Indiana University
CSCI-C212: Introduction to Software Systems (S25)
Joshua Crotts, Muazzam Siddiqui

Problem Set 3
February 19, 2025

Arrays, Collections, Generics

Important Dates:

- Assigned: February 19, 2025
- Deadline: March 4, 2025 at 11:59 PM EST,
  Note the date: this is one day earlier than normal!

Objectives:

- Students begin to understand the differences between classes in the Java Collections API.
- Students see the advantages and disadvantages over static arrays versus `ArrayList` objects.
- Students learn how generics allow them to write methods that work over any type.

What To Do:

For each of the following problems, create a class named `ProblemX`, where `X` is the problem number. E.g., the class for problem 1 should be `Problem1.java`. Write (JUnit) tests for each method that you design in corresponding test files named `ProblemXTest`, where `X` is the problem number. Additionally, write Javadoc comments explaining the purpose of the method, its parameters, and return value. **Do not round your solutions!**

This assignment contains ten required problems, with one extra credit problem worth 20 points, meaning the maximum possible score on this assignment is 120%/100%.

You must write sufficient tests and adequate documentation.

Problem 1:

Design the `String[] fizzBuzz(int min, int max)` method that iterates over the interval $[min, max]$ (you may assume $max \geq min$) and returns an array containing strings that meet the following criteria:

- If i is divisible by 3, insert `"Fizz"`.
- If i is divisible by 5, insert `"Buzz"`.
- If i is divisible by both 3 and 5, insert `"FizzBuzz"`.
- Otherwise, insert `"i"`, where `i` is the current number.

```
fizzBuzz(1, 12)  => {"1", "2", "Fizz", "4", "Buzz",
                     "Fizz", "7", "8", "Fizz", "Buzz",
                     "11", "Fizz"}
fizzBuzz(15, 18) => {"FizzBuzz", "16", "17", "Fizz"}
```

Problem 2:

Design the `int median(int[] A, int[] B)` that, when given two sorted (in increasing order) arrays of integers A and B, returns the median value of those two lists. You can use auxiliary data structures to help in solving the problem, but they are not necessary.

Problem 3:

Design the `boolean canSum(int[] A, int t)` method that, when given an array of integers A and a target t, determines whether or not there exists a group of numbers in A that sum to t. For example, if $A = \{2, 4, 10, 8\}$ and $t = 9$, then `canSum` returns false because there is no possible selection of integers from A that sum to 9. On the other hand, if $A = \{3, 7, 4, 5, 9\}$ and $t = 8$, then we return true because $3 + 5 = 8$. If $A = \{2, 4, 2, 1, 5, 4\}$ and $t = 9$, then we return true because $4 + 1 + 4$, but also $4 + 5 = 9$, $5 + 4 = 9$, and $4 + 1 + 2 + 2 = 9$.

Problem 4:

The correlation coefficient r is a measure of the strength and direction of a linear relationship between two variables x and y. The value of r is always between -1 and 1. When $r > 0$, there is a positive linear relationship between x and y. When $r < 0$, there is a negative linear relationship between x and y. When $r = 0$ or is approximately zero, there is no (or little) linear relationship between x and y. The formula for the correlation coefficient is as follows:

$$r = \frac{1}{n-1} \cdot \frac{1}{S_x \cdot S_y} \cdot \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})$$

Where n is the number of data points, x_i and y_i are the i^{th} data points, $\bar{x}$ and $\bar{y}$ are the means of x and y respectively, and S_x and S_y are the sample standard deviations of x and y respectively. To compute the sample standard deviation of a set of values S, we use the formula:

$$S_x = \sqrt{\frac{\sum_{i=1}^{|S|} (x_i - \bar{x})^2}{|S| - 1}}$$

Design the `double correlationCoefficient(double[] xs, double[] ys)` method that, when given two arrays of doubles *xs* and *ys*, returns the correlation coefficient between the two arrays. Assume $|xs| = |ys|$, and that $|xs|, |ys| \geq 2$.

Problem 5:

For this problem you are not allowed to use an `ArrayList` or any helper methods, e.g., `.contains`, or methods from the `Arrays` class. You *may* (and should) use a `Set<Integer>` to keep track of previously-seen peaks.

