Indiana University
CSCI-C212: Introduction to Software Systems (F25)
Joshua Crotts

Problem Set 12
November 19, 2025

Exceptions and I/O

Important Dates:

- Assigned: November 19, 2025
- Deadline: December 3, 2025 at 11:59 PM EST

Objectives:

- Students begin to understand the distinction between unchecked and checked exceptions.
- Students employ both exception types to write programs that involve file I/O.
- Students learn to write test cases that involve file I/O.

What To Do:

Design classes with the given specification in each problem, along with the appropriate test suite. **Do not round your solutions!**

What You Cannot Use:

You cannot use any content beyond Chapter 5.2. Please contact a staff member if you are unsure about something you should or should not use. Any use of anything in the above-listed forbidden categories will result in a **zero** (0) on the problem set. You cannot use the "modern Java I/O features" from section 5.3. For problem 4, you cannot use `Deque` or `ArrayDeque`.

How to Write Tests for Methods Using File I/O

This is very important to read, and I will redirect anyone that asks questions about how to write unit tests for file I/O to this section!

The way to write unit tests involving file I/O is, unfortunately, not very straightforward.

Suppose we are testing a method called `fact(int n, String outFile)`. This method receives an integer n and the name of an output file. The method will compute $n!$ and write it out to the file of the given name. Below are the steps for how to write *one* unit test:

1. Create a file called `fact001.exp`, where `.exp` means "expected output." This file will contain what you expect your file to output. For example, suppose we want to test `fact(5, "fact001.out")`. The generated output file `"fact001.out"` would, therefore, contain 120.
2. Create a JUnit test method, like we have created before.
3. Inside that method, call your method:

```
class FactTester {

  @Test
  void testFact001() {
    Fact.fact(5, "fact001.out");
  }
}
```

4. Below this, write code that reads in the lines of text from both the expected output file (denoted by the `.exp` extension) and the actual output file (denoted by the `.out` extension). Read the lines into two separate lists.
5. Assert that the contents of the lists are equal.

```
class FactTester {

  @Test
  void testFact001() {
    Fact.fact(5, "fact001.out");
    List<String> expLines = ...; // Read from "fact001.exp"
    List<String> actLines = ...; // Read from "fact001.out"
    assertEquals(expLines, actLines);
  }
}
```

Again, I said that these are the steps for writing *one* unit test. You need to do these multiple times to write comprehensive tests. It might be helpful to break the logic into smaller methods so as to avoid duplicating logic. Do not get lazy and simply omit tests because you don't want to write them. If you do, you will lose a substantial number of points!

Warning:

It is very tempting to ChatGPT/AI this assignment. ChatGPT/AI has a funky way of solving problems that involve file I/O, which makes it ostensibly obvious that you're using AI. So, don't do it. Everything in the problem set can be solved with the techniques from the book and class. Please don't make our lives any harder.

Problem 1:

Recall the `Optional` class and its purpose. In this exercise you will reimplement its behavior with the `IMaybe` interface with two subtypes `Just` and `Nothing`, representing the existence and absence of a value, respectively. Design the generic `IMaybe` interface, which contains the following three methods: `T get()`, `boolean isJust()`, and `boolean isNothing()`. The constructors of these subtypes receive either an object of type `T` or no parameter, depending on whether it is a `Just` or a `Nothing`. Throw an `UnsupportedOperationException` when trying to get the value from an instance of `Nothing`.

Problem 2:

Design the `ApplyOperations` class, which contains one method: `static void applyOperations(String inputFileName, String outputFileName)`. This method, when given input and output file names, reads a file of the following format:

```
n1 op n2
n3 op n4
...
```

Namely, each line (possibly) contains an integer, a space, an operator, another space, and a second integer. The method should parse this data and write the result of applying the operations to the output file. Though, there is the possibility for malformed input:

(1) If there are not exactly three components to a line, throw an `IllegalStateException`.

(2) If either of the operands are not integers, throw an `InputMismatchException`,

(3) If the operator is not one of `"+"`, `"-"`, `"*"`, or `"/"`, throw an `UnsupportedOperationException`, and

(4) If the operator is a division and the right-hand operand is 0, throw an `ArithmeticException`.

If **any** exceptions are thrown, do not produce an output file. As an example of valid input, an input file named `"test1.in"` with:

```
3 + 4
5 * 10
16 / 3
```

Then running `applyOperations("test1.in", "test1.out")` produces a file `"test1.out"` with the contents:

```
7
50
5
```

Problem 3:

Design the `RowNumbering` class, which contains one method: `static void outputRowNumbering(String filename)` method that, when given a file name containing an extension that is not `.out`, creates a new file of the same name with the `.out` extension, where each line from the original file is prefaced with the line number containing padded zeroes. For example, suppose a file `test.in` contains 473 lines of text. The generated output file should be named `test.out`, with its contents as follows:

```
001 ...
002 ...
...
471 ...
472 ...
473 ...
```

So, the number of leading zeroes of each line depends on what is the line number, as well as how many lines are in the original input file.

