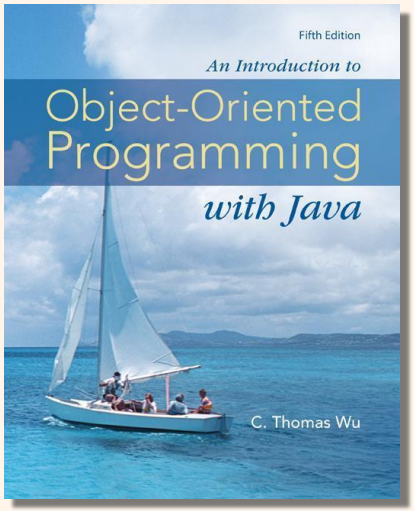


Chapter 12

File Input and Output

Chapter 12 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 DataOutputStream 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

The File Class

- To operate on a file, we must first create a File object (from java.io).

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

Opens the file sample.dat in the current directory.

```
File inFile = new File
    ("C:/SamplePrograms/test.dat");
```

Opens the file test.dat in the directory C:\SamplePrograms using the generic file separator / and providing the full pathname.

Some File Methods

```
if ( inFile.exists( ) ) {
```

To see if inFile is associated to a real file correctly.

```
if ( inFile.isFile() ) {
```

To see if inFile is associated to a file or not. If false, it is a directory.

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

String filename[] = directory.list();

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

List the name of all files in the directory C:\JavaProjects\Ch12

The **JFileChooser** Class

- A **javax.swing.JFileChooser** object allows the user to select a file.

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

To start the listing from a specific directory:

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


Getting Info from JFileChooser

```
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");
}
```

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

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


Applying a File Filter

- A *file filter* may be used to restrict the listing in JFileChooser to only those files/directories that meet the designated filtering criteria.
- To apply a file, we define a subclass of the **javax.swing.filechooser.FileFilter** class and provide the **accept** and **getDescription** methods.

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

- See the JavaFilter class that restricts the listing to directories and Java source files.

Low-Level File I/O

- To 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 a sequence of data items, usually 8-bit bytes.
- 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 are going.

Streams for Low-Level File I/O

- **FileOutputStream** and **FileInputStream** are two stream objects that facilitate file access.
- **FileOutputStream** allows us to output a sequence of bytes; values of data type **byte**.
- **FileInputStream** allows us to read in an array of bytes.

Sample: Low-Level File Output

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

FileOutputStream
      outStream = new FileOutputStream( outFile );

//data to save
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();
```


Sample: Low-Level File Input

```
//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();
```

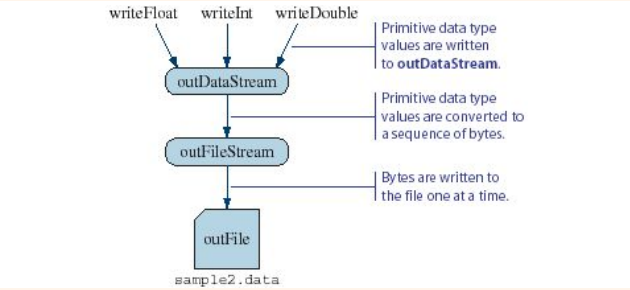

Streams for High-Level File I/O

- FileOutputStream and DataOutputStream are used to output primitive data values
- FileInputStream and DataInputStream are used to input primitive data values
- To read the data back correctly, we must know the order of the data stored and their data types

Setting up DataOutputStream

- A standard sequence to set up a DataOutputStream object:

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



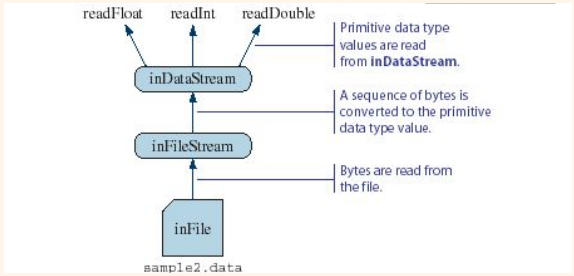
Sample Output

```
import java.io.*;
class Ch12TestDataOutputStream {
    public static void main (String[] args) throws IOException {

    . . . //set up outDataStream

    //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();
    }
}
```


Setting up DataInputStream

- A standard sequence to set up a DataInputStream object:

```
File             inFile       = new File( "sample2.data" );
FileOutputStream inFileStream = new FileOutputStream(inFile);
DataOutputStream inDataStream = new DataOutputSteam(inFileStream);
```

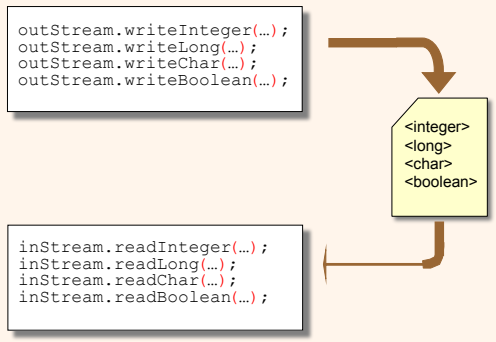


Sample Input

```
import java.io.*;
class Ch12TestDataInputStream {
    public static void main (String[] args) throws IOException {

        . . . //set up inDataStream

        //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();
    }
}
```

Reading Data Back in Right Order

- The order of write and read operations must match in order to read the stored primitive data back correctly.



Textfile Input and Output

- Instead of storing primitive data values as binary data in a file, we can convert and store them as a string data.
  - This allows us to view the file content using any text editor
- To output data as a string to file, we use a PrintWriter object
- To input data from a textfile, we use FileReader and BufferedReader classes
  - **From Java 5.0 (SDK 1.5), we can also use the Scanner class for inputting textfiles**

Sample Textfile Output

```
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("Hello, world.");
    outStream.println(true);

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


Sample Textfile Input

```
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;

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

    //similar process for other data types

    bufReader.close();
    }
}
```

Sample Textfile Input with Scanner

```
import java.io.*;

class Ch12TestScanner {

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

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

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

    //similar process for other data types

    scanner.close();
    }
}
```


Object File I/O

- It is possible to store **objects** just as easily as you store primitive data values.
- We use ObjectOutputStream and ObjectInputStream to save to and load objects from a file.
- To save **objects from a given class**, the class declaration must include the phrase **implements Serializable**. For example,

```
class Person implements Serializable {
 . . .
}
```


Saving Objects

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

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

```
account1       = new Account();
bank1          = new Bank();
outObjectStream.writeObject( account1 );
outObjectStream.writeObject( bank1    );
```

Could save objects from the different classes.

Reading Objects

```
File               inFile
                   = new File("objects.data");
FileInputStream    inFileStream
                   = new FileInputStream(inFile);
ObjectInputStream inObjectStream
                   = new ObjectInputStream(inFileStream);
```

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

Must type cast to the correct object type.

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

Must read in the correct order.

Saving and Loading Arrays

- Instead of processing array elements individually, it is possible to save and load the whole array at once.

```
Person[] people = new Person[ N ];
                  //assume N already has a value
//build the people array
...
//save the array
outObjectStream.writeObject ( people );
```

```
//read the array
Person[ ] people = (Person[]) inObjectStream.readObject( );
```


Problem Statement

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

Development Steps

- We will develop this program in four steps:

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

Step 1 Design

- We identify the data members and define a constructor to initialize them.
- Instead of storing individual Person objects, we will deal with a AddressBook object directly using Object I/O techniques.

Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter12/Step1

Source Files: AddressBookStorage.java
TestAddressBookStorage.java

Step 1 Test

- We include a temporary output statement inside the setFile method.
- We run the test main class and verify that the setFile method is called correctly.

Step 2 Design

- Design and implement the write method
- The data member filename stores the name of the object file to store the address book.
- We create an ObjectOutputStream object from the data member filename in the write method.
- The write method will propagate an IOException when one is thrown.

Step 2 Code

Directory: Chapter12/Step2

Source Files: AddressBookStorage.java
TestAddressBookWrite.java

Step 2 Test

- We run the test program several times with different sizes for the address book.
- We verify that the resulting files indeed have different sizes.
- At this point, we cannot check whether the data are saved correctly or not.
  - We can do so only after finishing the code to read the data back.

Step 3 Design

- Design and implement the read method.
- The method returns an AddressBook object read from a file (if there's no exception)
- The method will propagate an IOException when one is thrown.

Step 3 Code

Directory: Chapter12/Step3

Source Files: AddressBookStorage.java
TestAddressBookRead.java

Step 3 Test

- We will write a test program to verify that the data can be read back correctly from a file.
- To test the read operation, the file to read the data from must already exist.
- We will make this test program save the data first by using the TestAddressBookWrite class from .

Step 4: Finalize

- We perform the critical review of the final program.
- We run the final test