Introduction to Algorithms and Data Structures

Lecture 8 – File I/O

Streams

- A stream is an object that allows for the flow of data between a program and some I/O device (or a file).
  - If the flow is into a program it's an input stream.
  - If the flow is out of a program, it's an output stream.
- **`System.in`** and **`System.out`** are examples of input and output streams respectively.

Text and Binary Files

- Text files are sequences of characters that can be viewed in a text editor or read by a program.
- Files whose contents are binary images of information stored in main memory are called binary files. These include the computer's representation of:
  - numbers
  - pictures
  - sound bites
  - machine instructions
- The advantage of text files is that they are human-readable and can be moved easily from computer system to computer system.

Writing To A Text File

- The preferred class for writing to text files is **`PrintWriter`**, whose methods include **`println`** and **`print`**.
- A **`PrintWriter`** object can be created by writing:

```
outputStream = new PrintWrite
     (new FileOutputStream("stuff.txt"));
```

- A **`FileOutputStream`** is needed when calling the constructor. It, in turn, needs the name of the file to which output is going to be written.
- Since opening a file might lead to a **`FileNotFoundException`**, it should be done in a **`try`** block.

Every File Has 2 Names

- Every file has 2 names: the name by which it is known the operating system of its computer and the name of the stream connect to the file (by which the program knows it).

`PrintWriter` Methods

- There are three constructors:
  - **`PrintWriter`** *`objectStream`* **`= new PrintWriter(OutputStream`** *`streamObject`***`);`**

    is the standard constructor
  - **`PrintWriter`** *`objectStream`* **`= new PrintWriter(new FileOutputStream (`***`FileName`***`));`**

    allows us to construct a file to which it will write.
  - **`PrintWriter(new FileOutputStream(`***`FileName`***`, true));`**

    which we will use for appending to a file.

`PrintWriter` Methods (continued)

- **`public void println(argument)`**
  - Prints the argument which can be a string, integer, floating-point number, or any object for whch a toString() method exists.
- **`public void print(argument)`**
  - Works like println with the newline at the end.
- **`public PrintWriter void printf()`**
  - Formatted output
- **`close()`** – closes the stream
- **`flush()`** – flushes the stream, forcing any other data to be written that has been buffered but not yet written.

`TextFileOutputDemo.java`

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

public class TextFileOutputDemo  {
  public static void main(String[] args) {
    PrintWriter outputStream = null;
    try {
      outputStream = new PrintWriter
               (new FileOutputStream("stuff.txt"));
    }
    catch (FileNotFoundException e) {
      System.out.println
          ("\"Error opening the file stuff.txt\".");
      System.exit(0);
    }
```

```
    System.out.println("Writing to file.");
    outputStream.println("The quick brown fox ");
    outputStream.println
                    ("jumped over the lazy dogs.");

    outputStream.close();
    System.out.println("End of program.");
  }
}
```

Reading a Text File using Scanner

- The same Scanner class that we have been using to read input from the keyboard can be used to read from a text file.
- Although the Scanner has to be declared outside the try block, the constructor should be called inside it.
- Example

