Department of Computer Science and Software Engineering
Concordia University

COMP 249: Object-Oriented Programming II - Winter 2020

Instructors: Dr. Aiman Hanna, Dr. Yuhong Yan,
Dr. Abdelwahab Elnaka, Dr. Zixi Quan, Dr. Rabe Abdalkareem

24-hour Take-home Final Examination

Submission Due date and time (EXACT):
Thursday, April 23nd 9:00 AM EST

Exam Rules: Please read very carefully:

1. General Information

Date and time of exam release: **Wednesday, April 22, 2020 at 9:00 AM EST**

Deadline for exam submission: **Thursday, April 23, 2020 at 9:00 AM EST**

Exam is a 24-hours exam. You must submit your solution by 9:00 AM on Thursday April 23. Submission received at, or after 9:01 AM EST, will be rejected and FNS grade will automatically be issued for the course. [In fact, the system will reject any submissions beyond 9:00 AM EST]. After the exam, you *may* be called for an oral evaluation. If you are called for an online oral evaluation, you **MUST** present yourself for this evaluation. Failing to do that will result in your grade being suspended and eventually FNS grade will be assigned.

Quality of Submission: You will be marked based on correctness, comprehension, completeness and quality of produced work, and the solution.

You must produce and submit Java code that compiles and runs. The exam has multiple tasks examining different subjects. All tasks are required and you should attempt all of them. In case you are unable to finish any of the tasks, make sure that your incomplete work on these tasks is commented out; so the submission works correctly for the successfully completed tasks. More details on what you need to submit is indicated below.

⇨ **Very Important:** When a name is given for a class, attribute, method, etc., you **MUST** use this name as given. Do **NOT** change these names. Doing so will cost 50% of the mark! No exceptions will be considered as well.

Submissions: You need to submit "exactly" **the following 3 files.** The files must be submitted to Moodle of your section under the title: ***Take-Home Final Exam Submission.***

1) A **signed copy of this exam itself**. You need to enter your name, ID# and **sign** it (besides that, do NOT alter or change anything else). This file must be submitted as a **PDF** file.
2) **Your UML Diagram**. This file must be submitted as a **PDF** file. The file must include you name, ID, and you must **sign** it. Signatures must be identical.
3) **Your entire Java project of your solution**. This file must be submitted as a **.ZIP** file. You need to include all files in a single zip file that is named as **<Your-ID-#_Your-Name>.zip**. For instance: 9080332_Mike_Simon.zip.

⇨ **VERY IMPORTANT: Once again, please notice that you need to sign this exam document itself and include it with your submission as well. Your exam will not be marked unless this exam is signed and returned back.**

⇨ **If you do not submit any of these two signatures; we will have to ask you for it manually before your exam can be considered. This request will cost you 10% of the mark (no exceptions will ever be considered)! So; in simple words, return the exam signed, the UML file signed, and of course the entire project of your solution.**

2. Terms and Conditions

The exam is an individual exam. The Exam is open-book. You can consult any materials, including textbooks, notes, Moodle materials, tutorial notes, and online materials. However; you are **NOT** permitted to communicate with any persons neither offline nor online (or through any other type of communication), including your classmates.

In simple words, **plagiarism will NOT be tolerated under any conditions** and it will officially be reported to the University for disciplinary actions.

3. Questions that arise during the exam

It is important to note that there will be no support from the instructors or the TAs during the exam. The exam contents are clear; however, in case you find anything ambiguous, you must express your assumptions very clearly and continue with the solution. We may however send you some announcements if needed through Moodle and the mailing list, so you should check from time to time. In simple words, Avoid hesitations! The text of the exam is very clear and the requirements are detailed carefully. **You should not send requests to your professors requesting clarifications.** They may not even have a chance to see these requests on-time.

4. Academic Integrity

Concordia University takes academic integrity very seriously and expects its students to do the same.
You need to declare and sign the following:
I understand that any form of cheating, or plagiarism, as well as any other form of dishonest behaviour, intentional or not, related to the obtention of gain, academic or otherwise, or the interference in evaluative exercises committed by a student is an offence under the Academic Code of Conduct.
I understand that the above actions constitute an offence by anyone who carries them out, attempts to carry them out or participates in them.

By way of example only, academic offences include:

- Plagiarism;
- Contribution by a student (or anyone else) to another student's work with the knowledge that such work may be submitted by the other student as their own;
- Unauthorized collaboration;
- Obtaining the questions or answers to an exam or other unauthorized resource;
- Use of another person's exam during an exam;
- Communication with anyone (in any shape or form) during an exam and any unauthorized assistance during an exam;
- Impersonation;
- Falsification of a document, a fact, data, or reference.

