CSC-240 Lesson 6: More on Classes and Objects

Assignment 6

Please do the exercise defined below. It requires you to define and use two regular Java classes and also to define and use one enumeration class. The classes are to be placed in the designated packages. You should submit exactly three files for the assignment:

- `Prefix.java` - the implementation of an enumeration class as defined below that will compile into the package `edu.frontrange.csc240.a6`.
- `Course.java` - the implementation of the **Course** class that will compile into the package `edu.frontrange.csc240.a6`.
- `Section.java` - the implementation of the **Section** class that will compile into the package `edu.frontrange.csc240.a6`.

For this exercise, you are being provided with the Java source code for a set of already-written classes defined in the packages `edu.frontrange.csc240.a6` and the `edu.frontrange.util` (see The Test Programs and Other Resources Provided below) . They are:

- `Date.java`: this is the class shown in Figure 8.7 in the textbook, re-formatted and re-documented, and with two extra methods (which you will not need for this exercise).
- `Time2.java`: this is the class shown in Figure 8.5 in the textbook, re-formatted and re-documented.
- `Student.java`: is a class that represents a student record within the Front Range Community College.
- `Degree.java`: is a class defining the target degrees towards which students (for this exercise) may be working. This class is used by the class **Student**, and serves as a template for the implementation of the **Prefix** class.
- `ObjectCounter.java`: You are not required to use this class beyond the assignment of an instance to a field variable, as described in the the *Student Class* topic, and also under the heading ***Counting Objects*** below.

As well, you are given the Java source code for two already-written classes defined in the package `edu.frontrange.csc240.a6.test` (see The Test Programs and Other Resources Provided below) . They are:

- `StudentTest.java`: which is a very simple test program, provided as an example, to demonstrate just some of the workings of the class **Student** (and note that the class **Student** also uses the classes **Date** and **Degree**). You are not required to do anything more with this test program.
- `Assignment6Test.java`: which tests the workings of the three classes **Prefix**, **Course**, and **Section** (the classes which you are to write). The intent is that once you have written the classes asked for above, you should then run this test program together with the classes you have written and the provided classes. (Note that the test program `StudentTest.java` may be run successfully soon as you have the files unpacked, yet the test program `Assignment6Test.java` can be run, but it will report errors until you have written the “missing” classes.)

All the supplied files should compile correctly and work correctly.

The classes that you write will need to use the three supplied classes, *i.e.*, the **Time2** and **Date** classes from the textbook, and the **Student** class from the Learning Module examples (which uses the **Degree** class). Note that all these classes are in the package `edu.frontrange.csc240.a6`.

The supplied classes are provided in a zip format file. These classes, as well as the compiled classes you create for this assignment, should reside in a four-tiered folder structure (edu/frontrange/csc240/a6). NetBeans will create these folders if you instruct it to create the package `edu.frontrange.csc240.a6`.

UML Specifications

Here are the UML specifications and the method descriptions for mostly the **public** members of the enumeration class and the classes that you are being asked to write.

Remember, additional **private** methods and other **private** members may be added to the class and used as you find them advisable or necessary. However, no new **public** methods should be added in your implementation.

In the UML, "+" means **public**, "-" means **private**, "#" means **protected**, and "~" means package-private (default access in Java).

The Prefix Enumeration

The following UML describes the **public** interface of the **Prefix** class. The constructor is mentioned as a reminder that the constructor of an **enum** class is normally **private.** You may implement this class using any additional **private** members as you require, but should not add to the **public** interface.

Prefix (enum)

+ <static> CIS: Prefix
+ <static> CNG: Prefix
+ <static> CSC: Prefix
+ <static> CWB: Prefix
- <constructor> Prefix(String prefixTitle)
+ getTitle(): String

Here are specific requirements for the **Prefix** enumeration:

- Each **Prefix** value represents one FRCC department, and is named for the prefix for that department.
- Each **Prefix** value has a three-letter name plus a corresponding **String-**type title.
- The four prefixes and their corresponding titles are:

  CIS Computer Information Systems
  CSC Computer Science
  CNG Computer Networking
  CWB Computer Web-based
- The **Prefix** constructor accepts a **String** as the title of the enumerated value.
- **Prefix** provides a *getTitle* method that returns the title as a **String** type value.