```
Scanner inputStream = null;
try {
   inputStream = new Scanner
     (new FileInputStream("stuff.txt"));
}
catch(FileNotFoundException e)  { …
```

Testing for the End of Line

- If you write
```
Scanner keyb = new Scanner(System.in);
int x = keyb.nextInt()
```
- and there is no more text, (or a non-integer), **`nextInt()`** will throw a **`NoSuchElementException`**
- You can work around this by using **`hasNextInt()`**(which return false if there isn't a proper integer there or **`hasNextLine()`**, which will return false if there is no next line.

Some **`Scanner`** Constructors

There are two constructors in which we can most interested:

- **`public Scanner(InputStream streamObject)`**
  - This can be used to create stream for a file by writing
```
public Scanner
      (new FileInputStream (fileName));
```
- **`public Scanner(File fileObject)`**
  - This can be used to create a **`File`** object for a file by writing
  - **`public Scanner(new File(fileName)`**

Some **`Scanner`** input Methods

- Scanner has several methods that can be used to obtain a single data item:
  - **`nextInt()`**
  - **`nextLong()`**
  - **`nextByte()`**
  - **`nextShort()`**
  - **`nextDouble()`**
  - **`nextFloat()`**
- All these methods will return a single value of the indicated type.
- If there are no more tokens, they will throw a **`NoSuchElementException`**. If the data is not a valid representation of the type, they will throw an **`InputMismatchException`**, and if the Scanner is closed, they will throw an **`IllegalStateException`**

Some **`Scanner`** Input Methods

- Scanner has several methods that can be used to detect that there is valid data waiting as input:
  - **`hasNextInt()`**
  - **`hasNextLong()`**
  - **`hasNextByte()`**
  - **`hasNextShort()`**
  - **`hasNextDouble()`**
  - **`hasNextFloat()`**
- All these methods will return a true if there is a well-formed string representation of the type of the data item ; false otherwise.
- If the Scanner is closed, they will throw an **`IllegalStateException`**

Some Other **`Scanner`** Methods

- Scanner has two methods that can be used to detect that there is valid data waiting as input:
  - **`next()`** - **`hasNext()`**
  - **`nextLine()`** - **`nextLine()`**
- **`next()`** and **`nextLine()`** return the next token and line respectively.
- **`hasNext()`** and **`hasNextLine()`** return true if there is another token or line respectively; otherwise false.
- These methods throw the appropriate exceptions.

`TextFileScannerDemo.java`

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class TextFileScannerDemo  {
  public static void main(String[] args) {
    System.out.println
         ("I will read three nunbers and a line");
    System.out.println
         ("of text from the file morestuff.txt");

    Scanner inputStream = null;

    try {
      inputStream = new Scanner
             (new FileInputStream("morestuff.txt"));
    }
```

```
    catch(FileNotFoundException e) {
      System.out.println
              ("File morestuff.txt was not found.");
      System.out.println("or could not be opened.");
      System.exit(0);
    }
    int n1 = inputStream.nextInt();
    int n2 = inputStream.nextInt();
    int n3 = inputStream.nextInt();

    inputStream.nextLine(); // Go to the next line

    String line = inputStream.nextLine();

    System.out.println
      ("The three numbers read from the file are:");
    System.out.println(n1 + ", "
                   + n2 + ", and " + n3);
```

```
    System.out.println
            ("The line read from the file is:");
    System.out.println(line);
    inputStream.close();
  }
}
```

HasNextLineDemo.java

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.FileOutputStream;

public class HasNextLineDemo  {
  public static void main(String[] args) {
    Scanner inputStream = null;
    PrintWriter outputStream = null;

    try {
      inputStream = new Scanner(new
                  FileInputStream("original.txt"));
      outputStream = new PrintWriter(new
                  FileOutputStream("numbered.txt"));
    }
```

```
    catch (FileNotFoundException e) {
      System.out.println("Problem opening files.");
      System.exit(0);
    }
    String line = null;
    int count = 0;

    while (inputStream.hasNextLine()) {
      line= inputStream.nextLine();
      count++;
      outputStream.println(count + " " + line);
    }
    inputStream.close();
    outputStream.close();
  }
}
```

HasNextIntDemo.java

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

public class HasNextIntDemo  {
  public static void main(String[] args) {
    Scanner inputStream = null;

    try {
      inputStream = new Scanner(new
                    FileInputStream("data.txt"));
    }
```

```
    catch (FileNotFoundException e) {
      System.out.println
               ("File data.txt was not found");
      System.out.println("or could not be opened.");
      System.exit(0);
    }

    int next, sum = 0;
    while (inputStream.hasNextInt()) {
      next= inputStream.nextInt();
      sum += next;
    }

    inputStream.close();
    System.out.println("The sum of the numbers is "
                    + sum);
  }
}
```

BufferedReader Class

- BufferedReader is the class that was used for reading text data before Scanner class was introduced in verion 5.0 of Java.
- A BufferedReader was used to open a file as shown below:

```
BufferedReader inputStream
= new BufferedReader
   (new FileReader("stuff.txt"));
```

Reading Text Using **BufferedReader**

- BufferedReader has methods that are used for input:
  - **readLine()** – reads a line of input that is returned as text. If there is no more text, it returns a null reference.
  - **read()** – reads a single character from the input stream that is returned as an integer.. If there is no more text, it returns -1
  - **skip(int n)** – skips n characters.
  - **close()** – closes the stream's connection to the file.
- These methods all throw **IOException**.