For more information about academic misconduct and academic integrity, refer to the Academic Code of Conduct and Concordia University's Academic Integrity webpage: https://www.concordia.ca/conduct/academic-integrity.html.

I have neither given nor received unauthorized aid on this exam and I agree to adhere to the specific Terms and Conditions that govern this exam.

Student Name **Signature:**

Shadi Jiha

40131284
ID #: --------------------------------

Shadi Jiha

An **infrastructure** (or simply a **structure**) of a city is defined as a physical facility (i.e. roads, buildings, bridges, power supplies, etc.).

In the scope of this exam, the following classes are defined:

- A **Structure** class has two attributes: *yearOfCreation* (int), and *cost* (double).
- A **Building** class has two attributes: *landSpace* (double), which describes the amount of land space of the building, and *material* (String), which describes the main material used for the building, such as concrete, wood, glass, etc.
- A **Bridge** class has two attributes: *length* (double), and height (double), which describes the peek height of the bridge above ground level.
- An **Overpass** is a bridge, which additionally has the following attribute: *maxLoad* (double), which indicated that maximum load/weight accommodated by the overpass.
- A **HighRise** class is a building that additionally has the following attributes: *height* (double), which describes the height of the high-rise.
- A **ResidentialBuilding** is a building that additionally has the following attributes: *maxNumberOfHabitants* (int), which describes the maximum number of expected habitants.
- A **House** is a residential building, which additionally has the following two attributes: *numberOfRooms* (double), which describes the number of rooms in the house (notice that this is a double since houses may describe a washroom as ½ a room), and *numOfFloors* (int), which describes the number of floors in the house.

- A **CondoBuilding** is a residential building that additionally has the following attribute: *numOfUnits* (int), which describes the number of condo units the building has.
- An **Airport** is a structure that additionally has the following two attributes: *numOfRunways* (int), and *code* (String), which gives the international airport code.
- A **CommercialAirport** is an airport that additionally has the following attributes: *numOfGates* (int), which describes the number of gates in the airport.
- A **CargoAirport** is an airport that additionally has the following attribute: *landedWeight* (double), which describes the weight of landed goods on the airport. (As an explanation side note, for an airport to be classified as a cargo airport, the airport must have a landed weight of 100 million pounds according to the FAA).

In this exam, you are required to perform few tasks. We very strongly recommend that you work on these tasks in the order they are given.

In your code, you must:
1) Place clear comments that indicate where your tasks are implemented;
2) include a *System.out.println()* statements before and after the execution of each task. These statements should be similar to the following: ***System.out.println("Starting Task 3.C")***; and ***System.out.println("End of Task 3.C");***

Task 1:

Provide a UML diagram describing the hierarchy of the above **Structure** classes. This UML diagram should be included in a file called **Infrastructure_UML.PDF** (Remember; you will need to write your name, ID and **sign** this file.)

Task 2:

Write the implementation of these classes. All these classes need to be included in a single package, with the exception of **CommerialAirport** class and **CargoAirport** class, which must be included in separate packages (so; you will have a total of 3 packages). Call the main package **FX_W20PKG** ,and call the other two packages **CargoAirport** and **CommercialAirport**. As a general rule, you must provide sufficient constructors, include all necessary methods in all of these **classes**, use appropriate access right, etc. **However, you are NOT allowed have any default constructors for any of these classes.** Additionally, you only need to include *set* and *get* methods (i.e. *getYearOfCreation()*, *setCost()*, etc.) if they are only needed. **Do NOT include these methods otherwise.**

Task 3:

A. Create a driver class called Infrastructure in the same package where most of the classes exist. In this class, you will have few static methods (will be created as you go through the tasks), as well as the main() method. In the main() method, you need to create at least **6** objects from **each** of the classes (when it makes sense, or possible, to create them!), initialize all of these objects with appropriate values at creation.

B. Following that, place all of these objects in a single array called **StrArr**. You *must* scatter your objects on the array (i.e. do not place all **House** objects for instance after each others).

