Com S 227
Spring 2024
Assignment 2
200 points

Due Date: Wednesday, February 28, 11:59 pm (midnight)
5% bonus for submitting 1 day early (by 11:59 pm February 27)
10% penalty for submitting late (by 11:59 pm March 3)
No submissions accepted after March 3, 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, try office hours, the schedule is on Canvas. Lots of help is also available through the Piazza discussions. Please start the assignment as soon as possible to get your questions answered right away. It is physically impossible for the staff to provide individual help to everyone the afternoon that the assignment is due!

This is a "regular" assignment so we are going to read your code. Your score will be based partly 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. You'll create one class, called **`FuzzballGame`**, that is a model of an obscure game played with a ball and a stick[1]. Although there is some resemblance to the American game of baseball, it's probably best if you forget everything you know about baseball as you read these instructions. Your job is to implement the rules of the game that *is specified in this document* even if that conflicts with your understanding of any other baseball game rules.

Here is how it works. There are two teams, which we will call "Team 0" and "Team 1" that take turns as the *batting team* and the *fielding team*. At each step of the game, a player called the *pitcher* from the fielding team, standing in a designated location, throws the ball towards a player on the batting team called the *batter*, also standing in a designated location called *home*. The batter, at his or her discretion, tries to hit the ball with a stick and make it go a very long distance. Another player from the fielding team, called the *catcher*, crouches behind the batter and catches the ball if the batter doesn't hit it. Behind the catcher is an official called the *umpire*, whose job is to decide whether the pitch was accurately thrown. Some possible outcomes are:

- The batter swings the stick at the ball and misses. This is a *strike*, and the batter is immediately *out*.
- The batter doesn't swing at the ball, but the umpire declares that it was accurately thrown. This is a *called strike*. The batter isn't immediately out, but the called strike is added to a count for that batter, and after a certain number of them, the batter is out.
- The batter doesn't swing at the ball, but the umpire declares that it wasn't accurately thrown. This is called a *ball*, and is added to the batter's count of balls. What this means is discussed below.
- The batter hits the ball, but it goes less than 15 feet, or flies behind the batter. This is a *foul*, and the batter is immediately out.
- The batter hits the ball, but someone on the fielding team catches it before it bounces. This is a *caught fly*, and the batter is immediately out.
- The batter hits the ball and it goes 15 feet or more, in the right direction, and isn't caught by the fielding team. This is a *hit*. Depending on the distance the ball goes, it is called a *single*, *double*, *triple*, or *home run*, the consequences of which are discussed below.

Outs

If a batter is out, one of two things happens: if the number of outs for the batting team has reached a designated maximum (usually 3), the teams switch sides: the fielding team becomes the batting team, and the batting team becomes the fielding team, and all the imaginary runners (see below) are cleared off the bases. In any case, the counts of balls and strikes are reset to zero for a new batter.

[1] https://en.wikipedia.org/wiki/Fuzzball_(sport)

Hits

Close your eyes, breathe deeply, and imagine three locations. Everyone playing the game has in his or her mind a picture of three locations, known as *first base*, *second base*, and *third base*. A base may be occupied by an *imaginary runner*. When a batter hits a *single*, an imaginary runner goes to first base. Any imaginary runners that were already on the imaginary bases automatically advance to the next one: first base to second base, second base to third base, and third base to "home". Each time an imaginary runner advances to home, the team earns a point. This is important, because it's by accumulating points that you win the game!

When the batter hits a *double*, the same process essentially happens twice: an imaginary runner goes to second base, leaving first base empty, and any runners already on the bases advance *twice*. (Note that an imaginary runner reaching home always stops there, and does not keep going around to first base again!) Likewise, in case of a triple, an imaginary runner goes to third base, and any runners already on base advance three times. In case of a home run, the runners advance four times, so the bases always end up empty, and the team always earns at least one point, and possibly up to four points.

After any hit or out, the counts of balls and strikes is reset to zero for a new batter.

Singles, Doubles, Triples, Home Runs

If the batter hits the ball but it travels less than 15 feet in front of the batter, or goes a negative distance (behind the batter), as noted above it's considered a *foul* and the batter is out. Otherwise, as long as it's not a *caught fly*, it's a hit, and the interpretation of the distance the ball travels is as follows:

1. At least 15 feet but less than 150: the hit is a *single*
2. At least 150 feet but less than 200: the hit is a *double*
3. At least 200 feet but less than 250: the hit is a *triple*
4. 250 feet or more: the hit is a *home run*

Balls

If the batting player's count of balls reaches a certain point (usually 5), the pitching stops and the batter gets what is called a *walk*. What this means is that, as with a single, an imaginary runner is placed on first base. However, the imaginary runners on other bases don't automatically advance one base as they do for a hit; they only do so if *forced* to by the advancement of another imaginary runner. As an example: suppose there are runners on first and third base and the batter gets a walk. The new imaginary runner is placed on first, the runner on first is forced to advance to second, but the batter on third stays there. After a walk, the counts of balls and strikes are reset to zero for a new batter.

