CS 5004: OBJECT ORIENTED DESIGN AND ANALYSIS

SPRING 2022

LECTURE 8

Divya Chaudhary

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Collaborating with GitHub
- Review
  - Parametric polymorphism
- Interactive programs in Java
- Processing command-line arguments
- Java input/output (I/O)
- Regular expressions in Java

COLLABORATING WITH GITHUB

CS 5004, SPRING 2022 – LECTURE 8

NOW YOU'RE WORKING IN GROUPS…

Good git hygiene is vital!

Why? Merge conflicts ☹

AVOIDING MERGE CONFLICTS

Golden rule #1: Do not touch code/files that someone else is likely to edit

If a teammate is working on SomeClass.java, DO NOT

- edit SomeClass.java
- move SomeClass.java
- rename/change the package containing SomeClass.java

Discuss who is working on what before touching anything!

AVOIDING MERGE CONFLICTS

Golden rule #2: ALWAYS start work with the following commands

```
git status
<commit and push any outstanding local changes>
git pull
```

Even if you only left your computer for a short time and did not shut your computer/files e.g.

- Left the IDE open overnight
- Went to the store
- Made a sandwich

AVOIDING MERGE CONFLICTS

Golden rule #3: NEVER work directly on the MAIN branch

When beginning a new assignment, create a new "topic branch" just for you (instructions on Canvas)

- Do all your work on that branch
- Merge to master only when your topic branch is working without compile time errors and is thoroughly tested

IF A MERGE CONFLICT HAPPENS…

1) Don't panic

2) DO NOT use –f or --force

No matter how many Stack Overflow posts tell you to do it!

PRACTICE WITH BRANCHES

In your own repo, using the command line (not the GUI or IntelliJ):

`cd path/to/your/repo`

`git status` (commit and push any outstanding changes, if needed)

`git pull`

`git checkout –b branch_name`

Write some code, commit and push

→ you will need to set the remote branch on first push:

```
git push --set-upstream origin branch_name
```

REVIEW

CS 5004, SPRING 2022 – LECTURE 8

REVIEW: THREE TYPES OF POLYMORPHISM

- Polymorphism – the ability to define different classes and methods as having the same name but taking different data types

- Three types of polymorphism:
  - Subtype polymorphism
  - Ad hoc polymorphism
  - Parametric polymorphism

REVIEW: WHAT EXACTLY IS BEING POLYMORPHIC

Parametric polymorphism (generics):

- "Enables **data types** (classes and interfaces) to be **parameters** when defining classes and interfaces."
- Especially useful when writing classes that are collections of other objects (e.g. List, Set, Stack etc).
  - Write one class that can handle multiple types of objects.

Enables a function or class to be written such that it handles values identically regardless of type

REVIEW: TYPE PARAMETERS

```
List<Type> name = new ArrayList<Type>();
```

Type parameter specifies type of element stored in the collection

- Allows the same class to store different types of objects
- Also called a *generic* class

```
List<String> names = new ArrayList<String>();
List<Integer> digits = new ArrayList<Integer>();
```

REVIEW: WHAT CAN BE A TYPE PARAMETER?

Objects only

- Setting a primitive as a type parameter → compile time error e.g.

```
List<int> digits = new ArrayList<int>(); //won't compile
```

- Instead, use a wrapper class type:

Primitive	Wrapper
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

REVIEW: USING TYPE PARAMETERS - A SHORTCUT

Right side Type argument is unnecessary:

```
List<Type> name = new ArrayList<Type>();
```

Instead, use the diamond operator, **<>**:

```
List<Type> name = new ArrayList<>();
```

Compiler auto populates each type parameter from the types on the left side

```
List<String> names = new ArrayList<>();
```

REVIEW: IMPLEMENTING GENERICS

```
// a parameterized (generic) class
public class Name<Type> {…}
public class Name<Type, Type, ..., Type> {…}
interface Name<Type, Type, ..., Type> {…}
```

- By putting the Type in < >, we are demanding that any client that constructs our object must supply a type parameter
- We can require multiple type parameters separated by commas
- The convention is to use a 1-letter name:
  - T for Type
  - E for Element
  - N for Number
  - K for Key,
  - V for Value
- The type parameter is instantiated by the client (e.g. E → String)

REVIEW: WILDCARDS

- A wildcard is essentially an anonymous type variable
- Each ? stands for some possibly-different unknown type
- Use a wildcard when you would use a type variable exactly once, so no need to give it a name
- Avoids declaring generic type variables
- Communicates to readers of your code that the type's "identity" is not needed anywhere else

- For a type-parameter instantiation (inside the <...>), can write:
  - ? is shorthand for `? extends Object`
  - ? extends `Type`, some unspecified subtype of `Type`
  - `? super Type`, some unspecified supertype of `Type`

REVIEW: TYPE ERASURE

- All generics types become type `Object` once compiled
  - One reason: Backward compatibility with old byte code
  - At runtime, all generic instantiations have the same type

```
List<String>  lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass()  // true
```

- You cannot use `instanceof` to discover a type parameter

```
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) {
    // illegal
}
```

ACKNOWLEDGEMENT

Acknowledgement: the following lecture notes were inspired by course material prepared by UW faculty members Z. Fung and H. Perkins.

INTERACTIVE PROGRAMS IN JAVA

CS 5004, SPRING 2022 – LECTURE 8

INTERACTIVE PROGRAMS

- Interactive programs – programs where a user interacts with a program by providing an input into the console, that a program then reads and uses for execution

- Interactive programs can (sometimes) be challenging
  - Computers and users think in very different way
  - Users misbehave
  - Users are malicious

TWO WAYS TO INTERACT

(…without a GUI)

- **Passing command line arguments when the program is run**
  - Assignment 8
- Scanner
  - When further input is required *while* the program is running

REMINDER: APPLICATION ENTRY POINT

- Some class, often called **`Main`**
- Must have a **`main`** method
  - Signature must be exactly as follows…

```
public static void main(String[] args) {
  // business logic here
}
```

Command line arguments passed in here

`java MyApp hello 9000`

`args` will equal **`["hello", "9000"];`**

COMMAND LINE: *TYPICAL* TERMINOLOGY

- **command** – what the program should do, includes arguments
- **option** – a "named" argument modifying the command
- **parameter/argument** – a value associated with an option/command

COMMAND LINE: *TYPICAL* TERMINOLOGY

- **command** – what the program should do, includes arguments
- **option** – a "named" argument modifying the command
- **parameter/argument** – a value associated with an option/command

```
git commit -m "a message"
```

command

option

argument associated with option

WRITING A PROGRAM THAT ACCEPTS COMMAND LINE INPUT

General rules:

- Define options to allow a user to control how the program runs
- Decide if options need parameters/arguments
- Allow users to enter the options in any order
- EXCEPT if an option requires a parameter/argument, it must follow the option

WHAT GOES IN MAIN()?

