Com S 227
Fall 2023
Assignment 2
200 points

Due Date: Friday, September 29, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Sept 28)

10% penalty for submitting 1 day late (by 11:59 pm Sept 30)

No submissions accepted after Sept 30, 11:59 pm

This assignment is to be done on your own. See the Academic Integrity policy in the syllabus, 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, October 2, which is just three 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 October 1.*

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. See the "More about grading" section.

Contents

Overview

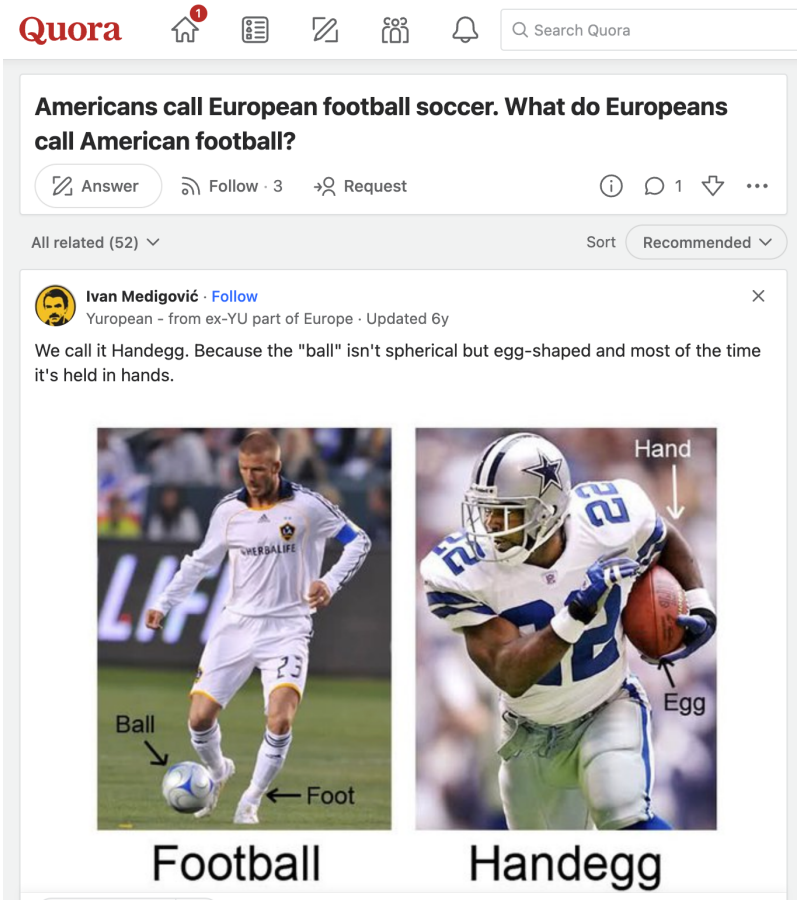
The purpose of this assignment is to give you lots of practice working with conditional logic and managing the internal state of a class.

For this assignment you will implement one class, called **`Handegg`**, that encapsulates the state of a *highly* simplified form of the charming and popular game often known as "American football". If you know nothing about American football, never fear, neither do we! We just looked it up on Wikipedia! Although the real-life game includes many complex and byzantine rules, we are going to ignore most of them. For a precise explanation of the simplified rules that we are using for this assignment, and the exact behavior of the **`Handegg`** methods, see the online Javadoc. Those of you who understand American football extremely well are going to have extra work cut out for you since you have to un-learn many of the rules you think you know.[1]

If you really and truly have no idea about American football at all, first read the javadoc; also, this overview could help, maybe:
https://www.downhomeinspiration.com/football-101-newbies-guide-basics/

Why are you calling it "Handegg"??

Number one, to emphasize to all the American football experts out there that this is a slightly different game than what you know, so you still have to read the spec. Second, because it makes total sense. See this Quora post:



[1] It could be worse; a couple of years ago we did the game of *cricket*. The baseball fans in that class still have not recovered from the psychological trauma.

Using helper methods

You are expected to make some effort to use procedural decomposition to simplify your logic and reduce code duplication. Look for places in your code where you are repeating the same steps, and see whether it makes sense to factor those steps into a helper method. For example, there are many places where the offense and defense are reversed, right? Does it make sense to define a helper method to carry out those steps? Similarly, there are many places where you have to add points to the score of whichever team is currently on offense. And there is a lot of similarity between `run()` and `pass()` that you should notice.

Remember that any method you write that is not in the public API needs to be declared `private`, and also that *all* methods need to be documented, whether public or private.

Specification

The specification for this assignment includes this pdf, the online Javadoc, and any "official" clarifications announced on Canvas.

Where's the main() method??

There isn't one! Like most Java classes, this isn't a complete program and you can't "run" it by itself. It's just a single class, that is, the definition for a type of object that might be part of a larger system. To try out your class, you can write a test class with a main method like the examples below in the getting started section.

