Unit 4, Part 2

File Processing

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

A Class for Representing a File

- The `File` class in Java is used to represent a file on disk.
- To use it, we need to import the `java.io` package:

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

- Here's how we typically create a `File` object:

```
File f = new File("<filename>");
```

- Here are some useful methods from this class:

```
public boolean exists()
public boolean canRead()
public boolean canWrite()
public boolean delete()
public long length()
public String getName()
public String getPath()
```

See the Java API documentation for more info.

Review: Scanner Objects

- We've been using a Scanner object to read from the console:

```
Scanner console = new Scanner(System.in);
```

tells the constructor to construct a Scanner object that reads from the console

- Scanner methods:

```
next()
nextInt()
nextDouble()
nextLine()
```

Reading from a Text File

- We can also use a Scanner object to read from a text file:

```
File f = new File("<filename>");
Scanner input = new Scanner(f);
```

tells the constructor to construct a Scanner object that reads from the file

- We can combine the two lines above into a single line:

```
Scanner input = new Scanner(new File("<filename>"));
```

- We use a different name for the Scanner (input), to stress that we're reading from an input file.
- All of the same Scanner methods can be used.

Scanner Lookahead and Files

- When reading a file, we often don't know how big the file is.
- Solution: use an indefinite loop and a `Scanner` "lookahead" method.
- Basic structure:

```
Scanner input = new Scanner(new File(<filename>));
while (input.hasNextLine()) {
    String line = input.nextLine();
    // code to process the line goes here…
}
```

- `hasNextLine()` returns:
  - `true` if there's at least one more line of the file to be read
  - `false` if we've reached the end of the file

Sample Problem: Printing the Contents of a File

- Assume that we've already created a `Scanner` called `input` that is connected to a file.
- Here's the code for printing its contents:

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

File-Processing Exceptions

- Recall: An *exception* is an error that occurs at runtime as a result of some type of "exceptional" circumstance.
- We've seen several examples:
  `StringIndexOutOfBoundsException`
  `IllegalArgumentException`
  `TypeMismatchException`
- When using a `Scanner` to process a file, we can get a `FileNotFoundException`
  - if the file that we specify isn't there
  - if the file is inaccessible for some reason

Checked vs. Unchecked Exceptions

- Most of the exceptions we've seen thus far have been *unchecked* exceptions.
  - we do *not* need to handle them
  - instead, we usually take steps to avoid them
- `FileNotFoundException` is a *checked* exception. The compiler checks that we either:
  1) handle it
  2) declare that we don't handle it
- For now, we'll take option 2. We do this by adding a `throws` clause to the header of any method in which a `Scanner` for a file is created:

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

Sample Program: Counting the Lines in a File

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

public class CountLines {
    public static void main(String[] args)
      throws FileNotFoundException {
        Scanner input = new Scanner(new File("romeo.txt"));

        int count = 0;
        while (input.hasNextLine()) {
            input.nextLine();  // read line and throw away
            count++;
        }

        System.out.println("The file has " + count +
          " lines.");
    }
}
```

Counting Lines in a File, version 2

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

public class CountLines {
    public static void main(String[] args)
      throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        System.out.print("Name of file: ");
        String fileName = console.next();

        Scanner input = new Scanner(new File(fileName));

        int count = 0;
        while (input.hasNextLine()) {
            input.nextLine();  // read line and throw away
            count++;
        }

        System.out.println("The file has " + count +
          " lines.");
    }
}
```

Counting Lines in a File, version 3

```
…
public static void main(String[] args)
  throws FileNotFoundException {
     Scanner console = new Scanner(System.in);
     System.out.print("Name of file: ");
     String fileName = console.next();
     System.out.println("The file has " +
        numLines(fileName) + " lines.");
}

public static int numLines(String fileName)
  throws FileNotFoundException {
     Scanner input = new Scanner(new File(fileName));
     int count = 0;
     while (input.hasNextLine()) {
         input.nextLine();  // read line and throw away
         count++;
     }
     return count;
}
```

- We put the counting code in a separate method (`numLines`).
- Both `numLines` and `main` need a `throws` clause.

Extracting Data from a File

- Collections of data are often stored in a text file.
- Example: the results of a track meet might be summarized in a text file that looks like this:

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Fran Flash,BU,800m,2:15:00
Tammy Turtle,UMass,800m,4:00:00
```

- Each line of the file represents a *record*.
- Each record is made up of multiple *fields*.
- In this case, the fields are separated by commas.
  - known as a CSV file – comma separated values
  - the commas serve as *delimiters*
  - could also use spaces or tabs (`'\t'`) instead of commas

Extracting Data from a File (cont.)

