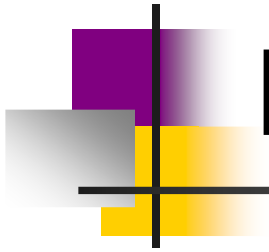
UNIVERSITY OF WASHINGTON

Building Java Programs

Chapter 6:
File Processing



Lecture outline

Lecture 14

- **File input using `Scanner`**
  - **`File` objects**
  - **throwing exceptions**
  - **file names and folder paths**
  - **token-based file processing**

Lecture 15

- Line-based file processing
  - processing a file line by line
  - examining the contents of an individual line
  - searching for a particular line in a file
  - handling complex and multi-line input records
  - handling the file-not-found case

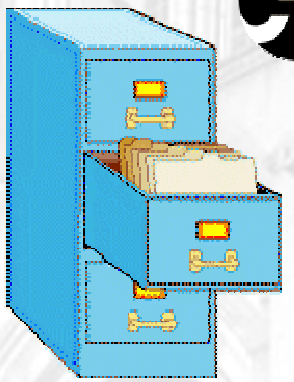
Lecture 16

- Complex input records
- File output using `PrintStream`

File input using Scanner

- suggested reading: 6.1 - 6.2

File objects



- Programmers refer to input/output as "I/O".

- Java's `File` class in the `java.io` package represents files on the user's hard drive.
  - `import java.io.*;`
  - To read a file, create a `File` object and pass it to 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"));
```

File and path names

- **relative path**: does not specify any top-level folder
  - `"names.dat"`
  - `"input/kinglear.txt"`
- **absolute path**: specifies drive letter or top `"/"` folder
  - `"C:/Documents/smith/hw6/input/data.csv"`
  - Windows systems also use backslashes to separate folders. How would the above filename be written using backslashes?

- When you construct a `File` object with a relative path, Java assumes it is relative to the *current directory*.
  - `Scanner input = new Scanner(new File("data/readme.txt"));`
  - If our program is in: `H:/johnson/hw6`,
    Java will look for: `H:/johnson/hw6/data/readme.txt`.

Compiler error with files

- The following program does not compile:

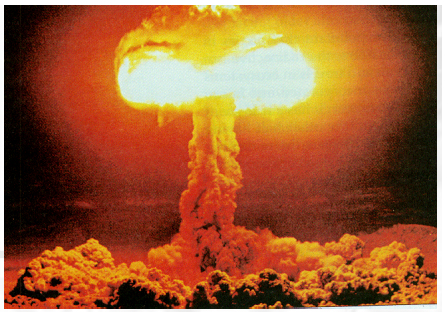
```
import java.io.*;     // for File
import java.util.*;   // for Scanner

public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- The following compiler error is produced:

```
ReadFile.java:6: unreported exception
java.io.FileNotFoundException; must be caught or declared
to be thrown
        Scanner input = new Scanner(new File("data.txt"));
                        ^
```

Exceptions



- **exception**: An object that represents a program error.
  - Programs that contain invalid logic cause ("*throw*") exceptions.
  - Trying to read a file that does not exist will throw an exception.

- **checked exception**: An error that Java forces us to handle in our program (otherwise it will not compile).
  - We must specify what our program will do to handle any potential file I/O failures.
  - We must either:
    - declare that our program will handle ("*catch*") the exception, or
    - explicitly state that we choose not to handle the exception (and we accept that our program will crash if an exception occurs)

Throwing exception syntax

- **`throws` clause**: Keywords on a method's header to state that it may throw an exception.
  - Somewhat like a waiver of liability form:
    "I hereby agree that this method might throw an exception, and I accept the consequences (crashing) if this happens."

- Throws clause, general syntax:

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

  - When doing file I/O, we use `FileNotFoundException`.