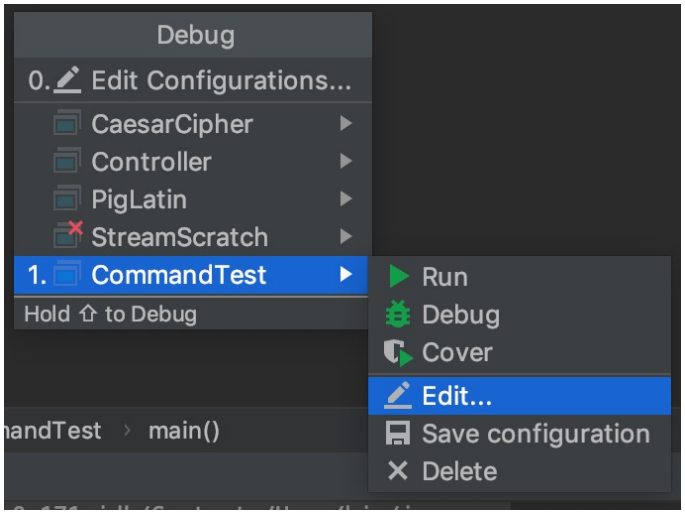
Very little!

- Send `args` elsewhere for processing
  - **`main()`** ***should not do the processing itself***
- Initiate any other business logic
  - e.g., call other classes to do something with the user input
- Print output to the user

INTELLIJ: SAVE SOME ARGS FOR DEBUGGING

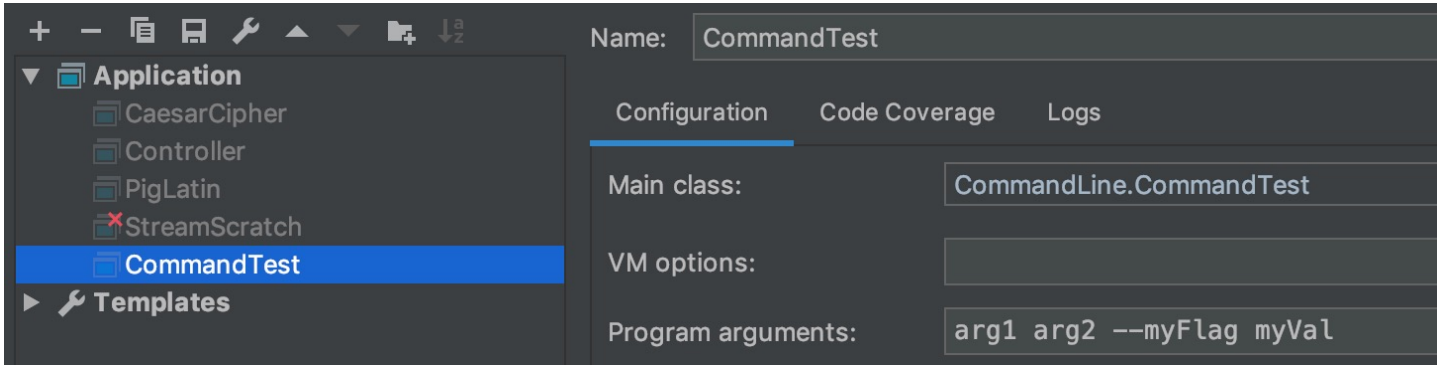
...these will be passed when you run the file in the IDE:

1. Select the file to pass arguments to (must have a main method)
2. Go to: Run > Run...
3. In the popup menu, select your file and click Edit...



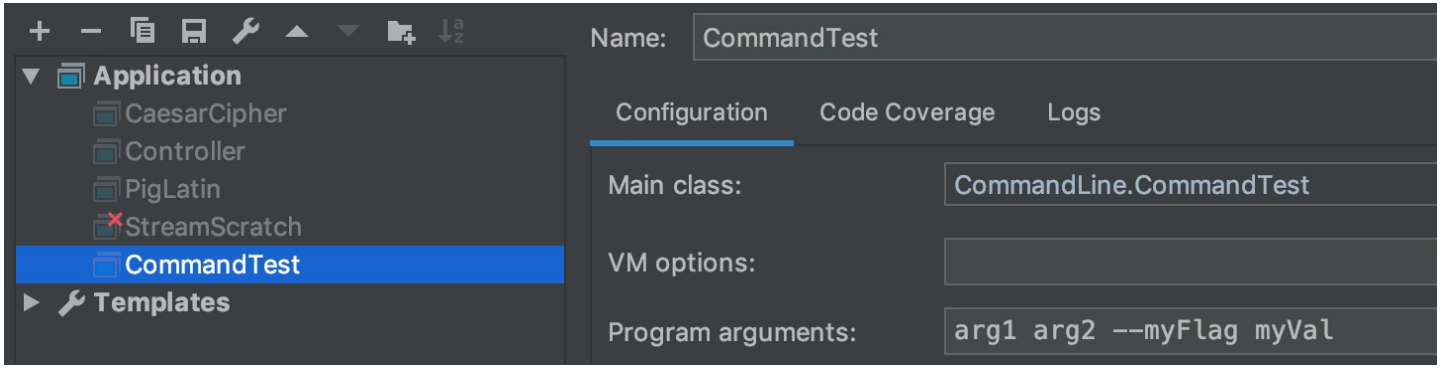
INTELLIJ: SAVE SOME ARGS FOR DEBUGGING

In the next popup, enter the command line args in the Program arguments field then click OK:



INTELLIJ: SAVE SOME ARGS FOR DEBUGGING

Next time you run in IntelliJ, the specified arguments will be passed to `main`



EXAMPLE: COOKIE COUNTER

A program that calculates the amount of calories & sugar consumed based on the number of cookies the user enters

- Options: cookie names
  - e.g. “--samosas”
- Required argument: number eaten
  - Must follow cookie name

ALTERNATIVE: `SCANNER`

Java typically handles user input using `System.in` but:

- `System.in` is not intended to be used directly
- Instead, we us a second object, `Scanner`, from package `java.util`

SCANNER METHODS

Method	Description
`nextInt()`	Reads a token of user input as an `int`
`nextDouble()`	Reads a token of user input as a `double`
`next()`	Reads a token of user input as a `String`
`nextLine()`	Reads a line of user input as a `String`

[Table credit: Paerson Education]

- Each method waits until the user presses *Enter*
  - The value typed is returned

CONSTRUCTING A SCANNER OBJECT

```
import java.util.*;
Scanner name = new Scanner(System.in);
```

Note: Only “name” should be changed

EXAMPLE SCANNER USAGE

```
import java.util.*;

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

      System.out.print("How many courses are you taking this term? ");
      int numCourses = console.nextInt();

      System.out.println(numCourses + "... That's too many!");
   }
}
```

Output (user input underlined):

How many courses are you taking this term? 4
4... That's too many!