TextFileInputDemo.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TextFileInputDemo {
  public static void main(String[] args)  {
    try {
      BufferedReader inputStream
          = new BufferedReader
                  (new FileReader("morestuff2.txt"));
      String line = inputStream.readLine();
      System.out.println
          ("The first line read from the file is:");
      System.out.println(line);
```

```
      line= inputStream.readLine();
      System.out.println
         ("The second line read from the file is:");
      System.out.println(line);
      inputStream.close();
    }
    catch(FileNotFoundException e) {
      System.out.println
            ("File morestuff2.txt was not found");
      System.out.println("or could not be opened.");
    }
    catch (IOException e) {
      System.out.println
             ("Error reading from morestuff2.txt");
    }
  }
}
```

TextEOFDemo.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class TextEOFDemo {
  public static void main(String[] args)  {
    try {
      BufferedReader inputStream = new
           BufferedReader
           (new FileReader("original.txt"));
      PrintWriter outputStream
             = new PrintWriter(new
                FileOutputStream("numbered.txt"));
```

```
      int count= 0;
      String line= inputStream.readLine();
      while (line != null) {
        count++;
        outputStream.println(count + " " + line);
        line = inputStream.readLine();
      }
      inputStream.close();
      outputStream.close();
    }
    catch (FileNotFoundException e) {
      System.out.println("Problem opening file.");
    }
    catch (IOException e) {
      System.out.println
          ("Error reading from original.txt");
    }
  }
}
```

Standard Streams

- All Java programs are assumed to have at least three open streams: **`System.in`**, **`System.out`** and **`System.err`** (the last one allows for error message and output to be redirects separately).

`RedirectionDemo.java`

```
import java.io.PrintStream;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

public class RedirectionDemo {
  public static void main(String[] args)  {
    PrintStream errStream = null;
    try {
      errStream = new PrintStream(new
             FileOutputStream("errormessage.txt"));
    }
    catch (FileNotFoundException e) {
      System.out.println
       ("Error opening file with FileOutputSteam.");
      System.exit(0);
    }
```

```
    System.setErr(errStream);

    System.err.println("Hello fgom System.err.");
    System.err.println("Hello from System.out.");
    System.err.println("Hello again from
  System.err.");

    errStream.close();
  }
}
```

File Class

- The File class is a bit like a wrapper class for file names.
- It also provides us with some methods that can be used to check some basic properties of files, such as whether it exists, does the program have permission to read or write it.
- Example:

```
File fileObject = new File(“data.txt”);
if (!fileObject.canRead())
   System.out.println
     ("file data.txt is not readable.");
```

File Class Methods

- **`exists()`** – returns true if it exists; false otherwise
- **`canRead()`** – returns true if the program can read data from the file; false otherwise.
- **`canWrite()`** - returns true if the program can write data from the file; false otherwise.
- **`isFile()`** – returns true if the file exists and is a regular file; false otherwise.
- **`isDirectory()`** - returns true if the file exists and is a directory; false otherwise.

File Class Methods (continued)

- **`length()`** – returns the number of bytes in the file as a long.If it doesn't exist or if it's a directory, the value returned isn't specified.
- **`delete()`** – tries to delete the file; returns true if it can, false if it can't.
- **`setReadOnly()`** – sets the file as read only; returns true if it is can; false if it can't.
- **`createNewFile()`** – creates an empty of that name; returns true if it can; false if it can't
- **`isFile()`** – returns true if the file exists and is a regular file; false otherwise.
- **`isDirectory()`** - returns true if the file exists and is a directory; false otherwise.

File Class Methods (continued)

- **`getName()`** – return the simple name (the last part of the path).
- **`getPath()`** – returns the abstract (absolute or relative) path name or an empty string if the path name is an empty string..
- **`renameTo(newName)`** – renames the file to a name represented by the file object **`newName`**; returns true if it is can; false if it can't.
- **`createNewFile()`** – creates an empty of that name; returns true if it can; false if it can't
- **`mkdir()`** – makes the directory named by the abstract path name. Will not create parent directories; returns true if the file exists and is a regular file; false otherwise.
- **`mkdirs()`** - makes the directory named by the abstract path name. Will necessary but nonexistent create parent directories; returns true if the file exists and is a regular file; false otherwise.

`FileClassDemo.java`

```
import java.util.Scanner;
import java.io.File;
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

public class FileClassDemo  {
  public static void main(String[] args)  {
    Scanner keyb= new Scanner(System.in);
    String line = null;
    String fileName = null;

    System.out.println
         ("I will store a line of text for you.");
    System.out.println("Enter the line of text:");
    line = keyb.nextLine();
```

```
    System.out.println
           ("Enter a file name to hold the line:");
    fileName = keyb.nextLine();
    File fileObject = new File(fileName);

    while (fileObject.exists())  {
      System.out.println
            ("There already is a file named "
                       + fileName);
      System.out.println
                  ("Enter a different file name:");
      fileName = keyb.nextLine();
      fileObject= new File(fileName);
    }

    PrintWriter outputStream = null;
```

```
    try {
      outputStream
           = new PrintWriter(new
                       FileOutputStream(fileName));
    }
    catch (FileNotFoundException e) {
      System.out.println("Error opening the file.");
      System.exit(0);
    }

    System.out.println("writing \"" + line + "\"");
    System.out.println("to the file " + fileName);
    outputStream.println(line);

    outputStream.close();
    System.out.println("Writing completed.");
  }
}
```

Binary Files

- Text files save data in the same form as they appear on the screen or printed page. This requires the computer to do some conversion when reading and writing them.
- Binary files save data in their internal form.
  - Strings have their unused bytes saved
  - Numbers are stored in binary form.

