
readFloat
readInt
readDouble
Primitive data type values are read from **inDataStream**.
inDataStream
A sequence of bytes is converted to the primitive data type value.
inFileStream
Bytes are read from the file.
inFile
sample2.data

Figure 12.6 A diagram showing how the three objects **inFile, inFileStream,** and **inDataStream** are related.

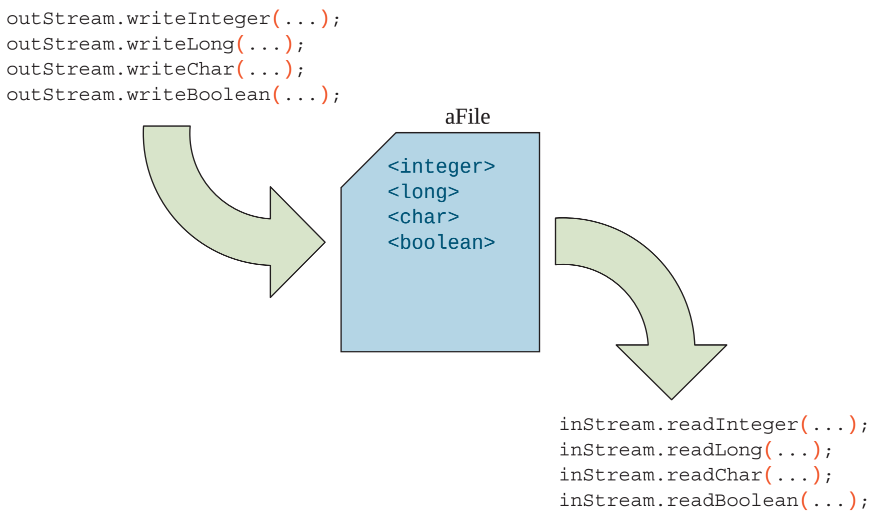

outStream.writeInteger(...);
outStream.writeLong(...);
outStream.writeChar(...);
outStream.writeBoolean(...);
aFile
<integer>
<long>
<char>
<boolean>
inStream.readInteger(...);
inStream.readLong(...);
inStream.readChar(...);
inStream.readBoolean(...);

Figure 12.7 The order of write and read operations must match to read the stored data back correctly.

benefit of a text file is that we can easily read and modify the contents of a text file by using any text editor or word processor.

PrintWriter is an object we use to generate a text file. Unlike DataOutputStream, where we have a separate write method for each individual data type, PrintWriter supports only two output methods: print and println (for print line). An argument to the methods can be any primitive data type. The methods convert the parameter to string and output this string value. The constructor of PrintWriter, similar to the one for DataOutputStream, requires an output stream as its argument. In the following program, the parameter is again an instance of FileOutputStream.

```
/*
    Chapter 12 Sample Program:
                 A test program to save data to a file using
                 PrintWriter for high-level I/O.

    File: Ch12TestPrintWriter.java
*/
import java.io.*;

class Ch12TestPrintWriter {
   public static void main (String[] args) throws IOException {

      //set up file and stream
      File             outFile        = new File("sample3.data");
      FileOutputStream outFileStream  = new FileOutputStream(outFile);
      PrintWriter      outStream      = new PrintWriter(outFileStream);
      //write values of primitive data types to the stream
      outStream.println(987654321);
      outStream.println(11111111L);
      outStream.println(22222222F);
      outStream.println(33333333D);
      outStream.println('A');
      outStream.println(true);

      //output done, so close the stream
      outStream.close();
   }
}
```

We use **print** and **println** with **PrintWriter.** The **print** and **println** methods convert primitive data types to strings before writing to a file.

To read the data from a text file, we use the FileReader and BufferedReader objects. The relationship between FileReader and BufferedReader is similar to the one between FileInputStream and DataInputStream. To read data back from a text file, first we need to associate a BufferedReader object to a file. The following sequence of statements associates a BufferedReader object to a file sample3.data:

```
File           inFile      = new File("sample3.data");
FileReader     fileReader  = new FileReader(inFile);
BufferedReader bufReader
                        = new BufferedReader(fileReader);
```

Then we read data, using the readLine method of BufferedReader,

```
String str = bufReader.readLine( );
```

and convert the String to a primitive data type as necessary.