SUMMARY OF DIFFERENCES

Command line

- Array of Strings passed to main() when the program is run
- Can pass multiple arguments in one go
- Use when input is only required when the program first runs
  - Assignment 8!

`Scanner`

- Different data types can be entered when prompted
- Stops "scanning" when a space is reached
- Use to get user input while program is still running

PROCESSING COMMAND-LINE INPUTS

CS 5004, SPRING 2022 – LECTURE 8

ACCESSING COMMAND LINE ARGUMENTS

- Question: How to pass information into a program before we run it?
- Answer: Use command–line arguments to `main()` method
- A command-line argument (input) is the information that directly follows the program name on the command line when we call the program to execute it
- Command-line arguments are stored as `String`s in the String array `args` passed to `main()`
- All command-line arguments are passed as `String`s → you need to convert them into a valid format

JAVA INPUT/OUTPUT

CS 5004, SPRING 2022 – LECTURE 8

KEY CONCEPT: INPUT/OUTPUT TAKES TIME & MEMORY

Data doesn't arrive / leave all at once

In Java:

IO – *stream* oriented

NIO – *buffer* oriented

JAVA IO

Stream oriented

- IO = “input/output”
- I/O stream – a communication channel (“pipe”) between a **source** and a **destination** that allows us to create a flow of **data**
- A stream is a sequence of data.
- Data is read 1+ bytes at a time directly from the stream

I/O STREAM

- Input **source**: e.g. a file, another program, device
- Output **destination**: e.g. a file, another program, device
- The kind of **data** streamed: e.g. bytes, ints, objects

INPUTSTREAMS AND OUTPUTSTREAMS

InputStreams

- FileInputStream
- AudioInputStream
- SequenceInputStream
- Etc
- Can create own

OutputStreams

- FileOutputStream
- PipedOutputStream
- Etc
- Can create own

INPUTSTREAMS

- FileReader – reads text files as characters
- FileInputStream – reads data as bytes e.g. image data
- AudioInputStream – input stream in a specified format
- SequenceInputStream – combines two or more InputStreams into one

JAVA NIO

Buffer oriented

- NIO = “new input/output”
- A buffer is an object which holds some data
- Data is read into and written from the buffer
- Client code interacts with the buffer

FILE PROCESSING USING JAVA IO

BASIC FILE I/O STEPS

1. Create variables for input and output streams
2. Try to read/write a file line by line
3. Catch exceptions
4. Finally, close streams and clean up

BASIC FILE I/O EXAMPLE

```
public static void main(String[] args) {
  FileReader inputFile = null;

  System.out.println(System.getProperty("user.dir"));

  try {
    inputFile = new FileReader( fileName: "country_code.csv");

    int character;
    while ((character = inputFile.read()) != -1) {
      System.out.println("Read : " + character);
    }
  } catch (FileNotFoundException fnfe) {
    System.out.println("*** OOPS! A file was not found : " + fnfe.getMessage());
  } catch (IOException ioe) {
    System.out.println("Something went wrong! : " + ioe.getMessage());
  } finally {
    if (inputFile != null) {
      try {
        inputFile.close();
      } catch (IOException e) {
        System.out.println("Failed to close input stream");
      }
    }
  }
}
```

BASIC FILE I/O EXAMPLE - FILEREADER

1. **Create a variable for the InputStream**

```
FileReader inputFile = null;
```

Debugging tip: see the directory the JVM is looking in

```
System.getProperty("user.dir")
```

BASIC FILE I/O EXAMPLE - FILEREADER

2. Try to read the file line by line – wrap it in a try block

```
try {
 inputFile = new FileReader("filename.csv");
}
```

Instantiate the FileReader and pass it the name of the file to open

BASIC FILE I/O EXAMPLE - FILEREADER

2. Try to read the file line by line – wrap it in a try block

```
try {
 inputFile = new FileReader("filename.csv");
 int character;
 while((character = inputFile.read() != -1) {
   System.out.println("Read: " + character);
 }
}
```

BASIC FILE I/O EXAMPLE - FILEREADER

2. Try to read the file line by line – wrap it in a try block

```
try {
 inputFile = new FileReader("filename.csv");
 int character;
 while((character = inputFile.read() != -1) {
   System.out.println("Read: " + character);
 }
}
```

…store each character read by the stream

BASIC FILE I/O EXAMPLE - FILEREADER

2. Try to read the file line by line – wrap it in a try block

```
try {
 inputFile = new FileReader("filename.csv");
 int character;
 while((character=inputFile.read()) != -1) {
   System.out.println("Read: " + character);
 }
}
```

As long as there are characters to read, read the next character

BASIC FILE I/O EXAMPLE - FILEREADER

3. Catch exceptions

```
catch (FileNotFoundException fnfe) {
 // What to do if the file doesn't exist
}
```

BASIC FILE I/O EXAMPLE - FILEREADER

3. Catch exceptions

```
catch (FileNotFoundException fnfe) {
 // What to do if the file doesn't exist
}
catch (IOException ioe) {
 // What to do if something else goes wrong…
}
```

BASIC FILE I/O EXAMPLE - FILEREADER

4. Close streams and clean up

```
finally {
 try {
   inputFile.close();
  }
 catch (IOException e) {
   // What to do if something else goes wrong…
  }
}
```

Always close your streams!

- Even if something went wrong

BUFFEREDREADER

Reads text from a character-input stream, **buffering characters** to provide efficient reading of characters, arrays, and lines.

