COMP 2522
Object Oriented Programming
Lab 2

Yue Wang
`yue_wang@bcit.ca`

Due before noon on Sunday, January 22, 2023

1 Introduction

Welcome to your second COMP 2522 lab.

This week you will design a system using UML and then implement a simple game for me. I don't think you can complete this during the two-hour lab – my hope is you will work on this for an hour or two each day. Fluency comes from using a language, and I'd like you to start using Java more often. Take your time. Help each other. Ask me lots of questions and ask Akila lots of questions in lab!

Good luck!

2 Submission requirements

This lab must be submitted to D2L at or before 11:59:59 AM on Sunday Januaray 22, 2023.

Don't wait until the last minute as you never know what might happen (computer issues, no Internet, power outage, ...). Submit often, and please make sure your comments are short and clear and specific.

You must not ignore Checkstyle warnings or the code style warnings produced by IntelliJ. **You must style your code so that it does not generate any warnings.**

When code is underlined in red, IntelliJ is telling us there is an error. Mouse over the error to (hopefully!) learn more.

When code is underlined in a different colour, we are being warned about something. We can click the warning triangle in the upper right hand corner of the editor window, or we can mouse over the warning to learn more. If the warning seems silly, tell me about it, and I will investigate and possibly ask the class to modify some settings to eliminate it once and for all!

3 Grading scheme

Figure 1: This lab, like all labs, will be graded out of 5

Your lab will be marked out of 5. On a high level, expect that marks will be assigned for:

1. (3 point) Meeting the functional requirements in this lab, i.e., does the code do what it is supposed to do?
2. (1 point) Meeting non-functional Java OOP design requirements such as encapsulation, information hiding, minimizing mutability, maximizing cohesion and minimizing coupling, making good choices about associations and thinking about Demeter, etc.
3. (1 point) Writing code that is self-documenting and well-styled that generates no warnings from IntelliJ or Checkstyle, with correct and complete Javadocs, using short and atomic methods correctly encapsulated in classes that are well defined.

4 Style Requirements

Continuing with this lab, there are code style requirements that you must observe:

1. Your code must compile.
2. Your code must not generate any IntelliJ code warnings or problems.
3. Your code must execute without crashing.
4. Your code must not generate any Checkstyle complaints (unless they are complains I have specifically said you can ignore).
5. Don't squeeze your code together. Put a blank space on either side of your operands, for example. I will be assessing the readability and clarity of your code.
6. All of your program classes must be in package ca.bcit.comp2522.xxx (replace xxx as required by the assignment, lab, quiz, etc.). For example, today's work should go in the ca.bcit.comp2522.labs.lab0X package.
7. All classes require Javadoc comments including the @author tag AND the @version tag. Class comments go after the package and import statements. There should be no blank lines between a class comment and the class it describes.
8. Public constants require Javadoc comments, private constants do not. Constants should be the first thing in your class.
9. Constants should be static and final, are often public, and should be followed by instance variables.
10. Instance variables have private or protected visibility, never public or default visibility.
11. Public and protected methods require Javadoc comments including @param tag(s)the @return tag, and the @throws tag where needed (we won't worry about throws until we talk about exceptions).
12. A method's comment must begin with verbs describing what the method does, i.e., Calculates, Returns, Sets, Prints, etc. Note that we use present tense in Java – Returns, not Return. Prints, not Print.
13. The @return and @param tags go AFTER the description.
14. Private methods require non-Javadoc comments (the comments that start with a slash and a single asterisk).
15. Do comment complicated logical blocks of code inside methods with sparse, clear inline comments.
16. Do not use magic numbers (you must use constants instead). Remember that a magic number is any numeric literal. A constant can be local in a method (use the final keyword with it) or class-level (make it static and ALL_CAPS).
17. All method parameters that are object references must be made final (so we don't forget parameters are passed by value):

    (a) Nice to prevent erroneous assignments, and necessary if parameter is referenced by inner class, but that is perhaps a little advanced for now.
    (b) References made final mean that the reference, once pointing to an object, cannot be changed to point at a different object.