Here's the program to read back from sample3.data, which was created by the program Ch12TestPrintWriter:

```
/*
    Chapter 12 Sample Program:
             A test program to load data from a file using the readLine
             method of BufferedReader for high-level String input.

    File: Ch12TestBufferedReader.java
*/
import java.io.*;

class Ch12TestBufferedReader {
   public static void main (String[] args) throws IOException {

      //set up file and stream
      File           inFile    = new File("sample3.data");
```

```
        FileReader     fileReader = new FileReader(inFile);
        BufferedReader bufReader  = new BufferedReader(fileReader);
        String str;

        //get integer
        str = bufReader.readLine();
        int i = Integer.parseInt(str);

        //get long
        str = bufReader.readLine();
        long l = Long.parseLong(str);

        //get float
        str = bufReader.readLine();
        float f = Float.parseFloat(str);

        //get double
        str = bufReader.readLine();
        double d = Double.parseDouble(str);

        //get char
        str = bufReader.readLine();
        char c = str.charAt(0);

        //get boolean
        str = bufReader.readLine();
        Boolean boolObj = new Boolean(str);
        boolean b = boolObj.booleanValue();

        System.out.println(i);
        System.out.println(l);
        System.out.println(f);
        System.out.println(d);
        System.out.println(c);
        System.out.println(b);

        //input done, so close the stream
        bufReader.close();
    }
}
```

Data are saved in ASCII format, so the conversion to the primitive data format is required.

Note: Here we only output, so there's no real need to perform data conversion. But in general we need to convert ASCII data to primitive data types to process them in the program.

Since Java 5.0, we can use the Scanner class introduced in Chapter 3 to input data from a text file. Instead of associating a new Scanner object to System.in, we can associate it to a File object. For example,

```
Scanner scanner = new Scanner(
                      new File("sample3.data"));
```

will associate scanner to the file sample3.data. Once this association is made, we can use scanner methods such as nextInt, next, and others to input data from the file.

The following sample code does the same as Ch12TestBufferedReader but uses the Scanner class instead of BufferedReader. Notice that the conversion is not necessary with the Scanner class by using appropriate input methods such as nextInt and nextDouble.

```
/*
    Chapter 12 Sample Program:
                 Illustrate the use of Scanner to input text file

    File: Ch12TestScanner.java
*/

import java.util.*;
import java.io.*;

class Ch12TestScanner {

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

        //open the Scanner

        Scanner scanner = new Scanner(new File("sample3.data"));

        //get integer
        int i = scanner.nextInt();

        //get integer
        long l = scanner.nextLong();

        //get float
        float f = scanner.nextFloat();

        //get double
        double d = scanner.nextDouble();

        //get char
        char c = scanner.next().charAt(0);

        //get boolean
        boolean b = scanner.nextBoolean();

        System.out.println(i);
        System.out.println(l);
        System.out.println(f);
        System.out.println(d);
```

```
            System.out.println(c);
            System.out.println(b);

            //input done, so close the scanner
            scanner.close();
        }
    }
```

The FileManager Class

In the Chapter 9 sample development and in Section 12.1, we used the helper class FileManager. A FileManager object provides file I/O operations for String data. To refresh our memory, here are the public methods of the class:

Public Methods of FileManager

```
public String openFile(String filename)
                   throws FileNotFoundException, IOException
```
Opens the text file `filename` and returns the content as a `String`.

```
public String openFile( ) throws IOException
```
Opens the text file selected by the end user using the standard file open dialog and returns the content as a `String`.

```
public String saveFile(String filename, String data)
                   throws IOException
```
Saves the string `data` to `filename`.

```
public String saveFile(String data) throws IOException
```
Saves the string `data` to a file selected by the end user using the standard file save dialog.