- Increases efficiency of read operations
- Example > MainBasicBuffered.java

CHANGES FOR BUFFEREDREADER

1. **Create a variable for the InputStream**

```
BufferedReader inputFile = null;
```

CHANGES FOR BUFFEREDREADER

2. Try to read the file line by line – wrap it in a try block

```
try {
 inputFile = new BufferedReader(
      new FileReader("filename.csv"));
}
```

BufferedReader takes a FileReader as a parameter

CHANGES FOR BUFFEREDREADER

2. Try to read the file line by line – wrap it in a try block

```
try {
 inputFile = new BufferedReader(
      new FileReader("filename.csv"));
 String line;
 while((line=inputFile.readLine()) != null) {
   System.out.println("Read: " + line);
 }
}
```

Now we can read by line

APPROACH SO FAR

Initialize reader as null outside `try-catch-finally`

... so it can be closed in the `finally` block

```
BufferedReader reader = null;
try {
  // instantiate reader object
  // read file
} catch (FileNotFoundException…
catch (IOException…
finally {
 // close reader if not null
}
```

“TRY WITH RESOURCES”

Automatically closes the reader (or writer)

- No need to set reader to null outside try-catch-finally
- No need for a finally block to close the reader

"TRY WITH RESOURCES"

```
try (// instantiate stream objects in parentheses;
   // can have multiple lines ) {
  //processing in here
}
catch…
```

“TRY WITH RESOURCES”

```
try (BufferedReader reader =
   new BufferedReader(new FileReader(“filename”))) {
  //processing in here
}
catch…
```

WRITING FILES

- Instead of `Buffered`**`Reader`** and `File`**`Reader`** → `Buffered`**`Writer`** with `File`**`Writer`**
- Instead of `reader.readLine()` → `writer.write("contents")`
- Still need to catch same exceptions
- Still need to close stream in finally block unless using try with resources

READING & WRITING IN THE SAME TRY…

```
try ( BufferedReader reader =
   new BufferedReader(new FileReader("filename"));
   BufferedWriter writer =
   new BufferedWriter(new FileWriter("filename"))) {
  String line = "";
  while ((line = reader.readLine()) != null) {
    writer.write(line);
  }
}
catch…
```

FILE PROCESSING USING JAVA NIO

Just scratching the surface…

YET ANOTHER APPROACH: FILES & PATH

Read / write all lines in a file at once

```
try {
  Path in = Paths.get("somefile.csv");
  Path out = Paths.get("somefile_out.csv");
  List<String> lines = Files.readAllLines(in);
  Files.write(out, lines);
} catch (NoSuchFileException nsf) {
  // handle the exception
} catch (IOException ioe) {
  // handle the exception
}
```

YET ANOTHER APPROACH: FILES & PATH

Read / write all lines in a file at once

```
try {
  Path in = Paths.get("somefile.csv");
  Path out = Paths.get("somefile_out.csv");
  List<String> lines = Files.readAllLines(in);
  Files.write(out, lines);
} catch (NoSuchFileException nsf) {
  // handle the exception
} catch (IOException ioe) {
  // handle the exception
}
```

Still need try-catch

YET ANOTHER APPROACH: `FILES` & `PATH`

Read / write all lines in a file at once

```
try {
  Path in = Paths.get("somefile.csv");
  Path out = Paths.get("somefile_out.csv");
  List<String> lines = Files.readAllLines(in);
  Files.write(out, lines);
} catch (NoSuchFileException nsf) {
  // handle the exception
} catch (IOException ioe) {
  // handle the exception
}
```

Create a `Path` object for each file to read/write

- `Paths.get()` throws `NoSuchFileException`

YET ANOTHER APPROACH: `FILES` & `PATH`

Read / write all lines in a file at once

```
try {
  Path in = Paths.get("somefile.csv");
  Path out = Paths.get("somefile_out.csv");
  List<String> lines = Files.readAllLines(in);
  Files.write(out, lines);
} catch (NoSuchFileException nsf) {
  // handle the exception
} catch (IOException ioe) {
  // handle the exception
}
```

Read all lines at once to a `List<String>`

- Not a good idea for very large files

YET ANOTHER APPROACH: `FILES` & `PATH`

Read / write all lines in a file at once

```
try {
  Path in = Paths.get("somefile.csv");
  Path out = Paths.get("somefile_out.csv");
  List<String> lines = Files.readAllLines(in);
  Files.write(out, lines);
} catch (NoSuchFileException nsf) {
  // handle the exception
} catch (IOException ioe) {
  // handle the exception
}
```

Write all lines using `List<String>`

- Not a good idea for very large files

YET ANOTHER APPROACH: FILES & PATH

Read / write all lines in a file at once

```
try {
  Path in = Paths.get("somefile.csv");
  Path out = Paths.get("somefile_out.csv");
  List<String> lines = Files.readAllLines(in);
  Files.write(out, lines);
} catch (NoSuchFileException nsf) {
  // handle the exception
} catch (IOException ioe) {
  // handle the exception
}
```

Resources automatically cleaned up

WHICH APPROACH TO USE?

Claim: Java NIO is faster than traditional IO

Experiment:

- See Code_from_lectures > timing > Main
- Compares:
  - Basic I/O using FileReader only
  - IO using BufferedReader + FileReader
  - NIO
- Read the code then run Main

WHICH APPROACH TO USE?

Claim: Java NIO is faster than traditional IO

→ under certain circumstances which we won't encounter in this course

In this course:

- Try-with-resources
- BufferedReader/BufferedWriter

REGULAR EXPRESSIONS

CS 5004, SPRING 2022 – LECTURE 8

SO WE HAVE INPUT…WHAT DO WE DO WITH IT?

In country_code.csv, each line has the format:

<country_name>, <country_code>

What if we want to do something with each piece?

SO, WE HAVE INPUT…WHAT DO WE DO WITH IT?

In country_code.csv, each line has the format:

<country_name>, <country_code>

What if we want to do something with each piece?

```
String[] parts = line.split( regex: ",");
```

SO, WE HAVE INPUT…WHAT DO WE DO WITH IT?

```
String[] parts = line.split(regex: ",");
```

…splits a whole line into parts at the comma.

What if we want to do something more complicated?

- E.g., remove all numeric characters

REGULAR EXPRESSIONS

- Abbreviated as “re” or “RE”
- Allows searching / matching by *pattern* rather than a literal String
- A sequence of characters used to describe some set of strings (pattern)
  - E.g., RE “,” used inside a document to search for the character “,”
- We can also use **operators** to express more complex patterns
  - e.g., All ”words” made up of exactly 5 numbers

REGULAR EXPRESSION BASIC OPERATORS

*	Means 0 or more of the preceding RE e.g. “a*” matches 0 or more “a” characters in the sequence
|	Matches if either the left or the right RE match e.g. “A|B” matches an “A” or a “B”
.	Matches any single character, e.g. “a.b” matches and string of 3 where the first is “a” and the third is “b”
[ ]	Matches a single character found within the square brackets e.g. “[abcd]” matches “a” or “b” or “c” or “d”
[^]	Match any character that is NOT in the set e.g ”[^abcd]” matches any character that is not ”a” or “b” etc.

