Com S 227
Fall 2022
Assignment 2
180 points

Due Date: Friday, September 30, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Sept 29)

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

No submissions accepted after Oct 2, 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.

Please start the assignment as soon as possible and 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!

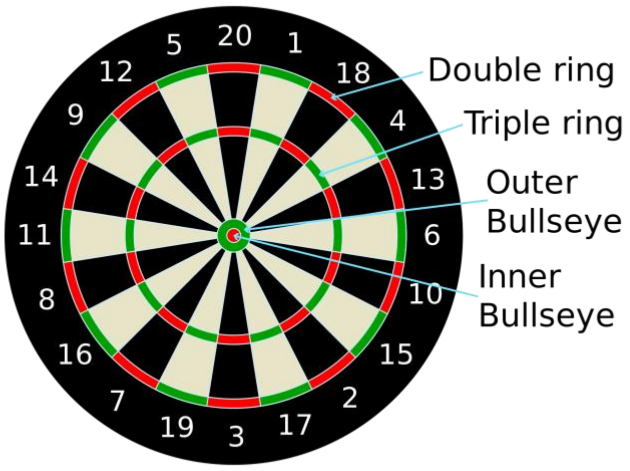
This is a "regular" assignment so we are going to read your code. Your score will be based partly (about a third) on the specchecker's functional tests and partly on the grader's assessment of the quality of your code. See the "More about grading" section.

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. You'll create one class, called `DartGame`, that is a model of a typical game of "darts".

If you have never played darts, don't worry, neither have I! I just looked up the rules on Wikipedia! It works like this. Two players, whom we will call by the charming names of "Player 0" and "Player 1", typically in some state of inebriation, take turns throwing these dangerously sharp missiles at a round target hanging on the wall. Here is what the dartboard looks like (courtesy of Wikipedia):



Each player has a fixed number of darts, typically 3, and starts with some number of points, typically 501 or 301. Both these values will be configurable in our game. The object is to be the first player to reduce their score to zero. The player throws a dart at the dartboard and their score is reduced by a number that depends on where the dart lands. You can see the board is divided into pie-shaped segments, each labeled with a number, and there is also a "double ring" and a "triple ring". As an example, if the dart lands in the white area of the segment labeled 15, the player gets 15 points (i.e., their score goes *down* by 15 points). But if it lands in the green area of the double ring within that segment, the 15 is doubled, so the player gets 30 points off their score; likewise for the triple ring it would be 45 points. The outer bullseye is always 25 points and the inner bullseye is 50 points.

In a player's turn they normally throw all of their darts, and then it's the other player's turn, and so on, unless the player either *wins* or *goes bust*. In order to win, the player has to get their score down to exactly zero, ending with a throw that lands in either the double ring or in the inner bullseye. This is called "doubling out". The game ends at that point and the player does not have to use their remaining darts. If the score goes negative, gets to 1, or gets to zero but not on a double or inner bullseye, the player has gone *bust*, and their score is reset to whatever it was at the beginning of their turn, and the turn ends even if they haven't thrown all their darts.

We also implement the rule called "doubling in", which is that at the beginning, none of the player's throws will be counted until they have first made one throw that lands in either the double ring or the inner bullseye.

The **`DartGame`** class has to keep track of the score for each player, whose turn it is (the "current" player), and how many darts they have left to throw. Most of the logic is in the method:

```
  public void throwDart(ThrowType type, int number)
```

which simulates the current player throwing a dart and adjusts the score, the player's dart count, and whose turn it is, as needed. The arguments represent where the dart lands on the dartboard, indicating which segment (the **`number`**) and whether it was a single, double, triple, or bullseye (the **`type`**). (The **`number`** parameter is ignored in case of a miss or bullseye.). The **`type`** parameter is a constant of the type **`ThrowType`**, described below.