The class uses the PrintWriter and BufferedReader classes for text (String) output and input. Notice that all public methods throw an IOException, and only the openFile method that accepts a filename as an argument throws FileNotFound-Exception also. Here is the class listing:

```
/*
    Chapter 9 and Chapter 12 Helper Class

    File: FileManager.java
*/

import java.io.*;
import javax.swing.*;

class FileManager  {
```

```
private static final String EMPTY_STRING = "";
private static String lineTerminator
                              = System.getProperty("line.separator");

public FileManager( ) {
}

public String openFile( ) throws FileNotFoundException,
                                 IOException {
    String filename, doc = EMPTY_STRING;

    JFileChooser chooser = new JFileChooser(
                              System.getProperty("user.dir"));
    int reply = chooser.showOpenDialog(null);

    if(reply == JFileChooser.APPROVE_OPTION) {

          doc = openFile(chooser.getSelectedFile().getAbsolutePath());
    }

    return doc;
}

public String openFile(String filename)
            throws FileNotFoundException, IOException {

    String         line;
    StringBuffer   document = new StringBuffer(EMPTY_STRING);

    File           inFile     = new File(filename);
    FileReader     fileReader = new FileReader(inFile);
    BufferedReader bufReader  = new BufferedReader(fileReader);

    while (true) {
        line = bufReader.readLine();

        if (line == null) break;

        document.append(line + lineTerminator);
    }

    return document.toString();
}

public void saveFile(String data) throws IOException {
    String filename, doc = EMPTY_STRING;

    JFileChooser chooser = new JFileChooser(
                              System.getProperty("user.dir"));
    int reply = chooser.showSaveDialog(null);

    if(reply == JFileChooser.APPROVE_OPTION) {

        saveFile(chooser.getSelectedFile().getAbsolutePath(),
                 data);
    }
}
```

openFile

saveFile

```
    public void saveFile(String filename, String data)
                throws IOException {

        File              outFile       = new File(filename);
        FileOutputStream  outFileStream = new FileOutputStream(outFile);
        PrintWriter       outStream     = new PrintWriter(outFileStream);

        outStream.print(data);

        outStream.close();
    }
}
```

Quick CHECK

1. Which type of files can be opened and viewed by a text editor?
2. Which class is used to save data as a text file? Which class is used to read text files?
3. Assume bufReader, a BufferedReader object, is properly declared and created. What is wrong with the following?

```
double d = bufReader.readDouble( );
```

12.4 Object I/O

With Java, we can store objects just as easily as we can store primitive data values. There are object-oriented programming languages that won't allow programmers to store objects directly. In those programming languages, we must write code to store individual data members of an object separately. For example, if a Person object has data members name (String), age (int), and gender (char), then we have to store the three values individually, using the file I/O techniques explained earlier in the chapter. (*Note:* String is an object, but it can be treated much as any other primitive data types because of its immutability.) Now, if the data members of an object are all primitive data types (or a String), then storing the data members individually is a chore but not that difficult. However, if a data member is a reference to another object or to an array of objects, then storing data can become very tricky. Fortunately with Java, we don't have to worry about them; we can store objects directly to a file.

ObjectOutputStream

ObjectInputStream

In this section, we will describe various approaches for storing objects. To write objects to a file, we use **ObjectOutputStream**; and to read objects from a file, we use **ObjectInputStream**. Let's see how we write Person objects to a file. First we need to modify the definition of the Person class in order for ObjectOutputStream and

ObjectInputStream to perform object I/O. We modify the definition by adding the phrase implements Serializable to it.

```
import java.io.*;
class Person implements Serializable {
   //the rest is the same
}
```

Serializable is defined in **java.io.**

Whenever we want to store an object to a file, we modify its class definition by adding the phrase implements Serializable to it. Unlike other interfaces, such as ActionListener, there are no methods for us to define in the implementation class. All we have to do is to add the phrase.

Things to Remember

If we want to perform an object I/O, then the class definition must include the phrase **implements Serializable.**

To save objects to a file, we first create an ObjectOutputStream object:

```
File               outFile
                         = new File("objects.dat");
FileOutputStream   outFileStream
                         = new FileOutputStream(outFile);
ObjectOutputStream outObjectStream
                         = new ObjectOutputStream
                                          (outFileStream);
```

To save a Person object, we write

```
Person person = new Person("Mr. Espresso", 20, 'M');

outObjectStream.writeObject(person);
```

