Com S 227
Spring 2018
Assignment 2
200 points
Due Date: Thursday, September 27, 11:59 pm (midnight)
"Late" deadline: Friday, September 28, 11:59 pm
(Remember that Exam 1 is MONDAY, October 1.)

General information

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

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

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

Note: Our first exam is Monday, October 1, which is just four days after the due date for this assignment. It will not be possible to have the assignments graded and returned to you prior to the exam. We can post a sample solution on September 29.

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

Introduction

The purpose of this assignment is to give you lots of practice working with conditional logic. For this assignment you’ll create two classes, TennisGame and Set, that comprise a simplified model of the game of tennis. (The project also uses a type called BallDirection, but it is provided for you and you should not modify it.)
Whoa, you say, but I don't know anything about tennis. Do not worry, until the day before yesterday neither did we! We just read about the rules on Wikipedia, made some simplifications, and wrote up a specification. As with our previous assignment, all you really need to do is implement the methods as specified and you'll be fine.

**Overview of the game**

Here is a high-level overview of the game: Two people on opposite sides of a net use funny-looking paddles to hit a ball back and forth until one of them messes up. The one that doesn't mess up gets a point. As soon as one of them gets enough points, that player wins the game. A "set" is a sequence of games between the same two players. As soon as one of the players wins enough games, that player wins the set. (Then a "match" is a sequence of some number of sets, and the player who wins the majority of sets wins the match, at which point they can say "game, set, and match" and mean it literally. However, in this homework we are not implementing matches.)

More precisely, the two players are called the *server* and the *receiver*, whose roles remain the same throughout a game. The server initially hits, or *serves*, the ball toward the receiver. If the server makes an error in serving the ball, called a *service fault*, then the ball is served again. However, if there are two service faults in a row, the receiver is awarded a point; then the server starts again (with the count of faults reset to zero). Once the ball is served without a fault, it is said to be *in play*, and a *rally* begins, which is where the players hit the ball back and forth towards one another. The ball has to go over the net and land within the boundary rectangle on the other side, and is allowed to bounce once before the other player hits it back. Various things can end the rally:

1. A player *faults* by hitting the ball into the net or into the stands or somewhere other than across the net, or by allowing the ball to bounce twice before hitting it. Then the other player is awarded a point.

2. A player *misses* the ball, perhaps intentionally. If the ball then lands out of bounds, the player gets a point. However, if the ball lands within the boundary, the other player gets the point.

Note that even when a player hits the ball in such a way that its trajectory *would* take it out of bounds, that fact does not immediately mean that the player loses the point, because the other player might decide to hit the ball before it bounces.

The game ends when one player has more than three points *and* has a margin of at least two points more than the other player.
Sets

A set consists of a sequence of games, with the two players alternating in the role of server. We will call them by the charming names "player 0" and "player 1". A player wins the set by winning some minimum number of games and having at least two more wins than the other player. Traditionally, this minimum number is six, but in our model the number is configured in the constructor. (Note also that there is no "tiebreak" game when the score reaches 6-6, as occurs in some traditional forms of the game that you might read about on the internet.)

High-level overview of the Game API

*Full details for all methods and their arguments can be found in the online javadoc.*

Note that at any given moment during the tennis game the ball is either moving in the direction of the server, moving in the direction of the receiver, or else is not in play (i.e., when waiting to be served, or when the game is over). The basic operations of the game are `serve`, `hit`, or `miss`, represented by the methods:

```java
public void serve(boolean serviceFault)
public void hit(boolean fault, boolean headedOutOfBounds)
public void miss()
```

After a call to `serve(false)` (no service fault) the direction of the ball is always toward the receiver. In case of a service fault, the ball remains out of play; as noted in the game overview, when there are two service faults in a row, a point is awarded to the receiver and the count of service faults is reset to zero.

When there is a call to `hit(false, false)` or `hit(false, true)`, the ball simply switches direction. The first argument is used to indicate that the player making the hit has faulted, so after `hit(true, ...)`, a point is awarded to the other player, and the ball is then out of play. The second argument to hit may be `true` to indicate that the ball is moving on a trajectory that would cause it to land out of bounds. We have to keep track of this piece of information since it may affect who gets a point, if the next operation is `miss()`. (Note the second argument to `hit()` should be ignored when the first argument is `true`.)