Problem 4:

A *stack-based* programming language is one that uses a stack to keep track of variables, values, and the results of expressions. These types of languages have existed for several decades, and in this exercise you will implement such a language.

Design the `StackLanguage` class, whose constructor receives no parameters. The class contains two instance methods: `void readFile(String f)` and `double interpret()`.

- The `readFile` method reads a series of "stack commands" from the file. These can be stored however you feel necessary in the class, but you should not interpret anything in this method, nor should you throw any exceptions. You may want to create a private static class for storing commands.
- The `interpret` method interprets the stored list of instructions. If no instructions have been received by a `readFile` command, throw an `IllegalStateException`. Here are the following possible instructions:

  (a) `DECL` *v* *X* declares that *v* is a variable with value *X*.

  (b) `PUSH` *X* pushes a number *X* to the stack.

  (c) `POP` *v* pops the top-most number off the stack and stores it in a variable *v*. If *v* has not been declared, an `IllegalArgumentException` is thrown.

  (d) `PEEK` *v* stores the value at the top of the stack in the variable *v*. If *v* has not been declared, an `IllegalArgumentException` is thrown.

  (e) `ADD` *X* adds *X* to the top-most number on the stack.

  (f) `SUB` *X* subtracts *X* from the top-most number on the stack.

  (g) `XCHG` *v* swaps the value on the top of the stack with the value stored in variable *v*. If *v* has not been declared as a variable, an `IllegalArgumentException` is thrown.

  (h) `DUP` duplicates the value at the top of the stack.

  If the command is none of these, then throw an `UnsupportedOperationException`. You may assume that all commands, otherwise, are well-formed (i.e., they contain the correct number of arguments and the types thereof are correct). After interpreting all instructions, the value that is returned is the top-most value on the stack. If there is no such value, throw a `NoSuchElementException`.

  Hint: use a `Map` to store variable identifiers to values.

Problem 5:

A maze is a grid of cells, each of which is either open or blocked. We can move from one free cell to another if they are adjacent. The top-left of a maze is the start, and the goal is to move to the bottom-right. For example, below is a maze, and below that is an example of a solved maze.

```
.########
........#
#.###.###
#...#...#
###.###.#
#...#...#
###.#.###
#...#...#
#######..
```

```
*********
******..#
#.###*###
#...#***#
###.###*#
#...#***#
###.#*###
#...#***#
#######**
```

Design the `MazeSolver` class, which has the following methods:

(a) `MazeSolver(String fileName)` is the constructor, which reads a description of a maze from a file. The file contains a grid of characters, where `'.'` represents an open cell and `'#'` represents a blocked cell. The file is formatted such that each line is the same length. Read the data into a `char[][]` instance variable. Assume that the maze dimensions (i.e., the number of rows and the number of columns) are on the first line of the file, separated by a space.

(b) `char[][] solve()` returns a `char[][]` that represents the solution to the maze. The solution should be the same as the input maze, but with the path from the start to the end marked with `'*'` characters. The start is the top-left cell, and the end is the bottom-right cell. If there is no solution, return `null`.

We can use a backtracking algorithm to solve this problem: start at a cell and mark it as visited. Then, recursively try to move to each of its neighbors, marking the path with a `'*'` character. If you reach the maze exit, then return `true`. Otherwise, backtrack and try another path. By "backtrack," we mean that you should remove the `'*'` character from the path. If you have tried all possible paths from a cell and none of them lead to the exit, then return `false`.

(c) `void output(String fileName, char[][] soln)` outputs the given solution to the maze to a file specified by the parameter. Refer to the above description for the format of the output file and the input `char[][]` solution.

Problem 6:

We can represent simple images using the PPM image format. The PPM image format is a primitive way to encode pixel data for an image. Each pixel has a color value from 0 to 255, representing the amount of red, green, and blue in each channel. Therefore, each pixel is represented by 3 distinct integers corresponding to the red, green, and blue color channels.

(a) Design the `Pixel` class, which has three instance variables: `int red`, `int green`, and `int blue`. The constructor should receive three integers, which are the values for each channel. If any of the values are not in the range $[0, 255]$, throw an `IllegalArgumentException`. Design the respective getters and setters for these instance variables.

(b) Design the abstract `Image` class, which has one instance variable: a `Pixel[][]` array representing the image's pixels. The constructor should receive a `Pixel[][]` array and assign it to the instance variable. Design the respective getters and setters for this instance variable. You should also write the `int getWidth()` and `int getHeight()` methods, which return the width and height of the image, respectively.

(c) Design the `PpmImage` class, which extends `Image`. The constructor should receive a file name and read the PPM image data from the file into a `Pixel[][]` array. The file format is as follows:

```
P3
width height
255
r1 g1 b1
r2 g2 b2
...
```

Note that because the `Image` class constructor receives the `Pixel[][]` array, we unfortunately cannot call `super` in the `PpmImage` constructor because we do not yet have the pixel data. Instead, you will need to call the `super.setImageData(Pixel[][] p)` method after reading the pixel data from the file. Finally, the width of an image is the number of columns in the pixel array, and the height is the number of rows. Do not accidentally reverse those!