The following sample program saves 10 Person objects to a file:

```
/*
    Chapter 12 Sample Program: Illustrate the use of ObjectOutputStream

    File: Ch12TestObjectOutputStream.java
*/

import java.io.*;
```

```
class Ch12TestObjectOutputStream {
   public static void main (String[] args) throws IOException {

       //set up the streams
       File               outFile  = new File("objects.dat");
       FileOutputStream   outFileStream
                                   = new FileOutputStream(outFile);
       ObjectOutputStream outObjectStream
                                   = new ObjectOutputStream(outFileStream);

       //write serializable Person objects one at a time
       Person person;
       for (int i = 0; i < 10; i++) {
           person = new Person("Mr. Espresso" + i, 20+i, 'M');

           outObjectStream.writeObject(person);
       }

       //output done, so close the stream
       outObjectStream.close();
   }
}
```

It is possible to save different types of objects to a single file. Assuming the Account and Bank classes are defined properly, we can save both types of objects to a single file:

```
Account   account1, account2;
Bank      bank1, bank2;

account1 = new Account(); //create objects
account2 = new Account();
bank1    = new Bank();
bank2    = new Bank();

outObjectStream.writeObject(account1);
outObjectStream.writeObject(account2);
outObjectStream.writeObject(bank1   );
outObjectStream.writeObject(bank2   );
```

We can even mix objects and primitive data type values, for example,

```
outObjectStream.writeInt   (15      );
outObjectStream.writeObject(account1);
outObjectStream.writeChar  ('X'     );
```

To read objects from a file, we use FileInputStream and ObjectInputStream. We use the method readObject to read an object. Since we can store any types of objects to a single file, we need to typecast the object read from the file. Here's an example of reading a Person object we saved in the file objects.data.

```
File              inFile
                     = new File("objects.dat");

FileInputStream   inFileStream
                     = new FileInputStream(inFile);

ObjectInputStream inObjectStream
                     = new ObjectInputStream(inFileStream);

Person person = (Person) inObjectStream.readObject();
```

Need to typecast to the object type we are reading

ClassNot-Found-Exception

Because there is a possibility of wrong typecasting, the readObject method can throw a **ClassNotFoundException** in addition to an IOException. You can catch or propagate either or both exceptions. If you propagate both exceptions, then the declaration of a method that contains the call to readObject will look like this:

```
public void myMethod( )
              throws IOException, ClassNotFoundException {
   ...
}
```

The following sample program reads the Person objects from the objects.dat file:

```
/*
    Chapter 12 Sample Program: Illustrate the use of ObjectInputStream

    File: Ch12TestObjectInputStream.java
*/

import java.io.*;

class Ch12TestObjectInputStream {
    public static void main (String[] args) throws ClassNotFoundException,
                                                   IOException {

        //set up file and stream
        File              inFile   = new File("objects.dat");

        FileInputStream   inFileStream
                                   = new FileInputStream(inFile);

        ObjectInputStream inObjectStream
                                   = new ObjectInputStream(inFileStream);
        //read the Person objects from a file
        Person person;
        for (int i = 0; i < 10; i++) {
            person = (Person) inObjectStream.readObject();

            System.out.println(person.getName() + "      " +
                               person.getAge()  + "      " +
                               person.getGender());
        }

        //input done, so close the stream
        inObjectStream.close();
    }
}
```

If a file contains objects from different classes, we must read them in the correct order and apply the matching typecasting. For example, if the file contains two Account and two Bank objects, then we must read them in the correct order:

```
account1 = (Account) inObjectStream.readObject();
account2 = (Account) inObjectStream.readObject();
bank1    = (Bank)    inObjectStream.readObject();
bank2    = (Bank)    inObjectStream.readObject();
```

Now, consider the following array of Person objects where N represents some integer value:

```
Person[] people = new Person[N];
```

Assuming that all N Person objects are in the array, we can store them to file as

```
//save the size of an array first
outObjectStream.writeInt(people.length);

//save Person objects next
for (int i = 0; i < people.length; i++) {
   outObjectStream.writeObject(people[i]);
}
```

We store the size of an array at the beginning of the file, so we know exactly how many Person objects to read back:

```
int N = inObjectStream.readInt();

for (int i = 0; i < N; i++) {
   people[i] = (Person) inObjectStream.readObject();
}
```

We can actually store the whole array with a single writeObject method, instead of storing individual elements one at a time, that is, calling the writeObject method for each element. The whole people array can be stored with a single statement as

```
outObjectStream.writeObject(people);
```

and the whole array is read back with a single statement as

```
people = (Person[]) inObjectStream.readObject( );
```

Notice how the typecasting is done. We are reading an array of Person objects, so the typecasting is (Person[ ]). This approach will work with any data structure object such as a list or map.