For a call to `miss()`, a point is awarded to one of the players as follows: If the ball is headed out of bounds based on a previous call to `hit(false, true)`, the point is awarded to the player
toward whom the ball was moving, since it landed out of bounds; otherwise, the point is awarded to the other player. After a call to `miss()`, the ball is always out of play.

Note: You may notice that the `serve()` method doesn't include an argument to indicate whether the serve is headed out of bounds. One of the rules of tennis is that the receiver always has to allow the serve to bounce once, and if it doesn't bounce within the specified rectangle, it's counted as a service fault.

**The enum type BallDirection**

**NOTE: The BallDirection type is provided for you and you should not modify it.**
The method `getBallStatus()` is supposed to tell us whether the ball is moving toward the receiver, moving toward the server, or out of play. The return type of this method is `BallDirection`, which is an `enum` type. You can read about `enum` types in "Special Topic 5.4" at the end of Section 5.4 of the text, but the idea is very simple. The `BallDirection` type just defines three constants that we can use for indicating the three possible states of the ball. You can use them in your code similar to the way you'd use a numeric constant. You don't construct them with `new` (they already exist) and you can compare them using `==` and `!=`. Variables of type `BallDirection` are "type-safe": they can only store one of the three predefined values. As an example, suppose you have an instance variable, say `direction`, of type `BallDirection`. You could write code like this to switch the ball's direction:

```java
if (direction == BallDirection.TOWARD_RECEIVER)
{
    direction = BallDirection.TOWARD_SERVER;
}
...```

Here is a tip: if you put the declaration

```java
import static hw2.BallDirection.*;
```

at the top of your file, then you can omit the prefix "BallDirection."

**The Set API**

A `Set` always maintains a private reference to a "current game". This reference is initialized to a new game when the `Set` is constructed and is re-initialized to a new game whenever the method `newGame()` is called (provided the existing current game is over and the set has not ended). The game is controlled indirectly through the `Set` methods `serve()`, `hit()`, and `miss()`, which are analogues to the `TennisGame` methods of the same name. In fact, the effect of calling `serve(false)` on a `Set` is that it in turn just `serve(false)` on its current game (after making
ensure the set is not over and the current game is not over). The logic is similar for hit() and miss().

One of the things that the Set is supposed to simulate is that the two players take turns being the server, so we'll have to keep track of who is currently the server, and arrange for this value to switch (from 0 to 1 or from 1 to 0) whenever newGame() creates a new game. There is a public accessor method whoIsServing() that returns 0 or 1 to indicate who is the server in the current game. There is an argument to the constructor to indicate whether player 0 or player 1 is the server of the first game.

The Set keeps a count of games won by each player. The set ends when one of the players has reached the minimum number of wins and has at least two more wins than the other player. Traditionally it takes six games to win a set, but in our model this minimum value is configured by an argument to the constructor.

Testing and the SpecCheckers

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

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

SpecChecker 1

Your class must conform precisely to this specification. The most basic part of the specification includes the class name and package, the required constants, the public method names and return types, and the types of the parameters. We will provide you with a specchecker to verify that your class satisfies all these aspects of the specification and does not attempt to add any public attributes or methods to those specified. If your class structure conforms to the spec, you should see a message such as "3 out of 3 tests pass" in the console output. (This specchecker will not offer to create a zip file for you). Remember that your instance variables should always be declared private, and if you want to add any additional “helper” methods that are not specified, they must be declared private as well.

SpecChecker 2

In addition, since this is the second assignment and we have not had a chance to discuss unit testing very much, we will also provide a specchecker that will run some simple functional tests for you. This is similar to the specchecker you used in Assignment 1. It will also offer to create
a zip file for you to submit. *Specchecker 2 should be available by September 24. Please do not wait until that time to start testing!*

**More about grading**

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

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

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

**Note on defining constants**

In implementing the game scoring you will have to refer specifically to the numbers 0, 1, 2, and 3, and to the string values "love", "15", "30", "40", "advantage in", etc. You are NOT required to define symbolic constants for all of these, you can leave the literal values in your code.

**Style and documentation**

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

- Each class, method, constructor and instance variable, whether public or private, must have a meaningful and complete Javadoc comment. Class javadoc must include the **@author** tag, and method javadoc must include **@param** and **@return** tags as appropriate.
  - Try to state what each method does in your own words, but there is no rule against copying and pasting the descriptions from this document.
  - Run the javadoc tool and see what your documentation looks like! You do not have to turn in the generated html, but at least it provides some satisfaction :(
• All variable names must be meaningful (i.e., named for the value they store).
• Your code should not be producing console output. You may add `println` statements when debugging, but you need to remove them before submitting the code.

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

If you have questions

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

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

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

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

It probably makes sense to start with **TennisGame**, since you need to have it partially implemented to test your **Set** class. However, to some extent you can work independently on:

- a) the logic for serves, hits, misses, and keeping track of the ball direction
- b) the logic for checking whether the game is over and who won
- c) the logic for `getCallString()`
- d) the implementation of `Set`, which really only depends on (b)

1. Create a new Eclipse project and within it create a package `hw2`. Put the provided code for the **BallDirection** class in your `hw2` package. Do not modify the code for **BallDirection**.

2. Go ahead and create the **TennisGame** and **Set** classes in the `hw2` package. Add stubs for all the methods and constructors described in the Javadoc. 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.

3. Import and run the first specchecker. Your code should successfully pass the basic tests.

4. The most basic feature of tennis is that the ball goes back and forth. Check it out:

   [https://www.youtube.com/watch?v=JgAgfC8Tqok](https://www.youtube.com/watch?v=JgAgfC8Tqok)

so maybe the first thing to try to implement is keeping track of the ball's direction. Refer to the subsection about the **BallDirection** type on page 4. As always, write some simple test cases (there is source code posted for all these simple examples):