Joe the mountain climber has come across a large mountain range. He wants to climb only the tallest mountains in the range. Design the `int[] peakFinder(int[] H)` method that returns an array H' of all the peaks in an `int[]` of mountain heights H. A peak p is defined as an element of h at index i such that $p[i-1] < p[i]$ and $p[i] > p[i+1]$. If $i = 0$ or $i = |H| - 1$, Joe will not climb $p[i]$. Joe doesn't want to climb a mountain of the same height more than once, so you should not add any peaks that have already been added to H'. We present some test cases below.

```
peakFinder({9, 13, 7, 2, 8})                        => {13}
peakFinder({8, 7, 8, 7, 8, 7, 8, 7})                => {8}
peakFinder({111, 27, 84, 31, 5, 9, 4, 3, 2, 1, 64}) => {84, 9}
peakFinder({})                                      => {}
peakFinder({1})                                     => {}
peakFinder({1, 2})                                  => {}
peakFinder({1, 2, 1})                               => {2}
peakFinder({1, 2, 3, 2, 1})                         => {3}
```

Problem 6:

Design the `List<String> tokenize(String s, char d)` method that, when given a string *s* and a `char` delimiter *d*, returns an `ArrayList` of tokens split at the delimiter. You must do this by hand; you **cannot** call any `String` methods (except `.length` and `.charAt`).

Problem 7:

Design the `Map<String, Integer> wordCount(String s)` method that, when given a string *s*, counts the number of words in the list, then stores the resulting frequencies in a `HashMap<String, Integer>`. Assume that *s* is not cleaned. That is, you should first remove all punctuation (periods, commas, exclamation points, question marks, semi-colons, dashes, hashes, ampersands, asterisks, and parentheses) from *s*, producing a new string *s′*. Then, split the string based on spaces (remember `tokenize`?), and produce a map of the words to their respective counts. Do not factor case into your total; e.g., `"fAcToR"` and `"factor"` count as the same word. The ordering of the returned map is not significant.

```
String s = "Hello world, the world is healthy, is
            it not? I certainly agree that the world
            is #1 and healthy."
wordCount(s) => [<"hello" : 1>, <"world" : 3>, <"the" : 2>
                 <"is" : 3>, <"healthy" : 2>, <"it" : 1>,
                 <"i" : 1>, <"certainly" : 1> <"agree" : 1>
                 <"that" : 1>, <"1" : 1>, <"and" : 1>, <"not" : 1>]
```

Problem 8:

Design the `double postfixEvaluator(List<String> l)` method that, when given a list of binary operators and numeric operands represents as strings, returns the result of evaluating the postfix-notation expression. You will need to write a few helper methods to solve this problem, and it is best to break it down into steps. First, write a method that determines if a given string is one of the four binary operators: `"+"`, `"-"`, `"*"`, or `"/"`. You may assume that any inputs that are not binary operators are operands, i.e., numbers. Then, write a method that applies a given binary operator to a list of operands, i.e., an `ArrayList<Double>`.

```
postfixEvaluator({"5", "2", "*", "5", "+", "2", "+"}) => 17
postfixEvaluator({"1", "2", "3", "4", "+", "+", "+"}) => 10
postfixEvaluator({"12", "3", "/"})                    => 4
```

Problem 9:

Minesweeper is a simple strategy game where the objective is to uncover all spaces on a board without running into mines. If you are not familiar with the mechanics, we encourage you to find a version online and play it for a bit to understand its gameplay. In this exercise you will implement the minesweeper game as a series of methods.

(a) First, design the static `boolean isValidMove(char[][] board, int mx, int my)` method that receives a board and a move position, and determines whether the move is valid. A move is valid if it is located within the bounds of the board.

(b) Design the static `List<int[]> getValidNeighbors(char[][] board, int mx, int my)` method that receives a board and a move position, and returns a list of all the immediate neighbors to the cell (mx, my). Each element of the list is a two-element integer array containing the x and y coordinates of the neighbor. Consider the diagram below, where $(0, 0)$ is the move position, and the surrounding cells are its neighbors, represents as offsets. Note that `getValidNeighbors` should only return neighbors that are *in bounds*. Hint: use `isValidMove`.

(-1, 1)	(0, 1)	(1, 1)
(-1, 0)	(0, 0)	(1, 0)
(-1, -1)	(0, -1)	(1, -1)

(c) Design the static `List<int[]> getNonMineNeighbors(char[][] board, int mx, int my)` method that receives a board and a move position, and returns a list of all the neighbors that are not mines. You *must* use `getValidNeighbors` in your definition.

