**HW 2: More Objects and Classes**

**Liang Chapters 9 and 10**

**Out 9/25/16**

**Due by 10/5/16 (later is OK)  
10 points**

**Foundations of Software Development  
SEIS 601.02**

Fall Semester, 2016

Eric V. Level, Professor

Download and import the zipped Eclipse project **602\_hw\_2.zip** from Class 3 on our Blackboard site.

This project contains starting Java code for the problems below. Follow the instructions below, completing each requested program.

When you are finished, submit by doing **either** of the following:

1. Export your entire project to **601\_hw\_2**, and export it to a .**zip** file. Then email it to me as an attachment.

**OR**

2. Attach each individual **.java** file to an email to me. (Include all files in the project - even those you don't create yourself.) You can find these within the workspace's project folder (same name as project), inside the **src** subfolder.

In either case, put **602 - HW 2** in the Subject: line of your email. Then send it to **evlevel@stthomas.edu.**

Please submit only one HW per email. Don't submit both Labs and HW in the same email.

And remember: each student should submit their **own** work for this homework assignment.

Each of the following problems is worth a maximum of 2 points. Help videos are posted for each of the following problems, to help you get started on them. They're under the new **Help Videos** section on Bb.

[1] (**QueueOfInteger.java**) Do Problem 10.10 in our Liang textbook. You don't have to draw a UML diagram. The provided code begins this implementation, except that it uses an array of **Integer** (**Integer[]**) to hold both **int** and **Integer** values. Otherwise the provided fields are the same as Chapter 10's **StackOfInteger** implementation; study this to get ideas on how to implement the requested methods **enqueue(int)**, **dequeue()**, **empty()**, and **getSize()**.

Also add Javadoc comments immediately before each of your methods, briefly describing it does. Your **QueueOfInteger.main()** method should implement the problem's test program; use a **for**-loop to generate your 20 numbers.  
  
[2] (**Grade.java**) Implement a class **Grade.java** that represents a course grade for our 602 class. Provide **double** array fields **rdQ[12]** and **rvQ[12]** storing the scores for Reading Quizzes and Review Quizzes, with **int** fields **numRdQ** and **numRvQ** indicating how many such quizzes have actually been taken. Similarly **double labs[12]** and **int numLabs** represent your Lab scores, and **double hw[12]** and **int numHw** your Homework scores. Finally, fields **double project** and **double attendance** will store your Final Project and Attendance scores. Note that the provided code declares all of the above instance fields. You will still need to initialze each inside your constructors.

Implement **public** instance methods **addRdQ(double score)**, **addRvQ(double score)**, **addLab(double score)**, **addHw(double score)**. Each of these should add an additional score to the appropriate array, then increment the count for that array.

Assume that each of the Reading Quizzes, Labs and Homework are worth a maximum of 10 points each. Review Quizzes are worth 15 points each, the Project is worth 100 points, and Attendance is worth 14 points. Provide constants in your code to represent each of these maximum values. Provide a public instance method **calcGradePct()** which calculates and returns the **double** weighted average of all stored grades, with weights as found in our Course Syllabus. Also provide **calcCourseGrade()**, which returns a **String** letter grade as per the curve, also as found in the Syllabus. You should sum only those RdQ, RvQ, Lab, and HW scores that have been entered.

Finally, provide a **Grade.main()** method to test your implementation: create a **Grade** object, then add to it each of your three Reading Quiz and first Review Quiz scores (all available after Class 4), as well as your three Lab scores and first HW score. Add perfect scores for your Project and Attendance, then print out both the grade percentage and the calculated final course grade. I'll be grading Labs 1-3 and Homework 1 this coming week, so you'll have values for these to enter.

[3] (**Zip.java**) Implement a class **Zip** that represents the USPS ZIP code. A **Zip** object should be able to represent either the original 5-digit ZIP format, as well as the newer and optional ZIP+4 format (which includes a hyphen and 4 additional digits). Examples are **54747** and **55105-1048**.

Provide a **Zip(String s)** constructor with **s** initializing your two **private** fields **String zipCode** and **boolean isPlus4**, with the latter indicating if the format of **zipCode** is of the newer variety. Also provide a default constructor (no arguments) which sets the **zipCode** to **00000** and **isPlus4** to **false**. Your first constructor should also validate the format of parameter **s**. That is, if **s** is a valid original format zip code, set **isPlus4** to **false**; otherwise, if it's a valid new ZIP+4 format, set **isPlus4** to **true**. If it isn't either of these valid formats, then set the **zipCode** to **00000** and **isPlus4** to false.

Also provide **public** instance methods **getZipCode()** to return the stored zip code, whatever its format; **getZipPrefix()** to return only the first five digits of the stored code if **isPlus4==true** or just the zip code otherwise; and a **toString()** method which returns the entire **zipCode**. Finally, provide your own version of **public boolean equals(Zip code)**, which returns **true** for **Zip** instances **a** and **b**, when they represent the same ZIP, no matter if they differ in formats or not: **a.equals(b)**.