The enumerated type api.ThrowType

To describe the different kinds of throws, we use the six constants:

```
ThrowType.MISS
ThrowType.SINGLE
ThrowType.DOUBLE
ThrowType.TRIPLE
ThrowType.OUTER_BULLSEYE
ThrowType.INNER_BULLSEYE
```

These are defined as an "enumerated" type or **`enum`**. You can basically use them as though they had been defined as **`public static final`** constants. You can't construct instances of this type, you just use the six names as literal values, as though they were primitives. Use **`==`** to when checking whether two values of this type are the same, e.g. if **`t`** is a variable of type **`ThrowType`**, you can write things like

```
      if (t == ThrowType.DOUBLE)
      {
        // do cool stuff
      }
```

An **`enum`** type can also be used as the switch expression in a **`switch`** statement.

Tip: add the line

```
import static api.ThrowType.*;
```

to the top of your file. Then you can refer to the six constants without having to type "**`ThrowType.`**" in front of them.

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.

Using helper methods

You are expected to make some effort to use procedural decomposition to simplify your logic and reduce code duplication. in steps 4 and 5 of the Getting Started section there are a couple of *possible* examples illustrating how private “helper methods” might be used in this assignment, but those are just suggestions. Remember that any method you write that is not in the public API needs to be declared **`private`**, and 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 two packages called `hw2` and `api`. ***Be sure to choose "Don't Create" at the dialog that asks whether you want to create module-info.java***.

2. Copy the `ThrowType` class into the `api` package. Go ahead and create the `DartGame` class in the `hw2` package. You can copy the code from the posted skeleton (that has the `toString` method already implemented). Add stubs for all the methods and constructors described in the Javadoc. Document each one. For methods that need to return a value, just return a "dummy" value as a placeholder. At this point there should be no compile errors in the project.
Note: do not modify the `ThrowType` *or add anything else to the api package.*

3. Try a very simple test like this:

```
public class SimpleTests
{
  public static void main(String[] args)
  {
    DartGame g = new DartGame(1, 100, 3);
    System.out.println(g.getScore(0));        // expected 100
    System.out.println(g.getScore(1));        // expected 100
    System.out.println(g.getCurrentPlayer()); // expected 1
    System.out.println(g.getDartCount());     // expected 3
  }
}
```

This will get you thinking about what instance variables you need to store things like the score for each player, which is the current player, and how many darts they have left to throw in the current turn. The values you should see are shown above.

You can also very easily check all the values above by using the `toString()` method, e.g.

```
      System.out.println(g.toString());
```

Which (due to some Java magic) can be abbreviated to:

```
      System.out.println(g);
```

You should see an output string like this,

```
      Player 0: 100  Player 1: 100  Current: Player 1  Darts: 3
```

The **`toString`** method itself is already implemented for you; notice it just calls your other accessor methods to get the values to put in the string.

4. Next you could start thinking about some of the things that have to happen in **`throwDart`**, e.g., adjusting the scores and counting the darts. For the sake of experimentation, until you get around to the **`calcPoints`** method (item #6) you might just hard-code the points associated with a given throw, e.g., 20 points for **`throwDart(ThrowType.DOUBLE, 10)`** so you can start on the logic of **`throwDart`**. Then try something like this,

```
    g = new DartGame(1, 100, 3);
    g.throwDart(DOUBLE, 10);
    System.out.println(g);
    g.throwDart(DOUBLE, 10);
    System.out.println(g);
    g.throwDart(DOUBLE, 10);
    System.out.println(g);
    g.throwDart(DOUBLE, 10);
    System.out.println(g);
```

which should produce the output:

```
Player 0: 100  Player 1: 80  Current: Player 1  Darts: 2
Player 0: 100  Player 1: 60  Current: Player 1  Darts: 1
Player 0: 100  Player 1: 40  Current: Player 0  Darts: 3
Player 0: 80  Player 1: 40  Current: Player 0  Darts: 2
```

Note we have assumed that you have the line: **`import static hw2.ThrowType.*;`** at the top of your file; otherwise, you need to type out **`ThrowType.DOUBLE`** instead of just **`DOUBLE`** as in the examples above.