Classes Used in Writing a Binary File

- There are two classes of objects used in writing binary files:
  - **FileOutputStream** – stream objects that write data to a file.
  - **ObjectOutputStream** - objects that take data of a certain type and converting it into a stream of bytes.

The **`write`***Primitive*`()` Methods

- The ObjectOutputStream class has several methods of **`write`***Primitive*`(di)` that allows the programmer to write the data item **`di`** of type *Primitive* into the file.
- They all throw **`IOException`**s
- The methods include:
  - **`oos.writeInt(i)`**- write **`integer`** value **`i`** into the binary file **`dos`**.
  - **`oos.writeDouble(x)`** - write **`double`** value **`x`** into the binary file **`dos`**.
  - **`oos.writeChar(c)`** - write char value **`c`** into the binary file **`dos`**.

`BinaryOutputDemo.java`

```
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class BinaryOutputDemo  {
  public static void main(String[] args)  {
    try {
      ObjectOutputStream outputStream
         = new ObjectOutputStream(new
                  FileOutputStream("numbers.dat"));

      int i;
      for  (i = 0;  i < 5; i++)
        outputStream.writeInt(i);
```

```
      System.out.println("\"Numbers written to the"
                + " file numbers.dat\"");
      outputStream.close();
    }
    catch(IOException e)  {
      System.out.println
           ("Problem writing with file output.");
    }
  }
}
```

Reading a Binary File

- We do this essentially the same way that we write a binary file.
- It is very important that we know ***exactly*** what the files structure is so we can read it accurately.

The **read***Primitive***()** Methods

- The ObjectInputStream class has several methods of **read***Primitive***(di)** that allows the programmer to read the data item **di** of type *Primitive* from the file.
- The methods include:
  - **ois.readInt(i)**- read **integer** value **i** from the binary file **dos**.
  - **ois.readDouble(x)** - read **double** value **x** from the binary file **dos**.
  - **ois.readChar(c)** – read char value **c** from the binary file **dos**.

BinaryInputDemo.java

```
import java.io.ObjectInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.FileNotFoundException;

public class BinaryInputDemo  {
  public static void main(String[] args)  {
    try {
      ObjectInputStream inputStream = new
  ObjectInputStream(new
  FileInputStream("numbers.dat"));

      System.out.println
          ("Reading the file \"numbers.dat\"");
      int n1 = inputStream.readInt();
      int n2 = inputStream.readInt();
```

```
      System.out.println("Numbers read from file:");
      System.out.println(n1);
      System.out.println(n2);
      inputStream.close();
    }
    catch(FileNotFoundException e) {
      System.out.println
              ("Cannot find file \"numbers.dat\"");
    }
    catch(IOException e)  {
      System.out.println
       ("Problem with input from \"numbers.dat\".");
    }
  }
}
```

Files And Records

- Many programming languages and computer systems treat files as a collection of records, where you read and write one record at a time.
- Java does not have anything exactly analogous to records; we use objects instead.
- We will create an implementation of the class **`Serializable`** so the properties of our objects can be converted into a byte stream and saved in a binary file.

SomeClass.java

```
import java.io.Serializable;

public class SomeClass implements Serializable  {
  private int number;
  private char letter;

  public SomeClass()  {
    number = 0;
    letter = 'A';
  }
```

```
  public SomeClass(int theNumber, char theLetter) {
    number = theNumber;
    letter = theLetter;
  }

  public String toString()  {
    return "Number = " + number +
      " Letter " + letter;
  }
}
```

ObjectIODemo.java

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

// Demonstrates binaryfile I/O of serializable
// class objects
public class ObjectIODemo {
  public static void main(String[] args)  {
    try {
      ObjectOutputStream outputStream
          = new ObjectOutputStream(new
                 FileOutputStream("datafile"));
```

```
      SomeClass oneObject = new SomeClass(1, 'A');
      SomeClass anotherObject
                  = new SomeClass(42, '2');

      outputStream.writeObject(oneObject);
      outputStream.writeObject(anotherObject);

      outputStream.close();
      System.out.println("Data sent fo file.");
    }
    catch(IOException e) {
      System.out.println
              ("Problem with file output.");
    }
    System.out.println("Now let\'s reopen the file "
           + " and display the data.");
```

