1.00 Lecture 30

Input/Output
Introduction to Streams

Reading for next time: Big Java 15.5-15.7

Sending information to a Java program

- **So far: use a GUI**
  - **limited to specific interaction with user**
  - **sometimes tedious (entering matrix elements)**
- **What if you have lots of data to send to the program?**
- **What if the program will generate lots of data?**
- **How do you tell Java where to find the data? How does Java get the data?**

Streams

- **Java can communicate with the outside world using *streams***
- **Picture a pipe feeding data into your Java program**
  - **where can the data come from?**
  - **keyboard input, files, other programs, network sockets, other streams**
- **Picture a pipe leading out of your Java program**
  - **where can the data go to?**
  - **screen output, files, other programs, network sockets, other streams**

Java I/O

- **I/O -- input/output, how you get data into and out of your program**
- **Streams abstract away the details of I/O**
  - **have the same methods whatever your data source or destination**
- **Streams work in one direction only**
  - ***input streams* control data coming into program from some source**
  - ***output streams* control data leaving the program for some destination**
  - **if you want to both read and write data, you'll need two separate streams**

Java Stream Classes

- **Java provides a hierarchy of classes for streams (in java.io.*)**
  - **Abstract, top-level classes that define general methods for different types of streams**
    - **InputStream -- reads bytes**
    - **OutputStream -- writes bytes**
    - **Reader -- reads characters**
    - **Writer -- writes characters**
  - **Many, many subclasses that implement streams**
    - **Some are tailored for specific data sources or destinations (FileReader reads chars from a file)**
    - **Some add functionality to existing streams (BufferedReader buffers input for more efficiency)**

Characteristics of Streams

- **FIFO queues**
  - **input streams deliver data to program in the order it was read from source**
  - **output streams deliver data to destination in order it was generated from program**
- **Basic streams provide sequential access**
  - **no rewind or backup**
  - **some streams (like RandomAccessFile) provide more functionality**

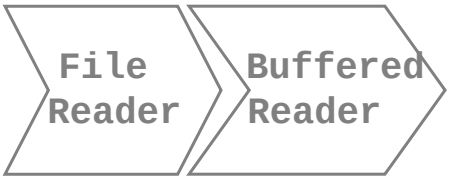
System.out

- What exactly is System.out.println()?
- Turns out you've been using streams all along...
  - System is a special class that is automatically instantiated once when your program runs
  - It has three static member variables
    - in -- InputStream (connected to terminal input)
    - out -- PrintStream (connected to terminal output)
    - err -- PrintStream (connected to error output -- screen or special window in IDE)
  - println() is a (overloaded) method in PrintStream that takes a String (or primitive data type) as an argument, prints it to a stream and adds a line termination character.

Connecting Streams

- Sometimes the stream you use to input/output data doesn't have all the functionality you need
- Some streams can be connected by using one stream as the constructor argument to another
  - i.e., add BufferedReader to FileReader
  - this reads a file more efficiently

Stream Pipeline



- `FileReader` reads characters from a text file
- `BufferedReader` buffers the character stream for efficiency and allows you to read line by line (`readLine()`)

Exercise 1: Download and Run

```
import java.io.*;            // Move TestIn.txt to C: or similar
public class SimpleReader {
    public static void main(String[] args) {
        try {
            FileReader fin = new FileReader("C:/TestIn.txt");
            BufferedReader b = new BufferedReader(fin);
            FileWriter fout = new FileWriter("C:/TestOut.txt");
            String currentLine = "";
            int i = 1;
            while ((currentLine = b.readLine()) != null) {
                fout.write((i++) + " " + currentLine + "\n");}
            fin.close();
            fout.close();
            System.out.println("Done");
        }
        catch (FileNotFoundException ef) {
            System.out.println("File not found");}
        catch (IOException ei) {
            System.out.println("IO Exception"); }
    }
}
```

Exercise 2

- Create new text file of 10 lines for your program to read
  - What happens if it doesn't exist?
- Try to change the order of the catch{} blocks
  - What happens, and why?
- Change the while statement to (a bad idea):

```
while (b.readLine() != null) {
   fout.write((i++) + " " + b.readLine() + "\n");}
```

  - What happens, and why? (It's a common error)
- "Accidentally" write to your input text file (e.g. TestIn.txt)
  - Make a copy of your input text file first
  - What happens?
- Other notes:
  - Always check for end of file (EOF):
    - readLine() returns null
    - read() returns –1 (when reading characters)
  - Always close your streams when done: save system resources, avoid file corruption if system crashes
  - Use Wordpad, not Notepad to look at your files. (end of lines)

