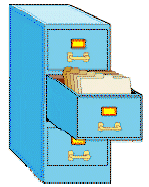
Topic 18
File Processing

"We have also obtained a glimpse of another crucial idea about languages and program design. This is the approach of stratified design, the notion that a complex system should be structured as a sequence of levels that are described using a sequence of languages. Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail. "
- Hal Abelson and Gerald Sussman

Based on slides for Building Java Programs by Reges/Stepp, found at http://faculty.washington.edu/stepp/book/

File objects



- Java's File class represents files on the user's computer.
  - Creating a File object does not actually create that file on your hard drive. (But a program can be used to create files.)
  - When we want to read data out of a file, we create a File object representing that file and open it with a Scanner.
- Creating a Scanner for a File, general syntax:

```
Scanner <name> = new Scanner(new File("<file name>"));
```

  - Example:

```
Scanner input = new Scanner(new File("numbers.txt"));
```

- The File class is in Java's `java.io` package, so we must write this at the top of our program:

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


Exceptions, throwing

- **exception**: A Java object that represents a program error.
  - they occur when the program is running
  - a.k.a. "runtime error"
- **checked exception**: An error that Java forces us to handle in our program. Otherwise the program will not compile.
  - Java forces us to specify what our program should do to handle potential failures when opening a file.
- **`throws` clause**: Keywords that can be added to the header of our methods to explain to Java that we plan NOT to handle file input failures. (We'll just let the program crash in such a case.)
- Throws clause, general syntax:

```
public static <type> <name>(<params>) throws <type> {
```

  - Example:

```
public static void main(String[] args)
                            throws FileNotFoundException
{
```


Sample file input program

```
import java.io.*;

// Displays each number in the given file,
// and displays their sum at the end.
public class Echo2 {
    public static void main(String[] args)
                                throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        for (int i = 1; i <= 5; i++) {
            double next = input.nextDouble();
            System.out.println("number " + i + " = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Input File numbers.dat:

```
308.2 14.9 7.4
 2.8

3.9 4.7    -15.4
    2.8
```

Output:

```
number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
Sum = 337.19999999999993
```

Testing before reading

- Reminder: The Scanner has useful methods for testing to see what the next input token will be:

Method Name	Description
`hasNext()`	whether any more tokens remain
`hasNextDouble()`	whether the next token can be interpreted as type double
`hasNextInt()`	whether the next token can be interpreted as type int
`hasNextLine()`	whether any more lines remain

- You can call these methods as a condition of an `if` statement or `while` loop.

Example test before read

```
import java.io.*;

// Displays each number in the given file,
// and displays their sum at the end.
public class Echo3 {
    public static void main(String[] args)
                                throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNextDouble()) {
            double next = input.nextDouble();
            System.out.println("number " + i + " = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Input File numbers.dat:

```
308.2 14.9 7.4
 2.8

3.9 4.7    -15.4
    2.8
```

Output:

```
number 1 = 308.2
number 2 = 14.9
number 3 = 7.4
number 4 = 2.8
number 5 = 3.9
number 6 = 4.7
number 7 = -15.4
number 8 = 2.8
Sum = 329.29999999999995
```


Files and input cursor

- Recall that a Scanner views all input as a stream of characters, which it processes with its *input cursor*:

```
- 308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
  ^
```

- Each call to next, nextInt, nextDouble advances the cursor to the end of the current token (whitespace-separated)

```
input.nextDouble()
- 308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
       ^
input.nextDouble()
- 308.2 14.9 7.4\n2.8\n\n\n3.9 4.7 -15.4\n2.8\n
            ^
```


File processing problem

- Write a program that reads a file of integers and prints their average. For added challenge, make your program able to skip over any non-integer tokens in the file and ignore them.

Example input file:

```
1 TEST 2 3 4 5 3.7 6 many bad tokens 7

27.5

8 9 Hello
10
```

Output:

```
Average = 5.5
```

File processing problem

- Write a program that accepts an input file containing integers representing daily high temperatures.
  Example input file:
  ```
  42 45 37 49 38 50 46 48 48 30 45 42 45 40 48
  ```
- Your program should print the difference between each adjacent pair of temperatures, such as the following:

```
Temperature changed by 3 deg F
Temperature changed by -8 deg F
Temperature changed by 12 deg F
Temperature changed by -11 deg F
Temperature changed by 12 deg F
Temperature changed by -4 deg F
Temperature changed by 2 deg F
Temperature changed by 0 deg F
Temperature changed by -18 deg F
Temperature changed by 15 deg F
Temperature changed by -3 deg F
Temperature changed by 3 deg F
Temperature changed by -5 deg F
Temperature changed by 8 deg F
```


File names and Java editors

- Relative path: "readme.txt" or "input\readme.txt"
- Absolute path: "C:\\Documents\\smith\\hw6\\input\\readme.txt"

- In most editors when you construct a File object with just a file name, Java assumes you mean a file in the *current directory*.
  - Scanner input = new Scanner(new File("readme.txt"));
  - If our program is in the folder C:\Documents and Settings\johnson\hw6, that is where Java will look for readme.txt.

