Concordia University GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE	**Concordia University** **Comp 249 – Winter 2021** **Object-Oriented Programming II - Assignment 2** **Due 11:59 PM - Friday March 5, 2021**

Purpose: The purpose of this assignment is to practice class inheritance, composition, and other Object Oriented Programming concepts such as constructors, access rights, method overriding, etc. The assignment would also allow you to practice the notion of package creation. This assignment contains two parts. You need to complete Part I to be able to do Part II. You should however handle each part as if it was a separate assignment (i.e. you should not modify Part I after finishing with it; rather create a copy of that part and use it to create Part II. This is needed so that you can finally submit both parts!

⇨ **IMPORTANT NOTE:** IN THIS ASSIGNMENT YOU ARE NOT PERMITTED TO USE ANY OF THE JAVA BUILT-IN CLASSES, SUCH AS ArrayLists, Hash Maps, etc. Using any of these will result in zero marks! In other words, you need to code whatever needed by yourself!!!

The Classes!

An **Address** represents information that can be used to contact a Party. A Party represents a person or an organization. In general, an Address may specify business, home, out-of-hours contact, emergency contact, and so on.

Each address specifies the usable time of the Address (i.e., limit the context in which the Address is applicable/valid. In other words, an Address has the following attributes: **validFrom** (String type representing a date in the format `"YYYY-MM-DD"`) and **validTo** (String type representing a date in the format `"YYYY-MM-DD"`) to specify and limit the use of an Address. Any address that falls out of the date limits is still and address, but considered to be obsolete. Notice that the dates can be future dates (yet to become valid). These Addresses are also obsolete for instance compared to the current date.

There are six (6) kinds of address objects: **WebPageAddress**, **EmailAddress**, **TelecomAddress**, **GeographicAddress**, **PostOfficeBoxAddress**, and **GeneralDeliveryAddress**. They can be described as follows:

- A **WebPageAddress** represents the **URL** for a Web page related to the Party. This consists of a Uniform Resource Locator (URL) that locates a page on the World Wide Web. A URLs has the following form "www.domainName/resourceName" as String type where the resourceName is optional.

- An **EmailAddress** specifies a way of contacting a Party via email. Each **EmailAddress** has 5 String parts: userName, "@", domainName, ".", and TLD in the following form "username@domainName.TLD" (where Top Level Domain (TLD) can be com, org, gov, etc.).

- A **TelecomAddress** represents a number that can contact a telephone, mobile phone, fax, pager, or other telephonic device. The International Telecommunication Union (www.itu.int) provides standards for TelecomAddresses. Each TelecomAddress is made up of the parts shown below as an example:

+1 (0)208 1234567 ext. 789 mobile

The "+1" is the International Direct Dialing (IDD) for a country. Almost each country in the world has its own country calling code, it is a single- or up to three-digit number which you have to dial to make a call to that country. Note this country code is not the same as the ISO 3166 country code (see below). For example, "+1" represents all countries in North America, "+2" represents all countries in Africa, etc.

The "(0)" is the National Direct Dialing prefix (NDD).
The "208" is the area code.
The "1234567" is the number.
The "789" is the extension.
The "mobile" is the physicalType.

In summary, a **TelecomAddress** has a **countryCode** (the number you must use to direct dial a particular country), a **nationalDirectDialingPrefix** (the prefix to make a call within a country between different cities or areas), an **areaCode** (the code for an area or city), a **number** (the telephone number), an **extension** accessible via the number, and a **physicalType** (the type of device, such as phone, fax, mobile, pager, and so on). The area code and the extension are of integer type; the number is a long integer type. The rest of the parts are String types.

- A **GeographicAddress** represents a geographic location at which a Party may be contacted. It is a postal address for the Party. Each **GeographicAddress** has an **addressLine** (String type), a **city** (String type), a **regionOrState** (String type), a **zipOrPostCode** (String type), and a **Locale** (Class type).
    - ⇨ A **Locale** is inspired by the ISO 3166 standard (see the note on ISO 3166 below) and has an alphabetic two-letter country code (String type), a numeric country code (int type), and official English language country name (String type). For example:

Canada Locale is: "CA" 124 "Canada"
USA Locale is: "US" 840 "The United States of America"
Iran Locale is: "IR" 364 "The Islamic Republic of Iran"
India Locale is: "IN" 356 "The Republic of India"
Etc.

