CS 5004: Lecture 9

Northeastern University, Spring 2021

At the start of every lecture

1. Pull the latest code from the lecture-code repo
2. Open the Evening_lectures folder
3. Copy this week's folder somewhere else
   - So you can edit it without causing GitHub conflicts
4. Open the code:
   1. Find the build.gradle file in the folder called LectureX
   2. Double click it to open the project

Agenda

- Java file I/O
  - Reading/writing files using Java IO
  - Reading/writing files using Java NIO
- Interactive programs
- RegEx
- Testing I/O
- File processing good practice

Intro to Java I/O

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

InputStream examples

- Lecture 9 > iostreams
- Many different ways to do the same thing!

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

```java
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 `BufferedReader` and `FileReader` → `BufferedWriter` with `FileWriter`
- 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…
```

Example try-with-resources

- iostreams > MainTryWithResourcesReadWrite.java

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 lecture-code > 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

Interactive programs

Command line & Scanner

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

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

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. “--samoas”
- Required argument: number eaten
  - Must follow cookie name

GitHub > Lecture9 > cookiecounter

Running a Java class from the command line

e.g.: `java packagename.FileName`

- Run's the file's main method (if it exists)
- Anything printed (or input via Scanner) will show in the command line
- The file must be **compiled** first

Compiling in IntelliJ

IntelliJ compiles your project when your run it

- Compiled classes are in:
  project_folder > build > classes > java > main > package_folder
- The file endings will be .class instead of .java

Running your project from the command line

In the (system) terminal:

- `cd path/to/project_folder/build/classes/java/main/`
  - Change the red text to match your project/system
- `java packagename.FileName`

Passing arguments at the command line

Anything after the filename is an option/argument

```
java packagename.FileName arg1 arg2
```

Where do these arguments go?

Arguments are passed to `main` method

…as an array of Strings

```
public static void main(String[] args) {
  for (String arg: args) {
    System.out.println("You entered: " + arg);
  }
}
```

Option/argument distinction is terminology only—Java handles both the same way

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`

Break (10 mins)

Exercise 2 (walkthrough, follow along)

Write a program that reads country_code.csv line by line

- Allow the user to pass the file path in at the command line
- Informally test your program by running it
  - using Terminal (Command Prompt)
  - using IntelliJ configurations
- ***What do you notice about the file path?***

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

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`

Constructing a Scanner object

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

Note: Only “name” should be changed

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

- Returns what the user typed

Example Scanner Usage

- GitHub > scanner > Interactive.java

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

RegEx

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`

Exercise 3: Basic regex example

- GitHub > scanner > Regex.java
- Run it, try a few things
- **Can you work out what each `Matcher` method does?**
  - `.matches()`
  - `.lookingAt()`
  - `.find()`

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*”`**

Testing I/O

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 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"
```

Exercise 4: File I/O & regex

Finishing the country code processor from earlier

- User enters file path at command line
- Use regex to put each country into a HashMap
  - Key: country code
  - Value: country name
- Final values should not contain any extra quotes
- Write tests