Then provide a **main()** method for **Zip**. This should create three new **Zip** objects: one with the older 5-digit format, one with the newer 5+4 format but with the same prefix as the first, and one which is initialized with an invalid **String** (and which will get the value of **00000** as a result). Invoke each of your instance methods against each of your three instances, and print out the results - except for **equals()**. Use it to test the equality of the first two zip codes, printing out the result. Then test **equals()** for each valid **Zip** against the **00000** **Zip**, again printing out the result.

[4] (**Gene.java**) Write a program that reads a **String** from the user representing a DNA molecule as a sequence of letters **A**, **C**, **G**, and **T**, each letter representing a different base. Then determine whether it represents a potential gene, based on the following criteria:

* It begins with the start codon **ATG**.
* Its length is a multiple of 3.
* It ends with one of the stop codons **TAG**, **TAA**, or **TGA**.
* It has no intervening stop codons.

The starting code for this problem has a **main()** method that reads the input **String** using **StdIn.readLine()**, and also provides a method **isValidDNA(String str)**, which returns **true** if all characters in **str** are one of **A**, **C**, **G**, and **T**. Otherwise return **false.**

Finish this **main()** implementation. You should validate the input **String** and print **Not valid DNA** if it contains any characters other than one of **A**, **C**, **G**, and **T.** Otherwise check it and print **Is potential gene** if it satisfies the previous criteria, and **Is NOT potential gene** otherwise.

[5] (**Koch.java**) A Java class **Turtle.java** is provided, which implements simple **turtle graphics**. It uses the **stdlib** library that's included with the provided code. For a description of turtle graphics: **https://en.wikipedia.org/wiki/Turtle\_graphics**.

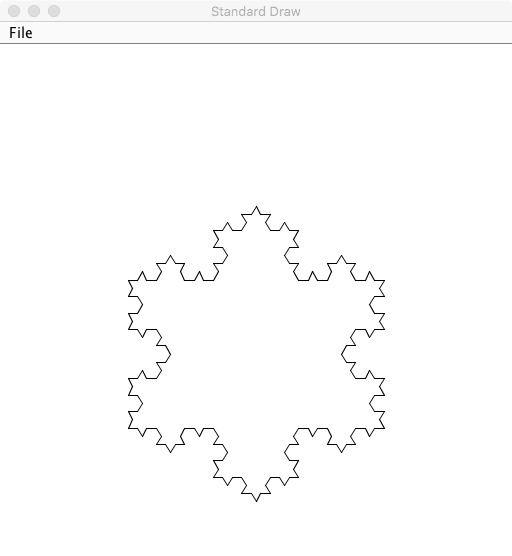
The default coordinate system of **Turtle** has the lower left corner at **(0.0,0.0)** and the upper right at **(1.0,1.0)**. The Turtle constructor **Turtle(x0,y0,head)** puts the turtle at coordinates **(x0,y0)**, with an initial heading of **head** degrees counterclockwise from the x-axis and set the state of the turtle's "drawing pen" as "up". For a turtle reference **t**, **t.goForward(dist)** moves the turtle forward a distance of **dist**, leaving a trail of ink behind it if the pen is down; otherwise, it just changes its location. **t.penUp()** and **t.penDown()** change the turtle's drawing state, and **t.turnLeft(deg)** changes the turtle heading by **deg** degrees in the counterclockwise direction from the current heading.

**Turtle.main()** draws an example triangle using this **Turtle** API.

**Koch.java** is also provided, drawing the famous recursive Koch "snowflake curve" along the x-axis at the bottom of the drawing area. Historically, this was one of the first mathematical examples of a **fractal**, which is a curve with a fractional dimension. See: **https://en.wikipedia.org/wiki/Koch\_snowflake**.

The recursive method **koch(int n, double step, Turtle turtle)** draws one side of the Koch curve, with **n** as the order of the curve (recursive depth), **step** as the distance turtle moves when drawing the curve, and **turtle** as the **Turtle** object that does the actual drawing.

Run the provided **Koch.main()** to see how it draws an order 3 curve, then try setting **order** as both greater than and less than 3, running it multiple times to see the effect. Finally, modify this code so it draws something like the following picture (which is a more realistic snowflake):



Hint: Call **koch()** three times, turning the turtle the same angle after each call. Note that **koch()** preserves the turtle's heading: it's the same after the call as before.

**Extra Credit (2 points each)**

Each of the following is optional. Submit one or more of these within your Eclipse project above. Name each .**java** file as indicated.

[5] (**Palindrome2.java**) Write an application which checks if an input **String** is a palindrome. Do so by using **StackOfInteger** modified to **StackOfChar**, and similarly for your **QueueOfInteger** class, modified to handle individual **char** and named **QueueOfChar**. You should iterate over each **char** of the input **String**, pushing it both onto a stack and adding it to a queue. Then examine both data structures, popping and dequeing to perform the palindrome check, printing out the result.

[6] (**Zip2.java**) Extend your **Zip** class above by adding a method **getState()** which returns the two letter abbreviation of the state corresponding to the represented ZIP code. This code should be the standard two capital letters abbreviation for the state (Examples: **MN** for Minnesota, **ME** for Maine). For details: **https://en.wikipedia.org/wiki/ZIP\_Code**

[7] (**RecursiveArt.java)**Using the **Turtle** class above, draw some interesting recursive picture. You must implement some method which is recursive (like **koch()** in the above problem), with your drawing within this method. You might draw something like the following: