Com S 227
Spring 2022
Assignment 2
200 points

Due Date: Monday, March 7, 11:59 pm (midnight)
5% bonus for submitting 1 day early (by 11:59 pm March 6)

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

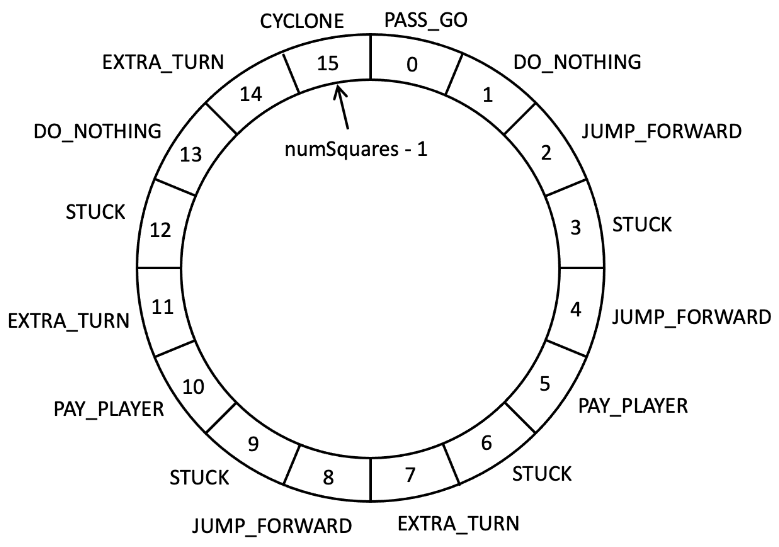
Overview

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 **`CyGame`**, that models a Monopoly-like game. The game is based around the idea of two players taking turns rolling dice and each moving a game piece over the squares on a board. The squares their piece lands on can have different rules (for example, one type of square gives a player an extra turn). Each player also has a pile of game money and can use that money to by property units. The game ends when one of the players has at least **`MONEY_TO_WIN`** or one of the players to bankrupt (negative money).

Here is how the game works. The board consists of one or more squares which are arranged sequentially and wrap around forming a circular pattern. The squares are numbered starting at 0 and ending at one less than the number of squares. Each square is assigned a single type based on the following rules. The rules are listed in order of highest to lowest precedence (i.e., when multiple rules match a square number, apply the one higher on the list). Rules:

- Square 0 is type **`PASS_GO`**.
- The very last square (before wrapping back to 0) is **`CYCLONE`**.
- Every 5th square (e.g., 5, 10, 15...) is **`PAY_PLAYER`**.
- Every 7th and every 11th square is **`EXTRA_TURN`**.
- Every 3rd square is **`STUCK`**.
- Every 2nd square is **`JUMP_FORWARD`**.
- All remaining squares are **`DO_NOTHING`**.

The figure below shows the configuration of a 16 square board.



Both players start with their game piece on square 0. They take turns rolling a dice and after each roll they move their piece forward on the board by value rolled. An exception to this rule is when a player is current on a **`STUCK`** type square, in that case they can only move forward if the dice value is an even number. When a player passes over the **`PASS_GO`** square they are awarded the amount of **PASS_GO_PRIZE** to their pile of money. The type of square their piece lands on determines some action they must perform. The actions are:

- **PASS_GO** If the player lands on **PASS_GO** award **PASS_GO_PRIZE**. A player cannot both land on **PASS_GO** and pass over **PASS_GO** (i.e., the player only gets the **PASS_GO_PRIZE** *once* each time around the board.
- **CYCLONE** CYCLONE WEATHER ALERT: The current player is moved to the same square as the other player. If the player passes go during the move no prize is awarded. Do not apply the rules of the square that is moved to.
- **PAY_PLAYER** The current player pays the other player (**PAYMENT_PER_UNIT** * number_of_other_player's_units). That is to say, the current player's money is decreased and the other player's money is increased by that amount.
- **EXTRA_TURN** The current player remains the current player for one more move (i.e., don't change the current player after this move).
- **STUCK** Stuck in the campa-meal: No additional action is taken for the remainder of the players turn. The player will not be able to leave this square until they roll an even number.
- **JUMP_FORWARD** Walk light is on to cross: The player moves forward an additional 4 squares. Do not apply the rules of the square that is moved to. If the player passes over the **PASS_GO** square add **PASS_GO_PRIZE** to the player's money.
- **DO_NOTHING** No additional action is taken.

If a player is on a **DO_NOTHING** square, at the beginning of their turn they can purchase one property unit at a cost of **UNIT_COST**. Doing so ends their current turn. At the beginning of any turn (regardless of what type square they are on) they can sell back one unit at the same price of **UNIT_COST** to increase their pile of money. Doing so ends their current turn.

The game ends when one of two conditions are meet 1) one of the players has reached a winning amount of money (**MONEY_TO_WIN**) or 2) one of the players has gone bankrupt (a negative money balance).

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. 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 the `CyGame` class in the `hw2` package. You can copy the code from the posted skeleton that has the constants and the string methods. 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.

