Assignment #3 - Tic-Tac-Toe

Due: Fri, Feb 12

Objective

Practice with classes in Java, including String libraries, command-line arguments, and inheritance techniques.

Task

Write a Tic-Tac-Toe game, which will be played and drawn with text characters on the console. This assignment is based on exercise 8.19, but the specifications here will be more specific.

The main program should be in a class called `TicTacToe`, so this will have filename:

- TicTacToe.java

You may build other classes, in other files. The design of this program is up to you. Everything will be packed up into a runnable jar file at the end.

Requirements

1. **Storage and setup**: You have the freedom to design this with as many or few classes as you like. The game must begin with the `main()` method in class `TicTacToe`. You must use a 3 X 3 two-dimensional array (member data of a class) to store the game board information. (You can choose the type of storage).
   Suggestion: Try to design your classes for re-usability. For example, if you were to port your game logic to a GUI interface in the future, how easy would it be?
   Another suggestion: Your main game loop will be easier if you set up base class variables for any generic type of player, then attach objects for the player types for any given game. You can make good use of polymorphism here.

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

```
3.   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 mode). Here are the possible combinations:

```
  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, like:

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

4. **Board output and player interface**: The output of the board should be in ascii text format. It needs to be user friendly. When asking a human player for a move, let them type in just the number of a square. 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'. A sample of suggested output appears farther down the page.

5. **Computer Player Rules**: The computer player must play to a certain level of intelligence (i.e. not just random moves). Here are the decisions that a computer player must implement, in this order of priority:
    1. If a winning move is available, take it.
    2. If the opponent is threatening a winning play, block it. (Note that if the opponent has two winning plays, only one can be blocked).
    3. If the center square is available, take it
    4. Else choose randomly between any remaining squares

    Notice that with the above logic, it *will* be possible to beat a computer player.

6. **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.

7. **Randomness**: Whenever the computer player has to make a random choice, each of the possible choices *must* have an equal chance of being picked. (This means that if it's a choice of 3 squares, you should pick a random number in a range of 3, etc.)

Some Suggested Libraries

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

Also notice that there's a good opportunity to use inheritance and polymorphism in the game play. (Hint: Create a `Player` class and then derive classes to represent the different types of players. We'll talk about this technique in class.)

Sample output format

(user input is underlined)

```
Game Board:                Positions:

   |   |                    1 | 2 | 3
-----------                -----------
   |   |                    4 | 5 | 6
-----------                -----------
   |   |                    7 | 8 | 9

Player 1, please enter a move (1-9): 5


Game Board:                Positions:

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

Player 2 (computer) chooses position 7


Game Board:                Positions:

   |   |                    1 | 2 | 3
-----------                -----------
   | X |                    4 | 5 | 6
-----------                -----------
 O |   |                    7 | 8 | 9

Player 1, please enter a move (1-9): 3


Game Board:                Positions:

   |   | X                  1 | 2 | 3
-----------                -----------
   | X |                    4 | 5 | 6
-----------                -----------
 O |   |                    7 | 8 | 9
```

Extra Credit

Add a command line option "-a", which stands for "advanced". This option, if used, makes any 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 play, 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**. Theoretically, any game on this setting should result in a tie (as long as a human player is playing "smart", too).

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 your class files **and** source code into a fully runnable JAR file called **`hw3.jar`**. I should be able to run your game from the jar file with the command:

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

Submit this jar file through the HW 3 submission link on the Canvas course page.