```
Mike Mercury,BU,mile,4:50:00
Steve Slug,BC,mile,7:30:00
Fran Flash,BU,800m,2:15:00
Tammy Turtle,UMass,800m,4:00:00
```

- We want a program that:
  - reads in a results file like the one above
  - extracts and prints only the results for a particular school
    - with the name of the school omitted
- Basic approach:
  - ask the user for the school of interest (the *target school)*
  - read one line at a time from the file
  - split the line into fields
  - if the field corresponding to the school name matches the target school, print out the other fields in that record

Splitting a String

- The `String` class includes a method named `split()`.
  - breaks a string into component strings
  - takes a parameter indicating what delimiter should be used when performing the split
  - returns a `String` array containing the components
- Example:

```
> String sentence = "How now brown cow?";
> String[] words = sentence.split(" ");
> words[0]
"How"
> words[1]
"now"
> words[3]
"cow?"
> words.length
4
```

Extracting Data from a File (cont.)

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

public class ExtractResults {
  public static void main(String[] args)
    throws FileNotFoundException {
      Scanner console = new Scanner(System.in);

      System.out.print("School to extract: ");
      String targetSchool = console.nextLine();

      Scanner input = new Scanner(new File("results.txt"));
      while (input.hasNextLine()) {
          String record = input.nextLine();
          String[] fields = record.split(",");

          if (fields[1].equals(targetSchool)) {
              System.out.print(fields[0] + ",");
              System.out.println(fields[2] + "," + fields[3]);
          }
      }
  }
}
```

- How can we modify it to print a message when no results are found for the target school?

Example Problem: Averaging Enrollments

- Let's say that we have a file showing how course enrollments have changed over time:

```
cs111 90 100 120 115 140 170 130 135 125
cs105 14 8
cs108 40 35 30 42 38 26
cs101 180 200 175 190 200 230 160 154 120
```

- For each course, we want to compute the average enrollment.
  - different courses have different numbers of values
- Initial pseudocode:

```
while (there is another course in the file) {
    read the line corresponding to the course
    split it into an array of fields
    average the fields for the enrollments
    print the course name and average enrollment
}
```

Example Problem: Averaging Enrollments (cont.)

```
cs108 40 35 30 42 38 26
cs111 90 100 120 115 140 170 130 135 125
cs105 14 8
cs101 180 200 175 190 200 230 160 154 120
```

- When we split a line into fields, we get an array of strings.
  - example for the first line above:

```
{"cs108", "40", "35", "30", "42", "38", "26"}
```

- We can convert the enrollments from strings to integers using a method called `Integer.parseInt()`
  - example:

```
String[] fields = record.split(" ");
String courseName = fields[0];
int firstEnrollment = Integer.parseInt(fields[1]);
```

  - note: `parseInt()` is a static method, so we call it using its class name (`Integer`)

Example Problem: Averaging Enrollments (cont.)

Other Details About Reading Text Files

- Although we think of a text file as being two-dimensional (like a piece of paper), the computer treats it as a one-dimensional string of characters.
  - example: the file containing these lines

```
Hello, world.
How are you?
I'm tired.
```

    is represented like this:

```
Hello, world.\nHow are you?\nI'm tired.\n
```

- When reading a file using a `Scanner`, you are limited to *sequential* accesses in the forward direction.
  - you can't back up
  - you can't jump to an arbitrary location
  - to go back to the beginning of the file, you need to create a new `Scanner` object.

Optional Extra Topic: Writing to a Text File

- To write to a text file, we can use a `PrintStream` object, which has the same methods that we've used with `System.out`:
  - `print()`, `println()`
- Actually, `System.out` *is* a `PrintStream` that has been constructed to print to the console.
- To instantiate a `PrintStream` for a file:

```
File f = new File("<filename>");
PrintStream output = new PrintStream(f);
```

- We can also combine these two steps:

```
PrintStream output = new PrintStream(
  new File("<filename>"));
```

- If there's an existing file with the same name, it will be overwritten.

Copying a Text File

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

public class CopyFile {
    public static void main(String[] args)
      throws FileNotFoundException {
        Scanner console = new Scanner(System.in);
        System.out.print("Name of original file: ");
        String original = console.next();
        System.out.print("Name of copy: ");
        String copy = console.next();

        Scanner input = new Scanner(new File(original));
        PrintStream output = new PrintStream(new File(copy));

        while (input.hasNextLine()) {
            String line = input.nextLine();
            output.println(line);
        }
    }
}
```

- How could we combine the two lines in the body of the `while` loop?

Our Track-Meet Program Revisited

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

public class ExtractResults {
  public static void main(String[] args)
    throws FileNotFoundException {
      Scanner console = new Scanner(System.in);

      System.out.print("School to extract: ");
      String targetSchool = console.nextLine();

      Scanner input = new Scanner(new File("results.txt"));
      while (input.hasNextLine()) {
          String record = input.nextLine();
          String[] fields = record.split(",");

          if (fields[1].equals(targetSchool)) {
              System.out.print(fields[0] + ",");
              System.out.println(fields[2] + "," + fields[3]);
          }
      }
  }
}
```

- How can we modify it to print the extracted results to a separate file?

Optional Extra Topic: Binary Files

- Not all files are text files.
- *Binary files* don't store the string representation of non-string values.
  - instead, they store their *binary* representation – the way they are stored in memory
- Example: 125
  - the text representation of 125 stores the string "125" – i.e., the characters for the individual digits in the number

  - the binary representation of 125 stores the four-byte binary representation of the integer 125