The Dorm class

In the Chapter 8 sample development, we used the helper class Dorm to manage a list of Resident objects. A Dorm object is capable of saving a Resident list to a file and reading the list from a file. The class uses object I/O discussed in this section to perform these tasks. A list of Resident objects is maintained by using a HashMap. Instead of saving Resident objects individually, the whole map is saved with a single writeObject method and is read by a single readObject method. (The map data structure was explained in Chapter 10.) Here's the complete listing:

```
/*
    Chapter 8 Sample Development Helper Class

    File: Dorm.java
*/

import java.io.*;
import java.util.*;

public class Dorm {

    private Map<String,Resident> residentTable;

    public Dorm( ) {
        residentTable = new HashMap<String,Resident>();
    }

    public Dorm(String filename)
                    throws FileNotFoundException,
                           IOException {

        openFile(filename);
    }

    public void add(Resident resident)
                        throws IllegalArgumentException{

        if (residentTable.containsKey(resident.getName())) {
            throw new IllegalArgumentException(
                "Resident with the same name already exists");
```

Constructors

add

```
        } else {
            residentTable.put(resident.getName(), resident);
        }
    }

    public void delete(String name) {

        residentTable.remove(name);
    }

    public Resident getResident(String name) {

        return residentTable.get(name);
    }

    public String getResidentList( ) {
        StringBuffer result = new StringBuffer("");

        String tab = "\t";
        String lineSeparator = System.getProperty("line.separator");

        for (Resident res: residentTable.values()) {
             result.append(res.getName()     + tab +
                           res.getRoom()     + tab +
                           res.getPassword() + tab +
                           lineSeparator);
        }

        return result.toString();
    }

    public void openFile(String filename)
                    throws FileNotFoundException,
                           IOException {

        File inFile = new File(filename);
        FileInputStream inFileStream =
                    new FileInputStream(inFile);
        ObjectInputStream inObjectStream =
                    new ObjectInputStream(inFileStream);

        try {
            residentTable = (Map<String,Resident>)
                              inObjectStream.readObject();
        } catch (ClassNotFoundException e) {
            throw new IOException(
                         "Unrecognized data in the designated file");
        }

        inObjectStream.close();
    }

    public void saveFile(String filename)
                    throws IOException {
```

delete

getResident

getResidentList

openFile

saveFile

```
            File outFile = new File(filename);
            FileOutputStream outFileStream =
                     new FileOutputStream(outFile);
            ObjectOutputStream outObjectStream =
                     new ObjectOutputStream(outFileStream);

            outObjectStream.writeObject(residentTable);

            outObjectStream.close();
        }
    }
```

Quick CHECK

1. When do you have to include the clause implements Serializable to a class definition?
2. You cannot save the whole array at once—you must save the array elements individually, true or false?

12.5 Sample Development

Saving an AddressBook Object

As an illustration of object I/O, we will write a class that handles the storage of an **AddressBook** object. The class will provide methods to write an **AddressBook** object to a file and to read the object back from the file.

Problem Statement

Write a class that manages file I/O of an **AddressBook** *object.*

Overall Plan

Before we begin to design the class, we must modify the definition of the class that implements the **AddressBook** interface by adding the phrase **implements Serializable,** such as

```
import java.io.*;
class AddressBookVer1 implements AddressBook,
                                 Serializable {
   //same as before
}
```

In the following discussion, we will use the implementation class **AddressBookVers1.** This modification allows us to store instances of the AddressBookVer1 class. We will use the expression "an **AddressBook** object" to refer to an instance of any class that implements the **AddressBook** interface.

Since the class handles the file I/O operations, we will call the class **AddressBookStorage.** Following the STO (single-task object) principle, this class will be responsible solely for file I/O of an **AddressBook** object. The class will not perform, for instance, any operations that deal with a user interface.

