Project 1 - Tic Tac Toe

We're going to build a Java Tic-Tac-Toe console game.

You will need to follow the design laid out in the assignment here. (See class diagram on last page.)

If you have other ideas, you can discuss them with me, but fair warning I will probably make you do it my way. It helps to reinforce specific concepts, helps prevent internet cheating, and is often the way it goes in the real world too. So, please read carefully and ask questions if clarifications are needed.

Stuff not in the textbook

There are some classes needed that aren't covered in the textbook, however we did discuss them in class:

- The ArrayList class is one of the Java Collections classes that can be thought of as an array that can grow and shrink as needed.
- The Scanner class is used to get easily get input from the command line.
- The Random class is used to get a random number.

If you need a review or missed the information in class, here are some web pages that may help: one for ArrayList class, one for the Scanner class, and one for the Random class.

Part I - Two Player Game

We will start by putting together a two player game that we can add onto later. We'll start with the assumption that we will have two human players, so all we really need to do is track their moves and watch for a winner.

The first classes we'll need to create are the Player, GameBoard, and the TicTacToe classes. I recommend you create and test them in that order. Put all of the classes created in this part of the game in the `edu.htc.tictactoe` package.

Add a main method to each class, and use that to test each public method that contains logic. (If a method just returns an instance variable, you do not need to test it.) Be thoughtful in your tests and use System.out.println to indicate what method you are testing, what you expect the result to be and what the actual result is. As you write each method in the class, you should test it to make sure it works correctly for all situations.

Note: To keep the main method from becoming too large, you should make a separate static method to test each class method, and then use the main method to call each of your test methods. (See the GameBoard test example included below.)

Player Class

In the class diagram, you will note there are two types of Player, for now ignore the inheritance and just create the player class with the assumption it is a human player.

The Player must be given a name and marker (X or O) when the class is created and these should not be modifiable after creation. The winCounter should be set to zero initially and must be incremented by one when the `addWin` method is called.

Test creating the Player with the name and marker, and also test incrementing the win counter.

The `getMove` method should use a Scanner to get the square that the player wants to play in. Make sure to add validation logic here to ensure that the player enters a number from 1-9. You do not need to validate here that the square is open. (Since the Player class has no access to the board, the TicTacToe class will ensure that the selected square is open.)

GameBoard Class

The GameBoard is the 3 x 3 tic tac toe grid. Each square is numbered 1 - 9 and will display that number in the grid until a player takes that square. Once the player takes the square, it should show the player's marker, either X or O.

The game board grid is stored as a simple array of characters. Remember that the array index is 0-based, so square 1 is at position 0 in the array. This translation should occur in the GameBoard class methods so that the outside code is unaware of how the grid is stored. (This is an example of using abstraction.)

When the GameBoard is created, initialize both the `char[] board` and the `ArrayList openSquares`. The `char[]` should contain characters for the numbers 1-9, and the `ArrayList` should contain the integers 1-9.

Since we'll also need to be able to test with different configurations of moves on the board, let's add a new constructor to the GameBoard class just for testing. This constructor should take one input parameter, a `char[]` that contains the desired state of the board. Make sure that the `ArrayList openSquares` is setup correctly to match the input array values.

The `display` method should print this grid using System.out.println(). Use underscore _ characters, vertical bar | characters and plus + characters (for where the lines cross) to print the lines. Use spaces to separate the number or the X or O from the grid lines.

The `updateSquare` method must take two input parameters, the square number and the marker, and update both the `char[] board` and the `ArrayList openSquares`. Updating both allows us to quickly get a list of the open squares without needing to check each item in the array.

The `int[][] winCombinations` is a two-dimensional array that should include all of the ways to win the game. There are 8 possible ways to win. Each of the 8 ways to win should be an array inside the `winCombinations` array.

```
private int winCombinations[][] = {{1, 2, 3}, {4, 5, 6}, ... add other wins ...};
```

The `winCombinations` array is used by the `isGameWon` method to see if the game has been won by a player. The `isGameWon` method should return `true` if a player has won and `false` otherwise. A tie (or full board) is not considered a win.

The `getOpenSquares` method must return an array of primitive ints. We do not want to return the actual `ArrayList` that is used internally as that would allow it to be manipulated by outside code. (Remember the `ArrayList` is an object, and returning it returns a reference to the object. That reference can then be used to change the values inside the ArrayList.) This Stack Overflow article: Convert ArrayList<Integer> to int[], shows how to do this simply using Java 8. Prior to Java 8, this would require a loop to make a new array.

The `isSquareOpen` method must take a single input parameter for the square number. It should return `true` if that square is open (does not contain an X or O), and `false` otherwise.

GameBoard Test Example

As you write each method, you should write code to test it. Some example code for testing the GameBoard is shown below. This is an example to illustrate what I'm looking for with testing. Not all code is shown.

```
public class GameBoard {

 ... some code removed ...

  public static void main(String[] args) {
     testGameBoardDisplay();
     testIsSquareOpen();

    ... some code removed ...

  }

  public static void testIsSquareOpen() {
     System.out.println();
     System.out.println("-------------------------------------------------");
     System.out.println("Testing is square open method");
     GameBoard board = new GameBoard();

     System.out.println("Testing with empty board - all squares should be open.");
     boolean error = false;
     for (int i=1; i<=9; i++){
        if (board.isSquareOpen(i) == false) {
           error = true;
           System.out.println("Error. Square " + i + " is NOT open but should be!");
        }
     }
     if (error == false){
        System.out.println("Correct. All squares are open.");
     }

     System.out.println();
     System.out.println("Testing after updates:");
     System.out.println("Update square 1 with X.");
     board.update(1, 'X');
     if (board.isSquareOpen(1)){
        System.out.println("Error. Square 1 is still open.");
     } else {
        System.out.println("Correct. Square 1 is no longer open.");
     }
```

```
        for (int i=2; i<=9; i++){
            if (board.isSquareOpen(i) == false) {
                System.out.println("Error. Square " + i + " is NOT open but should be!");
            }
        }

        ... some code removed ...
        System.out.println("End testing isOpenSquare");
    }

    public static void testGameBoardDisplay() {
        System.out.println();
        System.out.println("-------------------------------------------------");
        System.out.println("Testing display of board");

        GameBoard board = new GameBoard();

        System.out.println("Testing display of empty board");
        board.display();

        System.out.println("Testing display of board after adding O in top right, X in center and X in bottom left.");
        board.update(5, 'X');
        board.update(3, 'O');
        board.update(7, 'X');
        board.display();
        System.out.println("End testing display of board");
    }
        ... some code removed ...
}
```

TicTacToe Class

This class contains the code to play the game. It manages the creation of the game board and the players, as well as getting the input from the player(s) and the overall flow of the program. While this *could* be our "main" application class, we will still add a separate Main class to launch the game so that the main method in this class can be used for testing.

When the TicTacToe class is created, it should initialize the GameBoard and create a `Scanner` for user input. The `Scanner` is not shown in the class diagram. This is fairly common when there are multiple ways to solve a problem (getting user input) and the overall design is not dependent on the choice. However, for this project, you should use the `Scanner` (it works nicely with InteliJ) and you should create it and store it as an instance variable to avoid creating it multiple times throughout the code.

There is only one method specified in the class diagram, the `playGame` method. This method will be used to start the game. However, I recommend that you make other *helper* methods to simplify testing and for code readability. Some initial suggestions would be helper methods to handle getting information from the player such as their name and which square they want to play in. Keep in mind that you should validate any user input to ensure that it is acceptable. You should not allow a blank name and the square they want to play in should be a number from 1-9.

For the first part of this assignment, the game should require two players, so you should get two player names and create two Players, one assigned the marker X and the other the marker O. Player 1 will get the first move, and you should alternate asking each player for a move until the game is one or there are no open squares (the game is tied). When the game is over, either congratulate the winner or say it was a tie, then ask if they would like to play again. Repeat with a new board until the response is "No". When they are done playing, show how many times each player won.

When finished with Part I, you should have a playable two person game of Tic-Tac-Toe and that your final GameBoard class is bug free. The logic in the GameBoard class should not require changes for later parts of the project.

Part II - Single Player Support

For the second part of the project, we will extend the two player game to support a single player with a basic computer AI opponent.

We'll start by adding a new `edu.htc.tictactoe.player` package to contain all of the player classes. This is both organizational - it keeps the project looking cleaner - and functional. By separating the project code into multiple packages, we can also use default access control to help prevent unintended access to methods.

Move the Player class you created earlier into the new player package. Next add a new HumanPlayer class that extends the Player class. Move the code for the getMove method from the Player class to the HumanPlayer class. Then make the Player class abstract and make the Player getMove method abstract. The HumanPlayer class is our first *concrete* class, a class that completes or fully implements an abstract class.

Update your code to create HumanPlayer objects to resolve any compile issues, then test your game again. Everything should still work just as before after making this change.

Now we can add the second type of Player, the ComputerPlayer. The ComputerPlayer will use the Strategy Pattern to manage how it goes about selecting its move. This will allow us to add computer AI players of different degrees of smartness. For this part of the assignment, we will just have a fairly dumb AI player that just moves randomly. Later, you will smarten up the AI and allow the player to choose the difficulty.

Since our ComputerPlayer needs a strategy, let's make that first. First, we'll create another package `edu.htc.tictactoe.strategy` for our strategy classes. Then we'll add the abstract class TicTacToeStrategy which will contain a reference to the GameBoard and logic used by the other strategies.

Though we cannot create an instance of the abstract class, we'll add a constructor to ensure that every instance is created with a reference to the GameBoard.

We will also add an abstract `getBestMove` method which will be the interface used by the ComputerPlayer to get the move from the strategy.

Next we will add a little bit of common logic to this class as a protected helper method called getRandomMove. This method should get the open squares from the game board and select one randomly to return. To do this you'll need an instance of the Random class. To get the best random numbers, you should store this as an instance variable.

