File Input and Output

CS 18000
Sunil Prabhakar
Department of Computer Science
Purdue University

Persistent Data

- Suppose we write a Bank application.
- How do we remember the account balances?
- What if the bank application stops running?
  - Runtime exception
  - Power failure?
  - …
- How do we ensure that when we restart the application, the balances are set correctly?

File I/O

- Remember that a computer system has two types of memory
  - Fast, volatile main memory
  - Slower, non-volatile secondary memory (disk)
- When a program runs, its variables (primitive and objects) are stored in main memory.
- When the program terminates, all these variables are lost!
- If we would like to preserve the values of variables across multiple executions we must save them on disk and read them back in -- file input and output File I/O.

File I/O

- Files can also be used to prevent loss of data due to system failures.
- Files can serve as a means for sharing data between different programs.
- Different operating systems manage files differently.
- Since Java is a HLL, we are mostly shielded from these differences.
- In Java we use objects to perform file I/O.

The File Class

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

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

Opens the file data.txt in the current directory.

```
File inFile = new File
      ("/Users/sunil/test.txt");
```

Opens the file test.txt in the directory /Users/sunil/ using the generic file separator / and providing the full pathname.

File names

- The rules for file names are determined by the operating system on which the Java program is run.
- Thus the name may or may not be case sensitive, or require filename extensions…
- Java simply passes the string to the operating system.

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("/Users/sunil");

String filename[] = directory.list();

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

List the names of all files in the directory /Users/sunil

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("/Users/
sunil/Dropbox");

chooser.showOpenDialog(null);
```

Getting Info from JFileChooser

```
int status = chooser.showOpenDialog(null);
if (status == JFileChooser.APPROVE_OPTION) {
   JOptionPane.showMessageDialog(null, "Open is clicked");

} else { //== JFileChooser.CANCEL_OPTION
   JOptionPane.showMessageDialog(null, "Cancel is
clicked");
}
```

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

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

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.

FileOutputStream

- When creating a FileOutputStream object, it is possible that a FileNotFoundException may be thrown
  - Checked exception -- so it must be handled appropriately.
- When writing to, or closing a file, an IOException may be thrown
  - Checked exception -- must be handled appropriately.

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();
```

FileInputStream

- To input bytes from a file, we must create a FileInputStream object and attach it to a file.
  - Could throw a FileNotFoundException
- The number of bytes in the file can be determined using the length() method on the File object. Returns a long value.

