COMP 2522 Object oriented programming 1
Assignment 2

Christopher Thompson
cthompson98@bcit.ca

BCIT CST — **Due Wednesday February 12th 2020 at or before 23:59:59**

Introduction

Hooray! It took a few late nights, but you met your first milestone. You successfully demonstrated the first iteration of the simulation software to the ichthyologists and they are eager for you to finish.

The scientists have returned from their final fact-hunting expedition. They have given you the data you need to finish the program. And it's a rush job – they need a finished simulation in about two weeks! It's time to complete your contract and give the whole project a little bit of polish. You are ready to begin the next iteration. Did you have any idea that you'd be writing full-fledged Java programs after only a few weeks?

For assignment 2, you will implement a class that represents a hot spring Pool and 'manages' a collection of resident Guppy objects. You will make minor additions to the Guppy class, and create one Ecosystem class to rule them all. Let's begin!

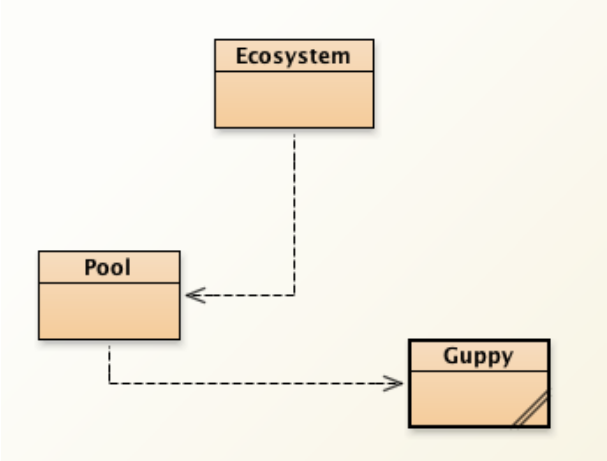

Figure 1: *Guppy Simulation class diagram (Proposed design)*

1 Submission requirements

This assignment must be completed by Wednesday February 12th 2020 at or before 23:59:59.

Your code must all be in the ca.bcit.comp2522.assignments.a2 package of your COMP 2522 IntelliJ project.

We will mark the code as it appears during the final commit before the due date and time.

2 Grading scheme

Your assignment will be marked out of 10:

1. 2 points for code style
2. 2 point the modifications to the Guppy class
3. 3 points for the Pool class
4. 3 points for the Ecosystem class

3 Implementation requirements

Implement the following:

1. Copy the **Guppy.java** and **GuppyTest.java** source files to the a2 package.
2. It's time to implement reproduction for the Guppies! You must implement a method called **public ArrayList<Guppy> spawn()** in the **Guppy.java** class. Here's how it must work:
   (a) Only female Guppies can have babies
   (b) Female Guppies must be 8 weeks of age or older to spawn
   (c) Each week, each female Guppy who is 8 weeks old or older has a 50 percent chance of spawning (hint: use Random)
   (d) A spawning female has between 0 and 100 fry (Note: Guppies don't lay eggs, they have live babies called fry! Really!)
   (e) Each fry is the same genus and species as its mother
   (f) Each fry has a 50 percent chance of being female
   (g) A new fry has a health coefficient equal to (1.0 + mother's health quotient) / 2.0
   (h) The generation number of a new fry is 1 + the mother's generation number.

   Here's a good way to approach this. The method should begin by checking if the Guppy is female. If not, or if the Guppy is 0 - 7 weeks old, return null immediately. Otherwise create a local ArrayList<Guppy> called babyGuppies. Use a Random object to generate a double between 0 and 1.0. If the random number is equal to or less than 0.50, the female will spawn some babies. If the female is going to have babies, use the Random object to generate a random int between 0 and 100 inclusive for the female. Use a loop statement to create this number of Guppies (0 - 100) with the correct attributes, and add them to the babyGuppies ArrayList. After the loop is done, return the newly created ArrayList. That's all.
3. You must create a class called **Pool.java** which contains the following elements:
   (a) The following public **symbolic constants**. Use these variable names and data types. Remember symbolic constants are **static** and **final**. Do not use different names or data types. Use these constants instead of magic numbers inside your code:
      i. **DEFAULT_POOL_NAME** a String equal to "Unnamed"
      ii. **DEFAULT_POOL_TEMP_CELSIUS** a double equal to 40.0
      iii. **MINIMUM_POOL_TEMP_CELSIUS** a double equal to 0.0
      iv. **MAXIMUM_POOL_TEMP_CELSIUS** a double equal to 100.0
      v. **NEUTRAL_PH** a double equal to 7.0
      vi. **DEFAULT_NUTRIENT_COEFFICIENT** a double equal to 0.50
      vii. **MINIMUM_NUTRIENT_COEFFICIENT** a double equal to 0.0
      viii. **MAXIMUM_NUTRIENT_COEFFICIENT** a double equal to 1.0.
   (b) The following 8 private **instance variables**. Use these variable names and data types. Do not use different names or data types, and do not add any other instance variables:

i. **name** a String
ii. **volumeLitres** a double
iii. **temperatureCelsius** a double
iv. **pH** a double that will always be between 0 and 14.0 inclusive
v. **nutrientCoefficient**, a double that will always be between 0 and 1.0 inclusive
vi. **identificationNumber** an int
vii. **guppiesInPool** an ArrayList of Guppy
viii. **randomNumberGenerator** a Random object.

(c) **numberOfPools** a private static variable with an initial value of 0 that is incremented by 1 each time a new Pool is constructed. Further details are provided in the section about constructors (remember that static variables are shared by all objects of a class).

(d) **Two (2) constructors**:

i. **Zero-parameter** constructor which sets **volumeLitres** to 0.0, and also sets:
   A. **name** is set to **DEFAULT_POOL_NAME**
   B. **temperatureCelsius** is set to **DEFAULT_POOL_TEMP_CELSIUS**
   C. **pH** is set to **NEUTRAL_PH**
   D. **nutrientCoefficient** is set to **DEFAULT_NUTRIENT_COEFFICIENT**
   E. **guppiesInPool** is initialized to be an empty ArrayList of Guppy, and **randomNumberGenerator** is initialized to be a new Random (use the zero-parameter constructor in the Random class).

ii. **Multi-Parameter** constructor which accepts a parameter for each instance variable (except the identification number, the ArrayList of guppies, and the random number generator) and validates it:
   A. The parameter passed for the **name** field must not be null, and must not be an empty String or a String composed only of whitespace. If the argument passed as the parameter is either null or an empty/whitespace String, throw an IllegalArgumentException. Otherwise, format the value correctly (remove the whitespace, capitalize the first letter, set the rest to lower case) before storing it in the instance variable.
   B. The parameter passed for the **volumeLitres** field must be positive, otherwise set the instance variable to 0.0.
   C. The parameter passed for the **temperatureCelsius** instance variable must be greater than or equal to **MINIMUM_POOL_TEMP_CELSIUS** and less than or equal to **MAXIMUM_POOL_TEMP_CELSIUS**, else the instance variable must be set to the default temperature.
   D. the parameter passed for the **pH** field must be within the range [0, 14.0], otherwise set the instance variable to **NEUTRAL_PH**.
   E. the parameter passed for the **nutrientCoefficient** must be within the range [0, 1.0] (between 0 and 1.0 inclusive), otherwise set the instance variable to **DEFAULT_NUTRIENT_COEFFICIENT**.
   F. **guppiesInPool** is initialized to be a new ArrayList of Guppy.
   G. **randomNumberGenerator** is initialized to be a new Random (use the zero-parameter constructor in the Random class).

iii. Both constructors must create and assign a unique numerical identification number to the **identificationNumber** instance variable. A good way to do this is to increment the static counter field from inside the constructor, and then assign that newly incremented value to the new Pool object's identification field.

(e) Implement **accessors** (getters) and **mutators** (setters). Create accessors for name, volumeLitres, temperatureCelsius, pH, nutrientCoefficient, and identificationNumber. We do not want to create an accessor for the ArrayList of Guppy because we want to keep that collection encapsulated and protected from tampering. Create mutators for volumeLitres. temperatureCelsius, pH, nutrientCoefficient. All mutators should ignore values that are unacceptable. For example. the pH

mutator must ignore input that does not fall within the range [0, 14], and the nutrientCoefficient mutator should ignore values that do not fall within the range [0, 1.0]. Instance variables which do not have mutators should be final.

(f) a method with the header **public void changeNutrientCoefficient(double delta)** which adds the value passed in the `delta` parameter to the Pool's nutrient coefficient. The parameter may be positive or negative, but if the new value of the nutrient coefficient would be less than the minimum, set the nutrient coefficient to the minimum. If the new value would be greater than the maximum, set the new value to the maximum.