There is also a specchecker (see below) that will perform a lot of functional tests, but when you are developing and debugging your code at first you'll always want to have some simple test cases of your own, as in the getting started section below.

Suggestions for getting started

*Remember to work **incrementally** and test new features as you implement them. Since this is only our second assignment, here is a rough guide for some incremental steps you could take in writing this class.*

1. Create a new, empty project and then add a package called `hw2`. ***Be sure NOT to create module-info.java***.

2. Download the Handegg.java skeleton from Canvas to somewhere on your machine, and drag it into the **`hw2`** package. Put in stubs for all the required methods and the constructor. Remember that everything listed in the Javadoc is declared **`public`**. For methods that are required to return a value, just put in a "dummy" return statement that returns zero or false. There should be no compile errors. ***DO NOT COPY/PASTE directly from this pdf or the online Javadoc!*** *This leads to insidious bugs caused by invisible characters that are sometimes generated by sophisticated document formats.)*

3. Briefly javadoc the class, constructor and methods. This is a required part of the assignment anyway, and doing it now will help clarify for you what each method is supposed to do before you begin the actual implementation. The class javadoc can be brief, but should include an **@author** tag with your name. (Repeating the wording from the online javadoc is perfectly acceptable; however, we strongly advise that you ***DO NOT COPY/PASTE directly from this pdf or online Javadoc!*** *This leads to insidious bugs caused by invisible characters that are sometimes generated by sophisticated document formats.)*

4. Now you'll need to start thinking about instance variables. Looking at the accessor methods often gives you clues about what information needs to be stored in the object. For example, how will you implement **`getLocation()`** ? The current location of the ball will have to be stored somehow in an instance variable. Make sure you initialize it correctly in the constructor. This is easy to illustrate and check with a simple test case:

```
Handegg game = new Handegg();
System.out.println(game.getLocation());
System.out.println("Expected 30");
```

5. Other obvious cases include **`getScore`**, **`getDown`**, and **`getOffense`**. Go ahead and add relevant instance variables and implement these accessors. Try it out:

```
System.out.println(game.whoIsOffense());
System.out.println("Expected 0");
System.out.println(game.whichDown());
System.out.println("Expected 1");
System.out.println(game.getScore(0));
System.out.println("Expected 0");
```

6. The most complex methods are **`run()`** and **`pass()`**, that is, you can tell from the Javadoc that there are a lot of cases to consider. So instead, it might make sense to think about a simpler method first and make sure you get it working correctly. The method **`punt()`** is pretty simple. The ball moves by the given amount, and then the other team becomes the offense. Start by writing a couple of simple test case such as these:

"After a punt, the other team becomes the offense."
"After a 50 yard punt from location 30 yards, the other team becomes the offense with the ball at location 20."

```
game.punt(50);
System.out.println(game.whoIsOffense());
System.out.println("Expected 1");
System.out.println(game.getLocation());
System.out.println("Expected 20");
```

(Aside: if you use an integer 0 or 1 to represent which team is the offense, you can switch between them easily with a line of code such as

```
theOffense = 1 - theOffense;
```

If you're using a boolean variable, you could do it like this with the "not" operator,

```
offenseIsTeam0 = !offenseIsTeam0;
```
)

7. Maybe next try **`fieldGoal`**, which is similar to **`punt`**, but also may update the score if the "success" parameter is true. Again, try a couple of scenarios in a simple test case:

"After a successful field goal, the offense earns 3 points."
"After a successful field goal, the other team becomes the offense with the ball 30 yards from its goal line."

```
game = new Handegg();
game.fieldGoal(true);
System.out.println(game.getScore(0));
System.out.println("Expected 3");
System.out.println(game.whoIsOffense());
System.out.println("Expected 1");
System.out.println(game.getLocation());
System.out.println("Expected 30");
```

The method **`extraPoint`** is similar too. (Although it is expected that **`extraPoint`** is only called by clients after a touchdown, there is no mechanism to enforce this, and the method should still work as expected.) Here is an example:

```
game = new Handegg();
game.extraPoint(true);
System.out.println(game.getScore(0));
System.out.println("Expected 1");
System.out.println(game.whoIsOffense());
```

```
System.out.println("Expected 1");
System.out.println(game.getLocation());
System.out.println("Expected 30");
```

8. Now you'd better start thinking about **`run()`**. The first thing to think about is updating the ball's location. At first, don't worry about counting "downs", just check that the location is updating. For example, if the offense runs 5 yards, what should you observe about the ball's location?

