Assignment 3

Due: Wednesday, February 15 by 11:59 PM

Objective

This assignment should give you practice with classes, command-line arguments, and inheritance.

Task

Write a Tic-Tac-Toe game which will be played and drawn with text characters on the console. Your main program should be in a class called `TicTacToe` and you **must** have at least the following two files:

- `TicTacToe.java`
- `Player.java`

You may build other classes in other files. How exactly you design this program and what other files you create will be up to you. Everything will be packed up into a runnable jar file at the end. **Each file should have a comment including your name at the top of the file. Each file should also have appropriate comments throughout.**

Requirements

1. **Storage and setup:** You can design this with as many or as few classes as you like, as long as you are using at least the two classes described above. The game must begin with the `main()` method in class `TicTacToe`. You must use a 3×3 two-dimensional array to store the game board information (store it as member data of a class, you can choose the type of storage).
   Suggestion: Try to design your classes for re-usability. If you changed things, such as porting your game logic to a GUI interface, how easy would it be?
   Another suggestion: Construct a main "game loop" that runs the game logic until game over. This will be easier if you set up base class variables for a generic player class, then attach objects for the player types used in a given game using polymorphism.

2. **Player options:** You should allow the options of two human players, a human player and a computer player, or two computer players. This will be controlled through command line options when starting the program. First player will have X, the second player will have O. The command format for player options is:

   ```
   java TicTacToe [-c [1|2]]
   ```

The "-c" option indicates computer player(s). The optional "1|2" allows the user to specify which player will be the computer player. Omitting the "-c" option means two human players (the default). The possible combinations are:

```
java TicTacToe  // two human players
java TicTacToe -c       // two computer players
java TicTacToe -c 1     // computer is player 1, human player 2
java TicTacToe -c 2     // human player 1, computer player 2
```

If the command is invalid usage (illegal options), print a usage message, as below, then exit the program:

```
Usage:  java TicTacToe [-c [1|2]]
```

3. **Board output and player interface:** The output of the board should be in ASCII text format and needs to be user friendly. When asking a human player for a move, let them type in just the number of a square (a single number). Whenever an invalid cell is chosen by a human player (already filled or out of range), print an error message and have them re-enter. Player 1 will always be X and will go first (whether human or computer). Player 2 will always be O. See below for some suggested output.

4. **Computer Player Rules:** The computer player should not just use random moves. It needs to follow the following rules, in this order of priority:

   (a) If a winning move is available, take it.

   (b) If the opponent is threatening a win, block it (Note that if the opponent has two winning plays, only one can be blocked).

   (c) If the center space is available, take it.

   (d) Choose randomly between any remaining squares.

   Note that with the above logic, it *is* possible to beat the computer player.

5. **End of Game:** If somebody wins, determine and print out which player was the winner. If the game ends with no winner, print out that it was a draw.

6. **Randomness:** Whenever the computer player has to make a random choice, each choice *must* have an equal chance of being picked (so, if there is a choice of 3 squares, you should pick a random number in a range of 3, etc.). If you try to choose a random number between 1 and 9, then check if the square is available, there is a small chance that your program could run a long time and never pick a valid square.

Some Suggested Libraries/Classes

- `String`
- `StringBuilder`
- `java.util.Random`
- `java.util.Scanner`
- `java.lang.Integer`

Also, notice that there is a good opportunity to use inheritance and polymorphism in the game play. It also may be useful to create an `enum` for the space type.

Sample Output

(user input is underlined)

```
Game Board:                      Positions:

   |   |                          1 | 2 | 3
-----------                      -----------
   |   |                          4 | 5 | 6
-----------                      -----------
   |   |                          7 | 8 | 9
Player 1, please enter a move(1-9): 9

Game Board:                      Positions:

   |   |                          1 | 2 | 3
-----------                      -----------
   |   |                          4 | 5 | 6
-----------                      -----------
   |   | X                        7 | 8 | 9
Player 2 chooses position 5

Game Board:                      Positions:

   |   |                          1 | 2 | 3
-----------                      -----------
   | O |                          4 | 5 | 6
-----------                      -----------
   |   | X                        7 | 8 | 9
Player 1, please enter a move(1-9): 1
```

```
Game Board:                       Positions:

 X |   |                          1 | 2 | 3
-----------                       -----------
   | O |                          4 | 5 | 6
-----------                       -----------
   |   | X                        7 | 8 | 9
Player 2 chooses position 4
Game Board:                       Positions:

 X |   |                          1 | 2 | 3
-----------                       -----------
 O | O |                          4 | 5 | 6
-----------                       -----------
   |   | X                        7 | 8 | 9
Player 1, please enter a move(1-9):
```

Extra Credit

Add a command line option "-a", which stands for "advanced." This option, if used, makes the computer players take their turns with more advanced logic, as follows:

1. If a winning move is available, take it
2. If the opponent is threatening a winning move, block it
3. Determine which squares can be played that will not eventually result in a loss, and choose randomly between them

In other words, on the advanced setting, a computer player will always play smart enough to **not lose**. On this setting, any game will result in a tie if a human is using the optimal strategy; otherwise, the computer will win. Exactly how you create the logic for the 3rd part of the computer's logic is up to you, as long as the computer can **never** lose (which is possible, as Tic-Tac-Toe is a *solved* game).

Sample command line usage:

```
java TicTacToe -c 1 -a  // player 1 is computer on advanced setting
```

Note that the -a option only makes sense if the -c option is used. If you do the extra credit, the usage message would now be:

```
Usage:  java TicTacToe [-c [1|2] [-a]]
```

Submitting

Pack all of your files (class files **and** source code) into a **fully runnable** JAR file called `hw3.jar`. The main program that the jar file should execute should be in class `TicTacToe`. I should be able to run the main() method from your file with the command:

```
java -jar hw3.jar
```

Submit your jar file via the Blackboard submission link for assignment 3.