Problem 7:

Design the `CharacterCounts` class, which contains one `static` method: `Map<Character, Integer> characterCounts(String filename, Set<Character> chars)`, which receives the name of a file, reads the contents thereof, and returns a map of specific characters to frequencies. That is, suppose `chars = {'\t', '\n', ' '}`, then the resulting map would contain an association of tabs, newlines, and blank characters with how many of each character there are.

Problem 8:

Design the `SpellChecker` class, containing the `static void spellCheck(String dict, String in)`. The `spellCheck` method reads two files: a "dictionary" and a content file. The content file contains a single sentence that may or may not have misspelled words. Your job is to check each word in the file and determine whether or not they are spelled correctly, according to the dictionary of (line-separated) words. If a word is not spelled correctly, wrap it inside brackets `[]`.

Output the modified sentences to a file of the same name, just with the `.out` extension (you must remove whatever extension existed previously). You may assume that words are space-separated and that no punctuation exists. Hint: use a `Set`! Another hint: words that are different cases are not misspelled; e.g., `"Hello"` is spelled the same as `"hello"`; how can your program check this? Assuming `dictionary.txt` contains a list of words, if we invoke the method with `spellChecker("dictionary.txt", "file3a.in")`, and `file3a.in` contains the following:

```
Hi hwo are you donig I am dioing jsut fine if I say so mysefl but I
will aslo sya that I am throughlyy misssing puncutiation
```

then `file3a.out` is generated containing the following:

```
Hi [hwo] are you [donig] I am [dioing] [jsut] fine if I say so
[mysefl] but I will [aslo] [sya] that I am [throughlyy] [misssing]
[puncutiation]
```

Problem 9 Extra Credit (25 points)

This problem has two parts. Note: this problem relies on having completed Problem Set 11, problems 3 and 4. When writing tests, write **your** tests in a file called `AsplTest.java`.

(a) *(5 points)* Having to manually type out the abstract syntax tree constructors when writing tests is extremely tedious. Design a *lexer* for the language described by the interpreter. That is, the text is broken up into tokens that are then categorized. For example, `'('` might become `OPEN_PAREN`, `"lambda"` might become `SYMBOL`, `"variable-name"` might become `SYMBOL`, `123.45` might become `NUMBER`, and `true`/`false` might become `BOOLEAN`. The output of the lexer is a list of tokens. Part of the trick is to ensure that after reading an open parenthesis, the next token is not grabbed as part of the open parenthesis.

Inside a class called `AsplLexer`, create a static method called `List<Token> lex(String in)`, which returns a list of `Token` objects. A `Token` is a class that stores a string for the type of token and a string for what the data is. (You will need to write this class yourself.) The class should have a constructor that receives the token type and the data as strings. Write the accessors as `getType()` and `getData()`. Override both `equals` and `hashCode`.

You are allowed and encouraged to modify the `lex` method you wrote in Problem Set 6.

You may assume that the only primitive operations applied are `+`, `-`, `*`, `/`, and `eq?`.

You may assume that there are no negative integer literals. To negate an integer, we apply `-` to an expression.

(b) *(15 points)* Design a *parser* for the language described by the interpreter. The idea is to tokenize a raw string, then parse the tokens to create an abstract syntax tree that represents the program Note: `lambda` corresponds to a `FuncNode`.

Inside a class called `AsplParser`, create a static method called `AstNode parse(List<Token> tokens)`, which returns an abstract syntax tree that can then be interpreted. It receives a list of tokens as generated by `lex`.

(c) *(5 points)* Design the `AsplReader` class, which contains the following method: `static AstNode evalProgram(String inFile)`, which reads in a file that contains an `ASPL` program, lexes it, parses it, then returns the `AstNode` result.

Note: to solve this problem, you must have completely solved parts (a) and (b).

Example Programs. Below are several examples of programs in the syntax of our language that your lexer and parser should be able to handle.

```
(+ 3 39)
```

```
(let (x 5)
  (+ x 37))
```

```
(let (x 5)
  (let (y 37)
    (* (+ y 2)
       (* x x))))
```

```
(let (x (+ 3 9))
  (let (y (+ x x))
    (- y x)))
```

```
(if true 42 0)
```

```
((lambda (x) (+ x 42)) 5)
```

```
((((lambda (x)
  (lambda (y)
    (lambda (z)
      ((x y) z))))
   (lambda (p)
    (lambda (q)
      (+ p q)))) 5) 10)
```

```
(let (x 0)
  (let (y x)
    (if (eq? x y) 42 0)))
```

```
(begin
  (define f
    (lambda (n)
      (+ n 42)))
  (f 5))
```

```
(begin
  (define !
    (lambda (n)
      (if (eq? n 0)
          1
          (* n (! (- n 1))))))
    (! 5))
```