Murdoch
UNIVERSITY

Topic 9 Streams and File I/O

ICT167 Principles of Computer Science

Murdoch
UNIVERSITY

Objectives

- Explain the concept of a **stream**
- Understand the difference between **text files** and **binary files**
- Be able to program **input/output of text files** using the Java I/O library class **PrintWriter** and **java.util.Scanner** class
- Be able to program **input/output of binary files** using Java I/O library classes **ObjectInputStream** and **ObjectOutputStream**

Objectives

- Be able to handle **I/O exceptions**, especially **FileNotFoundException**
- Be able to test for the ends of binary files using **EOFException**
- Be able to use the **File** class for directory management

- **Reading**

Savitch: Chapter 10.1 – 10.4

I/O and Streams

- Input = data coming in to the program
  - For example from keyboard, files on disk, other programs or network connections
- Output = data flowing out of the program
  - For example to the screen, files on disk, other programs or network connections
- I/O = managing the input and output of your program

I/O and Streams

- Advantages of file I/O:
  - Permanent copy
  - Output from one program can be input to another
  - Input can be automated (rather than entered manually)
- In Java, keyboard/screen I/O as well as file I/O is handled by **streams**

I/O and Streams

- A **Stream** = flow of input or output data (i.e. a series of values such as characters, numbers, or bytes consisting of binary digits)
- There are many similarities between I/O to:
  - Files on disk
  - Network connections
  - Pipes to other programs
  - To the user via the screen, keyboard and mouse

I/O and Streams

- Therefore in Java:
  - ***A Stream*** is an object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.) and delivers it to your program
  - It acts as a buffer between the data source and destination
- Streams are implemented in Java as objects of special stream classes

I/O and Streams

- ***Input stream*** is a stream that provides input to a program
- ***Output stream*** is a stream that accepts output from a program
  - `System.out` is an output stream
  - `Scanner` class object is an input stream
- A stream connects a program to an I/O object
  - `System.out` connects a program to the screen
  - `Scanner` object connects a program to the keyboard or a file

Text vs Binary Files

- We use files on disk to store data which is:
  - Needed before or after program runs
  - Needs to be transported
  - Too large to be handled by a program all at once
  - Needed several times when you don’t want to type it into your program more than once
- All files (data and programs) are ultimately stored as 0’s and 1’s but there are two general types of encodings which you choose between depending on your purposes

Text Files

- The bits represent printable characters
- Stores characters, one at a time
    - One byte per character for ASCII
    - Two bytes per character for Unicode
- Can be written, read and edited by programs and text editors
    - For example, Java source files are text files
- Are very transportable (eg: send by email)

Binary Files

- The bits represent other types of encoded information, such as executable instructions or numeric data
- All non-text files are called binary files
  - Examples include movie files, music files
- Are easily read by the computer but not humans
- Are not “printable” files (actually you can print them, but they will be unintelligible)

Binary Files

- Different types of values coded differently to maximize efficient use of space (eg: each integer takes 4 bytes)
- Can only be written and read by programs (eg: Java programs) which know the types of values being stored - can not normally be read by a text editor
- Are transportable (especially in Java)

Every File has Two Names

- In Java, the code to open the file creates two names for an output file
  - The name used by the operating system
    - For example: `out.txt`
  - The stream name variable
    - For example: `outputStream`
- Both are user/programmer defined names
- Java programs use the stream names (eg: `outputStream`)

Open – Loop – Close

- I/O in Java consists of:
  - OPENING: creating a stream object for each input source or output destination and associating the object with the external entity
  - LOOPING: getting values in or sending values out by calling methods on the stream object and then
  - CLOSING the file or connection by calling a close method on the stream

Open – Loop – Close

- Open once: you will need to create a stream object and say what external entity it corresponds to
- In doing the main work of the program just refer to the stream object
- At the end make sure that you close the stream
- There are different classes of stream objects appropriate to the task
  - Found in `java.io.*` library

Which Stream Object to Use?