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();
```

Opening and closing files

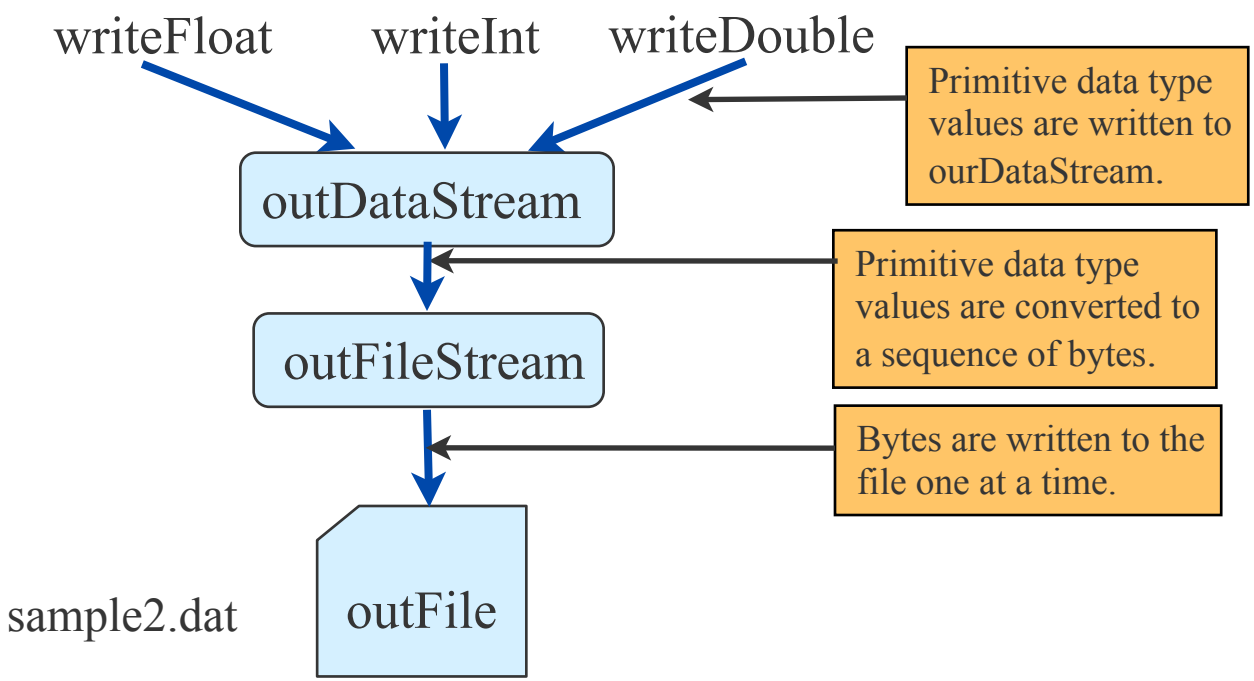
- When we create the stream object we open the file and connect it to the stream for input or output.
- Once we are done, we must close the stream. Otherwise, we may see corrupt data in the file.
- If a program does not close a file and terminates normally, the system closes the files for it.
- We must close a file after writing before we can read from it in the same program.

Streams for High-Level File I/O

- FileInputStream and FileOutputStream are used to input and output raw bytes, e.g., images.
- DataInputStream and DataOutputStream are used to input and output primitive data types other than bytes
- To read the data back correctly, we must know the order of the data stored and their data types

Setting up DataOutputStream

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

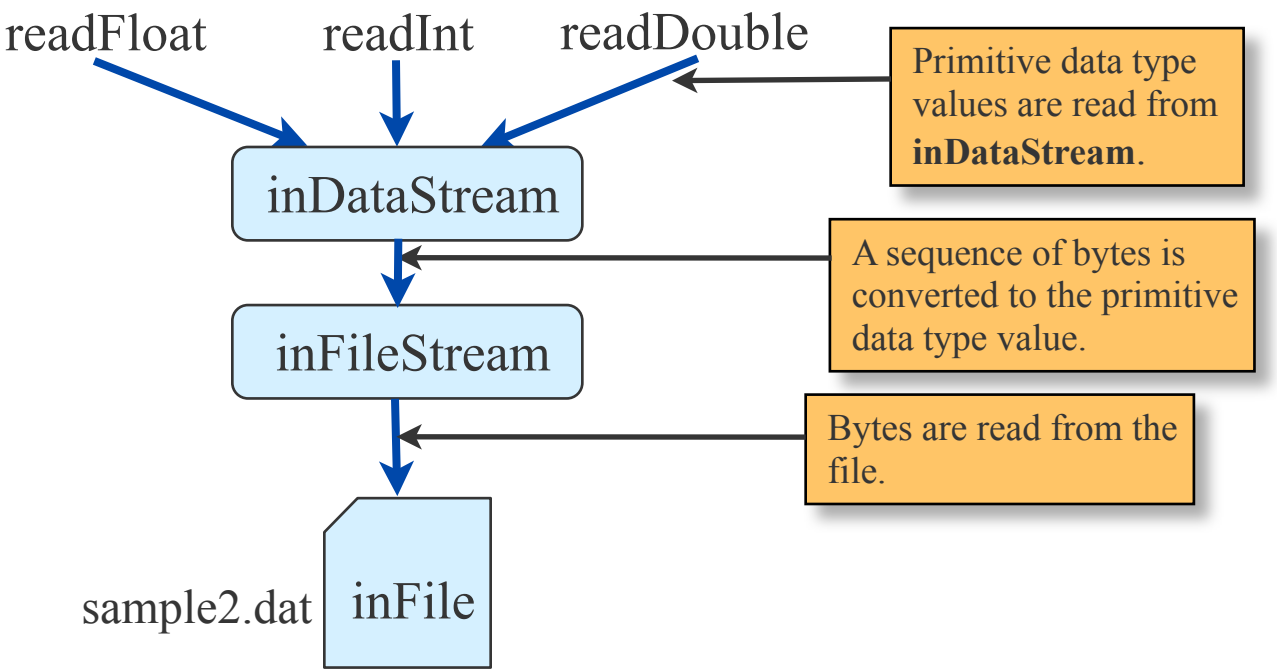


Sample Output

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

        . . . //set up outDataStream

        //write values of primitive data types to the stream
        outDataStream.writeInt(2343);
        outDataStream.writeLong(34545L);
        outDataStream.writeFloat(23.34F);
        outDataStream.writeDouble(456.4234D);
        outDataStream.writeChar('J');
        outDataStream.writeBoolean(true);

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

Setting up DataInputStream

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

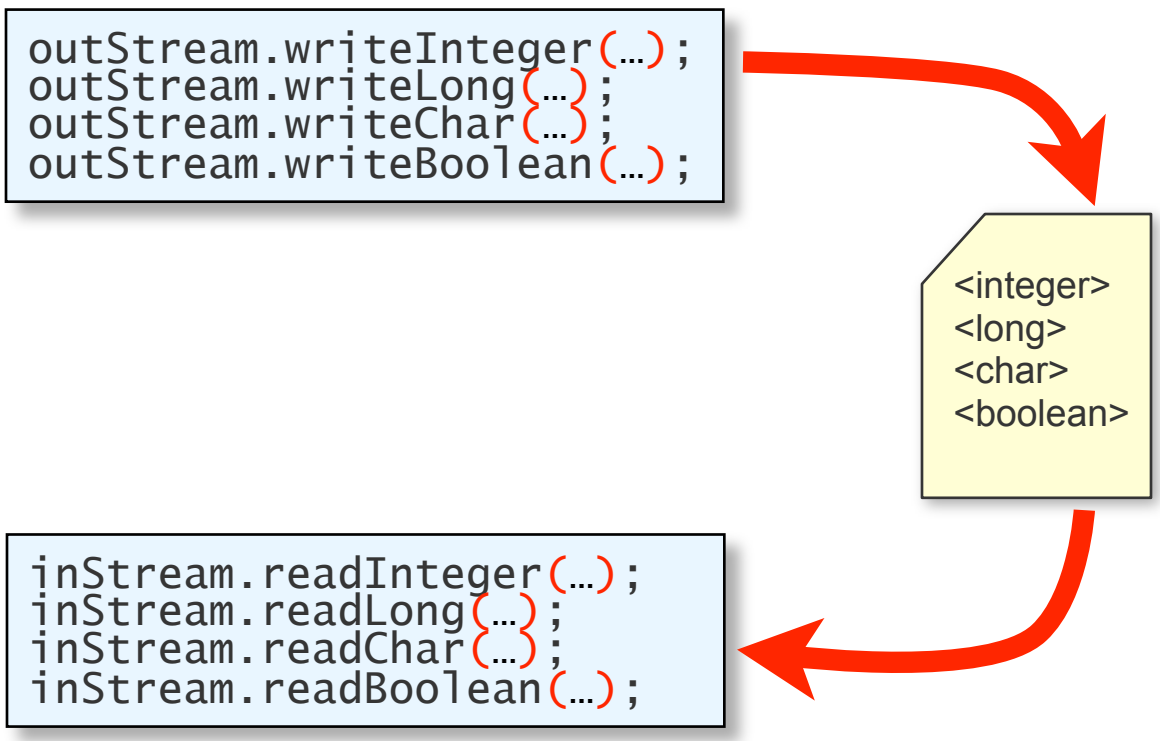


Sample Input

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

        . . . //set up inDataStream

        //read values back from the stream and display them
        int i = inDataStream.readInt();
        long l = inDataStream.readLong();
        float f = inDataStream.readFloat();
        double d = inDataStream.readDouble();
        char c = inDataStream.readChar();
        boolean bool = 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.



Types of files

- There are two main types of files
  - Text -- store characters (ASCII or UNICODE)
    - *.java
    - Usually platform independent, but less efficient.
  - Binary -- may store any type of data.
    - *.class, *.exe
    - Often depend on machine (OS, program)
    - Java binary files are platform independent.
- Text files are editable using editors and usually comprehensible to humans.
- So far, we have been dealing with binary files in this discussion.

Textfile Input and Output

- Instead of storing primitive data values as binary data in a file, we can convert and store them as 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 the **FileReader** and **BufferedReader** classes
  - From Java 5.0 (SDK 1.5), we can also use the Scanner class for reading input from text files

Sample Textfile Output

