Week 9 - Wednesday

COMP 2000

Last time

- What did we talk about last time?
- *N*-Queens
- Reading text files

Questions?

Project 3

Files

Reading

- Reading from a text file is straightforward
- We use **`Scanner`**, just like reading from the command line
- We just have to create a new **`File`** object that gives the file path we want to read from

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

- This code will read from some file called **`input.txt`**, as if someone were typing its contents into the command line

Scanner methods

- Recall that we can read correctly formatted text with a **`Scanner`** using the following methods
  - **`nextInt()`** Reads an **`int`** value
  - **`nextDouble()`** Reads a **`double`** value
  - **`next()`** Reads a white-space delimited **`String`**
  - **`nextLine()`** Reads a **`String`** up to the next newline (which can cause problems if there's a newline left over from previous reads)
- These methods are usually what you need to get the job done, but there are also **`nextBoolean()`**, **`nextByte()`**, **`nextFloat()`**, **`nextLong()`**, and **`nextShort()`** methods
- Note that all the integer reading methods have a second version that takes a base so that you can read values in bases 2-36

New `Scanner` powers

- When a **`Scanner`** is reading from the keyboard, it has no idea what the user will type next
- However, when a **`Scanner`** is reading from a file, it *can* examine the text it hasn't read yet
- A number of methods are available to tell you if there's some properly formatted data just ahead waiting to be read:
  - **`hasNextInt()`** — There's an **`int`** waiting to be read
  - **`hasNextDouble()`** — There's a **`double`** waiting to be read
  - **`hasNext()`** — There's a **`String`** waiting to be read
  - **`hasNextLine()`** — There's a line waiting to be read
- Such methods are often used in a **`while`** loop to keep reading data until the end of the file is reached

Back to opening the file…

- What if the file with name **`fileName`** doesn't exist?

```
Scanner in = new Scanner(new File(fileName));
```

- Creating a **`Scanner`** from a **`File`** object will throw a **`FileNotFoundException`** if the file doesn't exist or isn't accessible
- As you might recall, a **`FileNotFoundException`** is a **checked** exception
- You will need to surround the **`Scanner`** constructor in a **`try`** with a **`catch`** that handles this exception or you can add a **`throws`** declaration to your method
- You will often **want** to have a **`try-catch`** so that you can recover from the missing file and ask the user for a new file name (or similar)

File paths

- The file doesn't have to be a simple name like **`input.txt`**
- Instead, it can be a path, giving directories that lead up to the file
- Paths come in two kinds
- A **relative path** is relative to the current working directory, which is the project folder for Eclipse projects:
  - **`data\experiments\mutagens\ninja-turtles.txt`**
- An **absolute path** specifies the whole path, starting with a drive letter in Windows or a **`/`** in Linux/macOS:
  - **`C:\users\wittman1\Documents\Music\Beastie Boys\intergalactic.mp3`**

Path peculiarities

- In paths, `.` means the current directory and `..` means the parent directory
  - **`goats\.\boats\..\`** Is the same as **`goats\`**
- To separate directories (also called folders), Windows uses the backslash (**`\`**) and Linux/macOS uses the forward slash (**`/`**)
- To make it easier to write platform-independent code, Java generally accepts either one
- You will sometimes store path names inside String literals, requiring you to escape backslashes with another backslash:
- **`String path =`**
  `"C:\\users\\wittman1\\Documents\\Music\\Beastie Boys\\intergalactic.mp3";`
- It's easier to use all forward slashes so that you don't have to escape them

Badly formatted data

- What if you open a file thinking it's full of integers, but the numbers have decimal points…or are text, not numbers?
- One of the following exceptions can be thrown:
  - **`InputMismatchException`** The input isn't formatted right
  - **`NoSuchElementException`** You've reached the end of the file
  - **`IllegalStateException`** You're trying to read from a closed **`Scanner`**
- All of these exceptions are **unchecked** exceptions
- You don't have to have a **`try-catch`** or a **`throws`** declaration
- And you generally won't, unless you expect the input to be badly formatted?

Writing

- Writing to files uses a different sequence of steps
- If you want to write to a text file, you've got to create a **`PrintWriter`** object, based on a **`FileOutputStream`** object (which takes the file name as a parameter)

```
PrintWriter out = new PrintWriter(new
  FileOutputStream("output.txt"));
```

- Once you've got a **`PrintWriter`**, you can use it just like **`System.out`**

More exceptions!

- Just like making a **`Scanner`** from a **`File`**, making a **`PrintWriter`** from a **`FileOutputStream`** can potentially throw a **`FileNotFoundException`**
- Weird, isn't it? But Java throws this exception when you're unable to open the file for writing, for whatever reason
- Since it's a checked exception, you need a **`try-catch`** or a **`throws`**

Writing example

- This example opens a file called **goodbye.txt**, writes some text, and then closes the file
- Note that we are not showing the **try-catch** or **throws**

```
PrintWriter out = new PrintWriter(new
  FileOutputStream("goodbye.txt"));
out.println("So long!");
out.println("Farewell!");
out.println("Auf Wiedersehen!");
out.println("Goodbye!");
out.close();
```

Shut 'em down!

- You should always close files as soon as you're done reading from them or writing to them
- If you don't close files you're writing to before your program ends, output can be lost
- Keeping files open ties up system resources
- There's a maximum number of files one program can have open at a time
- Since we always want to close files, it's smart to put the closing in a **`finally`** block

Full example

- This example copies the text from **`input.txt`** to **`output.txt`**

```
Scanner in = null;
PrintWriter out = null;
try {
  in = new Scanner(new File("input.txt"));
  out = new PrintWriter(new FileOutputStream("output.txt"));
  while(in.hasNextLine())
     out.println(in.nextLine());
}
catch(FileNotFoundException e) {
  e.printStackTrace();
}
finally {
  if(in != null) in.close();
  if(out != null) out.close();
}
```

File I/O practice

- Prompt the user for an input file name and an output file name
- Read in every white-space delimited word from the input file
- If it's a palindrome, output it to the output file
- Close both files

More practice

- Prompt the user for an input file name
- If the file isn't accessible, prompt the user to enter the name again
- Read in all the integers in this file (until you run out)
- Close the file
- Store all the values in an **`ArrayList<Integer>`**
- Sort them
- Find the mode (the value that appears the most)

Upcoming

Next time…

- Reading and writing binary files
- Serialization

Reminders

- **Work on Project 3**
  - **Form teams if you haven't**
- Keep reading Chapter 20