Object Oriented Programming
Continuous Assessment 2
Assignment Questions
Semester 1 - 2024

BSc (Hons) in Computing in Games Development
BSc (Hons) in Computing in Software Development
Stage 2

Weighting: 15%

Deadline: (Moodle)

Attempt **all** Questions. Answer questions in order.

Use the starter code provide on Moodle.

Students will work in small groups to solve problems and design solutions and discuss solution design with their lecturer, but students **must complete their own implementation individually**.

Each student must create a Repository (Repo) on GitHub called "**oop-ca2-firstname-lastname**" and push commits of their work, with meaningful commit messages, on an ongoing basis. You must share this Repo with your lecturer for assessment purposes.

A zipped version of the source files must also be uploaded to Moodle by the deadline in addition to the GitHub Repo.

You must **reference the source of any code** that you have used by inserting comments with relevant URLs in your code.

Students will demonstrate their work and will be interviewed on their understanding of the material.

Marks will depend on the solutions provided, a demonstrated ongoing patter of work evidenced by git commit history and review meetings with lecturer, and a demonstrated understanding of the work during the final demo and interview.

Question 1 – Goods Containers (Interface) [6 marks]

We wish to create a system to represent various containers to store goods on a lorry. (e.g. Box, Cylinder, etc…). For these items we define a 'concrete' class for each type of item with fields:

- **Box** – length, width, depth and weight
- **Cylinder** – height, diameter, and weight
- **Pyramid** – length, sideLength and weight (Toblerone shape, all edges equal)

A class called **ContainerManager** will be used to store container objects (in an internal list). This manager class will provide the following services (or functionality):

- **add**(container) - to add a container to the list
- **totalWeight**( ) - to return the total weight of all objects currently in the ContainerManager
- **totalRectangularVolume**( ) – to return the total 'rectangular' volume of all objects
- **clearAll() –** remove all objects
- **getAllContainers() –** return a List of all containers

In order to allow the ContainerManager to deal with a variety of container types, we define an interface ***IMeasurableContainer*** that all container types must implement in order to use the manager services. The manager will only deal with objects of type ***IMeasurableContainer***. So, all references used in the ContainerManager will be of type IMeasurableContainer.

This structure results in the ContainerManager class having a dependency only on the interface IMeasurableContainer.

You must define the interface IMeasurableContainer with the following two methods:

- double weight();
- double rectangularVolume();

In the main() method, you will instantiate a ContainerManager class and a number of Box, Cylinder and Toblerone classes, and add those objects to the manager. You can then call the *totalWeight*() and *totalRectangularVolume*() methods on the manager object, and display the returned values. Call getAllContainers() and store the returned List. Iterate over the returned list and print all fields by calling their getter methods and do NOT use toString().

Note: "rectangular volume" for a object is the smallest rectangular box that an object will fit in.

1. Implement all of the classes as described above and test your program.
2. Draw a **UML diagram** to represent the classes, and consider the dependencies.

Question 2 – Car Parking (Stack) [6 marks]

A homeowner rents out parking spaces in a driveway during special events. The driveway is a “last-in, first-out” LIFO stack. Of course, when a car owner retrieves a vehicle that wasn’t the last one in, the cars blocking it must be temporarily move to the street so that the requested vehicle can leave. Write a program that models this behaviour, using one stack for the driveway and one stack for the street. Use integer values as license plate numbers (e.g. 1,2,3,4…).

User Interface: The user enters positive numbers to add a car (1,2,3…), negative numbers remove a specific car(so, “-2” means to remove car number 2), and zero stops the simulation. Print out the current state of the stack after each operation is complete.

So, entering “1” means – add car number 1 to the driveway, entering “-2” means - retrieve car number 2 from the driveway.

Question 3 – Nested HTML Tags (Stack) [6 marks]

Write a program that checks whether a sequence of HTML tags is properly nested. For each opening tag, such as <p>, there must be a closing tag </p>. Tags such as <p>, <ul>,<li> may have other tags nested inside, for example :

<p> <ul> <li> </li> </u> **<br>** </p>

The inner tags must be closed before the outer ones, in nested fashion. An exception is the <br> tag that has no closing tag. Your program should process a file containing tags. One is provided with the sample code, but you should expand this. For simplicity, assume that the tags are separated by spaces, and that there is no text between the tags.