No validation is required for the values of **Prefix** or the course titles. You, as the programmer, are supplying this class as a given system resource, and there are no user interactions to validate and verify.

The Course Class

The following UML describes mostly the **public** interface of the **Course** class. You may implement this class using any additional **private** members as you require, but should not add to the **public** interface.

Course

- counter: ObjectCounter
- courseName: String
- courseNum: String
- credits: int
+ <constructor> Course(coursePrefix: Prefix, courseNum: String, courseName: String)
+ <constructor> Course(coursePrefix: Prefix, courseNum: String, courseName: String, credits: int)
+ getCredits(): int
+ setCredits(numCredits: int)
+ getDetails(): String
+ toString(): String

[It is not usual to specify **private** fields, but in the interests of making the test program more capable of helping with the evaluation of the assignment, it is requested that these **private** fields be present and be used.]

Here are specific requirements for the **Course** class:

- Each **Course** object represents one course, such as CSC-240 (where the three letters are a prefix, and the three remaining characters are a course number).

- The **Course** class will keep track of how many total **Course** objects are currently created (possibly zero) by use of the **ObjectCounter** class. Nothing further beyond declaring and initializing the *counter* is required.

- Each **Course** has properties for prefix, course number, course name and credits.

- The prefix must be a value of type **Prefix**. (No error checking is required here.)

- The "course number" is a three-character text **String**, not a number. It is possible for some course "numbers" to contain alphabetic values. Assume that any **String** that is not **null** and of the correct length (3 characters) is valid. If no valid **String** is supplied, the value for the course number should be set to an ***empty*** **String**, and no error message is produced.

- The course name is a **String** that is not **null**, and of any length greater than zero. If a **null String** or a **String** that is otherwise invalid is supplied, the value for the course name should be set to an ***empty*** **String**, and no error message is produced.

- Course credits are whole number values from 0 to 5. The value zero indicates that the value has not been set (or that it will be set later). Invalid values should be replaced by zero (to indicate the absence of a valid value), and no error message produced.

- The *getDetails* method will return a **String** properly displaying all four attributes in a format similar to:
  ```
  CSC-240 Java Programming 3 Credits
  ```
  The example output below uses *getDetails*. This method is not required to give a correctly formatted string if any of the attributes used are invalidly set. It is a **String** not terminated by a `line.separator`.

- The *toString* method should return just the prefix, a hyphen and the course number as in "`CSC-240`". This method is not required to give a correctly formatted **String** if any of the attributes used are invalidly set.

The Section Class

The following UML describes mostly the **public** interface of the **Section** class. You may implement this class using any additional **private** members as you require, but should not add to the **public** interface.

Section

- counter: ObjectCounter
+ <constructor> Section(course: Course, sectionNumber: String)
+ addStudent(thisStudent: Student)
+ getDetails(): String
+ getRoster(): String
+ getRosterCount(): int
+ getSectionNumber(): String
+ setDates(startDate: Date, endDate: Date)
+ setTimes(startTime: Time2, endTime: Time2)
+ toString(): String

Here are specific requirements for this first version of the **Section** class:

- Each **Section** object represents one section. Each Section is associated with just one **Course** as given in the constructor. (No checking is required on this value.)

- The S**ection** class will keep track of how many total S**ection** objects are currently created (possibly zero) by use of the **ObjectCounter** class. Nothing further beyond declaring and initializing the *counter* is required.

- Assume each section is a classroom section that meets on a regular basis.

- Each section has properties for its section number, start/end dates (in the semester), and start/end times (daily), using data types as indicated by the UML.

- The “section number” is a three-character **String** (not **null**) that may include alphabetic characters. Assume any value of the correct length is valid. If a **null String** or a **String** that is otherwise invalid is supplied, the value for the section number should be set to an ***empty*** **String**, and no error message produced.

- Each section can have up to 30 students. (You will need to store these values in an internal array-like data structure capable of holding elements of type **Student**.)