18. Consider making your methods final:
    (a) Making a method final prevents subclasses (those that inherit the method) from changing its meaning.
    (b) Final methods are more efficient (the methods become inline, thus avoiding the stack and generating overhead).
19. Data and methods that work together must be encapsulated in the same class.
20. Code duplication must be minimized.
21. The values of local variables that are primitives are set when they are declared, and local variables are not declared until they are needed.
22. Every class that stores state needs an equals method, a hashCode method, and a toString method.
23. In general, we enforce a fairly hard maximum method length of 20 lines of code.

5 Design

1. **This Design exercise is collaborative**. Work together. Share ideas.
2. Draw a UML class diagram for the following situation.
3. **Submit your own work in a PDF called Demeter.PDF** by dragging the PDF into the project directly in IntelliJ.
4. You will design a system for a charter holiday airline, Demeter Airlines, to track its flights.
5. A charter holiday airline is an airline that charters or "rents" airplanes from the companies that own the planes.
6. Demeter Airlines must somehow manage information about all of its flights. You will design the core back end.
7. Demeter Airlines sells flights to desirable destinations on the planes that it rents from airline companies.
8. Demeter Airlines flights are piloted by a captain and a copilot.
9. Demeter Airlines flights are staffed by some flight attendants.
10. Demeter Airlines flights have a departure time and airport, and an arrival time and airport.
11. Demeter Airlines flights have a capacity which depends on the capacity of the plane booked for the flight.
12. Demeter Airlines sells seats to customers, who have names and credit card numbers.
13. At any given time, Demeter Airlines must have immediate and up-to-date access to the following information about each of its flights:
    (a) the ID of the plane rented for the flight
    (b) the flight number
    (c) the total number of seats
    (d) the customers booked on the flight
    (e) the number of available (unsold) seats

14. The Demeter Airlines system must also readily provide the departure time, arrival time, and flight duration for every flight.
15. Demeter Airlines also wants to maintain a list of employees who will be working on each flight.
16. For simplicity, assume each Demeter Airlines flight has two pilots and some flight attendants.
17. Pilots and flight attendants have names and IDs.
18. Planes have an ID (unique for each plane) and capacity (number of seats) and can be booked for a Demeter Airlines flight when they are available.
19. Flights have an ID composed of two letters followed by three numbers, like AC108.
20. Airports have a location and a 3 letter ID.
21. Seats have seat numbers composed of a row and a letter. Seats are available if they are unbooked, else they are unavailable and booked.
22. Seats can be business class, economy plus, or cattle class.
23. You do not need to include any information about money, cost, etc.
24. **Identify the classes including the instance variables, the constructors, and the methods. Identify the relationships between the classes using UML diagram arrows. Your UML diagram must fit on one possibly enormous PDF page I can view on my enormous monitor.**
25. Include all important private and public methods and attributes in the UML diagram. Methods and attributes are important if they help me understand the state being stored, and the relationships, and how information is passed between objects in the Demeter Airlines flight management system.
26. Maximize coherence and minimize coupling.
27. Minimize mutability and visibility.

6 Coding Exercise

This coding exercise is individual.

I've always been interested in the fable of the tortoise and the hare. I'd like you to re-create the classic contest between the slow steady reptile and the impetuous over-confident mammal. You'll use random number generation to develop a simulation of this historic and allegorical event.