Innings, "top" and "bottom", and winning the game

The game is divided into rounds called *innings*. In each inning, each team gets a chance to be the batting team, always starting with Team 0; when the designated number of outs is reached, Team 1 becomes the batting team. The maximum number of innings for a game is configurable via a constructor parameter. Whichever team has the most points after the last inning is the winner. It is possible to have a tie, in which case both teams are considered to "win" the game. As a matter of terminology, the first part of the inning when Team 0 is batting is called the *top*, and the second part when Team 1 is batting is called the *bottom*.

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

*Smart developers don't try to write all the code and then try to find dozens of errors all at once; they work **incrementally** and test every new feature as it's written. Here is example of 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 to choose "Don't Create" at the dialog that asks whether you want to create module-info.java***.

2. Create a package named **`hw2`**. Download the skeleton code version of **`FuzzballGame.java`** and copy it into your project. The easiest way to do this is by drag-and-dropping the file into **`src/hw2`** in Eclipse.

3. At this point, the **`FuzzballGame`** class will not even compile. Getting the code to compile should be your top priority. Add stubs for all the methods and the constructor which are described in the Javadoc provided with the assignment. Document each one. For methods that

need to return a value, just return a “dummy” value (e.g., 0 or false) as a placeholder for now. At this point there should be no compile errors in the project.

3. Try a very simple test like this:

```
public class SimpleTests {
  public static void main(String[] args) {
    FuzzballGame game = new FuzzballGame(3);
    System.out.println(game);
  }
}
```

You should see an output string like this, produced by the **`toString()`** method:

```
ooo Inning:0 [B] Score:0-0 Balls:0 Strikes:0 Outs:0
```

Note that in printing the variable **`game`**, the **`println`** method automatically invokes the method **`game.toString()`**. The **`toString`** method just calls your other accessor methods to get the values to put in the string.

4. The initial values seen in the output string above are not quite right. We should start out at the top of inning 1. You might start with the methods **`whichInning()`** and **`isTopOfInning()`**, which implies defining an instance variable to count innings, and an instance variable to keep track of whether it's the top or bottom of the inning. Once you have these initialized correctly in the constructor, the test above should give you the output:

```
ooo Inning:1 [T] Score:0-0 Balls:0 Strikes:0 Outs:0
```

5. Next you could think about counting the balls, strikes, and outs. You'll need three instance variables for these three values, and their values are returned by the corresponding accessor methods **`getBallCount()`**, **`getCalledStrikes()`**, and **`getCurrentOuts()`**. *Remember that accessor methods never modify instance variables, they only observe them and return information*.

6. You will have noticed that there are several mutator methods that will cause the game to change state by simulating what happens when the pitcher pitches the ball: **`ball()`**, **`caughtFly()`**, **`hit()`**, and **`strike()`**. Think about what happens after a sequence of strikes.

Remember that we distinguish between **`strike(true)`** (a "swung" strike) and **`strike(false)`** (a "called" strike). For the former, the batter is out immediately, but for the latter, the batter is out after two called strikes, so we should be able to check:

```
game.strike(false);
System.out.println(game); // one strike
game.strike(false);
System.out.println(game); // 0 strikes, one out, since it's a new batter
game.strike(false);
System.out.println(game); // one strike, one out
```

```
game.strike(false);
System.out.println(game); // 0 strikes, two outs, since it's a new batter
```

Add logic to the strike method to start a new batter after two called strikes as above.

7. Now, suppose you continue the test case above where we have two outs in the top of the first inning. When there is one more out, the teams switch, which we observe by seeing that `isTopOfInning` now returns false, and team 1 starts with no outs:

```
game.strike(true); // batter is immediately out for swung strike
System.out.println(game.isTopOfInning()); // should be false now
System.out.println(game);         // bottom of 1st inning, 0 outs
```

The final line of output above should be,

```
ooo Inning:1 [B] Score:0-0 Balls:0 Strikes:0 Outs:0
```

After that, if team 1 gets three outs, the game should transition to the top of inning 2. You should be able to create a short example that transitions all the way to the end of the game. At the end of a three-inning game, `whichInning()` would return 4, and `gameEnded()` would return true.

8. A batter could also be out because of a caught fly ball. Go ahead and implement `caughtFly()`, which should have the same effect as `strike(true)`.

9. What about `hit()` ? In the case that the distance is less than 15, it's an automatic out for the batter, which you already know how to handle as above. But if the distance is 15 or greater, we have to start thinking about the imaginary runners. How will you keep track, for bases 1, 2, and 3, whether there is an imaginary runner there or not? Decide how to do that, and implement the accessor method `runnerOnBase()`.