- The *addStudent* method accepts a **Student** object (a **null** reference is not a **Student**) and adds the student to the class if it is not full. If the student cannot be added to the section because it is full, the student is simply not added (no error message is required).

- The *getRoster* method returns a **String** that has one line per student in the course. The text on the line describing the student may be obtained using the *toString* method of the **Student** class. The lines of this roster **String** should be separated by the system-defined “`line.separator`” **String** (see the *getProperties* and *getProperty* methods of the **System** class).

- The *getRosterCount* method returns the number of students currently added to the section, possibly zero.

- The *getSectionNumber* method returns a **String** (which may include an ***empty*** **String** if no valid *sectionNumber* was provided) giving the current value of the *sectionNumber* property.

- The *getDetails* method returns a **String** that shows the properties of the section in a format similar to:

  ```
  Section: CSC-240-001
  Course: CSC-240 Java Programming 3 Credits
  Dates: 8/21/2007 to 12/10/2007
  Times: 10:00:00 AM to 11:15:00 AM
  Enrollment: 5
  ```

  Each line of this details **String** should be separated by the system-defined “`line.separator`” **String** (see the *getProperties* and *getProperty*, or the *lineSeparator* methods of the **System** class).

  The *getDetails* method is used for the details of each section. This method is not required to give a correctly formatted string if any of the attributes used are invalidly set.

- The *toString* method returns a **String** that has the **Course** *toString* value, a hyphen and the section number, as in “`CSC-240-500`”. This method is not required to give a correctly formatted string if any of the attributes used are invalidly set.

Counting Objects

In the Assignment for this Lesson, and the next, the test programs need to have access to a count of the number of objects created of the **Student**, **Course**, and **Section** types. To do this for each such class, an instance of type **ObjectCounter** is needed as a field. You will see an example of the needed code in the listing of the **Student** class. The instruction for implementing this counter in the each of the such classes are to create a field variable in the form of:

```
private final ObjectCounter counter = new ObjectCounter(getClass());
```

You may wish to simply copy and paste this element into the correct place in your code. The **ObjectCounter** class is provided with the other materials for the Assignment.

The Test Programs and Other Resources Provided.

As you develop these classes you will likely want to use some simple test programs of your own. But the assignment will be graded against the test program `Assignment6Test.java`, which you will find in the zip file (see below). Note that, as usual, your program will not be graded just on “getting the right answer,” but also on the structure of the program, use of the usual Java conventions, and the presence of useful commentary explaining your code.

All of the classes, in their respective files, necessary for the exercise are available in this zipfile. Additionally, there is the second test program, **StudentTest**, that is a test program for just the **Student** class. This is provided as an additional example. The zipfile also contains all the Javadoc for all the classes provided. The Javadoc may be compared to the UML in order to assist in understanding the programming interfaces that are to be implemented.

The same zipfile also provides the class **ObjectCounter**, which is needed to count objects (see *Counting Objects* above).

Once you have implemented the three class as requested above, you should be able to run the test program for Assignment 6. An example of output from this program is shown in the “About the Assignment” topic.

The following *About the Assignment* topic contains further explanation, directions, and hints about this assignment. Please read that topic before starting this assignment.

Important

Please follow the directions of Programming Assignment Identification in submitting your programming solutions.

If you have any questions or concerns about the Assignment or the *About the Assignment* topic, please use the Lesson Question discussion topic, or send a message by the D2L Internal Messaging system if you prefer.

Note: you may submit to the Assignment Submission Folders as many times and as often as you wish, up to the deadline time. Each submission is tagged with the date/time, and so each submission remains separate and distinct. Unless you leave instructions to the contrary, only the most recent of each file with the same name will be viewed for the purposes of grading. Details may be found in the topic *How To Submit and Get Feedback on Assignments* in the *How To* module.

Messages that accompany Assignment Submissions are read, and responded to, ***only*** when assignment submissions are graded (which is after the Assignment Submission Folder closing date/time). If you have a comment or question about an assignment, or a request for assistance, that needs an earlier response, then that comment, question or request should be made or asked *via* an Internal Message or the Discussion board, as these are usually read and answered every day.