The 3 Flavors of Streams

In Java, you can read and write data to a file:

- as text using `FileReader` and `FileWriter`
- as binary data using `DataInputStream` connected to a `FileInputStream` and as a `DataOutputStream` connected to a `FileOutputStream`
- as objects using an `ObjectInputStream` connected to a `FileInputStream` and as an `ObjectOutputStream` connected to a `FileOutputStream`

Parsing

- readLine() is okay if you want to read whole lines
- read() is okay if you want to read character by character
- What if you have structured data?
  - meaning is dependent on position or formatting
  - comma-separated values (or other delimiters/separators)
- Reading this data in a meaningful way is called *parsing*

Parsing

- When you parse (tokenize) a file, you are looking for *tokens*
  - sequences of one or more characters that "belong" together
  - sometimes tokens are separated by *delimiters* (",", "\t", " ", "\n"), sometimes not
- Two ways to parse in Java
  - StreamTokenizer : reads character by character, no delimiters
  - StringTokenizer : reads entire String, has delimiters

StringTokenizer

- In java.util
- Can tokenize any String, not just from streams
- 3 constructors, with 1, 2 or 3 arguments
  - One argument: String to be parsed.
    - Use default delimiter set " \t\n\r\f“
    - Space, tab, new line, carriage return, line feed
  - Two arguments: String to be parsed, String of delimiters
  - Three arguments: 3rd argument is flag to return delimiters, which are not returned normally
- nextToken() returns next token as a String
- hasMoreTokens() returns false when you don't have any more tokens left
- There is also a StreamTokenizer with similar features
  - Works with character streams, assembles tokens

Reading and writing Students

```
import java.io.*;

public class Student implements Serializable { // Object IO only
   private String name;
   private int year;
   private double gpa;
   public Student() {};                          // Constructors
   public Student(String n, int y, double g) {
        name = n; year = y; gpa = g; }
   public double getGpa() { return gpa; }       // Getters
   public String getName() { return name; }
   public int getYear() { return year; }
   public void setGpa(double d) { gpa = d; }    // Setters
   public void setName(String string) { name = string; }
   public void setYear(int i) { year = i; }
   public String toString() {
        return (name + "    \t" + year + "    \t" + gpa); }
}
```

Students in text files

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

public class StudentFile {
   public static void main(String[] args) {
        Student[] team= new Student[4];
        team[0]= new Student("Jennifer Wang", 1984, 5.0);
        team[1]= new Student("Helen Smithson", 1985, 5.0);
        team[2]= new Student("Rashika Mathews", 1983, 5.0);
        team[3]= new Student("Ferd Johnson", 1981, 5.0);
        try {
                FileWriter f= new FileWriter("student.txt");
                PrintWriter out= new PrintWriter(f);
                writeData(team, out);
                out.close();
                FileReader fin= new FileReader("student.txt");
                BufferedReader in= new BufferedReader(fin);
                Student[] newTeam= readData(in);
                in.close();
                for (int i=0; i < newTeam.length; i++)
                        System.out.println(newTeam[i]);
        } catch(IOException e) { System.out.println(e); }  }
```

Students in text files, p.2

```
   public static void writeData(Student[] s, PrintWriter out)
                throws IOException {
        out.println(s.length);
        for (int i= 0; i < s.length; i++) {
                String name= s[i].getName();
                int year= s[i].getYear();
                double gpa= s[i].getGpa();
                out.println(name + "|" + year + "|" + gpa); }
   }
   public static Student[] readData(BufferedReader in)
                throws IOException {
        int n= Integer.parseInt(in.readLine());
        Student[] sArr= new Student[n];
        for (int i=0; i < n; i++) {
                sArr[i]= new Student();
                String str = in.readLine();
                StringTokenizer t = new StringTokenizer(str, "|");
                sArr[i].setName(t.nextToken());
                sArr[i].setYear(Integer.parseInt(t.nextToken()));
                sArr[i].setGpa(Double.parseDouble(t.nextToken())); }
        return sArr;
   }
}
```

Exercise 3

- **Download and run StudentFile**
  - **Look at student.txt in Wordpad or other editor**
- **Questions:**
  - **Does it still work if you use just a FileReader, not a Buffered Reader? Remove it and see.**
  - **Does it work with just FileWriter, not a PrintWriter? Remove it and see.**
  - **What would change if we didn't have the number of students as the first line of the file?**

Exercise 4

- **Modify StudentFile so that it does not write or read the number of Students as the first line of the file**
  - **Use Exercise 1 as a guide**
  - **You may assume a maximum of 100 students if you need to at any point in the program.**
  - **Do you see the dilemma? And a solution?**