Com S 227
Spring 2023
Assignment 2
200 points

Due Date: Monday, February 24, 11:59 pm (midnight)
5% bonus for submitting 1 day early (by 11:59 pm February 23)
10% penalty for submitting 1 day late (by 11:59 pm February 25)
No submissions accepted after February 25, 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 **`ThreeCushion`**, that models the game of three-cushion billiards. The game is played on a billiard table and involves two players competing to be the first to reach a predetermined number of points. A player scores a point by making a successful shot. A shot is successful when the player strikes their cue ball with a cue stick, resulting in the cue ball impacting the tables cushions at least three time, while also striking into the other two balls. The minimum three cushion impacts must happen before the second ball is struck. Cushions surround the table on which the game is played (there are no pockets on the table, like in pool). The shot does not end until all balls stop motion. There are three balls; the players each chose a different cue ball from either white or yellow, the third ball is red. A player continues to play at the table, taking shots until they fail to score a point or they commit a foul. The failure or foul results in the end of one inning of the game, at which time the other player takes over taking shots at the table.

Figure 1: "2018/1 3-cushion World Cup in Bosphorus Sorgun Hotel, Antalya, Turkey", licensed under CC BY-SA 4.0

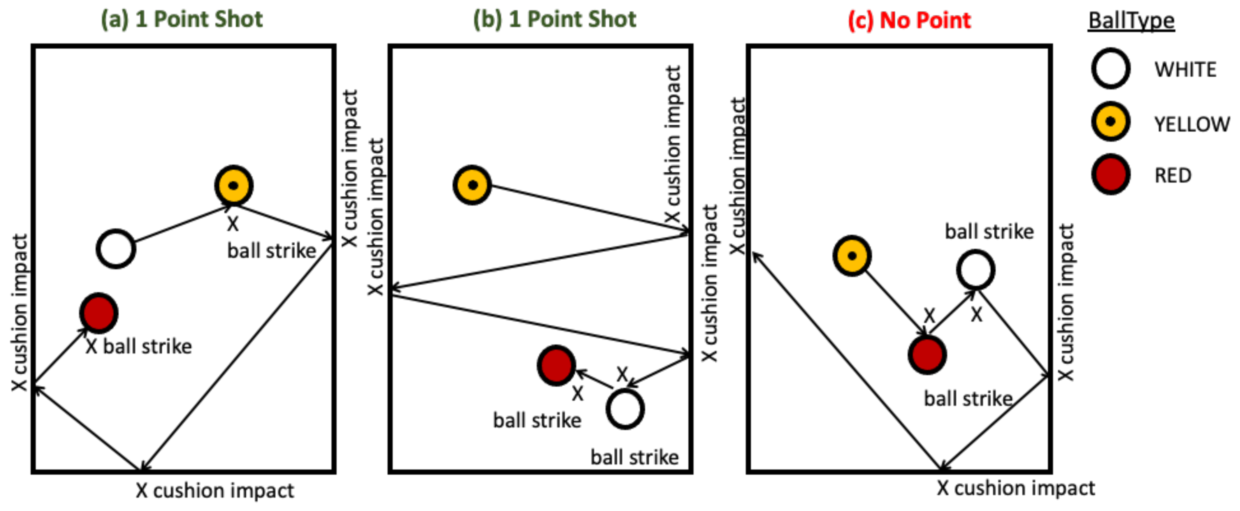

Figure 2: Examples of shots. (a) white is the cue ball, (b) and (c) yellow is the cue ball.

Terms

1. Cue stick – Pointy thing players use to hit the cue ball.
2. Cue ball – The only ball a player is allowed to strike with their cue stick. Each player is assigned a different cue ball (either white or yellow).
3. Object ball – During play, the two balls that are not the player's cue balls are that player's object balls. For example, if Player A's cue ball is yellow, then their object balls a white and red.
4. Cushion – The edges surrounding the table the balls impact and bounce off of to keep them from rolling off the table.
5. Shot – A player takes a shot by striking their cue ball. The shot ends only when all balls have stopped motion.
6. Inning – One player stays at the table taking consecutive shots until they miss a shot (Rule 2), commit a foul (Rule 4), or the game is over (Rule 6).
7. Inning player – The player currently at the table taking shots. This player keeps taking shots until their inning ends (see Inning).
8. Game – Consists of one or more innings. The game ends when one of the players reaches a predetermined number of points.
9. Break shot – The first shot of the game. This shot has specific rules different than a regular shot (Rule 3).
10. Lagging – A simple contest (similar to flipping a coin) that is used to determine who gets to take the break shot and pick their cue ball.