(d) Design the static `List<int[]> getMineNeighbors(char[][] board, int mx, int my)` method that receives a board and a move position, and returns a list of all the neighbors that are mines. You *must* use `getValidNeighbors` in your definition.

(e) Design the static `int countAdjacentMines(char[][] board, int mx, int my)` method that receives a board and a move position, and returns the number of mines that are adjacent to the given position. This method should be one line long and contain a call to `getMineNeigbors`.

(f) With the helper methods complete, we now need a method that searches through a position and reveals all non-mine adjacent positions. In general, this is a *traversal* algorithm called *depth-first search*. The idea is to recursively extend out the path until we hit a mine, at which point we unwind the recursive calls to extend another path.

Design the static `void extPath(char[][] board, int mx, int my)` method that receives a board and a move position, and extends the path from the given position using the following rules:

(i) If the given move position is invalid, then return.

(ii) If the character at `board[mx][my]` is not a dash, `'-'`, then return.

(iii) Otherwise, determine the number of adjacent mines to the move position. If the number of adjacent mines is non-zero, assign to `board[mx][my]` the number of mines at that move position.

(iv) If the number of adjacent mines *is* zero, then we can extend out the path to all non-mine neighbors. First, assign to `board[mx][my]` the character literal `'0'`, then loop over all non-mine neighbors to the move position. In the loop body, call `extPath` on each neighbor.

(g) Minesweeper board generation is an algorithmic problem in and of itself, and as such our implementation will be simple. Design the static `char[][] makeBoard(int N, int M, int B)` method that receives a board size of N rows, M columns, and B mines to place. To randomly place mines, create a `List<int[]>` of all the possible cells on the board, shuffle the list, retrieve the first B cells, and assign the character literal `'B'` to them. Assign the character literal `'-'` to all other cells.

(h) Finally, design the static `char[][] play(char[][] board, int mx, int my)` method that receives a board and a move position, and attempts to play the given move position on the board. If, at that position on the board, there is a mine, return `null`. Otherwise, call `extPath` on the board and position, then return `board`. In essence, `play` receives one game state and transitions it to the next state.

Problem 10:

Design the `<T extends List<Integer>> boolean areParallelLists(T t, T u)` method that, when given two types of lists t and u that store integer values, determines whether or not they are "parallel." In this context, Two integer lists are parallel if they differ by a single constant factor. For example, where $t = \{5, 10, 15, 20\}$ and $u = \{20, 40, 60, 80\}$, t and u are parallel because every element in t multiplied by four gets us a parallel element in u. This factorization is bidirectional, meaning that t could be $\{100, 200, 300, 200\}$ and u could be $\{10, 20, 30, 20\}$. Note that a list of all zeroes is parallel to every other list.

Extra Credit (20 points):

It's Black History Month! To celebrate, you are designing a system with querying functionality similar to a database language such as SQL. The database contains information about historic black figures in the STEM community, including computer science, math, and beyond. We have a 2D array of strings whose first row contains column headers to a database. Examples of such columns may be `"ID"`, `"Name"`, `"Age"`, `"Salary"`, and so forth.

Design the `List<String> query(String[][] db, String cmd)` method that, when given a "database" and a "Command", returns the data from the rows that satisfy the criteria enforced by the command.

```
A Command is "SELECT <count> <header> WHERE <predicate>"
```

The `SELECT` command receives a `<count>`, which is a number between 1 and *n*, or the asterisk to indicate everyone in the database. The `WHERE` clause receives a "Predicate". The `SELECT` command receives a `<count>` and a `<header>` to designate that the command should return `<count>` rows with data from the `<header>` column. An asterisk can be used to select all rows in the database. Your implementation should be flexible enough to work with any arbitrary column over the database. (You may assume that the input `<header>` is a valid column in the database, but it cannot be hard-coded to fit only a particular set of database columns, e.g., `"ID"`, `"Name"`, and so forth.)

```
A Predicate is "<header> <comparator> <value>"
```

Headers are one of the column headers of the database, and comparators are either `=`, `!=`, `<`, `<=`, `>`, or `>=`, or `LIKE`. Values are either numbers, floats, or strings.

Parsing a `LIKE` command is more complicated. There are four possible types of values:

```
'S'
'%S'
'S%'
'%S%'
```

The first matches an exact string, namely `S`. The second matches any string that ends with `S`. The third matches any string that begins with `S`. The fourth matches any string that contains `S`.

You may assume all commands are well-formed. However, it is possible that a command returns no results, e.g., `SELECT * Name WHERE Salary >= 10000000`.