10. Maybe a good place to start is with singles. What should happen here?

```
game = new FuzzballGame(3);
game.hit(15);
System.out.println(game.runnerOnBase(1)); // true
System.out.println(game.getBases()); // Xoo
```

(The `getBases` method, which is already implemented for you, returns a string representation of the bases, where "`Xoo`" indicates that there is a runner on first, but not on second or third.) If we get another single, there should be runners on first and second:

```
game.hit(15);
System.out.println(game.getBases()); // XXo
```

Continuing the example above, after another single, there should be runners on all three bases:

```
game.hit(15);
System.out.println(game.getBases()); // XXX
```

What is a good way to make that happen without going nuts considering all possible cases? For example, if you had a runner just on second ("**oXo**") and there was a single, you should end up with runners on first and third ("**XoX**"). One idea is to implement a *helper method,* e.g. **shiftRunners()**, that simply shifts all runners to the next base.

Tip: **shiftRunners()** is probably easiest to implement by updating third and working backward to first and second.

11. Finally, continuing the example above, what if there is one more single:

```
game.hit(15);
System.out.println(game.getBases()); // XXX
System.out.println(game.getTeam0Score()); // 1
```

There will still be runners on all three bases, but the imaginary runner on third base will have run to home, which counts as a point for team 0. Add instance variables to keep track of the score, and add logic to your helper method to adjust the score when a runner gets back to home.

12. Next, you could add the logic to the **hit()** method to take different action depending on whether the distance represents a double, triple, or home run. Your **shiftRunners()** method, if you chose to implement one, will be very useful here, since for a double all you really have to do is shift the runners twice (the first time, also adding a runner on first); for a triple you do it three times, and so on. So, continuing the example above, you should be able to do this:

```
// try hitting a double now
game.hit(150); System.out.println(game.getBases()); // oXX
System.out.println(game.getTeam0Score()); // 3
```

13. The **ball()** method is interesting. Normally it just increases the count of balls, going back to zero in case there is a hit or out.

```
game = new FuzzballGame(3);
game.ball();
System.out.println(game.getBallCount()); // 1
game.ball();
System.out.println(game.getBallCount()); // 2
game.ball();
System.out.println(game.getBallCount()); // 3
game.strike(true); // out!
System.out.println(game.getBallCount()); // 0, since it's a new batter
```

But if the ball count reaches the maximum, the batter gets a *walk* and an imaginary runner is placed on first base. But it is not quite like a single, where all the runner advance. On a walk, runners only advance if they are *forced* to do so. For example, if the bases are **oXo**, after a walk they would be **XXo**, not **XoX**, since the runner on second isn't forced to move. If the bases were **XoX** and there was a walk, it would become **XXX**. Thus, a walk only results in a point for the team if all the bases were all occupied to start with. Try something like this:

```
game = new FuzzballGame(3);
game.hit(225); // a triple
System.out.println(game.getBases()); // ooX
game.ball();
game.ball();
game.ball();
game.ball();
System.out.println(game.getBallCount()); // 4

game.ball(); // a walk
System.out.println(game.getBases()); // XoX
```

A helper method, e.g. **`shiftRunnersForWalk`**, could be helpful here.

14. There are a number of places where helper methods could simplify the code. Think about some of the tasks that occur repeatedly:

- switching to a new batter (resets balls and strikes)
- switching teams (switches top to bottom, or bottom to top plus next inning, and resets outs)
- updating score for the appropriate team, depending on whether it top or bottom
- shifting runners to next base (as described above, possibly increasing score)
- shifting runners for a walk (as described above, possibly increasing score)

The *requirements* for the class are embodied in the public API, so it is not mandatory that you use the ideas above. But they could be very helpful! And, of course, if you end up with a lot of unnecessarily duplicated code the grader may take off some points for style.

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 on the “syllabus” tab in Canvas, 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 two thirds) 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? Are you using a conditional statement when you could just be using `%`? Are you using

a loop for something that can be done with integer division? 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**.

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

Restrictions on Java Features For this Assignment

There are no restrictions on the Java features you can use to implement this assignment. However, see above the above section about using good programming practices. You may not use any outside libraries (i.e., external libraries such as Apache Commons; otherwise, we won't be able to compile your code when grading.

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 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. *However: **do not literally copy and paste from this pdf**! This leads to all kinds of weird bugs due to the potential for sophisticated document formats like Word and pdf to contain invisible characters.*
    - 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.

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 are considered to be part of the official 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

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Assignments page on Canvas, before the submission link will be visible to you.

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, **`FuzzballGame.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", linked on the Course Information page on Canvas.

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 files **FuzzballGame.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.