What kinds of core operations should this class support? Since the class handles the file I/O, the class should support two public methods to write and read an **AddressBook** object. Let's call the methods **write** and **read.** The argument will be an **AddressBook** object we want to write or read. If **filer** is an **AddressBookStorage** object, then the calls should be something like

```
filer.write(addressBook);
```

and

```
addressBook = filer.read( );
```

For an **AddressBookStorage** to actually store an **AddressBook** object, it must know the file to which an address book is written or from which it is read. How should we let the programmer specify this file? One possibility is to let the programmer pass the filename to a constructor, such as

```
AddressBookStorage filer
          = new AddressBookStorage("book.data");
```

Another possibility is to define a method to set the file, say, **setFile,** which is called as

```
filer.setFile("book.data");
```

Instead of choosing one over the other, we will support both. If we don't provide the **setFile** method, **filer** can input and output to a single file only. By using the **setFile** method, the programmer can change the file if she or he needs to. As for the constructor, we do not want to define a constructor with no argument because we do not want the programmer to create an **AddressBookStorage** object without specifying a filename. Yes, he or she can call the **setFile** method later, but as the **AddressBookStorage** class designer, we cannot ensure the programmer will call the **setFile** method. If the programmer doesn't call the method, then the subsequent calls to the **write** or **read** method will fail. Some may consider assigning a default filename in a no-argument constructor. But what will be the default filename? No matter which filename we choose, there's a possibility that a file with this filename already exists, which will cause the file to be erased. To make our class reliable, we will not provide a no-argument constructor.

We will implement the class in the following order:

develop-
ment steps

1. Implement the constructor and the **setFile** method.
2. Implement the **write** method.
3. Implement the **read** method.
4. Finalize the class.

12.5 Sample Development—*continued*

This order of development follows a natural sequence. We begin with the constructor as usual. Since the constructor and the **setFile** method carry out similar operations, we will implement them together. We will identify necessary data members in this step. The second step is to implement the file output routine, because without being able to write an **AddressBook** object, we won't be able to test the file input routine. For the third step, we will implement the file input routine.

Step 1 Development: Constructor and setFile

step 1 design

In step 1, we will identify the data members and define a constructor to initialize them. We will also implement the **setFile** method, which should be very similar to the constructor.

We need **File, FileInputStream, FileOutputStream, ObjectInputStream,** and **ObjectOutputStream** objects to do object I/O. Should we define a data member for each type of object? This is certainly a possibility, but we should not use any unnecessary data members. We need **ObjectInputStream** and **ObjectOutputStream** objects only at the time the actual read and write operations take place. We can create these objects in the **read** and **write** methods, only when they are needed. Had we used data members for all those objects, we would need to create and assign objects every time the **setFile** method was called. But calling the **setFile** method does not necessarily mean the actual file I/O will take place. Consider the case where the user changes the filename before actually saving an address book to a file. This will result in calling the **setFile** method twice before doing the actual file I/O. To avoid this type of unnecessary repetition, we will use one data member only, a **String** variable **filename** to keep the filename. The **setFile** method simply assigns the parameter to this variable. The constructor can do the same by calling this **setFile** method.

step 1 code

At this point, we have only one data member:

```
//---------------------------
//  Data Members
//---------------------------
private  String  filename; //name of the file to store
                           //an AddressBook object
```

The **setFile** method assigns the parameter to the data member. The class is defined as follows:

```
/*
   Chapter 12 Sample Program: Address Book Storage

   File: AddressBookStorage.java
*/
class AddressBookStorage {
   private String filename;

   public AddressBookStorage (String filename) {
      setFile(filename);
   }

   public void setFile(String filename) {
      this.filename = filename;
      System.out.println("Inside setFile. Filename is " + filename);
                                                            //TEMP
   }
}
```

step 1 test

To test this class, we have included a temporary output statement inside the **setFile** method. We will write a test program to verify that we can create an **AddressBookStorage** object and use the **setFile** method correctly:

```
/*
    Chapter 12 Sample Program: Driver class to test
                               the skeleton AddressBookStorage

    File: TestAddressBookStorage.java (Step 1)
*/

class TestAddressBookStorage {

   public static void main (String[] args) {

      AddressBookStorage fileManager;

      fileManager = new AddressBookStorage("one.data");
      fileManager.setFile("two.data");
      fileManager.setFile("three.data");
   }
}
```

Step 2 Development: Implement the write Method

step 2 design

In the second development step, we will implement the **write** method. From the data member **filename,** we will create an **ObjectOutputStream** object and write the parameter **AddressBook** object to it. A sequence of method calls to create an **ObjectOutputStream** object can throw an **IOException,** so we must either propagate it or handle it.