```
import java.io.*;
class TestPrintWriter {
    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(2343);
        outStream.println("Hello, world.");
        outStream.println(true);

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

PrintWriter

- If a file with the given name exists it is opened for output and its current contents are lost.
- If we want to retain the old data, and append to the end of the file:

```
FileOutputStream outFileStream
            = new FileOutputStream(outFile, true);
```

then create the PrintWriter object with this stream object.

```
PrintWriter outStream = new PrintWriter(outFileStream);
```

Sample Textfile Input

```
import java.io.*;
class TestBufferedReader {

    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();
    }
}
```

Notes

- FileReader is meant to be used for reading in character streams
- FileInputStream is for arbitrary streams
- BufferedReader is much more efficient than reading directly from an input stream
  - automatically buffers input from the stream
  - can be much more efficient due to disk I/O costs

Sample Textfile Input with Scanner

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

class TestScanner {

    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 of 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
(outFileStream);        = new ObjectOutputStream
```

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

outObjectStream.writeObject( person );
```

```
account1    = new Account();
bank1       = new Bank();

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

Can 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 entire 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
( );
```

Important Note

- All data is saved in memory and disk as bits.
- When we use Byte level I/O, the two representations are the same.
- When we do text I/O, the data has to be converted between the memory representation and the disk representation (i.e., the equivalent character strings).
  - this is similar to what we do with System.in and System.out which operate on characters.

Important Note (Cont.)

- When we write entire objects to disk
  - a representation of the object suitable for disk is created and is written out as bytes.
- When we read entire objects from disk
  - the disk bytes are used to create an object with similar values (but at a new location) in memory.

Exceptions

- File I/O methods throw various types of exceptions including
  - IOException
  - FileNotFoundException
- Please see Java API to become familiar with these.
- Many of these need to be handled in order to make programs more robust.
- Labs and projects will cover some

Knowing when to stop reading

- It is possible to try to read beyond the end of the file.
- Different reader classes signal the end of file in different ways.
- Binary file readers throw the EOFException.
- Text file readers return null or -1.
- You should be aware of how the common classes indicate the end of file condition.

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 filter, 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.