- For writing output to a text file, use an object of class `PrintWriter`
  - This class has methods needed to create and write to a text file
- For reading input from a text file use a `java.util.Scanner` object
- For writing output to a binary file, use a `ObjectOutputStream` object
- For reading input from a binary file, use a `ObjectInputStream` object

Which Stream Object to Use?

- Errors are very possible and should be handled via exceptions
- To use the classes `PrintWriter`, `ObjectOutputStream` and `ObjectInputStream` your program needs to import the java.io package:

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

- Or, import the specific class:

```
import java.io.PrintWriter;
```

Text File I/O: Writing

- To open the file:
  - Declare stream variable for referencing the stream
  - Invoke a `PrintWriter` constructor, pass the file name as an argument
  - Requires try and catch blocks

Text File I/O: Writing

```
String fileName = “out.txt”;
PrintWriter outputStream = null;
try {
  outputStream = new
  PrintWriter(fileName);
}
catch (FileNotFoundException e) {
  System.out.println(“Error opening”
                    + “ the file ” + filename);
  System.exit(0);
}
```

Text File I/O: Writing

- The second statement above declares `outputStream` as a variable of type `PrintWriter`
- The statement within the `try` block connects the object `outputStream` to the file named `out.txt`
- This is called **opening the file**
- If the file `out.txt` does not exist, a new empty file named `out.txt` will be created

Text File I/O: Writing

- If the file `out.txt` already exists, its (old) contents will be lost
- Data initially goes to memory buffer – when the buffer is full, it goes to the file
- Closing the file empties the buffer and disconnects from stream

Text File I/O: Writing

- Use via:

```
outputStream.println("This is a line.");
outputStream.print("A bit of a line.");
```

- Close via:

```
outputStream.close();
```

- An output file should be closed when you are done writing to it

Text File I/O: Writing

- If a program ends normally it will close any files that are open
- If a program automatically closes files when it ends normally, why close them with explicit calls to close?
- Two reasons:
  - To make sure it is closed if a program ends abnormally (it could get damaged if it is left open)
  - A file open for writing must be closed before it can be opened for reading

Text File I/O: Writing

- Although Java does have a class that opens a file for both reading and writing, it is not used in this unit

Example

```
/** TextFileOutputDemo.java from Savitch chapter 10.
   Input three lines of text and output them to a
   text file. */
import java.io.PrintWriter;
import java.util.Scanner;
public class TextFileOutputDemo {
  public static void main(String[] args) {
     String fileName = “out.txt”;
     // declare outputStream instance of PrintWriter
     PrintWriter outputStream = null;
```

Example

```
  // open out.txt and connect to object
outputStream
  try {
     outputStream= new PrintWriter(fileName);
  }
  // if unable to open file
  catch(FileNotFoundException e) {
     System.out.println("Error opening the
                                         file " + fileName);
     System.exit(0);
  }
```

Example

```
System.out.println("Enter three lines of text:");
Scanner keyboard = new Scanner(System.in);
for (int count=1;count <= 3;count++) {
   String line = keyboard.nextLine();
   outputStream.println(count+" "+line);
}
outputStream.close();
```

Example

```
     System.out.println("Those lines were
                       written to " + fileName);
  }// end main
}//end class
```

Java.io.PrintWriter Methods

- Some of the class `PrintWriter` methods for writing data to a text file:
- `PrintWriter(filename: String) – creates a PrintWriter object for the specified file`
- `print(s: String): void  – Writes a string`
- `print(c: char): void    – Writes a char`
- `print(i: int): void       – Writes an int`

  `print(d: double): void  – Writes a double`
- Also contains the overloaded `println` methods
- Also contains the overloaded `printf` methods
- See java API documentation for further details

Appending to a Text File

- If you connect a stream to an output file as in the above program example (`out.txt`), you always start with an empty file
- Sometimes you may want to add the program output to the end of an existing file
- This is called **appending to a file**
- This is achieved as follows:

```
outputStream = new PrintWriter(new
       FileOutputStream("out.txt", true));
```

Appending to a Text File

- The class `PrintWriter` does not have an appropriate constructor for this task, so we need to use class `FileOutputStream`
- The second parameter (**true**) of `FileOutputStream`'s constructor indicates that the file `out.txt` should not be replaced if it already exists
- If the file `out.txt` does not already exist, Java will create an empty file of that name
- The methods `print` and `println` will then append data at the end of the file

Opening a Text File: Reading

- To open a text file for input, we can use the `java.util.Scanner` class to connect the text file to a stream for reading
- So far, we have used the `Scanner` class to get input from the keyboard by passing `System.in` as an argument to the `Scanner`'s constructor
- Here we pass an instance of `File` class whose constructor can take a file name as parameter

Opening a Text File: Reading

- For example:

```
Scanner inputStream = new
              Scanner( new File("out.txt"));
```

- Note that we can not pass a file name to `Scanner`'s constructor directly
- The class **`File`** which has many useful methods (see later) can be used with file names
- If the file "`out.txt`" does not exist, `Scanner`'s constructor will throw a `FileNotFoundException`

Opening a Text File: Reading

- The following simple program from Savitch prompts the user to enter the name of a text file, reads data from that text file and writes them on to screen

Example

```
//TextFileInputDemo2.java from Savitch chapter 10
import java.io.*;
import java.util.*;
public class TextFileInputDemo2 {
  public static void main(String[] args) {
     System.out.println("Enter file name:");
     Scanner keyboard = new Scanner(System.in);
     String fileName = keyboard.next();
     Scanner inputStream = null;
```

Example

```
System.out.println("The file " + fileName
         + “contains the following lines: ”);
try {
   inputStream = new Scanner( new
                              File(fileName));
}
catch(FileNotFoundException e) {
   System.out.println("Error opening the
                           file " + fileName);
   System.exit(0);
}
```

Example

```
     while (inputStream.hasNextLine()) {
        String line = inputStream.nextLine();
        System.out.println(line);
     }
     inputStream.close();
  } // end main
} // end class TextFileInputDemo2
```

Testing for the End of Text Files

- There are several ways to test for end of file
- For reading text files in Java you can use one of the `Scanner` class methods as in the above program
- The following code loops around reading and then displaying each line in the file until the end of the file is reached
- The Scanner class method **`hasNextLine()`** returns true if there is another line (string) in the file available

Testing for the End of Text Files

```
while (inputStream.hasNextLine())
{
  String line = inputStream.nextLine();
  System.out.println(line);
}
```

- Note that all methods of the `Scanner` class that we have already used (eg, `nextLine()`, `next()`, `nextInt()`, `nextDouble()`, etc.) are available to us here and can be used as before

Testing for the End of Text Files

- Other methods of Scanner class which can be used to test for end of a file include:
- `Scanner_Object_Name`**`.hasNext()`** – returns true if more input data is available to be read by the method **`next()`**
- `Scanner_Object_Name`**`.hasNextInt()`** – returns true if more input data is available to be read by the method **`nextInt()`**

Testing for the End of Text Files

- `Scanner_Object_Name`**`.hasNextDouble()`** – returns true if more input data is available to be read by the method **`nextDouble()`**
- `Scanner_Object_Name`**`.hasNextFloat()`** – returns true if more input data is available to be read by the method **`nextFloat()`**
- See java API documentation for further details

Parsing Words in a String

- The class `StringTokenizer` can be used to parse a line into words
  - It is in the **`util`** library so you need to import `java.util.*;`
  - One of its useful methods is `hasMoreTokens` which can be used to check if there are more tokens
  - You can specify *delimiters* (the character or characters that separate words), the default delimiters are "white space" (space, tab, and newline)

Parsing Words in a String

- Eg: display words separated by any of the following characters:
  - Space
  - new line (\n)
  - period (.)
  - comma (,)