(g) a method with the header **public void changeTemperature(double delta)** which changes the temperature by the amount passed in the `delta` parameter. If the new Pool temperature would be less than the minimum, set the pool temperature to the minimum. If the new Pool temperature would be greater than the maximum, set it to the maximum.

(h) a method with the header **public static int getNumberCreated()** which returns the total number of Pools created.

(i) a method with the header **public boolean addGuppy(Guppy guppy)** which adds a Guppy to the Pool. If the parameter equals null, the method should return false. If the parameter is not equal to null, add the Guppy to the ArrayList of Guppy. If the Guppy is successfully added, return true, else return false.

(j) a method with the header **public int getPopulation()** which returns the number of living Guppies in the pool.

(k) a method with the header **public int applyNutrientCoefficient()** that calculates which Guppies in the Pool have died of malnutrition this week, and returns the number of deaths. Iterate over the ArrayList of Guppy in the Pool using the iterator returned by the ArrayList's iterator( ) method. For each Guppy, generate a different random number between 0.0 and 1.0 inclusive using the Random method `nextDouble()`. If this randomly generated number is greater than the Pool's nutrient coefficient, kill that Guppy by setting the appropriate boolean field in the Guppy. Note that this method does not remove any dead Guppies from the Pool, it just kills them. Do not do anything else.

(l) a method with the header **public int removeDeadGuppies()** which uses an ArrayList iterator and a loop to visit each Guppy in the collection and remove the Guppies that are not alive. This method must keep track of how many Guppies are removed. Return the number of Guppies that have been removed.

(m) a method with the header **public double getGuppyVolumeRequirementInLitres()** which returns the total number of litres required by all of the living Guppies in the Pool. Note that the Guppy class has a helpful method with returns the total number of milliLitres required per Guppy. You will need to convert this to litres (note that 1,000 mL = 1 L).

(n) a method with the header **public double getAverageAgeInWeeks()** which returns the average age of the living Guppies in the Pool.

(o) a method with the header **public double getAverageHealthCoefficient()** which returns the average health coefficient of the living Guppies in the Pool.

(p) a method with the header **public double getFemalePercentage()** which returns a double representing the percentage of living Guppies in the Pool that are female.

(q) a method with the header **public double getMedianAge()** which returns median age of the living Guppies in the Pool.

(r) a method with the header **public String toString()** which returns a String representation of the Pool. The version generated by IntelliJ is sufficient.

(s) Implement a method called **public int spawn()**. This method must Iterate over the Guppies in the Pool and invoke the spawn method on each one. Each Guppy will return null (if it is male, or too young to reproduce) or an ArrayList of Guppies containing the new fry. Add the new baby Guppies to the Pool's collection of Guppies. You can add each mother's new fry to the Pool like this: `guppiesInPool.addAll(newBabies);`. Count the total number of new Guppies that have been born and added to the Pool, and return the total.

(t) Implement a method called **public int incrementAges()**. This method increments the ages of every Guppy in the Pool and returns the number that have died of old age. **Do not remove the dead Guppies from the collection, just kill them**. The dead Guppies still need to be removed from the system.

(u) Implement a method called **public int adjustForCrowding( )**. This Pool method extinguishes the Guppies that have suffocated due to overcrowding. If the total volume needed by a collection of Guppies exceeds the volume of their Pool, then some of the Guppies must be extinguished. If the total volume required by the Guppies in the Pool exceeds the total volume of the Pool, then we must loop through the Guppies in the Pool and terminate the weakest, e.g., the Guppy with the lowest health coefficient. We must do this again and again, and keep terminating the weakest Guppy in the Pool until the total volume requirement of the group of Guppies that remain alive is equal to or less than than the volume of the Pool. Note that the crowded out Guppies are only terminated. They are not removed from the Pool. Return the total number of Guppies that have died due to crowding.

4. **Implement a new class called Ecosystem**. Ecosystem contains and drives the simulation and requires the following elements:

(a) One private instance variable, **private ArrayList<Pool> pools**. Do not add any other instance variables.

(b) Zero-parameter constructor that must initialize the ArrayList of Pools. Do not create a one-parameter constructor.

(c) Implement a method called **public void addPool(Pool newPool)** which adds the Pool passed as a parameter to the Collection of Pools. Ignore null parameters, e.g., do not add any nulls to the Pool collection.

(d) Implement a method called **public void reset( )** that resets the Ecosystem by emptying the ArrayList of Pool (hint: check out the clear() method available to the ArrayList).