Question 4 - Flood Fill (Stack, 2D Array) [10 marks]

In a paint program, a “flood fill” fills all empty pixels of a drawing with a given colour, stopping when it reaches occupied pixels. In this exercise, you will implement a simple variation of this algorithm, flood-filling a 10 × 10 array of integers that are initially set to 0. Prompt for the starting row and column (the starting cell for the flood fill).

- Push the (row, column) pair onto a stack. You will need to provide a simple Cell class with two fields (storing row and column; (x,y)?).
- Repeat the following operations until the stack is empty.
  - Pop the Cell (row, column pair) from the top of the stack.
  - If that cell has not yet been filled, fill the corresponding cell location with a number 1, 2, 3, and so on (this number is incremented at each step to show the order in which the squares are filled).
  - Push the coordinates of any unfilled neighbours in the north, east, south, or west direction on the stack.
- When you are done (i.e. when the stack is empty), print the entire 2D array.

Question 5 – Java Identifier Count (Map) [6 marks]

Write a program that reads a Java source file and constructs a Map **identiferCountMap** containing all of the identifiers in the file and a count of the number times each identifier occurs.

Create a second map that stores the identifier as a key and a corresponding list of all of the lines of code that contain that key. Add the code line number to the start of each line of code.

Output a list of all identifiers, sorted in ascending order of identifier, showing the identifier, its count and the list of lines of code that use that identifier.

Sample output:

```
boolean   2                          [i.e. "boolean" is the identifier]
5. boolean finished;
12. public Boolean getStatus() {
int 1
6. private int age;
```

Identifiers are variable names, class names and keywords etc.

Read from Java source file in a line-by-line manner.

For simplicity, we will consider each string consisting only of letters, numbers, and underscores an identifiers. Use a delimiter to filter e.g. "[^A-Za-z0-9_]+").

Question 6 – Airport Flights (Queue) [6 marks]

An airport has only one runway. When it is busy, planes wishing to take off or land have to wait. Implement a simulation, using **two** queues, one each for the planes waiting to take off and land. **Landing planes get priority**.

The user enters commands:

> takeoff *flightCode*

> land *flightCode*

> ***next***

> ***quit***

The first two commands place the flight in the appropriate queue. The ***next*** command finishes the current take-off or landing and enables the next one, printing the action (take off or land) and the flight symbol.

For example:

```
takeoff( “Flight-100”);   // is queued for ‘take off’
takeoff(“Flight-220”);    // is queued
land(“Flight-320”);       // is queued for landing
takeoff(“Flight-EI104”);  // queued for takeoff
next(); // will complete landing of the landing of Flight-320
next(); // will complete takeoff of Flight-100
```

Question 7 – Stock Shares Tax Calculation (Queue) [10 marks]

Suppose you buy 100 shares of a stock at \$12 per share, then another 100 at \$10 per share, and then sell 150 shares at \$15. You have to pay taxes on the gain, but exactly what is the gain? In the United States, the FIFO rule holds: You first sell all shares of the first batch for a profit of \$300, then 50 of the shares from the second batch, for a profit of \$250, yielding a total profit of \$550. Write a program that can make these calculations for arbitrary purchases and sales of shares in a single company. The user enters commands **buy quantity price,** and **sell quantity** (which causes the gain to be displayed), and quit. Hint: Keep a queue of objects of a class **Block** that contains the quantity and price of a block of shares.

Question 8 – Multi-Company Stock Shares Tax Calculation (Queue) [10]

Extend Question 6 to a program that can handle shares of multiple companies. The user enters commands ***buy symbol quantity price*** and sell symbol quantity. Hint: Keep a `Map<String, Queue<Block>>` that manages a separate queue for each stock symbol.