REGULAR EXPRESSION BASICS: EXAMPLE

*	Means 0 or more of the preceding RE e.g. “a*” matches 0 or more “a” characters in the sequence
|	Matches if either the left or the right RE match e.g. “A|B” matches an “A” or a “B”
.	Matches any single character, e.g. “a.b” matches and string of 3 where the first is “a” and the third is “b”
[ ]	Matches a single character found within the square brackets e.g. “[abcd]” matches “a” or “b” or “c” or “d”
[^]	Match any character that is NOT in the set e.g ”[^abcd]” matches any character that is not ”a” or “b” etc.

What does this match?

`"(a|b)b*"`

REGULAR EXPRESSION BASICS: EXAMPLE

*	Means 0 or more of the preceding RE e.g. “a*” matches 0 or more “a” characters in the sequence
|	Matches if either the left or the right RE match e.g. “A|B” matches an “A” or a “B”
.	Matches any single character, e.g. “a.b” matches and string of 3 where the first is “a” and the third is “b”
[ ]	Matches a single character found within the square brackets e.g. “[abcd]” matches “a” or “b” or “c” or “d”
[^]	Match any character that is NOT in the set e.g ”[^abcd]” matches any character that is not ”a” or “b” etc.

What does this match?

" (a|b)b* "

An “a” or “b”

REGULAR EXPRESSION BASICS: EXAMPLE

*	Means 0 or more of the preceding RE e.g. “a*” matches 0 or more “a” characters in the sequence
|	Matches if either the left or the right RE match e.g. “A|B” matches an “A” or a “B”
.	Matches any single character, e.g. “a.b” matches and string of 3 where the first is “a” and the third is “b”
[ ]	Matches a single character found within the square brackets e.g. “[abcd]” matches “a” or “b” or “c” or “d”
[^]	Match any character that is NOT in the set e.g ”[^abcd]” matches any character that is not ”a” or “b” etc.

What does this match?

"(a|b)b*"

An “a” or “b” followed by 0 or more “b”

REGEX IN JAVA WITH PATTERN & MATCHER

```
Pattern re1 = Pattern.compile("(a|b)b*");
re1.matcher(<someString>).matches()
```
→ true or false

REGEX IN JAVA WITH PATTERN & MATCHER

```
Pattern re1 = Pattern.compile("(a|b)b*");
re1.matcher(<someString>).matches();
```

`Matcher` invoked from "compiled" pattern

→ `re1.matcher(<someString>)` returns an instance of `Matcher`

REGEX IN JAVA WITH PATTERN & MATCHER

```
Pattern re1 = Pattern.compile("(a|b)b*");
re1.matcher(<someString>).matches();
```

- **`Matcher`** invoked from compiled pattern. Three operations:
  - **`matches()`** – does the *entire* input string match the pattern exactly?

REGEX IN JAVA WITH PATTERN & MATCHER

```
Pattern re1 = Pattern.compile("(a|b)b*");
re1.matcher(<someString>).lookingAt();
```

- **`Matcher`** invoked from compiled pattern. Three operations:
  - **`matches()`** – does the *entire* input string match the pattern exactly?
  - **`lookingAt()`** – does the pattern occur at the *start* of the input string?

REGEX IN JAVA WITH PATTERN & MATCHER

```
Pattern re1 = Pattern.compile("(a|b)b*");
re1.matcher(<someString>).find();
```

- **`Matcher`** invoked from compiled pattern. Three operations:
  - **`matches()`** – does the *entire* input string match the pattern exactly?
  - **`lookingAt()`** – does the pattern occur at the *start* of the input string?
  - **`find()`** – does the pattern occur *anywhere* in the input string?

ANOTHER EXAMPLE

```
String test = "aAaAaA";
Pattern a
    = Pattern.compile("a", Pattern.CASE_INSENSITIVE);
Matcher m = a.matcher(test);
```

`Pattern` allows use of certain flags

ANOTHER EXAMPLE

```
String test = “aAaAaA”;
Pattern a
    = Pattern.compile(”a”, Pattern.CASE_INSENSITIVE);
Matcher m = a.matcher(test);
m.matches()
```

← **What does this return? true / false?**

ANOTHER EXAMPLE

```
String test = "aAaAaA";
Pattern a
    = Pattern.compile("a", Pattern.CASE_INSENSITIVE);
Matcher m = a.matcher(test);
m.matches()
```

→ **FALSE**

The pattern doesn't match the entire string

ANOTHER EXAMPLE

```
String test = “aAaAaA”;
Pattern a
    = Pattern.compile(”a”, Pattern.CASE_INSENSITIVE);
Matcher m = a.matcher(test);
m.matches() // false
m.lookingAt()
```

← What does this return? true / false?

ANOTHER EXAMPLE

```
String test = "aAaAaA";
Pattern a
    = Pattern.compile("a", Pattern.CASE_INSENSITIVE);
Matcher m = a.matcher(test);
m.matches() // false
m.lookingAt() → TRUE
```

The beginning of the String matches the pattern

ANOTHER EXAMPLE

```
String test = “aAaAaA”;
Pattern a
    = Pattern.compile(”a”, Pattern.CASE_INSENSITIVE);
Matcher m = a.matcher(test);
m.matches() // false
m.lookingAt() // true
m.find()
```

← What does this return? true / false?