Rules

Note: The following rules are a modified and simplified version of the "United States Billiard Association Three-Cushion Rules" published by the United States Billiard Association, April 6, 2009. In most cases, these rules are similar to online examples of three-cushion billiards, but for the purposes of this assignment below are the "official" rules. This video gives a nice overview, but does not replace the full rules: https://www.youtube.com/watch?v=qRMCaOeWmqE

1) Lagging to determine break shot and cue ball assignments.
   a) Before the game starts, the players have a simple contest called lagging to determine who will take the break shot (first shot of the game) and to assign the cue balls.
   b) The winner of the lag has the right to the break shot or to assign the break shot to the opponent.
   c) The winner of the lag has the choice of cue balls (white or yellow), which is then used for the duration of the game.
2) A shot scores a point when the player's cue ball has struck both object balls and has made three or more cushion impacts before striking the last object ball. The following scenarios illustrate this rule:
   a) The cue ball strikes the first object ball and then strikes three or more cushions before striking the second object ball;
   b) The cue ball strikes three or more cushions and then strikes the two object balls;
   c) The cue ball strikes three or more cushions, then strikes the first object ball, then strikes one or more cushions and then strikes the second object ball;
   d) The cue ball strikes a cushion, then strikes the first object ball, and then strikes two or more cushions before striking the second object ball;

   e) The cue ball strikes two cushions, then strikes the first object ball, and then strikes one or more cushions before striking the second object ball.
   f) In a valid shot, the cue ball is allowed to strike the object balls any number of times, as long as both balls are each struck at least once.
   g) A three-cushion count means three impacts. These impacts need not be on three different cushions. A valid count may also be executed on one cushion or on two cushions.
   h) A shot is completed after all 3 balls have stopped moving and spinning.
3) The break shot.
   a) The player executes the break shot by stroking the cue ball intending to contact the red ball first. Failure to contact the red ball first and directly is a foul and the player's inning ends.
   b) On subsequent shots either the red ball or the opponent's cue ball may be used as the first object ball.
4) When the player at the table commits a foul during a shot, no point is scored, and the player's inning ends. The following are fouls that end a player's inning:
   a) Break shot foul (Rule 3a);
   b) Striking any ball other than the player's cue ball with the cue stick is a foul;
   c) Starting a new shot while balls are still in motion from the previous shot (Rule 5).
5) No shot shall be started while the balls are in motion or are spinning. If a player disregards this rule, it is a foul and the player's inning ends.
6) A game is ended when a player scores the number of points designated as constituting a game, although the opponent may have had one less turn at the table.

Common shots
From Rule 2, we see there are many possible ways to score a point. Some shots are more common than others. One particularly difficult shot is the bank shot. In this shot, the cue ball makes the required 3 cushion impacts before striking both object balls. There may be additional cushion impacts in between the two object ball strikes. An example is shown in Figure 2(b).

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.

What is BallType and PlayerPosition and how to use them?

An enum (short for enumeration) is a datatype that is restricted to predefined constants. For example, the enum `BallType` (defined in `api/BallType.java`) can be either `WHITE`, `YELLOW`, or `RED`. So, we can declare a variable of type `BallType`

```
BallType ball;
```

and then we can assign one of the predefined constants to this variable.

```
ball = WHITE;
```

Below is a more detailed example of how you can use enums in a method. Notice that the enum data types can be used for both the parameters and the return type of a method.

```
/**
 * An example method that returns a ball type for a given player.
 * Player A returns the yellow ball.
 * Player B returns the red ball.
 *
 * @param player the given player
 * @return the chosen ball color
 */
private static BallType getChosenBall(PlayerPosition player) {
      if (player == PLAYER_A) {
            return YELLOW;
      }
      // must be player B
      return RED;
}
```

One final feature of enum you may find useful, or should at least be aware of, is that like a class type variable, an enum type variable can be assigned the value `null`. In fact, an enum type instance variable is initialized to `null` by default. Notice that as a consequence the logic in the above method may not be entirely thought out, what happens if player is `null`? Here is an example of how null may be used intentionally.

```
public class Example {
      private BallType ball; // null by default

      public void displayBallMessage() {
            if (ball == null) {
                  System.out.println("No ball has been assigned yet.");
            } else {
                  System.out.println("The ball color is " + ball);
            }
      }
}
```

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. Since this is our first assignment, here is an 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 packages named `api` and `hw2`. Download the skeleton code provided with the assignment. Copy into your project the three files: `BallType.java`, `PlayerPosition.java`, and `ThreeCushion.java`. The easiest way to do this is by unzipping the skeleton zip file and drag-and-dropping the files into `src` in Eclipse. All of the code you write for this assignment will be in `ThreeCushion.java`.