Question 9 Arithmetic Expression Calculator (Stack) [10 marks]

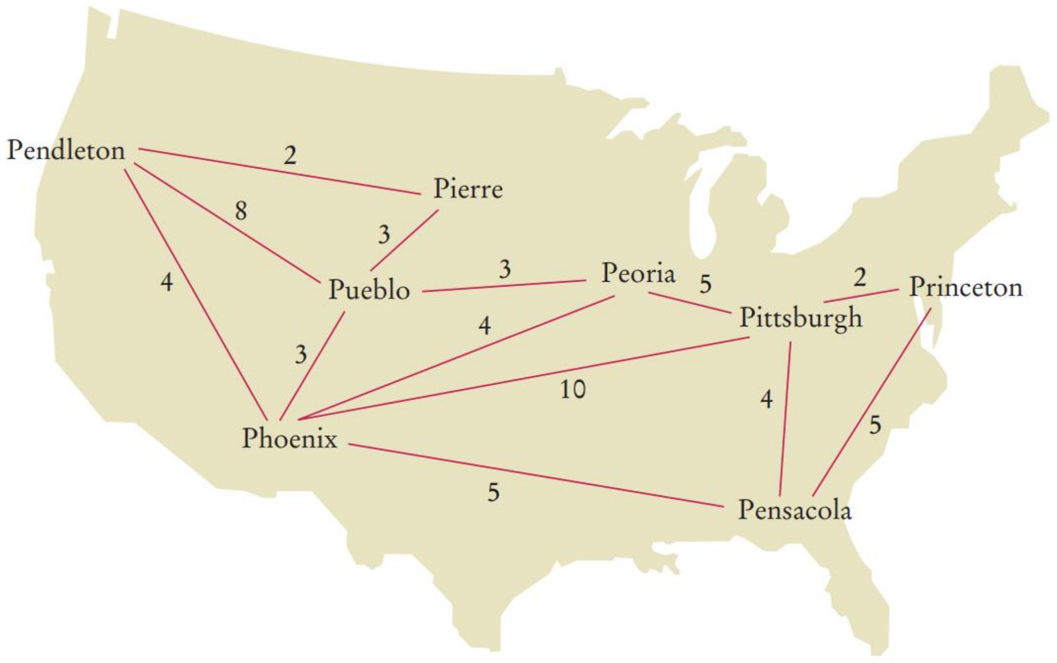
Implement a calculator to evaluate arithmetic expressions for the operators + - * / and parenthesis ( ). See the accompanying PDF document which is an extract from a book explaining the algorithm.

Question 10 – Backtracking through a Maze (Stack) [10 marks]

Implement a backtracking algorithm, using a Stack, to find a path through a maze from start to exit. Refer to description of algorithm (from textbook) in the accompanying PDF.

Question 11 – Shortest Distance to City [20 marks]
(Map, TreeSet, PriorityQueue)

Consider the problem of finding the least expensive routes to all cities in a network from a given starting point.



For example, in this network, the least expensive route from Pendleton to Peoria has cost 8 (going through Pierre and Pueblo). The following helper class expresses the distance to another city:

```
public class DistanceTo implements Comparable {
      private String target;
      private int distance;
      public DistanceTo(String city, int dist) {
            target = city; distance = dist; }
      public String getTarget() { return target; }
      public int getDistance() { return distance; }
      public int compareTo(DistanceTo other) {
            return distance - other.distance;
      }
}
```

All direct connections between cities are stored in a **Map<String,TreeSet<DistanceTo>>.**

The algorithm now proceeds as follows:

```
Let *from* be the starting point.
Add DistanceTo(from, 0) to a priority queue.
Construct a map shortestKnownDistance from city names to distances.
While the priority queue is not empty
    Get its smallest element.
    If its target is not a key in shortestKnownDistance
        Let d be the distance to that target.
        Put (target, d) into shortestKnownDistance.
        For all cities c that have a direct connection from target
            Add DistanceTo(c, d + distance from target to c) to the priority queue.
```

When the algorithm has finished, *shortestKnownDistance* contains the shortest distance from the starting point to all reachable targets. Your task is to write a program that implements this algorithm. Your program should read in lines, from a file, of the form:

city1 city2 distance

The starting point is the first city in the first line. Print the shortest distances to all other cities.

END