Line-by-line processing

- Scanners have a method named `nextLine` that returns text from the input cursor's current position forward to the nearest \n new line character.
  - You can use `nextLine` to break up a file's contents into each line and examine the lines individually.

- Reading a file line-by-line, general syntax:

```
Scanner input = new Scanner(new File("<file name>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    <process this line...>;
}
```


File processing problem

- Write a program that reads an email message from a text file and turns it into a quoted message by putting a > and space in front of each line. Example input:

```
Kelly,

Can you please modify the a5/turnin settings
to make CS305J Homework 5 due Wednesday,
July 27 at 11:59pm instead of due tomorrow
at 6pm?

Thanks, Joe
```

- Example output:

```
> Kelly,
>
> Can you please modify the a5/turnin settings
> to make CS305J Homework 5 due Wednesday,
> July 27 at 11:59pm instead of due tomorrow
> at 6pm?
>
> Thanks, Joe
```

Processing tokens of one line

- Often the contents of each line are themselves complex, so we want to tokenize each individual line using its own Scanner.
  - Example file contents:

```
Susan 12.5 8.1 7.6 3.2
Brad 4.0 11.6 6.5 2.7 12
Jennifer 8.0 8.0 8.0 8.0 7.5
```

- A Scanner can be constructed to tokenize a particular String, such as one line of an input file.

```
Scanner <name> = new Scanner(<String>);
```

- Processing complex input, general syntax:

```
Scanner input = new Scanner(new File("<file name>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    <process this line...>;
}
```


Line-based input, cont'd.

- Often we have to write programs that scan through a file, finding the appropriate piece of data, and process only that piece:

```
Enter a name: Brad
Brad worked 36.8 hours (7.36 hours/day)
```

- Often we have files with a particular token of importance on each line, followed by more data:

hours.txt

```
Susan 12.5 8.1 7.6 3.2
Brad 4.0 11.6 6.5 2.7 12
Jennifer 8.0 8.0 8.0 8.0 7.5
```


File processing problem

- Write a program that reads in a file containing pseudo-HTML text, but with the HTML tags missing their < and > brackets.
  - Whenever you see any uppercase token in the file, surround it with < and >, and output the corrected file text to the console.
  - You must retain the original orientation/spacing of the tokens on each line. (Is this problem line-based or token-based?)

Input file:

```
HTML
HEAD
TITLE My web page /TITLE
/HEAD
BODY
P There are pics of my cat here,
as well as my B cool /B blog,
which contains I awesome /I
stuff about my trip to Vegas.
/BODY /HTML
```

Output to console:

```
<HTML>
<HEAD>
<TITLE> My web page </TITLE>
</HEAD>
<BODY>
<P> There are pics of my cat here,
as well as my <B> cool </B> blog,
which contains <I> awesome </I>
stuff about my trip to Vegas.
</BODY> </HTML>
```


Searching a file for a line

- Recall: reading a file line-by-line, general syntax:

```
Scanner input = new Scanner(new File("<file name>"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    <process this line...>;
}
```

- For this program:

```
Scanner input = new Scanner(new File("hours.txt"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    String name = lineScan.next();

    // ... if this is the name we want, add the hours
}
```

Processing a particular line

- A solution to the 'hours' problem shown previously:

```
Scanner input = new Scanner(new File("hours.txt"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    Scanner lineScan = new Scanner(line);
    String thisName = lineScan.next();      // "Brad"

    double sum = 0.0;
    int count = 0;
    if (name.equals(thisName)) { // we found the right person
        while (lineScan.hasNextDouble()) {
            sum += lineScan.nextDouble();
            count++;
        }
        double avg = sum / count;
        System.out.println(name + " worked " + sum +
                           " hours (" + avg + " hours/day)");
    }
}
```


Prompting for a file name

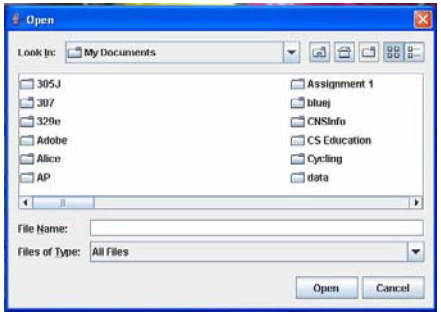
- Instead of writing the file's name into the program as a String, we can ask the user to tell us the file name to use.
  - Here is one case where we may wish to use the `nextLine` method on a console Scanner, because a file name might have spaces in it.

```
// prompt for the file name
Scanner console = new Scanner(System.in);
System.out.print("Type a file name to use: ");
String filename = console.nextLine();

// open the file
Scanner input = new Scanner(new File(filename));
```


Another Way of Prompting

- Java has a built in class named JFileChooser that can be used to select files.
- JFileChooser objects themselves are somewhat complicated to use, but a method can be written that returns the chosen file
- Benefit is files can be chosen using a traditional window