3. At this point, the `ThreeCushion` class will not even compile. Getting the code to compile should be your top priority. Add stubs for all the methods and the constructor 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) as a placeholder for now. At this point there should be no compile errors in the project.

4. In a separate file, try a very simple test like this:

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

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

```
Player A: 0, Player B: 0, Inning: 0 not started
```

Note that in printing the variable `game`, the `println` method automatically invokes the method `game.toString()`.

5. The initial values seen in the output string above are not quite right. We should start the inning count at 1. You might start with the constructor `ThreeCushion()` and the method `getInning()`, which implies defining an instance variable to keep track of the current player. Once you have it initialized correctly in the constructor, the test above should give you the output:

```
Player A: 0, Player B: 0, Inning: 1 not started
```

As you continue development, you may realize there are other instance variables that need to be initialized to specific values, update the constructor as needed.

6. Next you could implement the first action required to start the game, which is the lag winner deciding who will take the break shot and which cue ball they will use. Implement the method **`lagWinnerChooses()`**. It can be tested in the following way.

```
// The lag winner chooses to take the break shot and take the white
// cue ball.
game.lagWinnerChooses(true, WHITE);

System.out.println();
System.out.println("Test 2:");
System.out.println("The shot player is " + game.getInningPlayer()
      + ", expected PLAYER_A");
System.out.println("The cue ball is " + game.getCueBall()
      + ", expected WHITE");
System.out.println("This is the break shot " + game.isBreakShot()
      + ", expected true");
System.out.println("Stats:    " + game);
System.out.println(
      "Expected: Player A*: 0, Player B: 0, Inning: 1 not started");
```

You'll need additional instance variables to store the current inning player, the player's cue ball, and whether the shot is the break shot (first shot of the game). Their values are returned by the corresponding accessor methods **`getInningPlayer()`**, **`getCueBall()`**, and **`isBreakShot()`**. Implement the accessor methods now. *Remember that accessor methods never modify instance variables, they only observe them and return information.*

7. A shot starts with a call to **`cueStickStrike()`**. The shot is valid only if the stuck ball is the player's cue ball, otherwise, the player has committed a foul. Now is where you need to start thinking about how you will keep track of valid shots and fouls. There isn't a single right answer, in fact, there are likely dozens of correct approaches. If you aren't sure, start with something simple for now and improve on it as your understanding of the problem grows. For right now, call the method **`foul()`** when a foul is committed and add an instance variable to keep track of whether the shot is valid or not. You don't need to implement the **`foul()`** method yet, just calling the method stub is enough for now. You will also need to keep track of whether or not the shot is started so the information can be returned in the accessor method **`isShotStarted()`**. If the shot is the first of the inning, the accessor method **`isInningStarted()`** should also now return true. The following is an example test for a player taking a shot on their own cue ball. No point has been scored yet.

```
// Player A takes a shot.
game.cueStickStrike(WHITE);

System.out.println();
System.out.println("Test 3:");
System.out.println("The shot has started is " + game.isShotStarted()
      + ", expected true");
System.out.println("Stats:    " + game);
System.out.println(
      "Expected: Player A*: 0, Player B: 0, Inning: 1 started");
```

8. Now that we can start a shot, a good method to implement next is **`endShot()`**. A call to this method indicates that all balls have stopped motion. Several things need to happen at the end of a shot. For example, **`isShotStarted()`** should now return false and the break shot is also over so **`isBreakShot()`** should also return false. If the shot is not valid (no point scored), then the inning has ended, meaning the inning number should be incremented and **`getCueBall()`** should return the other player's cue ball. On the other hand, if the shot is valid, the players score should be incremented by one point.

```
// All balls have stopped motion.
game.endShot();

System.out.println();
System.out.println("Test 4:");
System.out.println("The shot has started is " + game.isShotStarted()
      + ", expected false");
System.out.println("This is the break shot " + game.isBreakShot()
      + ", expected false");
System.out.println("The cue ball is " + game.getCueBall()
      + ", expected YELLOW");
System.out.println("Stats:    " + game);
System.out.println(
      "Expected: Player A: 0, Player B*: 0, Inning: 2 not started");
```

9. In step 6, you called the **`foul()`** method, so now is a good time to work on its implementation. A foul can be called at any time during the game, either before or during a shot. If it is called during a shot the inning immediately ends and the other player's inning begins. However, be aware that the balls on the table may still be in motion, meaning **`endShot()`** will be called sometime after the call to **`foul()`**. In the previous step you starting to think of a strategy to keep track of whether a shot is valid or not, as you implement **`foul()`** you may find the need to reevaluate that strategy. For now, keep the implementation of foul() simple. When a foul is committed the inning changes to the other player, meaning the inning count needs to be incremented and the other player's cue ball become the current cue ball.

```
// A foul has been committed by Player B.
game.foul();

System.out.println();
System.out.println("Test 5:");
System.out.println("The cue ball is " + game.getCueBall()
      + ", expected WHITE");
System.out.println("Stats:    " + game);
System.out.println(
      "Expected: Player A*: 0, Player B: 0, Inning: 3 not started");
```