C. Write a static method called **findTallestHighRise()**, which should accept an array of Structure objects, search all objects in that array, display the info of the tallest/highest **HighRise** object found in the array, then returns the index of that object. If more than one **HighRise** objects have the same highest value, return the first one. Return -1 if no **HighRise** objects were found. Add a test case in your main method using **strArr** as a parameter that shows whether or not your implementation is correct. That is, your code should include something like that:

```
System.out.println("Starting Task 3.C");
int i = findTallestHighRise(strArr);
if (i!=-1)
      System.out.println("Tallest HighRise was found at index: " + i + " Here is the
                             info of that object");
else
      System.out.println("No HighRise objects were found in the array!");
      System.out.println(strArr[i]);
      System.out.println("End of Task 3.C");
```

The output may look like:

```
Starting Task 3.C
Tallest HighRise was found at index: 37 Here is the info of that object
HighRise object Info: year of Creation is: 2010, cost is: 5.5E7, landspace is: 2071.0, material is: Steel, and its height is:
255.0
End of Task 3.C
```

D. Write a method called **showBuildingInfo()**, which should accept an array of **Structure** objects, search all objects in that array, and display only the info of all found **Building** objects in that array. Add a test case in your main method using **strArr** as a parameter that shows whether or not your implementation is correct.

E. Write a static method called **displayAllObjects()**, which should accept an array of **Structure** objects and, **recursively**, display the info of all objects in the passed array. Additionally, the display of the objects must be made backward. That is from the object at the last index of the array to the one at index 0. Add a test case in your main method using **strArr** as a parameter that shows whether or not your implementation is correct.

F. Write a method called **copyStructures(),** which accepts an array of **Structure** objects and returns a copy of that array. Add a test case in your main method using **strArr** as a parameter that shows whether or not your implementation is correct. Use the **displayAllObjects()** method you created above to show the contents of the new copied array.

<u>Task 4:</u>

A. Create an ArrayList, called **strArrLst** of **Structure** objects. Initialize the ArrayLst to size **30**.

B. <u>Using the built-in ArrayList methods</u>, copy all your objects from the **StrArr** array in Task 3, into **strArrLst**.

C. <u>Using the built-in ArrayList methods</u>, remove any 5 objects from **strArrLst**.

D. <u>Using the built-in ArrayList methods</u>, find out whether or not the **strArrLst** contains any of the objects. If so, you must indicate the location/index where the object was found; otherwise, indicate that the object is not found. You must attempt to find one of the objects that are still in **strArrLst** and at least one that was removed.

<u>Task 5:</u>

A. Create a text file called **AirportCodes.txt** using the **PrintWriter** class.

B. Write a method called **findExistingAirportCodes()**, which accepts two parameters, a **Printwriter** object and an array of **Structure** objects. The method must find all airport codes of any **Airport** objects in the array and permanently store them in the **AirportCodes.txt** file. You should store one code per line.

C. Call the **findExistingAirportCodes()** method with the **PrintWriter** object you just created above, and the **StrArr** array as the second parameter.

D. Open the **AirportCodes.txt** but this time as an input for reading using the **BufferedReader** class.

E. Write a method called **displayAirportCodes()**, which accepts a **Bufferedreader** object. The method should read the input file and display the information in the file to the screen. <u>You are only allowed to use</u> the **read()** method from the **BufferedReader** class to perform what is needed.

F. Call the **displayAirportCodes()** method with the **Bufferedreader** objects created in #5.D above to display the contents of the **AirportCodes.txt** file.

<u>Task 6:</u>

For this task, you are **NOT** permitted to use any of the java Built-in types (i.e. Built-in Linked Lists). You will need to write your entire code from scratch to achieve what is needed.

A. Create a new class called **StructureList**, which has an <u>inner class</u> called **StructureNode** and one attribute called **head**.

B. The **StructureNode** class has two attributes, called **sObj**, which is a **Structure** object, and **next**, which is a pointer to a **StructureNode**.

C. In the **StructureList** class include the following methods (as well as any other necessary methods):

- **addToStart(),** which takes an **Structure** Object as a parameter and adds a node that includes that object to the start of the calling **StructureList** list object.

- **addAtEnd()**, which takes an **Structure** Object as a parameter and adds a node that includes that object to the end of the calling **StructureList** list object.

- **clone()**, which creates an independent copy of the calling object. The return type of the method must be declared as **Object**.

- **append()**, which takes as a parameter a **StructureNode** object, and appends the passed list to the calling object, then nullifies this passed list. For instance, if the method is called like that (for two lists sLst3 and sLst8) sLst3.append(sLst8), the method should concatenate/append sLst8 at the end of sLst3. sLst8 is then nullified (that is, the list will not have any nodes after that).

- **showListContents()**, which should display the contents of the calling **StructureList** list. Place `"  ===> \n"` after the display of each node, and place `"X"` at the end of the display.

D. In the **main()** method, create few **StructureList** list objects, test all the methods that you created above, and use the **showListContents()** after operations (addToStart, append(), etc.) are done to these lists.

This concludes the exam. Good Luck!