Parsing Words in a String

```
Scanner keyboard = new Scanner(System.in);
String inputLine = keyboard.nextLine();
StringTokenizer wordFinder = new
StringTokenizer(inputLine, " \n.,");

//the second argument is a string of the 4 delimiters
while(wordFinder.hasMoreTokens()) {
  System.out.println(wordFinder.nextToken());
}
```

Entering "Question, 2b. or !tooBee." in the above example, what output would you get:

Parsing Words in a String

- Entering "Question, 2b. or !tooBee." in the above example, would give the following output:
- Question
  2b
  or
  !tooBee
- Note that the `Scanner` class method **`next()`** can be used to parse an input String, so the `StringTokenizer` class is not needed for that purpose when the `Scanner` class is used

Binary File I/O

- Important classes for binary file **output** (to the file)
  - `ObjectOutputStream`
  - `FileOutputStream`
- Important classes for binary file **input** (from the file):
  - `ObjectInputStream`
  - `FileInputStream`

Binary File I/O

- Note that `FileOutputStream` and `FileInputStream` are used only for their constructors, which can take file names as arguments
- `ObjectOutputStream` and `ObjectInputStream` cannot take file names as arguments for their constructors

Binary File I/O

- To use these classes your program needs a line like the following:

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

- The classes `ObjectInputStream` and `ObjectOutputStream`:
    - Have methods to either read or write data one byte at a time
    - Automatically convert numbers and characters into binary

Binary File I/O

- Note that binary-encoded numeric files (files with numbers) are not readable by a text editor, but store data more efficiently
- **Remember:**
  - *input* means data into a program, not the file
  - similarly, *output* means data out of a program, not the file

Binary File I/O

- When writing to binary files using `ObjectOutputStream:`
  - The output files are binary and can store any of the primitive data types (int, char, double, etc.) and the String type
  - The files created can be read by other Java programs but are not printable
  - An `IOException` might be thrown

Binary File I/O

- To open a new output (binary) file:

```
ObjectOutputStream outputStream =
  new ObjectOutputStream(
     new FileOutputStream("numbers.dat"));
```

Binary File I/O

- Writing to an output (binary) file:
  - You can write data to an output file after it is connected to a stream class by using methods defined in `ObjectOutputStream` class
    - `writeInt(int n)`
    - `writeDouble(double x)`
    - `writeBoolean(boolean b)`
    - `writeChar(int c)` // takes int not char as argument
    - `writeUTF (String s)`
    - etc.

Binary File I/O

- Note that each write method throws `IOException,` which means we will have to write try-catch blocks for it

Binary File I/O

- Using `ObjectInputStream` to read data from binary files
  - Similar to opening an output file, but replace "output" with "input"

```
ObjectInputStream inputStream =
 new ObjectInputStream(
   new FileInputStream("numbers.dat"));
```

- For every output file method there is a corresponding input file method

Binary File I/O

- You can read data from an input file after it is connected to a stream class using methods defined `in ObjectInputStream`
  - `readInt()`
  - `readDouble()`
  - `readBoolean()`
  - `readUTF()`
  - `etc.`
- Note each write method throws `IOException`

Example

```
/** BinaryOutputDemo.java from Savitch chapter 10.
   Outputting to a binary file. */
import java.io.*;
import java.util.*;
public class BinaryOutputDemo {
  public static void main(String[] args) {
     String fileName = "numbers.dat";
     try {
     // open file numbers.dat as output stream
     // create ObjectOutputStream object connected to it
        ObjectOutputStream outputStream =
           new ObjectOutputStream(
              new FileOutputStream(fileName));
```

Example

```
Scanner keyboard=new Scanner(System.in);
System.out.println("Enter nonnegative
               integers, one per line.");
System.out.println("Place a negative
                     number at the end.");
int n;
do {
   n = keyboard.nextInt();
   // ObjectOutputStream objects have methods
   // for writing out primitive values to them
   outputStream.writeInt(n);
}while (n >= 0);
```