```
    try {
      ObjectInputStream inputStream
          = new ObjectInputStream(new
                 FileInputStream("datafile"));

      SomeClass readOne
             = (SomeClass) inputStream.readObject();
      SomeClass readTwo
             = (SomeClass) inputStream.readObject();

      System.out.println
         ("The following were read from the file:");
      System.out.println(readOne);
      System.out.println(readTwo);
    }
    catch (FileNotFoundException e) {
     System.out.println("Cannot find datafile.");
    }
```

```
    catch(ClassNotFoundException e) {
      System.out.println
             ("Problems with file input.");
    }
    catch(IOException e) {
      System.out.println
            ("Problems with file input.");
    }
    System.out.println("End of program.");
  }
}
```

ArrayIODemo.java

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

public class ArrayIODemo  {
  public static void main(String[] args) {
    SomeClass[] a = new SomeClass[2];
    a[0] = new SomeClass(1, 'A');
    a[1] = new SomeClass(2, 'B');
```

```
    try {
      ObjectOutputStream outputStream
        = new ObjectOutputStream(
                 new FileOutputStream("arrayfile"));
      outputStream.writeObject(a);
      outputStream.close();
    }
    catch(IOException e) {
      System.out.println("Error writing to file.");
      System.exit(0);
    }

    System.out.println
           ("Array written to file arrayfile.");
    System.out.println("Now let's re-open the "
              + "file and display the arrow.");
```

```
    SomeClass[] b = null;

    try  {
      ObjectInputStream inputStream
          = new ObjectInputStream(
                 new FileInputStream("arrayfile"));
      b = (SomeClass[])inputStream.readObject();
      inputStream.close();
    }
    catch (FileNotFoundException e) {
      System.out.println
                 ("Cannot find file arrayfile.");
      System.exit(0);
    }
    catch (ClassNotFoundException e) {
      System.out.println
               ("Problems with file input.");
      System.exit(0);
    }
```

```
    catch(IOException e) {
      System.out.println
                  ("Problems with file input.");
    }
    System.out.println("The following array
           + "elements were read from the file:");
    int i;
    for (i = 0;  i < b.length; i++)
      System.out.println(b[i]);
    System.out.println("End of program.");
  }
}
```

Random Access Files

- We usually read files from the beginning to the end. Files that are always accessed in this fashion as called sequential access files.
- Sometimes we need to read files from any particular point to which we may need access. These are called random access files.
- These have two methods that other files do not have: **`getFilePointer()`** (which takes us to the current location from which we are reading) and **`seek()`** (which moves us to another location in the file).

`RandomAccessDemo.java`

```
import java.io.RandomAccessFile;
import java.io.IOException;
import java.io.FileNotFoundException;

public class RandomAccessDemo  {
  public static void main(String[] args)  {
    try  {
      RandomAccessFile ioStream
          = new RandomAccessFile("bytedata", "rw");

      System.out.println
         ("Writing 3 bytes to the file bytedata.");
      ioStream.writeByte(1);
      ioStream.writeByte(2);
      ioStream.writeByte(3);
      System.out.println
           ("the length of the file is now = "
                          + ioStream.length());
```

```
System.out.println
    ("The file pointer location is "
                + ioStream.getFilePointer());

System.out.println
   ("Moving the file pointer to location 1.");
ioStream.seek(1);
byte oneByte = ioStream.readByte();
System.out.println
   ("The value at location 1is " + oneByte);
oneByte = ioStream.readByte();
System.out.println
   ("The value at the next location is "
                    + oneByte);

System.out.println
("Now we can move the file pointer back to ");
System.out.println
         ("location 1, and change the byte.");
```

```
ioStream.seek(1);
ioStream.writeByte(9);
ioStream.seek(1);
oneByte = ioStream.readByte();
System.out.println
      ("The value at location 1 is now "
                                + oneByte);

System.out.println
     ("Now we can go to the end of the file");
System.out.println
                   ("and write a double.");
ioStream.seek(ioStream.length());
ioStream.writeDouble(41.99);
System.out.println
    ("the length of the fiule is now = "
                    + ioStream.length());
System.out.println
           ("Returning to location 3, ");
```

```
      System.out.println
                    ("where we wrote the double.");
      ioStream.seek(3);
      double oneDouble = ioStream.readDouble();
      System.out.println
           ("The double version of location is "
                          + oneDouble);

      ioStream.close();
    }
    catch(FileNotFoundException e) {
      System.out.println("Problem opening file.");
    }
    catch (IOException e) {
        System.out.println
                          ("Problems with file I/O");
    }
    System.out.println("End of program.");
  }
}
```