CSCU9A3 Practical 3

Cycle Ride Calculator – Objectified!

Intro

In this practical class, we are going to study how to create classes, instantiate objects, and use them as parameters.

Getting Started

Remember how we had this giant method:

```
public double getDuration(int numMiles, String competency, int
numYearsExperience, boolean cyclingAlone, int temp, int windSpeed,
boolean isRaining)
```

There are rather too many parameters. How could we group some of this information together to result in fewer parameters, but the same information being passed?

One of the best ways to do this is to use the concepts of classes and objects. For instance, suppose we need to program a data structure to store the name, registration number, and username of all the students of the university. We could group this information together as a Student class that would look like the following:

Student
name registration number username

This information is encapsulated within the class, so instead of passing the name, registration number and username separately, we could just pass a Student object instead.

Weather

Download the code from Canvas.

Notice that this code is using this concept of classes and objects. Run the tests and ensure that they pass. Now take a look at the code and explore what has changed since last week.

We are now encapsulating the ride and the cyclist attributes in the `CycleRide` and `Cyclist` classes, respectively. Both are these classes are used as parameters, instead of using several separate parameters as before. Much better!

As you can see, we have some parameters that would naturally lend themselves to being encapsulated by a **Weather class**. Make it so!

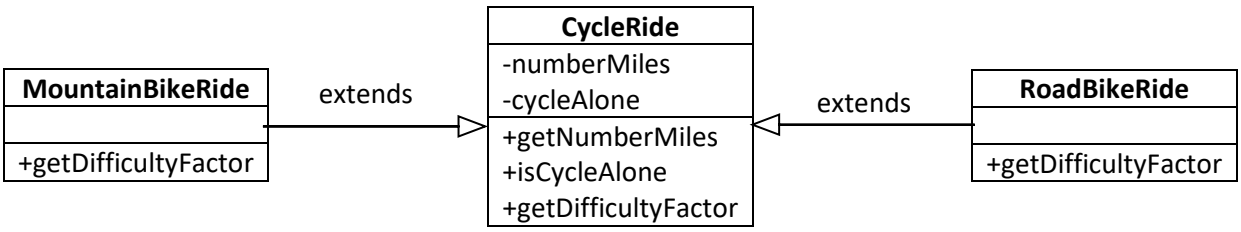
Amend the remaining code (including the `CycleCalculatorTest`) where needed.

[Checkpoint]

This is a good place to check your progress. Show a demonstrator your lovely new weather class, and the `getDuration` method that now uses it. Also, remember to check that your tests (in `CycleCalculatorTest`) all still pass.

Inheritance

Imagine that there could be different types of `CycleRide`. For example, you could go mountain biking or road racing. Each would have different difficulty factors. For example, a road route would have a small difficulty factor, whereas a mountain bike route would have a relatively high difficulty factor. ***Instead of storing the difficulty factor as a variable*** on `CycleRide`, design different subclasses of `CycleRide` that can return a value suitable for that type of ride. Diagrammatically, this might be:



The method to return the difficulty factor should return 1 for a generic `CycleRide`, 1.5 for a mountain bike ride, and 0.5 for a road bike ride. Implement `MountainBikeRide` and `RoadBikeRide`, inheriting from `CycleRide`. This will most likely require you to refer to notes, examples, the Internet, books. Do what it takes to get it all working. Ask for help if you really get stuck!

Note that you would also probably add more new features (methods and attributes) that would be specific to your new types of `CycleRide` that would differentiate them further from the base `CycleRide`. The general rule with inheritance is that the base class contains the core functionality and the sub-classes extend this functionality to provide their own specific behaviour.

After implementing the subclasses (`MountainBikeRide` and `RoadBikeRide`), change your `getDuration` method so it multiplies the calculated duration with the difficulty (retrieved from the `getDifficultyFactor` method). Check your implementation is working by using the `CycleCalculatorAdvancedTest`. Note that this advanced test class doesn't use your recently created Weather class. Amend this test class so it uses your Weather class. Furthermore, the advanced test class only tests the `CycleRide` and the `MountainBikeRide`. Amend this class and include a new test for the `RoadBikeRide`.

[Checkpoint]

Call a demonstrator and show your new `MountainBikeRide` and `RoadBikeRide` classes. Show the amended test class and that the tests pass for all cases.