Example

```
   System.out.println("Numbers and
                             sentinel value");
   System.out.println("written to file " +
                                      fileName);
   outputStream.close(); // always close
}
catch(FileNotFoundException e) {
   System.out.println("Problem opening
                     the file " + fileName);
}
```

Example

```
        catch(IOException e) {
           System.out.println("Problem with
                     output to file " + fileName);
        }
    } // end main
} // end class BinaryOutputDemo
```

Example: Client

```
/** BinaryInputDemo.java from Savitch chapter 10.
    Reading input from a binary file. */
import java.io.*;
public class BinaryInputDemo {
  public static void main(String[] args) {
     String fileName = "numbers.dat";
     try {
        ObjectInputStream inputStream =
           new ObjectInputStream(
              new FileInputStream(fileName));
```

Example: Client

```
System.out.println("Reading the non-
                     negative integers");
System.out.println(" in the file
                           numbers.dat.");
int n = inputStream.readInt();
while (n >= 0) {
   System.out.println(n);
   n = inputStream.readInt();
}
```

Example: Client

```
        System.out.println("End of reading
                                        from file.");
        inputStream.close();
    }
    catch(FileNotFoundException e) {
        System.out.println("Problem opening
                               the file " + fileName);
    }
```

Example: Client

```
        catch(EOFException e) {
           System.out.println("Problem reading
                              the file " + fileName);
           System.out.println("Reached end of
                                         the file.");
        }
        catch(IOException e) {
           System.out.println("Problem reading
                             the file " + fileName);
        }
   } // end main
} // end class BinaryInputDemo
```

I/O Exception Handling

- File I/O can produce several exceptions (all defined in java.io):
  - `FileNotFoundException` = trying to open a non-existent file for input
  - `EOFException` = trying to read in data after the binary file has ended (note that text files operate differently)
  - `IOException` is a class which includes as sub-classes these and other exceptions which may get thrown by I/O: you almost always have to handle `IOExceptions`

I/O Exception Handling

- Catching an `EOFException` is a good way to finish reading a binary data file
- In the following example also note:
  - Getting a file name from the user
  - Reading and writing Strings to binary files using the UTF (= Unicode Text Format) encoding (the recommended way of getting Strings represented in binary)

Example

```
import java.io.*;
import java.util.*;
public class StringIO {
  // uses binary file
  public static void main(String[] args) {
     System.out.println ("String storage
                                         manager.");
     char choice='q';
     Scanner keyboard = new
                             Scanner(System.in);
```

Example

```
  do {
      System.out.println("Choices are:");
      System.out.println("q to quit.");
      System.out.println("s to enter and save " + "a 
binary file of Strings");
      System.out.println("v to view a " +
                            "binary file of Strings");
      System.out.println("Enter choice:");
      choice = (keyboard.next()).charAt(0);
```

Example

```
      if (choice == 's') saveFile();
      else  if (choice == 'v') viewFile();
      else if (choice != 'q')
         System.out.println("Choice not
                                    recognized.");
   } while (choice != 'q');
   System.out.println("Thank you for
      using the String storage manager.");
} //end of main method
```

Murdoch
UNIVERSITY

Example

```
static void saveFile() {
   System.out.println("Please enter name of file " +
 "to save Strings in.");
   String fileName= getFileName(); // input
   try {
      ObjectOutputStream os =
         new ObjectOutputStream(
            new FileOutputStream(fileName));
```

Example

```
System.out.println("Enter Strings " +
            "to store, one per line.");
System.out.println("Enter an empty
                  line " + "to finish.");
String s;
Scanner keyboard=new Scanner(System.in);
do {
   s = keyboard.nextLine();
   if (! s.equals("")) os.writeUTF(s);
} while (! s.equals(""));
```

Example

```
      os.close();
      System.out.println("Data stored
   successfully in " + fileName);
   } // end try block
   catch (IOException e) {
      System.out.println("Input problem.");
   }
} //end of saveFile method
```