ANOTHER EXAMPLE

```
String test = "aAaAaA";
Pattern a
    = Pattern.compile("a", Pattern.CASE_INSENSITIVE);
Matcher m = a.matcher(test);
m.matches() // false
m.lookingAt() // true
m.find() → TRUE
```

There is at least one instance of the pattern in the String

ANOTHER EXAMPLE

Could also use `m.find()` to iterate through every occurrence of the pattern

```
while (m.find()) {
  System.out.println(
   test.substring(m.start(), m.end));
}
```

FIND AND REPLACE WITH REGEX

RegEx doesn't always have to be compiled as a `Pattern`

Common String methods using RegEx

- `String.split(<regex>)` - Splits a String into an array at every occurrence of `<regex>`
- `String.replaceAll(<regex>, replace_with)` - Replaces the first occurrence of `<regex>` with `replace_with`

GETTING FANCY: MORE REGEX METACHARACTERS

Regex	Description
X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

[Table credit: https://www.javatpoint.com/java-regex]

GETTING FANCY: MORE REGEX METACHARACTERS

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

[Table credit: https://www.javatpoint.com/java-regex]

REGEX CHALLENGE: MATCH WORDS BEGINNING WITH A VOWEL

Vowels: a,e,i,o,u,A,E,I,O,U

Useful metacharacters:

*	Means 0 or more of the preceding RE e.g. “a*” matches 0 or more “a” characters in the sequence
[ ]	Matches a single character found within the square brackets e.g. “[abcd]” matches “a” or “b” or “c” or “d”
\b	A word boundary
\w	A word charcter, short for [a-zA-Z_0-0]

REGEX CHALLENGE: MATCH WORDS BEGINNING WITH A VOWEL

Vowels: a,e,i,o,u,A,E,I,O,U

Useful metacharacters:

*	Means 0 or more of the preceding RE e.g. “a*” matches 0 or more “a” characters in the sequence
[ ]	Matches a single character found within the square brackets e.g. “[abcd]” matches “a” or “b” or “c” or “d”
\b	A word boundary
\w	A word charcter, short for [a-zA-Z_0-0]

Answer: “`\b[aeiouAEIOU]\w*`”

EXPLANATION: WORDS BEGINNING WITH VOWELS

Regex: **`"\b[aeiouAEIOU]\w*"`**

Test sentence: **`"Elephants like to eat apples."`**

EXPLANATION: WORDS BEGINNING WITH VOWELS

Regex: **"\b[aeiouAEIOU]\w*"**

Test sentence: **"Elephants like to eat apples."**

\b

Whatever follows must be at a word boundary (start of word)

EXPLANATION: WORDS BEGINNING WITH VOWELS

Regex: "`\b[aeiouAEIOU]\w*`"

Test sentence: "`Elephants like to eat apples.`"

`\b`

Whatever follows must be at a word boundary (start of word)

→ Without this – would match from the first vowel in every word

EXPLANATION: WORDS BEGINNING WITH VOWELS

Regex: “ `\b[aeiouAEIOU]\w*` ”

Test sentence: “**Elephants like to eat apples.**”

`[aeiouAEIOU]`

Matches any single vowel

EXPLANATION: WORDS BEGINNING WITH VOWELS

Regex: **"`\b[aeiouAEIOU]`** `\w*`"

Test sentence: **"`Elephants like to eat apples.`"**

`\b[aeiouAEIOU]`

Matches any single vowel *at the start of a word*

EXPLANATION: WORDS BEGINNING WITH VOWELS

Regex: "`\b[aeiouAEIOU]\w*`"

Test sentence: "`Elephants like to eat apples.`"

`\w*`

Whatever was previously matched *plus any word characters that follow*

REGEX IN JAVA: SOME CHARACTERS MUST BE "ESCAPED"

Certain characters have special meaning

E.g., "

- Used to encapsulate a String literal and distinguish it from a variable
  - `myVar` is a variable but `"myVar"` is a String literal

REGEX IN JAVA: SOME CHARACTERS MUST BE "ESCAPED"

Certain characters have special meaning

E.g., "

- Used to encapsulate a String literal and distinguish it from a variable
  - `myVar` is a variable but `"myVar"` is a String literal

What if we want to include a character with special meaning in a String?

E.g., **`"Computer says, "Hello, World!""`**

REGEX IN JAVA: SOME CHARACTERS MUST BE "ESCAPED"

What if we want to include a character with special meaning in a String?

E.g. **`"Computer says, "Hello, World!""`**

↑ end of String ↑ start of next String

everything between → error!

REGEX IN JAVA: SOME CHARACTERS MUST BE "ESCAPED"

Use \ to tell Java to treat a special character as a literal

E.g. **`"Computer says, \"Hello, World!\""`**

the following character is not special

the following character is not special

REGEX IN JAVA: SOME CHARACTERS MUST BE "ESCAPED"

Regex metacharacters containing \ need to be escaped

Regex: **`"\b[aeiouAEIOU]\w*"`**

Java regex: **`"\\b[aeiouAEIOU]\\w*"`**

REGULAR EXPRESSIONS IN JAVA – EXAMPLE 1

```
import java.util.regex.*;
public class RegexExample1{
public static void main(String args[]){
//1st way
Pattern p = Pattern.compile(".s");//. represents single character
Matcher m = p.matcher("as");
boolean b = m.matches();

//2nd way
boolean b2 = Pattern.compile(".s").matcher("as").matches();

//3rd way
boolean b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);
}}
```

[Example credit: https://www.javatpoint.com/java-regex]

REGULAR EXPRESSIONS IN JAVA – EXAMPLE 2

```
import java.util.regex.*;
class RegexExample2{
public static void main(String args[]){
System.out.println(Pattern.matches(".s", "as"));
   //true (2nd char is s)
System.out.println(Pattern.matches(".s", "mk"));
   //false (2nd char is not s)
System.out.println(Pattern.matches(".s", "mst"));
   //false (has more than 2 char)
System.out.println(Pattern.matches(".s", "amms"));
   //false (has more than 2 char)
System.out.println(Pattern.matches("..s", "mas"));
   //true (3rd char is s)
}}
```

[Example credit: https://www.javatpoint.com/java-regex]

REGEX CHARACTER CLASSES

No.	Character Class	Description
1	[abc]	a, b, or c (simple class)
2	[^abc]	Any character except a, b, or c (negation)
3	[a-zA-Z]	a through z or A through Z, inclusive (range)
4	[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
5	[a-z&&[def]]	d, e, or f (intersection)
6	[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
7	[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z](subtraction)

[Table credit: https://www.javatpoint.com/java-regex]

REGULAR EXPRESSIONS IN JAVA – EXAMPLE 3

```
import java.util.regex.*;

class RegexExample3{

public static void main(String args[]){
System.out.println(Pattern.matches("[amn]", "abcd"));//
false (not a or m or n)
System.out.println(Pattern.matches("[amn]", "a"));
//true (among a or m or n)
System.out.println(Pattern.matches("[amn]", "ammmna"));
//false (m and a comes more than once)
}}
```

[Example credit: https://www.javatpoint.com/java-regex]

REGEX QUANTIFIERS

Regex	Description
X?	X occurs once or not at all
X+	X occurs once or more times
X*	X occurs zero or more times
X{n}	X occurs n times only
X{n,}	X occurs n or more times
X{y,z}	X occurs at least y times but less than z times