3. Try a very simple test like this:

```
public class CyGameTest {
    public static void main(String args[]) {
        CyGame game = new CyGame(16, 200);
        System.out.println(game);
    }
}
```

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

```
Player 1: (0, 0, $0) Player 2*: (0, 0, $0)
```

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

4. The initial values seen in the output string above are not quite right. We should start at Player 1's turn. You might start with the constructor `CyGame()` and the method `getCurrentPlayer()`, 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 1*: (0, 0, $0) Player 2: (0, 0, $0)
```

5. Next you could think about tracking each players current square, money and number of property units. You'll need three instance variables for each player of these three values, and their values are returned by the corresponding accessor methods `getPlayerXSqaure()`,

getPlayerXMoney(), and **getPlayerXUnits()**. *Remember that accessor methods never modify instance variables, they only observe them and return information.*

6. Each square is assigned a type, now would be a good time to complete the **getSquareType()** method because it will be difficult to test any of the mutator methods without having square types. The rules for determining the square type are clearly described in the javadoc provided with the assignment files. The figure in the Overview Section above shows how square types should be assigned for a 16 square board. Try a simple test.

```
CyGame game = new CyGame(16, 200);
System.out.println(game.getSquareType(0));
System.out.println(game.getSquareType(1));
System.out.println(game.getSquareType(2));
System.out.println(game.getSquareType(3));
System.out.println(game.getSquareType(4));
System.out.println(game.getSquareType(5));
System.out.println(game.getSquareType(6));
System.out.println(game.getSquareType(7));
System.out.println(game.getSquareType(8));
System.out.println(game.getSquareType(9));
System.out.println(game.getSquareType(10));
System.out.println(game.getSquareType(11));
System.out.println(game.getSquareType(12));
System.out.println(game.getSquareType(13));
System.out.println(game.getSquareType(14));
System.out.println(game.getSquareType(15));
```

You should see the output:

```
1
0
5
6
5
3
6
4
5
6
3
4
6
0
4
2
```

7. You will have noticed that there are several mutator methods that will cause the game to change state by responding to action of the current player: **roll()**, **buyUnit()**, **sellUnit()**, and

`endTurn()`. The method `endTurn()` would be good to implement first because it is relatively simple and may be useful in other methods. A simple test would be:

```
CyGame game = new CyGame (16, 200);
System.out.println(game);
game.endTurn();
System.out.println(game);
game.endTurn();
System.out.println(game);
```

It should produce the result:

```
Player 1*: (0, 1, $200) Player 2: (0, 1, $200)
Player 1: (0, 1, $200) Player 2*: (0, 1, $200)
Player 1*: (0, 1, $200) Player 2: (0, 1, $200)
```

8. The next to implement would be `roll()` because it will allow us to move the players pieces which will be useful for testing other methods. There are several different rules that need to be implemented for `roll()`, create tests for each one and incrementally develop the method. For example:

```
CyGame game = new CyGame(16, 200);
System.out.println(game);

// Player 1 to JUMP_FORWARD square
game.roll(2);
System.out.println("Expect Player 1 on sqaure 2 + 4 = 6.");
System.out.println(game);

// Player 2 to PAY_PLAYER square
game.roll(5);
System.out.println("Expect Player 1 money 220.");
System.out.println("Expect Player 2 money 180.");
System.out.println(game);

// Player 1 is now on STUCK, roll an odd value
game.roll(5);
System.out.println("Expect Player 1 on sqaure 6 (not
moved).");
System.out.println(game);

// Player 2 to EXTRA_TURN
game.roll(2);
System.out.println("Expect Player 2 is still current
player.");
System.out.println(game);
game.roll(6);
```

```
System.out.println("Expect Player 1 is now current
player.");
System.out.println(game);

// Player 1 passes turn
game.endTurn();
System.out.println("Expect Player 1 has not changed.");
System.out.println(game);

// Player 2 buys property unit
game.buyUnit();
System.out.println("Expect Player 2 has 2 units and it is
now Player 1's turn.");
System.out.println(game);

// Player 1 passes turn and Player 2 to CYCLONE
game.endTurn();
game.roll(2);
System.out.println("Expect Player 2 is in same location as
Player 1.");
System.out.println(game);

// Player 1 passing though PASS_GO
game.roll(2);
game.roll(2);
game.roll(6);
System.out.println("Expect Player 1 has $420");
System.out.println(game);

// Player 1 has over $400, the game is over
System.out.println("Expect game over.");
System.out.println("Is game ended: " + game.isGameEnded());
```

It should produce the result:

```
Player 1*: (0, 1, $200) Player 2: (0, 1, $200)
Expect Player 1 on sqaure 2 + 4 = 6.
Player 1: (6, 1, $200) Player 2*: (0, 1, $200)
Expect Player 1 money 220.
Expect Player 2 money 180.
Player 1*: (6, 1, $220) Player 2: (5, 1, $180)
Expect Player 1 on sqaure 6 (not moved).
Player 1: (6, 1, $220) Player 2*: (5, 1, $180)
Expect Player 2 is still current player.
Player 1: (6, 1, $220) Player 2*: (7, 1, $180)
Expect Player 1 is now current player.
Player 1*: (6, 1, $220) Player 2: (13, 1, $180)
Expect Player 1 has not changed.
```

```
Player 1: (6, 1, $220) Player 2*: (13, 1, $180)
Expect Player 2 has 2 units and it is now Player 1's turn.
Player 1*: (6, 1, $220) Player 2: (13, 2, $130)
Expect Player 2 is in same location as Player 1.
Player 1*: (6, 1, $220) Player 2: (6, 2, $130)
passing go
Expect Player 1 has $420
Player 1: (2, 1, $420) Player 2*: (12, 2, $130)
Expect game over.
Is game ended: true
```

The above test uses `buyUnit()` so it will need to already be implemented. Now is also a good time to implement and test `sellUnit()`.

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

Submit the zip file to Canvas using the Assignment 2 submission link and **VERIFY** that your submission was successful. If you are not sure how to do this, see the document "Assignment Submission HOWTO", link #9 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 files **CyGame.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.