Example

```
static void viewFile() {
    System.out.println("Please enter name
                                    of file to view.");
    String fileName= getFileName(); // input
    try { // outer try block
        ObjectInputStream is =
            new ObjectInputStream(
                new FileInputStream(fileName));
```

Example

```
System.out.println("Here are the
         Strings stored in " +
         fileName + ", one per line.");
String s;
try { // inner try block
   do {
      s = is.readUTF();
      System.out.println(s);
   } while (true);
} // end inner try block
```

Example

```
   catch (EOFException e){ //empty block
   }
   is.close();
   System.out.println("That was the
               contents of " + fileName);
} // end outer try block
catch(FileNotFoundException e) {
   System.out.println("File " + filename
                           + " not found.");
}
```

Example

```
        catch (IOException e) {
            System.out.println("Output problem.");
        }
    } //end of viewFile
```

Example

```
  static String getFileName() {
     System.out.println("Enter file name:");
     Scanner keyboard = new Scanner(System.in);
     String fn = keyboard.nextLine();
     return fn;
  } //end of getFileName
} //end of class StringIO
```

File Management

- We have seen how to specify files using just their String names
- If more complicated management is needed then it is useful to make an object of the **`File`** class
- Eg: `File f = new File(“numbers.dat”);`
- `FileInputStream` and `FileOutputStream` classes have constructors that take a `File` argument as well as constructors that take a String argument

File Management

- We can:
  - Check whether the file exists or not via `f.exists()` (true or false)
  - Check whether the program can read the file (ie has permission) via `f.canRead()`
  - Find out the full path name of the file via `String path = f.getPath()` which might return "`C:\My Documents\Progs\numbers.dat`"
- Note that you should do such checks before writing to a file because an existing file with that name may be overwritten

Text File Input: BufferedReader

- You can also use the `BufferedReader` class for text file input (instead of the `Scanner` class)
- To open a text file for input, connect the text file to a stream for reading as follows:
  - Use a stream of the class `BufferedReader` and connect it to a text file
  - Use the `FileReader` class to connect the `BufferedReader` object to the text file

Text File Input: BufferedReader

- For example:

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

Text File Input: BufferedReader

- Then:
  - Read lines (Strings) with `readLine` (returns null when `eof` is reached)
  - `BufferedReader` has no methods to read numbers directly, so read numbers as Strings and then convert them (eg, `double d = Double.parseDouble (str);`)
  - Read a char with read (returns -1 when end of file is reached)

Text File Input: BufferedReader

- Note that you can only read Strings or single chars from a text file using the `BufferedReader` class
- The `Scanner` class is much more flexible

Example: LowerToUpper

```
/** Copies one text file to another changing lower case
   characters to upper case. Uses BufferedReader and
   FilerReader classes for input instead of the Scanner
   and File classes */
import java.io.*;
public class LowerToUpper {
  public static void main(String[] args) {
     System.out.println("Welcome to the lower -> " +
   "upper case converter.");
     System.out.print("Please enter the name
                          of file to process: ");
```

Example: LowerToUpper

```
  String inFileName = keyboard.next();
  System.out.println("Please enter the  name of " +
"file to save result in.");
  String outFileName= keyboard.next();
  try {
     PrintWriter pw = new
                    PrintWriter(outFileName);
     BufferedReader br =
        new BufferedReader(
           new FileReader(inFileName));
```

Example: LowerToUpper

```
   int nextCharVal=0;
   while((nextCharVal=br.read()) != -1)
      pw.print(Character.toUpperCase(
                        (char)nextCharVal));
   pw.close();
   br.close();
   System.out.println("Files converted
                                 and closed.");
}
```

Example: LowerToUpper

```
     catch(FileNotFoundException e) {
        System.out.println("File not found.");
     }
     catch(IOException e) {
        System.out.println("IO problem.");
     }
  } //end of main
} //end of class
```

End of Topic 9

Murdoch
UNIVERSITY