```java
TennisGame g = new Tennis Game();
// ball status should switch directions during rally
System.out.println(g.getBallStatus());  // NOT_IN_PLAY
g.serve(false);
System.out.println(g.getBallStatus());  // TOWARD_RECEIVER
g.hit(false, false);
System.out.println(g.getBallStatus());  // TOWARD_SERVER
g.hit(false, false);
System.out.println(g.getBallStatus());  // TOWARD_RECEIVER
```
If the next player faults, then the ball should be out of play:

```java
g.hit(true, false);
System.out.println(g.getBallStatus()); // NOT_IN_PLAY
```

To start out, ignore the second argument of `hit()` (trajectory out of bounds).

5. The other thing that should happen here is that the server gets a point, since the ball was moving toward the receiver:

```java
System.out.println(g.getServerPoints()); // expected 1
```

This suggests that you will need a couple of instance variables to keep track of the points for each player. Also implement `getReceiverPoints()`, `getServerPoints()` and `getScore()`.

6. Similarly, a call to `miss()` should always take the ball out of play and give someone a point.

```java
// miss should always take ball out of play
g.serve(false);
g.hit(false, false);
g.miss();
System.out.println(g.getBallStatus()); // NOT_IN_PLAY

// here, since ball was headed toward server (and not out of bounds),
// receiver gets a point
System.out.println(g.getReceiverPoints()); // expected 1
```

7. What about "service faults", i.e., when the argument to `serve()` is `true`?

```java
// serving with a service fault does not put the ball in play
g.serve(true);
System.out.println(g.getBallStatus()); // NOT_IN_PLAY

// after two service faults, receiver gets a point
g.serve(true);
System.out.println(g.getReceiverPoints()); // expected 2

// service faults start counting over again, so this doesn't
// give receiver another point
g.serve(true);
System.out.println(g.getReceiverPoints()); // expected 2
```

This example suggests you'll need another instance variable to count service faults.

8. What about the second argument to `hit()`, which is supposed to indicate that the ball is headed out of bounds? Think about what the behavior would be:
g = new TennisGame();
g.serve(false);
g.hit(false, true);
System.out.println(g.getBallStatus()); // TOWARD_SERVER

// this should be a point for server, since ball was headed out of bounds
g.miss();
System.out.println(g.getScore()); // 1-0

// ball headed out of bounds, but server hits it back anyway
g.serve(false);
g.hit(false, true);
g.hit(false, false);
System.out.println(g.getBallStatus()); // TOWARD_RECEIVER

// receiver misses, but this is a point for server since
// ball is *not* headed out of bounds
g.miss();
System.out.println(g.getScore()); // 2-0

Again the example suggests that whenever the ball is in play, you need to keep track of whether
its trajectory is out of bounds, because that piece of information may be relevant for the next
method call.

9. Independently of code above related to the ball direction, service faults, and out-of-bounds
trajectory, you can work on the methods to detect
whether the game is over and who wins.
Notice that there is a method setScore() that directly sets the number of points for the server
and receiver. If you implement that first (it is easy) you can test the logic of isOver(),
serverWon(), and receiverWon(), for example:

```
g = new TennisGame();
g.setScore(1, 3);
System.out.println(g.getScore()); // 1-3
System.out.println(g.isOver()); // expected false
System.out.println(g.receiverWon()); // expected false
System.out.println(g.serverWon()); // expected false
```

```
g = new TennisGame();
g.setScore(2, 4);
System.out.println(g.getScore()); // 2-4
System.out.println(g.isOver()); // expected true
System.out.println(g.receiverWon()); // expected true
System.out.println(g.serverWon()); // expected false
```

Notice that your implementation of isOver() can figure out whether the game is over just by
looking at the points for each player. You don't need another instance variable to represent the
game being over, and defining one will make the code more complicated.
10. As soon as you have `setScore()`, you could also work on `getCallString()`. This can be done anytime since since nothing else in `TennisGame` depends on it. As usual just start with a few test cases.

```java
// try out some call strings
TennisGame g = new TennisGame();
g.setScore(0, 3);
System.out.println(g.getCallString());  // "love-40"

g = new TennisGame();
g.setScore(2, 1);
System.out.println(g.getCallString());  // "30-15"

g = new TennisGame();
g.setScore(3, 4);
System.out.println(g.getCallString());  // "advantage out"
```

11. To test most of the logic of your `Set` class, the main thing you really need is the part of `TennisGame` that allows the score to be set and detects whether the game is over (as in step 9). As described in the Set API section on page 4, a Set always maintains a reference to a current instance of `TennisGame`. The `serve()`, `hit()` and `miss()` methods of `Set` just call the corresponding methods of `TennisGame` (after checking that the set is not over and that the current game is not over) then update the state of the Set as needed. The `fastForward()` method similarly just calls the `setScore()` method of the current game. With that, you can start to think about the basic logic of `Set`, for example:

```java
Set s = new Set(3, false);
System.out.println(s.whoIsServing());  // expected 0

// this should finish the current game with player 1 winning
s.fastForward(0, 4);
System.out.println(s.isCurrentGameOver());  // expected true
System.out.println(s.player1GamesWon());   // expected 1

// should be player 1 serving new game
s.newGame();
System.out.println(s.isCurrentGameOver());  // expected false
System.out.println(s.whoIsServing());  // expected 1
s.fastForward(4, 0);
System.out.println(s.isCurrentGameOver());  // expected true
System.out.println(s.player1GamesWon());   // expected 2

// check status string, current server is always listed first
System.out.println(s.getCurrentStatus(false)); // Set: 2-0 Game: 4-0

// should be player 0 serving new game
s.newGame();
System.out.println(s.whoIsServing());  // expected 0
s.fastForward(4, 0);
System.out.println(s.isCurrentGameOver());  // expected true
System.out.println(s.player0GamesWon());   // expected 1
```
// check status string, current server listed first
System.out.println(s.getCurrentStatus(false)); // Set: 1-2 Game: 4-0

// this set should end when player 1 wins one more game
s.newGame();
System.out.println(s.whoIsServing()); // expected 1
s.fastForward(4, 0);
System.out.println(s.isCurrentGameOver()); // expected true
System.out.println(s.player1GamesWon()); // expected 3
System.out.println(s.isSetOver()); // expected true

So you can see that the Set object needs to keep track of the number of games won for each player, and also needs to keep track of which player is serving.

12. It's simplest to check the logic of the Set class itself using the fastForward() method as above, but don't forget to also check that the basic methods are hooked up correctly, for example:

```
s = new Set(3, false);
s.serve(false);
s.hit(false, false);

// this should be a point for receiver
s.miss();
System.out.println(s.isCurrentGameOver()); // expected false
System.out.println(s.getCurrentStatus(false)); // Set: 0-0 Game: 0-1
```

**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 three files, TennisGame.java and Set.java and BallDirection.java. The file BallDirection.java should be the one provided as sample code and should not be modified. Please LOOK at the file you upload and make sure it is the right one!

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

We recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory hw2, which in turn should contain
the files TennisGame.java and Set.java and BallDirection.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.