Homework 1 - Classes, Methods, and Interfaces

Due Oct 29, 2019 by 6pm   **Points** 0

Before you begin, please make sure you are familiar with the guidelines discussed on the [Expectations on Homework](#) page and the "Homework Assignments" section of the [CS 2102 FAQ Page](#).

Assignment Goals

- To be able to define classes with methods
- To use JUnit to create a test suite
- To be able to construct a hierarchy of classes and interfaces

Problem Description and Context

Many competitions are played as a collection of matches or contests between either individual players or teams. In this assignment, we will use Java to model and implement programs over single matches.

Each *match* involves two contestants and contains the *results* (the scores for each contestant). The details of contestants and scores differ across kinds of contests (sports, quiz shows, robot battles, etc), but the structure of a match does not.

For this assignment, we will consider two kinds of contestants:

- In the Rugby World Cup, contestants are teams, each of which represents a *country* and wears a specific *jersey color*. A team may or may not have a formal *intimidation ritual* at the start of each match (this is a reference to the **[haka](https://www.youtube.com/watch?v=IqfHv9FOpXw) (https://www.youtube.com/watch?v=IqfHv9FOpXw)**, the traditional pre-match ritual by the New Zealand team); just use a boolean to record whether a team has such a ritual. Each team has a certain number of *wins* and *losses* for the season.

- Robotics competitions also feature teams. Each team comes from a particular *school* and competes with a robot that has some *special feature* (such as a powerful arm or the ability to spray water, which we describe as a string). We also care about the *previous score that the team had in their last competition* (using 0 if a team has never competed before).

The results of a match consist of the two teams and the scores for each contestant. For purposes of this assignment, the following information makes up the score for a contestant in each kind of event:

- A rugby score is just the number of *points* scored.

- A robotics score contains a number of *points*, a number of *attempted tasks*, and an indication of whether the robot *fell down* during the competition.

Programming Problems

1. Develop Java class and interface definitions to capture matches, contestants, and scores as described above. Your data should capture all of the italicized concepts from the description. You should have one match definition that can capture each of Rugby and Robotics matches, but different definitions for each kind of contestant and score.

   In order to let us run tests against your code, everyone needs to use standard names for interfaces and classes. Use the following:

   - `IContestant` and `IResult` for interfaces

   - `Match` for the match class

   - `RugbyTeam` and `RoboticsTeam` for the contestant classes

   - `RugbyResult` and `RoboticsResult` for the results classes

   - You may structure the fields of the results classes however you wish, but your results-class constructors should take all of the components of the results for both players. This means that, in addition to the two teams, the `RugbyResult` constructor should take two numbers (the points for each team) and the `RoboticsResult` constructor should have six numbers:

   ```
   RugbyResult(RugbyTeam team1, RugbyTeam team2, double team1points, double team2point) {
       ...
   }

   RoboticsResult(RoboticsTeam team1, RoboticsTeam team2,
                  double team1points, int team1tasks, boolean team1fell,
                  double team2points, int team2tasks, boolean team2fell) {
       ...
   }
   ```

2. Write a method `isValid` in each results class that determines whether the results are expected or reasonable according to the individual score components. For rugby scores, a result is valid only if BOTH teams are under 150 points. For robotics scores, a result is valid only if BOTH teams have fewer than 8 attempted tasks AND no more than 16 points.

3. Write a method `getScore` in `RoboticsResult` that takes in the number of points, the number of tasks, and whether the robot fell down. It calculates a final score by summing the points and tasks and then applying a 5 point deduction if the robot fell down.

4. Write a method `winner` in the `Match` class that returns the contestant that won the match according to the results. You may assume that there are no matches with ties. The winner of a Rugby match is the team with more points. The winner of a robotics contest is the one with the highest score

calculated using the method in the previous step. If the results are invalid, however, the method returns **null**.

To make `winner` work, write a common method called `getWinner` in both results classes and then make it available through the `IResult` interface. You will then have to call `getWinner` and `isValid` in your `winner` method.

5. Write a method `expectToBeat` in each contestant class that takes another contestant as input and returns a boolean indicating whether the contestant would be expected to win against the given/input contestant. For a rugby match, if only one team has an intimidation ritual, that team is the expected winner; if neither or both teams have such rituals, the expected winner is the one with the bigger gap between number of wins and number of losses (i.e. the team for whom the value of [number of wins - number of losses] is higher). The expected winner of a robotics competition is the one with a higher previous score, i.e. a higher value in the "previous score" field. For either type of competition, if there is no clear expected winner, return false. *You do not need to put this method in your IContestant interface.*

6. Create a test suite for your work. Put all of your tests and test data in a class called `Examples`. *Your class must be called this so that the auto-grader can find it!*

Note on testing Doubles: When you want to use assertEquals to compare doubles, you include a third argument which is the allowable difference between the two values for them to still be considered equal. For example:

```
assertEquals(5.0, 4.995, .01)
```

returns `true`. Doubles can be imprecise due to the way they are represented within the computer, hence the need for this third argument.

Note on Writing Tests that Compare Objects: A subtlety to JUnit (that we will talk about next week) affects how you write tests that compare objects. When writing these tests, name the objects and use the names in the `assertEquals` test. For instance, let's say we have a hypothetical class called ShootingResult that contains a hypothetical method called bestRound() that returns a ShootingRound:

```
public class Examples {

  ShootingRound longRound = new ShootingRound(...);
  ShootingResult goodResult = new ShootingResult(... longRound ...);

  ...

  @Test
  public void testLongBest() {
    assertEquals(longRound, goodResult.bestRound());
  }
}
```

You should NOT make a new `ShootingRound` for the expected answer in the `assertEquals`. Such a test would fail, even if the two rounds had the same contents (again, for reasons we will explain in detail in week 2).

Just because your program passes all of your JUnit tests does not guarantee that it will pass all of ours. However, you can minimize the risk that your methods will fail our JUnit tests by sticking with the proper naming conventions and writing as many edge cases as possible. The more testing you do, the less the likelihood of failure!

Support Files

Here are a couple of files that may be helpful.

- a skeletal **Examples.java** ↓ **(https://canvas.wpi.edu/courses/16466/files/2217271/download? download_frd=1)** file showing you the shape of a test case. You should create your own Examples.java file by right-clicking your project in Eclipse and going to New -> JUnit Test Case. **Make sure JUnit 4 is selected!**
- a **CompileCheck.java** ↓ **(https://canvas.wpi.edu/courses/16466/files/2217272/download? download_frd=1)** file that attempts to create objects from the expected classes and call the expected methods within those classes. Including this file when you compile will check that you have the class and method names that our grading tools expect, which saves you from losing points.

If you get a compilation error involving this file on a class or method that you defined, **do not edit this file. Edit your files instead!** As you are working, you may wish to comment out sections of the file that check methods you haven't written yet (that's fine). The final work you turn in should, however, compile against the entire contents of this file.

Just because your programs compiles does not mean that your program will pass all of the auto-grader tests! A program can compile but still be full of errors. The CompileCheck file is there only to make sure your naming conventions are correct, not that your classes and methods work correctly.

You are welcome to leave this file in the directory when you submit your work. It can serve as the main method for your program.

Grading

Homework 1 Grading Rubric ↓ **(https://canvas.wpi.edu/courses/16466/files/2236234/download? download_frd=1)**

Here are some details on what we will look for in grading this assignment:

- Did you create classes with the fields required in the problem?
- Do your methods produce the answers expected based on the problem statements?
- Is your code neatly indented and presented in a clean, readable manner?

- Are your test cases correct relative to the problem statement?
- Are your test cases thorough, covering different situations (based on input data) and exercising all of your code?

Programs must compile in order to receive credit. If you submit a program that doesn't compile, the grader will notify you and give you one chance to resubmit within 24 hours; a penalty (25% of the total points for the assignment) will be applied as a resubmission penalty. Code that is commented out will not be graded.

What to Turn In

Submit (via **InstructAssist** **(https://ia.wpi.edu/cs2102/)**) a single zip file (not tar, rar, 7zip, etc) containing all of your .java files that contain your classes, interfaces, and examples for this assignment. Do not submit the .class files. You may put all of your other classes and interfaces either into a single file or into separate ones (as you prefer). If you have separate src and test subdirectories, you may retain that structure in your zip file.

Make sure all of your tests are in separate files from your code, as explained in the **Expectations on Homework** page.