Note on ISO 3166: The International Organization for Standardization (ISO) has created and maintains the ISO 3166 standard. This standard uses codes for the representation of names of countries and their subdivisions. The ISO 3166 standard contains three elements (simplified for this assignment):
Element 1 is the two-letter country code (String type).
Element 2 is the three-digit country code (int type).
Element 3 is the English language country name (String type).
(Ref. https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes)

− A **PostOfficeBoxAddress** (also known as P.O. box), is special type of a geographic address, that is uniquely addressable to a lockable box that is located on the premises (box lobby) of a post office station. In some regions, there is no door to door delivery of mail. The lockable box is accessible after hours by customers with a code to the box lobby's door keypad. Each **PostOfficeBoxAddress** has an **addressLine** (String type), a **city** (String type), a **regionOrState** (String type), a **zipOrPostCode** (String type), a **Locale** (as described above), and a **boxLobbyDoorCode** (int type).

− A **GeneralDeliveryAddress** is a service where the post office holds the mail until the recipient calls for it. It is a common destination for mail for people who are visiting a particular location and have no need, or no way, of having mail delivered directly to their place of residence at that time. Each **GeneralDeliveryAddress** has an **addressLine** (String type), a **city** (String type), a **regionOrState** (String type), a **zipOrPostCode** (String type), and a **TelecomAddress** to call the service.

Part I:

1. Draw a UML representation for the hierarchy of the above-mentioned classes. Your representation must also be accurate in terms of UML representation of the different entities and the relation between them. You must use a software to draw your UML diagrams (no hand-writing/drawing is allowed).

2. Write the implementation of the above-mentioned classes using inheritance/composition and according to the specifications and requirements given below:

● You must have 3 different Java packages for the classes. The first package will include the **Address** class. The second package will include the **WebPageAddress**, **EmailAddress** and the **TelecomAddress** classes. The third package will include the **GeographicAddress**, **PostOfficeBoxAddress**, and **GeneralDeliveryAddress** classes.

● For each of the classes, you must have at least three constructors, a default constructor, a parameterized constructor (which will accept enough parameters to initialize ALL the attributes of the created object from this class) and a copy constructor. For instance, the parameterized constructor of the **TelecomAddress** class must accept 6 parameters to initialize the *international direct dialing*, the *national direct dialing*, the *area code*, the *number*, the *extension*, and the *physical type*.

● An object creation using the default constructor must trigger the default constructor of its ancestor classes, while creation using parameterized constructors must trigger the parameterized constructors of the ancestors.

● For each of the classes, you must include at least the following methods: accessors, mutators, ***toString()*** and ***equals()*** methods (notice that you are always overriding the last two methods).

− The ***toString()*** method must return a clear description and information of the object (For example: *"This email address dave.velop@cleancode.app is valid from 2018-01-31 to 2021-12-31 and therefore still usable today"*).

− The ***equals()*** method must first verify if the passed object (to compare to) is null and if it is of a different type than the calling object. The method would clearly return false if any of these conditions is true; otherwise the comparison of the attributes is conducted to see if the two objects are actually equal. Two objects are equal if all the values of all their attributes are equal.

● For all classes you **must** use the appropriate access rights, which allow most ease of use/access **without compromising security**. Do not use most restrictive rights unless they make sense!

● When accessing attributes from a base class, you must take full advantage of the permitted rights. For instance, if you can directly access an attribute by name from a base class, then you must do so instead of calling a public method from that base class to access the attribute.

3. Write a driver program (that contains the **main()** method) that would utilize all of your classes. The driver class can be in a separate package or in any of the already existing packages. Besides the main() method, create a method called **traceObsoleteAddresses**(). This method takes four parameters, an array of Addresses, and three integer values representing a date (YYYY, MM, DD). The method must search in the array of Addresses and display all addresses that are/were obsolete in comparison the passed date. For example, if the passed date is 2021, 02, 15 and an Address has date limits of 1998-05-21 and 2010-10-03, then this Address is obsolete. Again, you should notice that Addresses can have future dates (yet to become effective). For instance, if the passed date is 2021, 02, 15 and an Address has date limits of 2021-10-18 and 2023-4-20, then this Address is also obsolete in relation to the passed date.

4. In the main() method you must:

● Create various objects from the 6 classes, and display all their information (you must take advantage of the ***toString()*** method). Some of them must be obsolete addresses, limiting their usage only between **validFrom** and **validTo** dates.

● Test the equality of some of the created objects using the ***equals()*** method.

● Create an array of 15 to 20 these address objects (**HINT**: Do you need to add something else to the classes described above? If so; go ahead with that!) and fill that array with various objects from these classes (each class must have at least one entry in that array).

● Call the **traceObsoleteAddresses**() method with the array created above, along with any date of. The method should trace and display all information of all obsolete Addresses along with their location (index) in the array.

Part II

In that part, you need to modify/expand the implementation from Part I as follows:

1. In the driver program, you need to add another static method (add it above the main() method), called ***copyAddresses***. The method will take as input an array of these objects (the array can be of

any size) and returns a copy of that array. That is to say, the method needs to create an array of the same length as the passed array, copies all objects from the passed array to a new array, then returns the new array. Your copy of the objects **will automatically** depend on the copy constructors of the different listed classes. You **must** consider the following restrictions: **Do NOT attempt to explicitly find the exact type of the objects being copied, do NOT attempt to find the object type inside the copy constructors and Do NOT use clone().**

2. In the driver program, create an array of 15 to 20 objects (must have at least one from each of the classes), then call the *copyAddressObjects()* method to create a copy of that array.

3. Display the contents of both arrays, then add some comments indicating **whether or not the copying is correct. If not; you need to explain why it has not been successful or as you might have expected.**

General Guidelines When Writing Programs

- Include the following comments at the top of each class you are writing.

```
// -------------------------------------------------
// Part: (include Part Number)
// Written by: (include your name(s) and student ID(s))
// -------------------------------------------------
```

- Use **JavaDoc** to create the documentations of your program. Use appropriate comments when needed.
- Display clear prompts for the user whenever you are expecting the user to enter data from the keyboard.
- All outputs should be displayed with clear messages and in an easy to read format.
- End your program with a closing message so that the user knows that the program has terminated.

Submitting Assignment 2

- For this assignment, you are allowed to work individually, or in a group of a maximum of 2 students (i.e. you and one other student). Groups of more than 2 students = zero mark for all members! Submit only ONE version of an assignment. If more than one version is submitted the first one will be graded and all others will be disregarded.
- Students will have to submit their assignments (one copy per group) using Moodle. Assignments must be submitted in the right DropBox/folder of the assignments. **Assignments uploaded to an incorrect DropBox/folder will not be marked and result in a zero mark. No resubmissions will be allowed.**
- Naming convention for zip file: Create one zip file, containing all source files and produced documentations for your assignment using the following naming convention:

The zip file should be called *a#_StudentName_StudentID*, where # is the number of the assignment and *StudentName/StudentID* is your name and ID number respectively. Use your "official" name only - no abbreviations or nick names; capitalize the usual "last" name. Inappropriate submissions will be heavily penalized. For example, for the first assignment, student 12345678 would submit a zip file named like: *a1_Mike-Simon_123456.zip.* if working in a group, the name should look like: *a1_Mike-Simon_12345678-AND-Linda-Jackson_98765432.zip.*

- If working in a team, only one of the members can upload the assignment. Do **NOT** upload the file for each of the members!

IMPORTANT (Please read very carefully): Additionally, which is very important, a demo will take place with the markers afterwards. Markers will inform you about the details of demo time and how to book a time slot for your demo. If working in a group, both members must be present during demo time. Different marks may be assigned to teammates based on this demo.

- **If you fail to demo, a zero mark is assigned regardless of your submission.**
- **If you book a demo time, and do not show up, for whatever reason, you will be allowed to reschedule a second demo but a penalty of 50% will be applied.**
- **Failing to demo at the second appointment will result in zero marks and no more chances will be given under any conditions.**

Evaluation Criteria

Part I	
UML representation of class hierarchy	2 pts
Proper use of packages	1 pt
Correct implementation of the classes	1 pt
Constructors	1 pt
toString() & equals()	1 pt
Part II	
copyAddressObjects() and its behavior	2 pts
Driver program & general correctness of code	2 pts
Total	**10 pts**