Now, if you do **`g.throwDart(DOUBLE, 10)`** again at this point, it should reduce player 0's score. Notice there is some logic involved in reducing the score of the *current* player. Rather than clutter up your **`throwDart`** method with a lot of conditionals to check whether it's currently player 0 or player 1 throwing, you might consider making a private helper method to take care of that logic, e.g.,

```
  /**
   * Reduces the score for the current player by the given amount.
   * @param amount
   *   number of points to subtract
   */
  private void adjustScore(int amount)
```

5. Similarly, there may be several places where you need to switch the current player and reinitialize things like the number of darts left in the current player's turn. Again, rather than repeat the same code in multiple places in **`throwDart`**, it might make sense to create a helper method for this task, e.g., something like,

```
/**
 * Switches players and resets the dart count and
 * the starting score for the current player's turn.
 */
private void switchPlayer()
```

6. In general, the logic to determine the number of points to deduct for a given combination of throw type and number is a straightforward calculation that doesn't depend on anything about the current state of the game, so we have isolated it as the static method **`calcPoints`**. You can implement this anytime. Start with a simple test, of course:

```
System.out.println(DartGame.calcPoints(MISS, 42));// expected 0
                                                  //(second arg ignored)
System.out.println(DartGame.calcPoints(SINGLE, 7));  // expected 7
System.out.println(DartGame.calcPoints(DOUBLE, 10)); // 20
System.out.println(DartGame.calcPoints(TRIPLE, 8));  // 24
System.out.println(DartGame.calcPoints(OUTER_BULLSEYE, 42)); // 25
System.out.println(DartGame.calcPoints(INNER_BULLSEYE, 42)); // 50
```

7. Now, what if a player "goes bust"? That's what happens if the score goes down to 1, goes negative, or goes to zero but not on a double or inner bullseye. In those cases, you have to
   a) restore the player's score to whatever it was at the start of their turn, and
   b) immediately switch to the other player
Hmm. We need a test case.

```
g = new DartGame(1, 50, 3);
g.throwDart(DOUBLE, 15);
System.out.println(g);
g.throwDart(DOUBLE, 12);
System.out.println(g);
```

This should give the output,

```
Player 0: 50  Player 1: 20  Current: Player 1  Darts: 2
Player 0: 50  Player 1: 50  Current: Player 0  Darts: 3
```

That is, after the first throw Player 1 has 20 points and two darts left. But since the second throw is 24 points, that puts them below zero, so their original score (50 at start of the turn) is restored, and it switches to Player 0's turn. Come up with some additional test cases, for example, when the score at the start of the turn isn't the original game starting score, or when the player really reaches 0, but not on a double or inner bullseye. For example,

```
g = new DartGame(1, 50, 3);
g.throwDart(DOUBLE, 15);
System.out.println(g);
g.throwDart(SINGLE, 5);
g.throwDart(SINGLE, 5);
System.out.println(g);  // player 0's turn, player 1 score now 10
g.throwDart(DOUBLE, 1);
g.throwDart(DOUBLE, 1);
g.throwDart(DOUBLE, 1); // player 0 having a bad day
System.out.println(g);
g.throwDart(SINGLE, 5);
g.throwDart(SINGLE, 5); // player 1 gone bust
System.out.println(g);  // player 0's turn, player 1 score back to 10
```

8. Next, how do you detect a win? After you do that, notice that the only way a player ever ends up with a score of zero after a turn is by winning (since otherwise it's a bust and their previous score is restored). That makes it easy to implement the methods **isOver** and **whoWon**. Note two additional things about the spec: the winning player should remain the "current" player (according to getCurrentPlayer), and after the game ends, the **throwDart** method should do nothing.

9. The next twist in the rules is "doubling in". None of the player's throws affect their score until they throw either a double or an inner bullseye. A first test case:

```
g = new DartGame(1, 100, 3);
g.throwDart(SINGLE, 10);
System.out.println(g);   // score still 100, not doubled in yet
g.throwDart(DOUBLE, 7);
System.out.println(g);   // ok, score now 86
```

10. What if a player doubles in but simultaneously goes bust? This can't happen in a "real" game where the starting score is 301 or 501, but since the starting score is configurable for us, we need to account for the possibility. We specify that if the player goes bust on their first double or inner bullseye, we count them as having doubled in, but still end their turn and switch to the other player.

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.

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, **`DartGame.java`**. Please **LOOK** at the file you upload and make sure it is the right one!

Your submission does not include the **`api`** package.

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 **DartGame.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.