[Table credit: https://www.javatpoint.com/java-regex]

REGULAR EXPRESSIONS IN JAVA – EXAMPLE 4

```
import java.util.regex.*;
class RegexExample4{
public static void main(String args[]){
System.out.println("? quantifier ....");
System.out.println(Pattern.matches("[amn]?", "a"));
   //true (a or m or n comes one time)
System.out.println(Pattern.matches("[amn]?", "aaa"));
   //false (a comes more than one time)
System.out.println(Pattern.matches("[amn]?", "aammmnn"));
   //false (a m and n comes more than one time)
System.out.println(Pattern.matches("[amn]?", "aazzta"));
   //false (a comes more than one time)
System.out.println(Pattern.matches("[amn]?", "am"));
   //false (a or m or n must come one time)

}}
```

[Example credit: https://www.javatpoint.com/java-regex]

REGEX METACHARACTERS

Regex	Description
.	Any character (may or may not match terminator)
\d	Any digits, short of [0-9]
\D	Any non-digit, short for [^0-9]
\s	Any whitespace character, short for [\t\n\x0B\f\r]
\S	Any non-whitespace character, short for [^\s]
\w	Any word character, short for [a-zA-Z_0-9]
\W	Any non-word character, short for [^\w]
\b	A word boundary
\B	A non word boundary

[Table credit: https://www.javatpoint.com/java-regex]

REGULAR EXPRESSIONS IN JAVA – EXAMPLE 5

```
import java.util.regex.*;
class RegexExample5{
public static void main(String args[]){
System.out.println("metacharacters d....");\\d means digit

System.out.println(Pattern.matches("\\d", "abc"));
   //false (non-digit)
System.out.println(Pattern.matches("\\d", "1"));
   //true (digit and comes once)
System.out.println(Pattern.matches("\\d", "4443"));
   //false (digit but comes more than once)
System.out.println(Pattern.matches("\\d", "323abc"));
   //false (digit and char)

System.out.println("metacharacters D....");
   //D means non-digit
```

[Example credit: https://www.javatpoint.com/java-regex]

```
}}
```

REGULAR EXPRESSIONS IN JAVA – EXAMPLE 5

```
import java.util.regex.*;
class RegexExample5{
public static void main(String args[]){
System.out.println(Pattern.matches("\\D", "abc"));
   //false (non-digit but comes more than once)
System.out.println(Pattern.matches("\\D", "1"));
   //false (digit)
System.out.println(Pattern.matches("\\D", "4443"));
   //false (digit)
System.out.println(Pattern.matches("\\D", "323abc"));
   //false (digit and char)
System.out.println(Pattern.matches("\\D", "m"));
   //true (non-digit and comes once)

System.out.println("metacharacters D with quantifier....");
System.out.println(Pattern.matches("\\D*", "mak"));
   //true (non-digit and may come 0 or more times)
}}
```

[Example credit: https://www.javatpoint.com/java-regex]

TESTING I/0

CS 5004, SPRING 2022 – LECTURE 8

HOW TO TEST I/O?

Command line:

- Is input handled appropriately?
- Is output as expected?

File reading/writing:

- File processing?
- File output format?

COMMAND LINE KEY CHALLENGE: MAIN() CANNOT BE TESTED

Solution: separate communication with user from processing of input/output

In practice

- Send **`args`** to another class to process
- Format output elsewhere

COMMAND LINE KEY CHALLENGE: MAIN() CANNOT BE TESTED

Solution: separate communication with user from processing of input/output

Cookie counter example

`main()`
- Receives input

`CommandLineParser`
- Checks input is valid
- Stores validated input in appropriate format

`Calculator`
- Performs calculations based on input

COMMAND LINE KEY CHALLENGE: MAIN() CANNOT BE TESTED

Solution: separate communication with user from processing of input/output

Cookie counter example

`main()`
- Receives input
- Triggers parsing of input

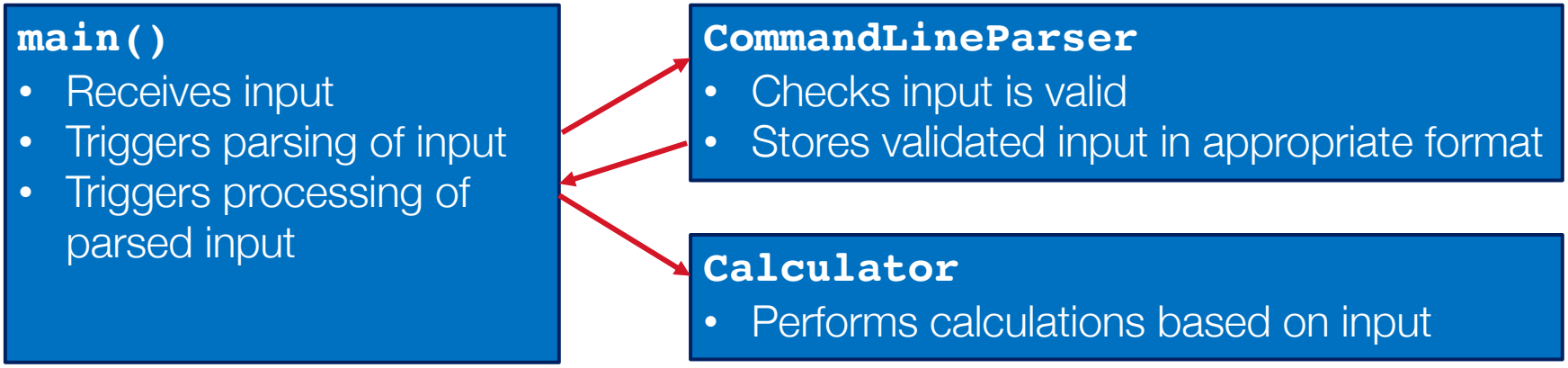
`CommandLineParser`
- Checks input is valid
- Stores validated input in appropriate format