"After running 5 yards, the ball should be 5 yards ahead."

```
game = new Handegg();
game.run(5); // advance the ball 5 yards
System.out.println(game.getLocation());
System.out.println("Expected 35");
```

9. Next you might go back to **`fieldGoal`**. After an unsuccessful field goal attempt, the score does not change and the defense gets the ball at the location it had when the field goal was attempted. So you could come up with a test case like this:

```
game = new Handegg();
game.run(25); // advance the ball 25 yards
System.out.println(game.getLocation());
System.out.println("Expected 55");
game.fieldGoal(false);
System.out.println(game.getScore(0));
System.out.println("Expected 0");
System.out.println(game.whoIsOffense());
System.out.println("Expected 1");
System.out.println(game.getLocation());
System.out.println("Expected 45");
```

10. Next, check for running to a touchdown (location is more than **`FIELD_LENGTH`**), and adjust the score accordingly. (You don't switch the offense or adjust the ball location after a touchdown, since the client is supposed to call **`extraPoint`** to handle that.)

```
game = new Handegg();
game.run(100); // advance the ball 100 yards
System.out.println(game.getLocation());
System.out.println("Expected 130");
System.out.println(game.getScore(0));
System.out.println("Expected 6");
```

11. Next, how would you add a mechanism for counting “downs”? At this point you'll need to figure out how to detect whether the ball has advanced 10 yards since the first down. If so, the offense gets to start another first down. If not, the other team gets the ball. How are you going

to keep track of where the ball started, so you can tell if it eventually moved 10 yards? Essentially, you have to implement **`getYardsToFirstDown()`**. Depending on how you have set things up so far, you will need another instance variable or two. For example, try a scenario like this:

```
game = new Handegg();
System.out.println(game.whichDown());
System.out.println("Expected 1");
System.out.println(game.getYardsToFirstDown());
System.out.println("Expected 10");
game.run(-4);
System.out.println(game.whichDown());
System.out.println("Expected 2");
System.out.println(game.getYardsToFirstDown());
System.out.println("Expected 14");
game.run(9);
System.out.println(game.whichDown());
System.out.println("Expected 3");
System.out.println(game.getYardsToFirstDown());
System.out.println("Expected 5");
game.run(20);
System.out.println(game.whichDown());
System.out.println("Expected 1");
System.out.println(game.getYardsToFirstDown());
System.out.println("Expected 10");
```

12. In the example above, the offense eventually makes 10 yards, and the downs counter starts over? What happens if they don't make it 10 yards? Try something like this:

```
game = new Handegg();
System.out.println(game.whichDown());
System.out.println("Expected 1");
System.out.println(game.getYardsToFirstDown());
System.out.println("Expected 10");
game.run(1);
game.run(1);
game.run(1);
game.run(1);
System.out.println(game.whoIsOffense());
System.out.println("Expected 1");
System.out.println(game.whichDown());
System.out.println("Expected 1");
System.out.println(game.getLocation());
System.out.println("Expected 66");
```

13. The **`pass()`** method does exactly the same thing as **`run()`**, except that there is the possibility of an *interception*. That means the other team gets the ball at its location after the pass.
Try something like this, maybe:

```
game = new Handegg();
game.pass(4, false);
System.out.println(game.whichDown());
System.out.println("Expected 2");
System.out.println(game.getYardsToFirstDown());
System.out.println("Expected 6");
game.pass(21, true);
System.out.println(game.whoIsOffense());
System.out.println("Expected 1");
System.out.println(game.getLocation());
System.out.println("Expected 45");
```

The SpecChecker

You can find the SpecChecker on Canvas. Import and run the SpecChecker just as you practiced in Lab 1. It will run a number of functional tests and then bring up a dialog offering to create a zip file to submit. Remember that error messages will appear in the *console* output. There are many test cases so there may be an overwhelming number of error messages. ***Always start reading the errors at the top and make incremental corrections in the code to fix them.*** When you are happy with your results, click "Yes" at the dialog to create the zip file. See the document "SpecChecker HOWTO", link #12 on our Canvas front page, if you are not sure what to do.

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 TA'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 unnecessary instance variables
    - All instance variables should be **`private`**.
- **Accessor methods should not modify instance variables**.

- You should not have a lot of redundant code. If a set of actions is repeated in multiple places, create a helper method.

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

Style and documentation

Roughly 15% of the points will be for documentation and code style. Here are some general requirements and guidelines in addition to those noted above.

- **Each class, method, constructor and instance variable, whether public or private, must have a meaningful javadoc comment**. The javadoc for the class itself can be very brief, but must include the **`@author`** tag. The javadoc for methods must include **`@param`** and **`@return`** tags as appropriate.
    - Try to briefly state what each method does in your own words. However, there is no rule against copying the descriptions from the online documentation.
    - 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 an internal 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.

Using arrays?

If you are already know what an array is and are comfortable using arrays in Java, you might notice that there are a couple of places where you could save a few lines of code by using an array. *That is ok if you want to do it*.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`hw2`**. 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 **`hw2`**. *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. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "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 announcements from the instructors on Canvas 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 Canvas. (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, **`Handegg.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", link #11 on our Canvas front page.

> 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 **Handegg.java**. 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.