(e) Implement a method called **public int getGuppyPopulation( )** which returns the total number of living Guppies in the Ecosystem. Hint: use a for each loop to invoke the getPopulation() method on each pool and add the result to a local accumulator variable.

(f) Implement a method called **public int adjustForCrowding( )** (same name as the Pool method!) which iterates over all the Pools, invokes each Pool's **adjustForCrowding( )** method, and adds the number of Guppies that have been crowded out to a running total. Once all the Pools have been "de-crowded," adjustForCrowding should return the total number of Guppies that have been squeezed out of the entire ecosystem.

(g) Implement a method called **public void setupSimulation( )**. This method should create the following Ecosystem:

i. The first pool is called Skookumchuk. It has a total volume of 3,000 litres. Its temperature is 42 degrees Celsius and its pH is 7.9. The nutrient coefficient is 0.9.

ii. The second pool is called Squamish. It has a total volume of 15,000 litres. Its temperature is 39 degrees Celsius and its pH is 7.7. The nutrient coefficient is 0.85.

iii. The third and final pool is called Semiahmoo. It has a total volume of 4,500 litres. Its temperature is 37 degrees Celsius and its pH is 7.5. The nutrient coefficient is 1.0.

iv. There are 300 Guppies in Skookumchuk Pool. They are all Poecilia reticulata. Use your Random object and the multi- parameter Guppy constructor to create each new Guppy with an age between 10 and 25 weeks inclusive and a health coefficient between 0.5 and 0.8 inclusive. Each original Guppy has a 75 percent chance of being female.

v. There are 100 Guppies in Squamish Pool. They are all Poecilia reticulata. Use your Random object and the multi-parameter Guppy constructor to create each new Guppy with an age be- tween 10 and 15 weeks inclusive and a health coefficient between 0.8 and 1.0 inclusive. Each original Guppy has a 75 percent chance of being female.

vi. There are 200 Guppies in Semiahmoo Pool. They are all Poecilia reticulata. Use your Random object and the non-default Guppy constructor to create each new Guppy with an age between 15 and 49 weeks inclusive and a health coefficient between 0.0 and 1.0 inclusive. Each original Guppy has a 35 percent chance of being female.

(h) Implement a method called **public void simulate(int numberOfWeeks)**. It uses a loop to invoke the next method, `simulateOneWeek`, the specified number of times.

(i) Implement a method called **public void simulateOneWeek( )** that runs the simulation for one week. This method must:

i. Invoke **incrementAges( )** on each Pool and add the return value to a local int called **diedOfOldAge**.

ii. Invoke **removeDeadGuppies( )** on each Pool and add the return value to a local int called **numberRemoved**.

iii. Invoke **applyNutrientCoefficient( )** on each Pool and add the return value to a local int called **starvedToDeath**.

iv. Invoke **removeDeadGuppies( )** again (second time) on each Pool and add the return value to the local int called `numberRemoved`.

v. Invoke **spawn( )** on each Pool and add the return value to a local int called **newFry**.

vi. Invoke **adjustForCrowding( )** and assign the return value to a local int called **crowdedOut**.

vii. Invoke **removeDeadGuppies( )** again (third time!) and add the return value to **numberRemoved**.

viii. Check that **diedOfOldAge** + **starvedToDeath** + **crowdedOut** == **numberRemoved**. If they are not equal, you have a logic error.

ix. Print in a lovely and easy-to-read format:

A. Week number

B. Number of deaths this week due to old age

C. Number of deaths this week due to starvation

D. Number of deaths this week due to overcrowding

E. Number of births (new fry) this week

F. List of pools and their current populations at the end of the week

G. Total population of the Ecosystem at the end of the week

5. **Create a Driver class**. The Driver class should contain a main method, nothing else. Inside the main method, instantiate an Ecosystem object. Ask the user how many weeks they would like to run the simulation, and then invoke the Ecosystem object's **public void simulate(int numberOfWeeks)** method passing in the user's choice.

6. **A bonus of up to 10% is available for submissions that do one more thing**. Modify the setupSimulation method in Ecosystem. Modify the Pool temperatures, pHs, and nutritional coefficients so that the populations in the Pool eventually reach a thriving balance point. I define a thriving balance point as a combination of environmental factors that result in a steady population where the number of deaths equals the number of births and the median health coefficient of the Guppies in the Ecosystem is greater that 0.6.

That's it! Good luck, and have fun!