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

Fixed compiler error

- The following corrected program does compile:

```
import java.io.*;     // for File, FileNotFoundException
import java.util.*;   // for Scanner

public class ReadFile {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

Files and input cursor

- Consider a file `numbers.txt` that contains this text:

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

- A `Scanner` views all input as a stream of characters, which it processes with its *input cursor*:
  - `308.2\n   14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n`
    `^`

  - When you call the methods of the `Scanner` such as `next` or `nextDouble`, the `Scanner` breaks apart the input into *tokens*.

Input tokens

- **token**: A unit of user input. Tokens are separated by whitespace (spaces, tabs, new lines).

- Example: If an input file contains the following:

```
23   3.14
  "John Smith"
```

  - The tokens in the input are the following, and can be interpreted as the given types:

Token	Type(s)
1. `23`	`int`, `double`, `String`
2. `3.14`	`double`, `String`
3. `"John`	`String`
4. `Smith"`	`String`

Consuming tokens

- Each call to `next`, `nextInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace.
  - We call this *consuming* input.

```
input.nextDouble()
```

- **308.2**\n 14.9 7.4 2.8\n\n\n3.9 4.7 -15.4\n2.8\n
  ^

```
input.nextDouble()
```

- 308.2\n **14.9** 7.4 2.8\n\n\n3.9 4.7 -15.4\n2.8\n
  ^

File input question

- Consider an input file named `numbers.txt` that contains the following text:

```
308.2
   14.9 7.4  2.8

3.9 4.7    -15.4
    2.8
```

- Write a program that reads the first 5 values from this file and prints them along with their sum. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
Sum = 337.19999999999993
```

File input answer

```
// Displays the first 5 numbers in the given file,
// and displays their sum at the end.

import java.io.*;   // for File, FileNotFoundException
import java.util.*;

public class Echo {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.txt"));
        double sum = 0.0;
        for (int i = 1; i <= 5; i++) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Testing before reading

- The preceding program is impractical because it only processes exactly 5 values from the input file.
  - A better program would read the entire file, regardless of how many values it contains.

- 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

Test before read question

- Rewrite the previous program so that it reads the entire file. Its output:

```
number = 308.2
number = 14.9
number = 7.4
number = 2.8
number = 3.9
number = 4.7
number = -15.4
number = 2.8
Sum = 329.29999999999995
```

Test before read answer

```
// Displays each number in the given file,
// and displays their sum at the end.

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

public class Echo2 {
    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 = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

File processing question

- Modify the preceding program again so that it will handle files that contain non-numeric tokens.
  - The program should skip any such tokens.

- For example, the program should produce the same output as before when given this input file:

```
308.2  hello
   14.9 7.4  bad stuff 2.8

3.9 4.7  oops  -15.4
:-)    2.8  @#*($&
```

File processing answer

```
// Displays each number in the given file,
// and displays their sum at the end.

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

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.hasNext()) {
            if (input.hasNextDouble()) {
                double next = input.nextDouble();
                System.out.println("number = " + next);
                sum += next;
            } else {
                input.next();   // consume / throw away bad token
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

File processing question

- 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 processing answer

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

public class Temperatures {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.dat"));
        int temp1 = input.nextInt();

        while (input.hasNextInt()) {
            int temp2 = input.nextInt();
            System.out.println("Temperature changed by " +
                               (temp2 - temp1) + " deg F");
            temp1 = temp2;
        }
    }
}
```

Lecture outline

Lecture 14

- File input using `Scanner`
  - `File` objects
  - throwing exceptions
  - file names and folder paths
  - token-based file processing

Lecture 15

- **Line-based file processing**
  - **processing a file line by line**
  - **examining the contents of an individual line**
  - **searching for a particular line in a file**
  - **handling complex and multi-line input records**
  - **handling the file-not-found case**

Lecture 16

- Complex input records
- File output using `PrintStream`

Line-based file processing

- suggested reading: 6.3

Line-by-line processing

- Scanners have a method `nextLine` that returns from the input cursor's position to the nearest `\n` character.
  - You can use `nextLine` to break up a file's contents by line and examine each line 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...>;
}
```

Line-based input

- `nextLine` consumes and returns a line as a `String`.
  - The `Scanner` moves its cursor until it sees a `\n` new line character, and returns the text found.
    - The `\n` character is consumed but not returned.
    - `nextLine` is the only non-token-based `Scanner` method.
    - Recall that the `Scanner` also has a `hasNextLine` method.
  - Example:

```
23   3.14 John Smith   "Hello world"
          45.2      19

input.nextLine()
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
                                  ^

input.nextLine()
23\t3.14 John Smith\t"Hello world"\n\t\t45.2  19\n
                                          ^
```

File processing question

- Write a program that reads a text file and "quotes" it by putting a > in front of each line. Example input:

```
Kelly,

Can you please modify the a5/turnin settings
to make CSE 142 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 CSE 142 Homework 5 due Wednesday,
> July 27 at 11:59pm instead of due tomorrow
> at 6pm?
>
> Thanks, Joe
```

File processing answer

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

public class QuoteMessage {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("message.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            System.out.println(">" + line);
        }
    }
}
```

Processing tokens of one line

- Many input files have a data record on each line.
  - The contents of each line contain meaningful tokens.
  - Example file contents:

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

- Consider the task of computing the total hours worked for each person represented in the above file.

```
Enter a name: Brad
Brad (ID#456) worked 36.8 hours (7.36 hours/day)
```

  - Neither line-based nor token-based processing is quite right.
  - The better solution is a hybrid approach in which we break the input into lines, and then break each line into tokens.

Scanners on Strings

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

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

  - Example:

```
String text = "1.4 3.2 hello 9 27.5";
Scanner scan = new Scanner(text);    // five tokens
```

- We can use this idea to tokenize each line of a file.

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

Complex input question

- Write a program that computes the total hours worked and average hours per day for a particular person represented in the following file.
  - Input file contents:

```
123 Susan 12.5 8.1 7.6 3.2
456 Brad 4.0 11.6 6.5 2.7 12
789 Jennifer 8.0 8.0 8.0 8.0 7.5 7.0
```

  - Example log of execution:

```
Enter a name: Brad
Brad (ID#456) worked 36.8 hours (7.36 hours/day)
```

  - Example log of execution:

```
Enter a name: Harvey
Harvey was not found
```

Searching 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...>;
}
```

- If we are looking for a particular line, often we look for the token(s) of interest on each line.
  - If we find the right value, we'll process the rest of the line.
  - Example: If the second token on the line is "Brad", process it.

Complex input solution

```
// This program searches an input file of employees' hours worked
// for a particular employee and outputs that employee's hours data.

import java.io.*;    // for File
import java.util.*;  // for Scanner

public class HoursWorked {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        System.out.print("Enter a name: ");
        String searchName = console.nextLine();   // e.g. "BRAD"
        boolean found = false;                    // a boolean flag

        Scanner input = new Scanner(new File("hours.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            Scanner lineScan = new Scanner(line);
            int id = lineScan.nextInt();          // e.g. 456
            String name = lineScan.next();        // e.g. "Brad"
            if (name.equalsIgnoreCase(searchName)) {
                processLine(lineScan, name, id);
                found = true;                     // we found them!
            }
        }

        if (!found) {    // found will be true if we ever found the person
            System.out.println(searchName + " was not found");
        }
    }
```

Complex input solution 2

```
    ...

    // totals the hours worked by one person and outputs their info
    public static void processLine(Scanner lineScan,
            String name, int id) {

        double sum = 0.0;
        int count = 0;
        while (lineScan.hasNextDouble()) {
            sum += lineScan.nextDouble();
            count++;
        }

        double average = sum / count;
        System.out.println(name + " (ID#" + id + ") worked " +
                sum + " hours (" + average + " hours/day)");
    }
}
```

File processing question

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

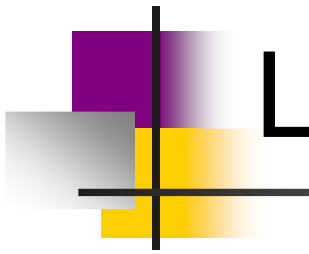
Input file:	Output to console:
HTML HEAD TITLE My web page /TITLE /HEAD BODY P There are pics of my cat here, as well as my B cool /B blog, which contains I awesome /I stuff about my trip to Vegas. /BODY /HTML	<HTML> <HEAD> <TITLE> My web page </TITLE> </HEAD> <BODY> <P> There are pics of my cat here,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

File processing solution

- The following code solves the HTML problem:

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

public class WebPage {
    public static void main(String[] args)
            throws FileNotFoundException {
        Scanner input = new Scanner(new File("page.html"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            Scanner lineScan = new Scanner(line);
            while (lineScan.hasNext()) {
                String token = lineScan.next();
                if (token.equals(token.toUpperCase())) {
                    // this is an HTML tag
                    System.out.print("<" + token + "> ");
                } else {
                    System.out.print(token + " " );
                }
            }
            System.out.println();
        }
    }
}
```



Lecture outline

Lecture 14

- File input using `Scanner`
  - `File` objects
  - throwing exceptions
  - file names and folder paths
  - token-based file processing

Lecture 15

- Line-based file processing
  - processing a file line by line
  - examining the contents of an individual line
  - searching for a particular line in a file
  - handling complex and multi-line input records

Lecture 16

- **handling the file-not-found case**
- **Complex input records**
- **File output using `PrintStream`**

Prompting for a file name

- We can ask the user to tell us the file to read.
  - We should use the `nextLine` method on the console `Scanner`, because the 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();

Scanner input = new Scanner(new File(filename));
```

- What if the user types a file name that does not exist?

Fixing file-not-found issues

- `File` objects have an `exists` method we can use:

```
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.print("File not found! Try again: ");
    filename = console.nextLine();
    file = new File(filename);
}
Scanner input = new Scanner(file);  // open the file
```

Output:

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

IMDB movie ratings problem

- Consider the following Internet Movie Database (IMDB) Top-250 data from a text file in the following format:

```
1  196376   9.1  Shawshank Redemption, The (1994)
2   93064   8.9  Godfather: Part II, The (1974)
3   81507   8.8  Casablanca (1942)
```

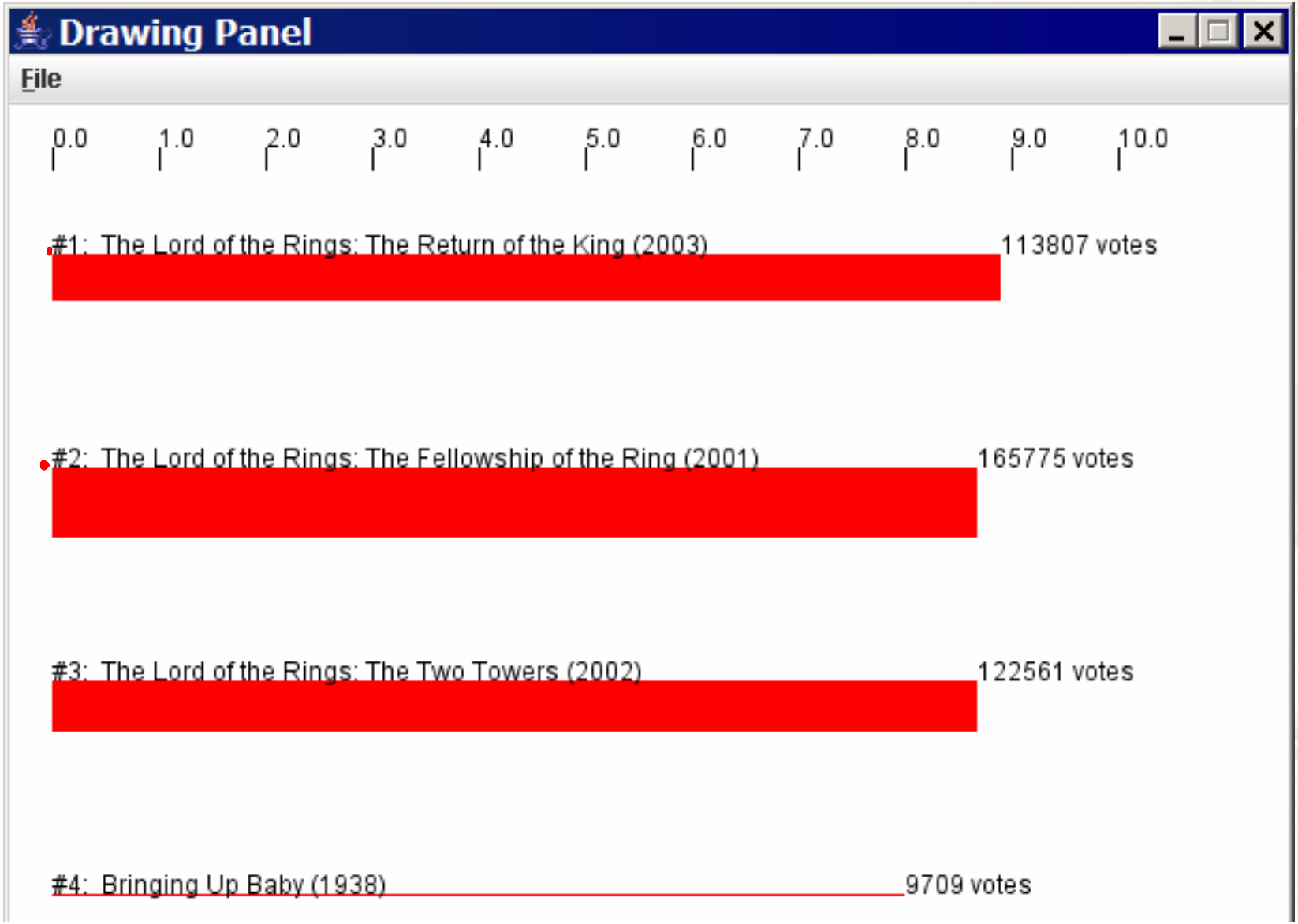
- Write a program that prompts the user for a search phrase and displays any movies that contain that phrase.

```
This program will allow you to search the
IMDB top 250 movies for a particular word.

search word? kill
Rank    Votes    Rating   Title
40      37815    8.5      To Kill a Mockingbird (1962)
88      89063    8.3      Kill Bill: Vol. 1 (2003)
112     64613    8.2      Kill Bill: Vol. 2 (2004)
128     9149     8.2      Killing, the (1956)
4 matches.
```

Graphical IMDB problem

- Consider making this a graphical program.



Mixing graphical, text output

- When solving complex file I/O problems with a mix of text and graphical output, attack the problem in pieces.

  Do the text input/output and file I/O first:

  - Handle any welcome message and initial console input.
  - Write code to open the input file and print some of the file's data. (Perhaps print the first token of each line, or print all tokens on a given line.)
  - Write code to process the input file and retrieve the record being searched for.
  - Produce the complete and exact text output.

  Next, begin the graphical output:

  - First draw any fixed items that do not depend on the user input or file results.
  - Lastly draw the graphical output that depends on the search record from the file.

More with lines and tokens

Mixing line-based with tokens

- It is not generally recommended to use `nextLine` in combination with the token-based methods, because confusing results occur.

```
23   3.14
Joe   "Hello world"
             45.2  19

int n = console.nextInt();                          // 23
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
  ^
double x = console.nextDouble();                    // 3.14
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
        ^

// receives an empty line!
String line = console.nextLine();                   // ""
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
          ^
// Calling nextLine again will get the following complete line
String line2 = console.nextLine();   // "Joe\t\"Hello world\""
23\t3.14\nJoe\t"Hello world"\n\t\t45.2  19\n
                              ^
```

Line-and-token example

- Here's another example of the confusing behavior:

```
Scanner console = new Scanner(System.in);
System.out.print("Enter your age: ");
int age = console.nextInt();
System.out.print("Now enter your name: ");
String name = console.nextLine();
System.out.println(name + " is " + age + " years old.");
```

Log of execution (user input underlined):

```
Enter your age: 12
Now enter your name: Marty Stepp
 is 12 years old.
```

- Why?
  - User's overall input: `12\nMarty Stepp`
  - After `nextInt()`: `12\nMarty Stepp`
    ```
      ^
    ```
  - After `nextLine()`: `12\nMarty Stepp`
    ```
        ^
    ```

Complex multi-line records

- Sometimes the data consists of multi-line records.
  - The following data represents students' courses.
  - Each student's record has the following format:
    - *Name*
    - *Credits Grade Credits Grade Credits Grade ...*

```
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?

File output using PrintStream

- suggested reading: 6.4

Output to files

- **`PrintStream`**: An object in the `java.io` package that lets you print output to a destination such as 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`.

- Printing into an output file, general syntax:

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

  - If the given file does not exist, it is created.
  - If the given file already exists, it is overwritten.

Printing to files, example

- Example:

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

  - You can use similar ideas about prompting for file names here.

- Do not open a file for 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!