Com S 227
Spring 2019
Assignment 2
200 points

Due Date: Thursday, February 21, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Feb 20)

10% penalty for submitting 1 day late (by 11:59 pm Feb 22)

No submissions accepted after February 22, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy questionnaire* on the Assignments page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Note: Our first exam is Monday, February 25, which is just four days after the due date for this assignment. ***It will not be possible to have the assignments graded and returned to you prior to the exam****. We can post a sample solution on February 23 if anyone is interested.*

Please start the assignment as soon as possible and get your questions answered right away!

Introduction

The purpose of this assignment is to give you lots of practice working with conditional logic. For this assignment you'll create one class called `CricketGame` that is a simplified model of the game of *cricket*.

Whoa, you say, but I don't know anything about cricket. Do not worry, until the day before yesterday neither did we! We just read about the rules on Wikipedia, made some simplifications, and wrote up a specification. In the past we have made assignments out of games like baseball

and American football, so this semester we thought we'd level the playing field (haha) for the non-US students.

As with our previous assignment, all you really need to do is implement the methods as specified and you'll be fine. Write simple test cases to make sure you know what you want the code to do. Remember you can always post your *test* code on Piazza for comments if you are not entirely sure you have the right idea of what the spec is saying.

Overview and specification

The specification for this assignment is provided in the form of an online javadoc:

http://web.cs.iastate.edu/~cs227/assignments/hw2/doc/

which also includes a concise overview of the game as relevant for this assignment. If you want a more in-depth understanding, a good place to look is the article, "Cricket Explained (An American Viewpoint)":

http://static.espncricinfo.com/db/ABOUT_CRICKET/EXPLANATION/CRICKET_EXPLAINED_AMERICAN.html

The `api` package

NOTE: The two types `Outcome` and `Defaults` in the `api` package are provided for you. Do not modify them and do not add any code to the `api` package. Your `CricketGame` class needs to be in a package named `hw2`.

In Assignment 1, recall that we defined two *constants* as `public static final` values in the `UberDriver` class. Here we are also defining some constants. One set of constants is defined in the class `Defaults` as `public static final` values. You will need to use these when you implement the no-argument constructor of `CricketGame`. The only new thing to learn is that since they are not defined within `CricketGame`, you need to qualify them with the class name when you use them. For example, you'd write something like

```
someInstanceVariable = Defaults.DEFAULT_NUM_INNINGS;
```

after importing `api.Defaults`.

The other set of constants are defined as an `enum` type called `Outcome`. This type is used as the argument type for the `bowl` method. They are really just like the numeric constants we have seen before, with the advantage that they are *type-safe*: it is impossible to accidentally pass an invalid

or out of range value to the **`bowl`** method, since the argument *must* be one of the constants listed in the **`enum`** type **`Outcome`**. The way you use them is just like integer constants. For example, in your implementation of the **`bowl`** method you might have code such as

```
if (outcome == Outcome.WICKET)
{
  playersOut = playersOut + 1;
  // etc...
```

Tip: Of course, you need an import for **`api.Outcome.`** If you ALSO add the line:

```
import static api.Outcome.*;
```

to the top of your **`CricketGame`** class, you don't have to keep typing "**`Outcome.`**" in front of each one. For more information on enums see "Special Topic 5.4" which is in Section 5.5 of the textbook.

Testing and the SpecCheckers

As always, you should try to work incrementally and write tests for your code as you develop it.

Since the test code is not a required part of this assignment and does not need to be turned in, **you are welcome to post your test code on Piazza for others to check, use and discuss.**

SpecChecker 1

Your class must conform precisely to this specification. The most basic part of the specification includes the class name and package, the required constants, the public method names and return types, and the types of the parameters. We will provide you with a specchecker to verify that your class satisfies all these aspects of the specification and does not attempt to add any public attributes or methods to those specified. If your class structure conforms to the spec, you should see a message such as `"2 out of 2 tests pass"` in the console output. (This specchecker will not offer to create a zip file for you). Remember that your instance variables should always be declared **`private`**, and if you want to add any additional "helper" methods that are not specified, they must be declared **`private`** as well.

SpecChecker 2

In addition, since this is the second assignment and we have not had a chance to discuss unit testing very much, we will also provide a specchecker that will run some simple functional tests for you. This is similar to the specchecker you used in Assignment 1. It will also offer to create

a zip file for you to submit. ***Specchecker 2 should be available February 17. Please do not wait until that time to start testing!***

Getting started

*Remember to work **incrementally** and test new features as you implement them. Always start with simple test cases or usage examples. The "getting started" section from Assignment 1 provided a detailed, step-by-step example of this process. For this assignment, we will provide some general guidance below, but we expect that by now you can start writing your own test cases.*

1. Create a new, empty project and then add two packages called **`api`** and **`hw2`**. In the **`api`** package, create two classes named **`Defaults`** and **`Outcome`**. Copy and paste the given sample code for Defaults.java and Outcome.java. (You might notice that **`Outcome`** is declared as an **`enum`**, not a **`class`**; this is explained on page 2.)

2. In the **`hw2`** package, create a class named **`CricketGame`**. Based on the javadoc, put in stubs for all the required methods and the two constructors. Make sure there are no compile errors. You'll need an import for **`api.Outcome`**. Write a brief javadoc comment for each method. At this point you can download and run specchecker #1 to make sure you have all the method declarations correct. **Do not add to or modify the code in the `api` package.**

3. If you look over the accessor methods, you can get an idea what instance variables you are going to need - you'll need to keep track of the score for each side, the outs, the bowls, and so on. Remember to avoid redundancy. For example, since side 0 always starts, you don't need a separate variable to keep track of which side is batting - you can get it from the number of innings. You will probably also have to store the values given in the constructor. Each time you define an instance variable, name it carefully and write a brief comment stating what it represents.

4. You can't go much further without starting on the **`bowl`** method. It will have a number of cases based on the argument. The simplest to begin with might be the cases for **`Outcome.WIDE`**, and **`Outcome.NO_BALL`**, which affect only the batting side's score. You could write a simple test case like this:

```
CricketGame g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.WIDE);
g.bowl(Outcome.WIDE);
System.out.println(g.getScore(true)); // expected 2
```

(The boolean argument to **`getScore`** indicates whether you want the batting side's score or the other side's score. For the moment, you can just return the batting side's score and worry about the other team later. Similarly, the **`bowl`** method is specified to do nothing if the game is over or if the ball is already in play. You can also add this check later on.)

5. The cases for **`Outcome.BOUNDARY_SIX`** and **`Outcome.BOUNDARY_FOUR`** also affect the score, but also count towards the number of *bowls*. A simple test might look like this:

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.BOUNDARY_SIX);
System.out.println(g.getScore(true)); // expected 6
System.out.println(g.getBowlCount()); // expected 1
```

6. Once you start counting bowls, you also need to worry about the *overs*. Once the number of bowls reaches a certain value (2 in the example above, the first argument to the constructor), you increment the overs and start counting bowls again.

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
System.out.println(g.getBowlCount()); // expected 1
System.out.println(g.getOverCount()); // expected 2
```

7. Some of the cases in **`bowl`** immediately get the batsman out, such as **`Outcome.WICKET`**. Try incrementing the *outs* for the batting side. Write some test cases.

8. When the number of overs reaches a certain amount, or when the number of outs reaches a certain amount, you have to increment the *innings* and switch to the other side. (In order to test your code you'll need to have implemented **`whichSideIsBatting`** and make sure your **`getScore`** returns the correct value for either side.). Try this:

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
g.bowl(Outcome.BOUNDARY_SIX);
System.out.println(g.getBowlCount()); // expected 0
System.out.println(g.getOverCount()); // expected 0
System.out.println(g.getCompletedInnings()); // expected 1
System.out.println(g.getScore(true)); // expected 0
System.out.println(g.getScore(false)); // expected 36
```

9. Next you might continue with **`Outcome.HIT`**. Again, start with simple test cases that you can check:

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.HIT);
System.out.println(g.isInPlay()); // expected true
g.safe();
System.out.println(g.isInPlay()); // expected false
```

Continue in a similar way, writing some simple tests for **`tryRun`** and **`runOut`**, for example,

```
g = new CricketGame(2, 3, 4, 6);
g.bowl(Outcome.HIT);
g.tryRun();
g.tryRun();
g.tryRun();
g.runOut();
System.out.println(g.getScore(true));  // expected 2
System.out.println(g.getOuts());       // expected 1
```

10. Other details include detecting when the game has ended, making sure that **`bowl`** does nothing when the game is over, and so on.

More about grading

This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the grader's assessment of the quality of your code. This means you can get partial credit even if you have errors, and it also means that even if you pass all the specchecker tests you can still lose points. Are you doing things in a simple and direct way that makes sense? Are you defining redundant instance variables? Some specific criteria that are important for this assignment are:

- Use instance variables only for the "permanent" state of the object, use local variables for temporary calculations within methods.
    - You will lose points for having lots of unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

See the "Style and documentation" section below for additional guidelines.

Using "helper" methods

The public API for your class must exactly match the javadoc, but you can define additional **`private`** methods if you wish to simplify your implementation. For example, incrementing the score for the batting side means having to check which side is batting and then adding to the appropriate score variable. Instead of having this check in multiple places, create a private method that does it. If you look at your code, you'll likely see several other places where the same logic is duplicated; for example, after many operations, you'll need to check the number of bowls and see whether it's time to increment overs, check the overs or outs to see whether it's time to switch innings. When you see duplicated logic in several places, ask yourself whether it makes sense to move the common code into a helper method.

Style and documentation

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines:

- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the **`@author`** tag, and method javadoc must include **`@param`** and **`@return`** tags as appropriate.
    - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
    - Run the javadoc tool and see what your documentation looks like! You do not have to turn in the generated html, but at least it provides some satisfaction :)

- All variable names must be meaningful (i.e., named for the value they store).
- Your code should not be producing console output. You may add **`println`** statements when debugging, but you need to remove them before submitting the code.

- Internal (//-style) comments are normally used inside of method bodies to explain *how* something works, while the Javadoc comments explain *what* a method does. (A good rule of thumb is: if you had to think for a few minutes to figure out how something works, you should probably include a comment explaining how it works.)
    - Internal comments always *precede* the code they describe and are indented to the same level.
- Use a consistent style for indentation and formatting.
    - Note that you can set up Eclipse with the formatting style you prefer and then use Ctrl-Shift-F to format your code. To play with the formatting preferences, go to Window->Preferences->Java->Code Style->Formatter and click the New button to create your own “profile” for formatting.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`assignment2`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`assignment2`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in, and **for this assignment you are welcome to post and discuss test code**. (In the Piazza editor, use the button labeled "code" to have the editor keep your code formatting. You can also use "pre" for short code snippets.)

If you have a question that absolutely cannot be asked without showing part of your source code, change the visibility of the post to "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are considered to be part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

What to turn in

Please submit, on Canvas, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_hw2.zip`**. and it will be located in whatever directory you selected when you ran the SpecChecker. It should contain one directory, **`hw2`**, which in turn contains one file, **`CricketGame.java`**. Please LOOK at the file you upload and make sure it is the right one!

Submit the zip file to Canvas using the Assignment 2 submission link and verify that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", which can be found in the Piazza pinned messages under "Syllabus, office hours, useful links."

> We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **hw2**, which in turn should contain the file **CricketGame.java.** If you are using Eclipse, you can accomplish this by zipping up the **src** directory of your project. **Do not zip up the entire project**. The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and NOT a third-party installation of WinRAR, 7-zip, or Winzip.