Notes: Token-based Processing and File Output

File Paths

- Absolute vs Relative File Paths
    - **absolute path**: complete path to a file, can use that path anywhere on your system and it will locate the file
        - usually begins with `C:/` (on Windows) or `/` (on Mac/Linux)
        - e.g. for Windows: `C:/Documents/cs141/lectures/day11/numbers.txt`
        - e.g. for Mac/Linux: `/Users/chess/Documents/cs141/lectures/day11/numbers.txt`
    - **relative path**: path to the file from the current directly, if you used this path in a different directory it would not go to your file
        - usually are a single file name or a series of folders followed by a single file name
        - e.g. for Windows: `lectures/day11/numbers.txt`
        - e.g. for Mac/Linux: `lectures/day11/numbers.txt`
- Note that when you put File Paths into code, you need to make sure that all the slashes are forward slashes (/) because otherwise you could create escape sequences in your path Strings such as "\n" in "day11**\n**umbers.txt"
- Some directory (folder) commands that you should know:
    - `.` means the directory that you are in
    - `/` means the root directory (when at the front of a path)
    - `..` means the parent directory ("go back a folder")
    - Files end with an extension (.doc, .txt, .java, .pdf, etc) and Folders end with a slash (cs141/, homeworks/, Documents/, etc)
    - `*` means "wildcard"; meaning, anything can go here

File Processing

- To make a .txt file in jGRASP: File > New > Plain Text
  - We will use text files to store information that we later read out instead of reading from the console
- To read from a text file:
  - Use the File class. You need to have an import statement, import java.io.*;
  - Create a new File object by saying: File f = new File("numbers.txt");
  - Then connect your file to a Scanner object with: Scanner input = new Scanner(f);
  - Or you can combine these lines into one: Scanner input = new Scanner(new File("numbers.txt"));
- Once you have a Scanner object, you can then use the methods we have learned previously for reading in information: input.next(), input.nextInt(), input.nextDouble()
  - Note that Scanner objects can only go forwards, they cannot read information backwards; if you need to read a file twice then you would need to create two Scanner objects
- Because of some exception rules in Java you need to add throws FileNotFoundException to any method that constructs a Scanner from a File object or calls a method that does so

File Example

```
import java.io.*;   // to use the File class
import java.util.*; // to use the Scanner class
public class FileExample {
   public static void main(String[] args) throws FileNotFoundException {
      Scanner input = new Scanner(new File("numbers.txt"));
      double sum = 0.0;
      while (input.hasNextDouble()) {
         double n = input.nextDouble();
         System.out.println("n = " + n);
         sum += n;
      }
      System.out.println("sum = " + sum);
   }
}
```

Testing for valid input with Scanner

Assuming a Scanner object has already been created named *input*

Method name	**Description**
input.hasNext()	returns true if there are more tokens of input to read
input.hasNextInt()	returns true if there is a next token and it can be read as an int
input.hasNextDouble()	returns true if there is a next token and it can be read as an double

Common Errors/Exception Messages

- InputMismatchException: When you try and read a token of the wrong type
- NoSuchElementException: When you try and read a token that does not exist

Common File Methods

Method name	**Description**
f.canRead()	returns whether the file f is able to be read
f.delete()	removes the file f from the disk
f.exists()	returns true if the file f exists, otherwise returns false
f.getName()	returns the name of file f
f.length()	returns the number of bytes in the file f
f.renameTo(name)	changes the name of file f to *name*

File Output / PrintStream

- Requires creation of a PrintStream object
- PrintStream is an object in the java.io package that lets you print output to a destination (e.g., a file)
- All the methods you have been using for System.out can also be used on PrintStream objects
- Important PrintStream details
  - If a given file does not exist, then it will be created for you
  - If a given file already exists, then it will be overwritten
  - The output you print will no longer appear on the console (it will be written to the file instead)
  - Do not open the same file for both reading and writing at the same time

Code Example

```
PrintStream output = new PrintStream(new File("output.txt"));
output.println("hello world");
```

Scanning a String

- In addition to using a Scanner to read input from the console (Scanner console = new Scanner(System.in);) or using the Scanner to read input from an input file (Scanner input = new Scanner(new File("data.txt"));, you can also use a Scanner to read tokens from a simple String:
    - Scanner lineScan = new Scanner("scan this string literal");
- Consider the following ex which prints all words in the string that begin with "a"

```
// creates a new Scanner that scans through the String literal provided
Scanner lineScan = new Scanner("spider ant elephant aardvark antelope");
while(lineScan.hasNext()) {
    //reads in current token and advances to the next token
    String word = lineScan.next();
    if(word.startsWith("a")) {
        System.out.println(word);
    }
}
```

- Example which counts the number of words in a String:

```
public class ScanStringExample {
   public static void main(String[] args) {
      Scanner console = new Scanner(System.in);
      System.out.print("Enter a phrase > ");
      String line = console.nextLine();   // nextLine reads until a \n
      int count = 0;
      // creates a new Scanner that scans through the phrase entered
      Scanner lineScan = new Scanner(line);
      while(lineScan.hasNext()) {
          //reads in current token and advances to the next token
          String word = lineScan.next();
          count++;
      }
      System.out.println("Number of words entered = " + count);
   }
}
```

File Processing

- Scanner is the main tool to use for file processing

- When doing both token-based processing and line-based processing use two different Scanner objects

Token-based Processing

- Processes the data in tokens using nextInt(), nextDouble(), and next()
- Skips past any newline characters
- Should not be used if your input is line-based, because token-based processing ignores the line breaks and looks only at the tokens

Line-based Processing

- Process data by line using nextLine()
- When doing line-based processing you often use a while loop, because you are unsure of the number of lines you will read in; an example method is shown below. Note that the method below returns the empty string if nothing is found.

```
// searches for and returns the next line of the given input that contains
// the given phrase; returns an empty string if not found
public static String find(Scanner input, String phrase) {
   while (input.hasNextLine()) {
      String line = input.nextLine();
      if (line.toLowerCase().contains(phrase)) {
         return line;
      }
   }
   return "";
}
```

Hybrid Approach

- Use line-based processing on the file, but then use token-based processing for the individual lines of the file
- Do this by passing the line itself (a String) into a new Scanner object to use token-based processing
- An example of this approach is shown below

```
public static void print(String line) {
   Scanner data = new Scanner(line);
   String name = in.next();
   for
   System.out.print(rank + "\t" + rating + "\t" + votes + "\t");
   while (data.hasNext()) {
      System.out.print(data.next() + " ");
   }
   System.out.println();
}
```