Testing the TicTacToeStrategy

To test an abstract class, you need to be able to make an instance of that class. We can do that in our test code by making an anonymous inner class that adds a dummy implementation of any abstract methods. The code below shows to do this for testing the initial TicTacToeStrategy logic. Note that this is only necessary if your abstract classes contains common logic used by other concrete classes.

Another interesting point with testing this class is dealing with random behavior. Since the behavior is random, we need to test many times to ensure that no bad behavior is observed. The number of times you test should be relative to the number of possible outcomes. Note that no matter how large the number, it will still be possible to miss a bad random occurrence, though the higher the number, the less likely this will be.

```
public abstract class TicTacToeStrategy {

  ... some code removed ...

  public static void main(String[] args) {
     testGetRandomMove();
  }

  public static void testGetRandomMove() {

     System.out.println();
     System.out.println("-------------------------------------------------");
     System.out.println("Testing getRandomMove method");
     GameBoard board = new GameBoard(new char[]{'1', 'X', '3', '4', 'O', '6', 'X', '8', 'O'});

     // Creates anonymous inner class with dummy getBestMove behavior.
     TicTacToeStrategy strategy = new TicTacToeStrategy(board) {
        public int getBestMove() {
           return 0;
        }
     };

     // Since result is random, testing multiple times to validate good output
     boolean error = false;
     int numberOfRuns = 50;
     for (int i=0; i<numberOfRuns; i++){

        int result = strategy.getRandomMove();
        if (result < 1 || result > 9){
           error = true;
           System.out.println("Error! Result " + result + " is out of bounds. Must be between 1 and 9");
        } else if (board.isSquareOpen(result) == false) {
           error = true;
           System.out.println("Error! Square " + result + " is not open!");
        }
     }
     if (error == false) {
        System.out.println("Correct.  No bad output was found in " + numberOfRuns + " tries.");
     }
  }
}
```

Once you have tested the getRandomMove method, you can create the concrete RandomMoveStrategy class. This class will extend the TicTacToeStrategy class and implement the getBestMove method by getting a random move by using the getRandomMove method.

Consider: Why do you think we put the getRandomMove method in the parent class, not in the RandomMoveStrategy?

Update the TicTacToe class to ask if there will be two players. If there is only one, create a ComputerPlayer for player two. Adjust the game output as needed to ensure clear game flow both with a single player and two players.

Make sure to test the game both as a single player game and as a two player game to ensure correct behavior and that the output is clear.

Part III - Add New AI Strategies

The last part of the assignment will involve improvements to our AI player choices. We'll set up three new strategy classes that will gradually make our computer opponent smarter.

The BestOpenMoveStrategy will take the best remaining move based on the following:

- first look for the center spot, and select that if open
- if not, then find all open corners and select a random corner
- If the center and corners are taken, take a random remaining square

Add a helper method(s) to the TicTacToeStrategy class to do this and then use that method in the BestOpenMoveStrategy similar to what you did earlier for the RandomMoveStrategy. Make sure to add tests for the new method(s).

The BlockWinStrategy should also extend the TicTacToeStrategy, but it will add an additional instance variables: one for the player's marker and another for a list of winning combinations. (The list of winning combinations is the same as what you have in the GameBoard, but since that is not shared we will replicate it in the strategy.) These additional variables will be used to find potential wins for the other player and block them.

The BlockWinStrategy should:

- first look for an opponent win and block the first one found
- if no opponent win option is found, then take the best open move using the rules above.

This will require a little bit of thought. How can you identify a potential win for the other player? You will need to combine several pieces of information to do this: the opponent's marker, the possible win positions, and the current values at those positions on the game board.

The GoForWinStrategy should take this one step further and first look for a computer win, then block a win, then take the best open move.

Make sure that you add tests to support the new methods added for each strategy.

To finish things up, you'll need to add logic to the TicTacToe class to ask the player what difficulty they want to play at when the game is single player. (This does not apply to a two player game.) You should not give away the strategy while asking, just name them 1 to 4, or call them something like Simple, Easy, Medium, and Hard.

Make sure to test your game both with a single player and with two players after the updates to ensure the dialog is smooth and logical in both cases. You should also test playing a few games with each strategy to ensure that you haven't missed anything.

The last thing that you should do is add some additional validation to the GameBoard class `update` method. This method should now throw a checked exception if the input parameter is not a valid square number 1-9, and it should also throw a checked exception if the square is already taken. Your TicTacToe class should also watch for and prevent these situations, but the GameBoard class should not be trusting. Another, less trustworthy class may use it in the future.

Make sure to test that these exceptions occur. You can check for them using try/catch blocks in your test code. Update the TicTacToe class to handle these exceptions, but you should still keep the earlier logic that was intended to prevent them from occurring in the first place.

Before you submit, make sure to test thoroughly one last time, then make sure your final code is updated on GitHub with a pull request.

Complete Class Diagram