Fixing file-not-found issues

- What if the user types a file name that does not exist?
  - We can use the `exists` method of the File object to make sure that file exists and can be read.

```
Scanner console = new Scanner(System.in);
System.out.print("Type a file name to use: ");
String filename = console.nextLine();
File file = new File(filename);

while (!file.exists()) {
    System.out.println("File not found!  Try again: ");
    String filename = console.nextLine();
    file = new File(filename);
}
Scanner input = new Scanner(file);
```

Output:

```
Type a file name to use: hourz.text
File not found!  Try again: h0urz.txt
File not found!  Try again: hours.txt
```

Complex multi-line records

- Sometimes the data in the file consists of 'records', each of which is a group of information that may occupy multiple lines.
  - The following data represents students' courses. Each student has their name listed on a line, plus a group of courses, listed as a number of units and the student's grade in that course.

```
Erica Kane
3 2.8 4 3.9 3 3.1
Greenlee Smythe
3 3.9 3 4.0 4 3.9
Ryan Laveree
2 4.0 3 3.6 4 3.8 1 2.8
Adam Chandler
3 3.0 4 2.9 3 3.2 2 2.5
Adam Chandler, Jr
4 1.5 5 1.9
```

- How can we process one or all of these records?

IMDB movie ratings problem

- Write a program that reads Internet Movie Database (IMDB) top-250 data from a text file in the following format:

```
9.0 131336 The Godfather (1972)
9.0 159357 The Shawshank Redemption (1994)
8.9 113807 The Lord of the Rings: The Return of the King (2003)
8.9 75868 The Godfather: Part II (1974)
```

- Your program should prompt the user for a search phrase, and then read the IMDB file and output any movies that contain that phrase.

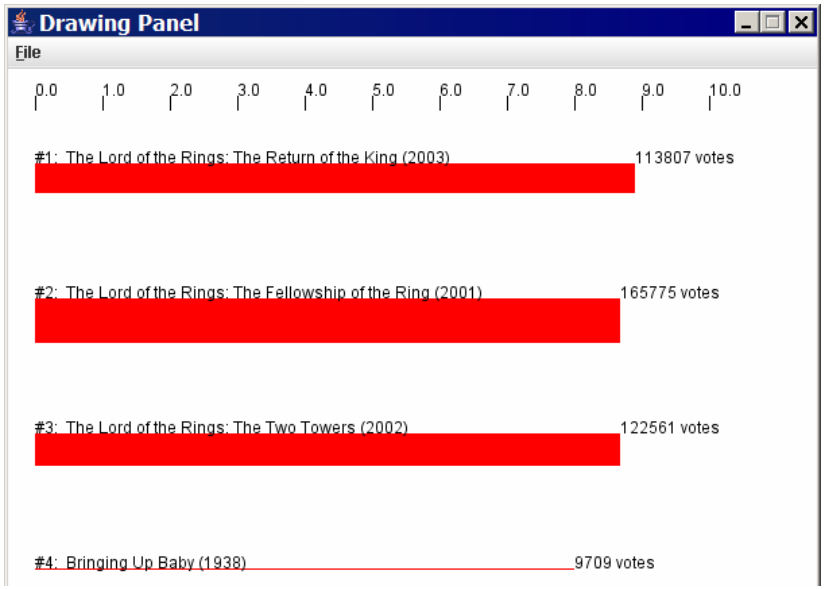
IMDB problem, continued

- Example execution dialogue (user input underlined):

```
Type a movie name or partial name: ring
#3: The Lord of the Rings: The Return of the King (2003)
    (113807 votes, 8.9 rating)
#10: The Lord of the Rings: The Fellowship of the Ring (2001)
    (165775 votes, 8.7 rating)
#14: The Lord of the Rings: The Two Towers (2002)
    (122561 votes, 8.7 rating)
#145: Bringing Up Baby (1938)
    (9709 votes, 8.0 rating)
```

- Expected graphical output:
  - top-left tick mark at (20, 20)
  - ticks 10 pixels tall, 50 pixels apart
  - first red bar top left corner at (20, 70)
  - 100 pixels apart vertically (max of 5)
  - 1 pixel tall for every 5000 votes earned
  - 50 pixels wide for each whole ratings point

Graphical Output



Output to files

- Java has an object named `PrintStream` (in the `java.io` package) that allows you to print output into a destination, which can be a file.
  - `System.out` is also a `PrintStream`.
  - Any methods you have used on `System.out` (such as `print`, `println`) will work on every `PrintStream`.
    - Why? `System.out` and the `PrintStream` objects you create are the same type of objects so they all have the exact same methods available.
  - You should explicitly end your file output by calling the `close()` method on the PrintStream when you are done with it.

- Printing into an output file, general syntax:

```
PrintStream <name> = new PrintStream(
        new File("<file name>"));
...
```


Printing to files, example

- Printing to a file, example:

```
PrintStream output = new PrintStream(
        new File("output.txt"));
output.println("Hello, file!");
output.println("This is a second line of output.");
output.close();
```

- Note: You should not open any file for both reading (`Scanner`) and writing (`PrintStream`) at the same time!
  - The result can be an empty file (size 0 bytes).
  - You could overwrite your input file by accident!