1. **Create a ca.bcit.comp2522.lab02.Hare class**:
   (a) A Hare has a position which can be represented as an integer.
   (b) A newly created Hare begins at position 0. Moving forward will increase the value of the position. Moving backward will decrease the value of the position.
   (c) The Hare has a public move( ) method which accepts no parameters and returns the Hare's new position after it moves. The move method will use a random number to determine what happens when the Hare is prompted to move:
      i. 50% of the time the Hare loafs and brags and doesn't move.
      ii. 25% of the time the Hare take a big hop and moves forward 4 units.
      iii. 5% of the time the Hare suffers a big slip and moves backward 3 units.
      iv. the rest of the time the Hare takes a small hop and moves forward 2 units.
   (d) The Hare needs an accessor and a mutator for its position.
   (e) The Hare needs a toString method an equals method, and a hashCode method. Ask IntelliJ to create these. They won't be perfect - you will have to tidy them to meet my requirements and to make Checkstyle happy.

2. **Create a ca.bcit.comp2522.lab02.Tortoise class**:
   (a) A Tortoise has a position which you should represent as an integer.
   (b) A newly created Tortoise begins at position 0. Moving forward will increase the value of the position. Moving backward will decrease the value of the position.
   (c) The Tortoise has a public move( ) method which accepts no parameters and returns the Tortoise's position after it moves. The move method will use a random number to determine what happens when the Tortoise is prompted to move:
      i. 75% of the time the Tortoise moves forward 2 units with a steady plod.
      ii. 20% of the time the Tortoise slips and moves backward 1 units.
      iii. the rest of the time, the Tortoise moves forward 1 unit with a slow plod.
   (d) The Tortoise needs an accessor and a mutator for its position.
   (e) The Tortoise needs a toString method, an equals method, and a hashCode method. Ask IntelliJ to create these. They won't be perfect - you will have to tidy them to meet my requirements.

3. **Create a ca.bcit.comp2522.lab02.Race class**:
   (a) A Race object manages a running competition between one Tortoise and one Hare. A Race has a length or distance, stored as an int. **You may need some additional instance variables and some may need accessors. Minimize the moving parts, though!**
   (b) The Race constructor must accept an integer that represents the length of the Race. It must also accept a new Tortoise racer and a new Hare racer and assign the new objects to instance variables.
   (c) Race requires a private method called onYourMark which accepts no parameters and instructs the Tortoise and the Hare to move to position 0.
   (d) Race requires a public simulateRace method. This method accepts no parameters. This method returns a String representation of the winner ("Tortoise" or "Hare"). This method should start by invoking onYourMark to bring the contestants to the starting line. And then it must invoke a helper method called race( ).
   (e) The private race( ) method accepts no parameters. It must use a loop (perhaps a while or better yet, a do-while). We will pretend that each iteration of the loop is one time unit in the race. Perhaps each iteration is a tick of the clock. With each tick of the clock, the method should invoke the move method for each animal. In order to be fair, randomly choose who moves first each time the clock ticks. The loop must end when the first competitor crosses the finish line, and the method must return a String representation of the winner ("Tortoise" or "Hare").

4. **Finally, create a ca.bcit.comp2522.lab02.Tournament class**. The Tournament class is a utility class which means it must have a private constructor that does nothing. Tournament must only contain a main method. Inside the main method, I'd like you to use the classes you just created to do the following:
   (a) Race the Tortoise and the Hare a distance of 100 units and report who won. Ensure the output is formatted carefully using Formatter, or Stringbuilder, or the PrintStream's printf method. Tell me the racers' final positions and how many 'clock ticks' elapsed before the winner crossed the finish line. Make the output pretty and easy to read. One line is best.
   (b) Simulate 100 races of length 100. Who won the most? Report the number of wins for each competitor.
   (c) Simulate 100 races of length 1000. Who won the most? Report the number of wins for each competitor.
   (d) I think we just created some duplicated logic. Extract the duplicated logic and create an additional method inside Tournament called simulateRaces that is static (it has to be in order to be accessed from main) and which accepts two parameters, the number of races and the length of the race. This method may or may not return a formatted string that reports the number of wins for each competitor.

5. **In the class comment for Tournament, please answer the million-dollar question: who would you bet on in a race between the Tortoise and the Hare?**

Test your code and submit before the deadline. Good luck, and have fun!