10. Going back to **`cueStickStrike()`**, it would be good to now test what happens when the player strikes the wrong cue ball. The inning count should be incremented and the inning flipped over to the other player. Don't forget to also update the cue ball.

```
// Player A strikes the wrong cue ball committing a foul.
game.cueStickStrike(YELLOW);
```

```
System.out.println();
System.out.println("Test 6:");
System.out.println("Stats:    " + game);
System.out.println(
      "Expected: Player A: 0, Player B*: 0, Inning: 4 not started");
```

11. We have tested many cases where the shot is not valid or a foul, finally it is time to test the case when the shot is valid. It is time to implement the methods `cueBallStrike()` and `cueBallImpactCushion()`. Once again, you will need to revisit the strategy you originally devised in step 6 for keeping track of whether or not the shot is valid. As stated before, there are possibly dozens of correct solutions. When the logic becomes too difficult it can help to list multiple different scenarios for how the shot can go and then think though how the program should respond to each scenario. Be willing to experiment with different instance variables to keep track of the shot. It's not an easy problem, you need to be willing to take some time thinking of a solution, implementing it, and then testing and possibly debugging your solution. Here is some test code to get started, but clearly these tests do not cover every possibility.

```
// All balls stop motion.
game.endShot();
// Player B strikes the correct cue ball starting the next shot.
game.cueStickStrike(YELLOW);

System.out.println();
System.out.println("Test 7:");
System.out.println("Stats:    " + game);
System.out.println(
      "Expected: Player A: 0, Player B*: 0, Inning: 4 started");
// The shot is valid.
game.cueBallStrike(RED);

game.cueBallImpactCushion();
game.cueBallImpactCushion();
game.cueBallImpactCushion();
game.cueBallStrike(WHITE);
game.endShot();

System.out.println();
System.out.println("Test 8:");
System.out.println("Stats:    " + game);
System.out.println(
      "Expected: Player A: 0, Player B*: 1, Inning: 4 started");
```

12. The API also requires detection of a special type of shot know as a bank shot. Implement the method `isBankShot()`. Again, you will need to revisit your instance variables and decide if they are sufficient for detecting a bank shot or if additional information is needed.

```
// Player B makes a bank shot
game.cueStickStrike(YELLOW);
game.cueBallImpactCushion();
game.cueBallImpactCushion();
game.cueBallImpactCushion();
game.cueBallStrike(RED);
game.cueBallImpactCushion();
```

```
        game.cueBallStrike(WHITE);
        game.endShot();

        System.out.println();
        System.out.println("Test 9:");
        System.out.println(
              "This is a bank shot " + game.isBankShot() + ", expected true");
        System.out.println("Stats:    " + game);
        System.out.println(
              "Expected: Player A: 0, Player B*: 2, Inning: 4 started");
```

13. Finally, it is time to know when the game has ended. Implement **`isGameEnded()`**.

```
        // Player B scores a third point and wins the game
        game.cueStickStrike(YELLOW);
        game.cueBallImpactCushion();
        game.cueBallStrike(WHITE);
        game.cueBallImpactCushion();
        game.cueBallImpactCushion();
        game.cueBallStrike(RED);
        game.endShot();

        System.out.println();
        System.out.println("Test 10:");
        System.out.println("This is a bank shot " + game.isBankShot()
              + ", expected false");
        System.out.println("Stats:    " + game);
        System.out.println(
              "Expected: Player A: 0, Player B*: 3, Inning: 4 started, game
        result final");
```

14. At this point, the main functionality has been implemented, but there are still some details that have not been tested. For example, the break shot (the first shot of the game) has special requirements (see Rule 3). Review the rules in the Overview section of this document to make sure you haven't missed anything.

15. At some point, download the SpecChecker, import it into your project as you did in lab 1 and run it. *Always start reading error messages from the top.* If you have a missing or extra public method, if the method names or declarations are incorrect, or if something is really wrong like the class having the incorrect name or package, any such errors will appear *first* in the output and will usually say "Class does not conform to specification." **Always fix these first.**

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 #10 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? Are you using a conditional statement when you could just be using **`Math.min`**? 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.

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 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

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, **`ThreeCushion.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 **ThreeCushion.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.