`Calculator`
- Performs calculations based on input

COMMAND LINE KEY CHALLENGE: MAIN() CANNOT BE TESTED

Solution: separate communication with user from processing of input/output

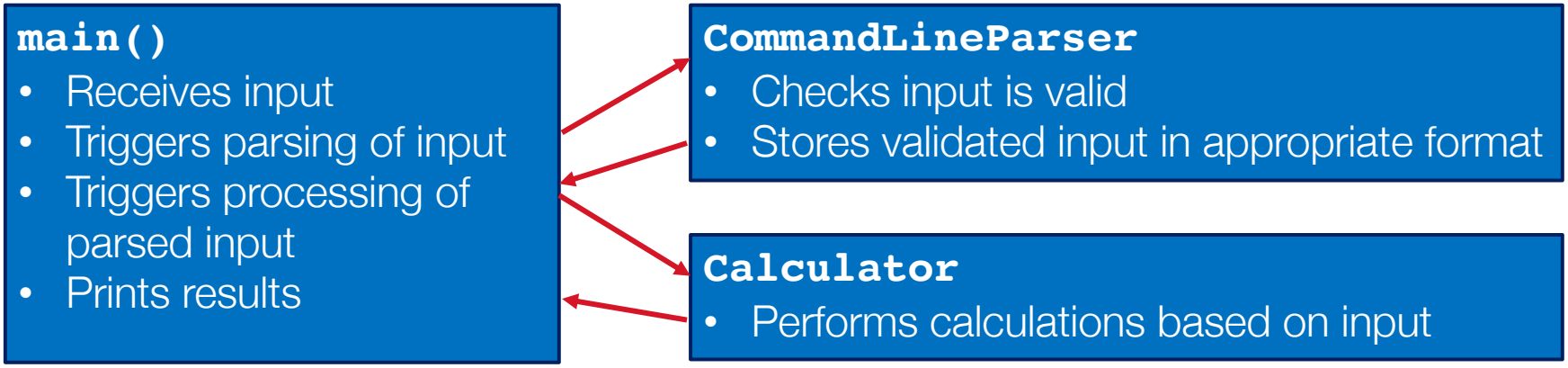
Cookie counter example



COMMAND LINE KEY CHALLENGE: MAIN() CANNOT BE TESTED

Solution: separate communication with user from processing of input/output

Cookie counter example



COMMAND LINE KEY CHALLENGE: MAIN() CANNOT BE TESTED

Solution: separate communication with user from processing of input/output

Takeaway → main() cannot be tested but all other functionality can be tested

HOW TO TEST READ/WRITE?

Built-in read/write functionality does not need to be tested

- Separate *reading* file from *processing* of file contents
  - Method containing file read should return the contents e.g. as a List<String>
- Separate *formatting* output from *writing* a file

→ Allows testing of custom functionality (processing, formatting) separate from built-in functionality

HOW TO TEST READ?

If code coverage *requires* that you test file read, consider creating some simplified test files

- Little content
- Just enough to check:
  - File read with expected format
  - File read with unexpected format
  - Missing file

HOW TO TEST READ/WRITE?

If code coverage *requires* that you test file read/write, try JUnit's `TemporaryFolder`

- Creates temporary storage for file input/output
- Files are destroyed after testing

USING TEMPORARYFOLDER

At the top of your test class:

```
@Rule
public TemporaryFolder tempFolder = new TemporaryFolder();
```

General steps:

- Write a file to the temporary folder
- Read the file into memory
- Check the contents in memory are as expected

To get the temporary folder path:

```
tempFolder.getRoot().getPath()
```

FILE I/O GOOD PRACTICE

- File I/O is expensive → use it carefully
- File processing should be cross-platform

FILE I/O IS EXPENSIVE

While a file is open, nothing else can access it.

→ Don't keep a file open longer than necessary

If possible, read the contents into memory, close the file, *then* process the contents

→ Don't process the contents line by line as you read the file

→ *Situations where this is not ideal are beyond the scope of this course*

FILE I/O IS EXPENSIVE

Avoid unnecessary file open/close operations

→ If possible, store contents in memory for other parts of the application to use. Don't reopen to re-access contents.

→ If possible, write an entire file at once.

→ Don't: open it, write one line, close it…open it again, write the next line, close…etc

FILE PROCESSING SHOULD BE CROSS-PLATFORM

Different operating systems have different conventions for:

- Line endings within a file
- File path separators

FILE PROCESSING SHOULD BE CROSS-PLATFORM

Line endings

- Mac: **`"\n"`**
- Windows: **`"\r\n"`**

Instead of String literals, use **`System.lineSeparator()`** e.g.:

DON'T: **`"First line\nSecond line"`**

DO: **`"First line" + System.lineSeparator() + "Second line"`**

FILE PROCESSING SHOULD BE CROSS-PLATFORM

File path separator

- Mac: **`"path/to/file.txt"`**
- Windows: **`"path\to\file.txt"`**

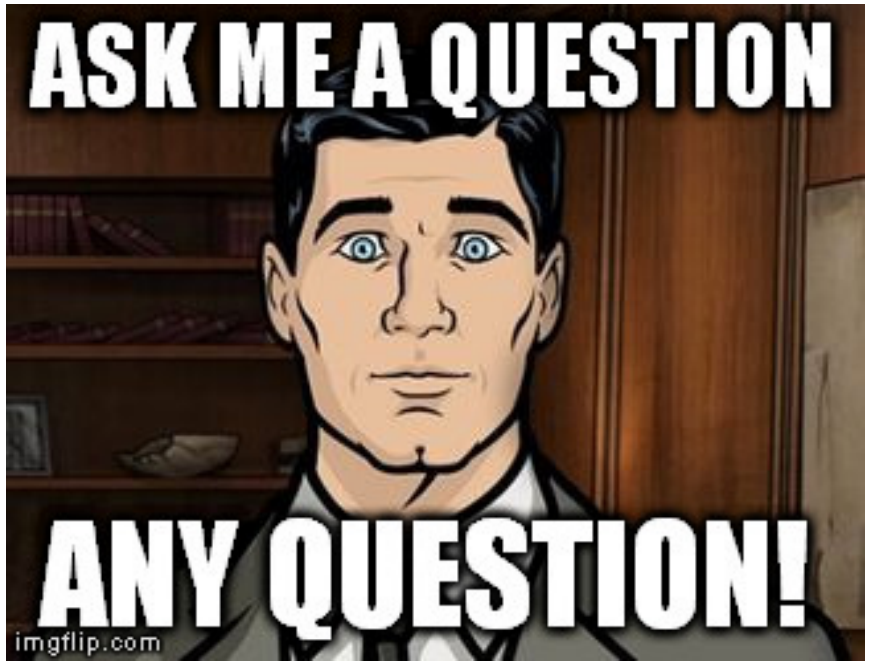
Instead of String literals, use **`File.separator`** e.g.:

DON'T: **`"path/to/file.txt"`**

DO:

```
"path" + File.separator + "to" + File.separator + "file.txt"
```

YOUR QUESTIONS



[Meme credit: imgflip.com]