# Finishing the Tic-Tac-Toe Program

*by Sophia*

WHAT'S COVERED

In this lesson, you will finish our Tic-Tac-Toe program to make use of methods. Specifically, this lesson covers:

## Table of Contents


- 1. Methods to Improve the Board
- 2. Methods to Check for a Winner


# 1. Methods to Improve the Board

Over the course of the game, the board will need to be redrawn many times to reflect the current state. This task is best encapsulated in a separate method. As with previous versions of the game, the method can use a loop to iterate over the array. It can also print out the positions marked by each player.

The method needs data about the board, so the two-dimensional array will need to be passed as a parameter. Since this method will print out the current state of the game to the console, there is no need for a return value.

Here is a reasonable design for the method:

EXAMPLE

```
public static void drawGame(String[][] board) {
 for(int row = 0; row < board.length; row++) {
  System.out.println(Arrays.toString(board[row]));
 }
 System.out.println(); // Add a blank line after drawing the board
}
```

Since this is a static method, no variable is needed to capture the value returned.

The board is initially declared like this in the `main()` method:

↗ EXAMPLE

```
String[][] board = {{" - ", " - ", " - "},
                    {" - ", " - ", " - "},
                    {" - ", " - ", " - "}};
```

The drawGame() method can be called like this (also from within the body of the main() method) drawGame(board), by checking that the selection is a valid row and column.

Another aspect of working with the Tic-Tac-Toe board is checking that a selected position is valid. The size of the board (three rows and three columns) is a fixed quantity, so the method that checks if a selected position is a valid position on the board just needs the requested row and column. This method needs to determine if the position is valid or invalid (a two-way distinction). The method can indicate this by returning a `boolean` value (`true` or `false`).

A method that returns true if the selected position is not out of the valid range of rows and columns:

↗ EXAMPLE

```
public static boolean isValidSelection(int row, int column) {
  // ! needed to return true if selection is valid position
  return !(row < 1 || row > 3 || column < 1 || column > 3);
}
```

Another approach to checking that the player has chosen a valid row and column is to turn the logic around and check if the selection is within the valid range of rows and within the valid range of columns.

The if() statement in this case looks like this:

↗ EXAMPLE

```
return (row >= 1 && row <= 3 && column >= 1 && column <= 3);
```

CONCEPT TO KNOW

Note that this second version needs to use logical && (and) rather than || (or) since it tests that all of the tests pass (rather than checking if any of the tests fail).

There is another common functionality related to the board that needs to be considered as well. When the user selects a space on the board, the game needs to check and make sure that the space isn't already taken. For this method to do its work, it needs the array with the data about the current state of the board and the integers for the row and column. This method, too, can return a boolean to indicate if the space is already taken. A position on the board is open if its current value is the String " - ".

Here is a potential version of such a method:

↗ EXAMPLE

```
public static boolean isPositionOpen(String[][] board, int row, int column) {
  return board[row - 1][column - 1].equals(" - ");
}
```

# 2. Methods to Check for a Winner

The next important piece of the game's functionality is determining if the game has been won. This is a more complicated process than checking that the selected position is a valid, open space. However, the process can be broken down to make it manageable.

HINT

In the previous lesson on debugging methods, you learned that methods can check the rows for a win and check the columns for a win. You also learned that a method can be used to check for a win on a diagonal. Here are the methods that the previous tutorial arrived at after making the needed corrections.

The checkForRowWin() method ended up like this:

EXAMPLE

```
public static int checkForRowWin(String[][] board, char player) {
  // Variable to track row that holds a win
  int rowNumber = 0;
  for(int row = 0; row <= 3; row++) {
    if(board[row][0].equals(" " + player + " ") &&
      board[row][1].equals(" " + player + " ") &&
      board[row][2].equals(" " + player + " ")) {
      // Add 1 to row number to match human count (starting with 1)
      rowNumber = row + 1;
      break;
    }
  }
  // If rowNumber > 0, there is a winning row
  return rowNumber;
}
```

The final version of the method for checking the columns for a win looks like this:

EXAMPLE

```
public static int checkForColumnWin(String[][] board, char player) {
  // Variable to track row that holds a win
  int colNumber = 0;
  for(int col = 0; col < 3; col++) {
```

```
  if(board[0][col].equals(" " + player + " ") &&
   board[1][col].equals(" " + player + " ") &&
   board[2][col].equals(" " + player + " ")) {
   // Correct for human count (starting with 1, not 0)
   colNumber = col + 1;
   break;
  }
 }
 // If colNumber > 0, there is a winning row
 return colNumber;
}
```

Now the game needs a method to check for a diagonal win. While there are three rows to check for a win and three columns, there are only two ways to win diagonally. The first is a run of three of the same character from the top left corner to the lower right corner, and the second is a run of three of the same character running from the top right corner to the lower left. In either case, a win is only possible if the winner has taken the middle square (the intersection of the second row and the second column).

Like the other two methods that check for a win, this method takes two parameters. These include the array containing the current state of the board and a character representing the player for whom the method is checking for a win.

The resulting method should be set up like this:

↗ EXAMPLE

```
public static int checkForDiagonalWin(String[][] board, char player) {
 int diagonalNumber = 0;
 if(board[1][1].equals(" " + player + " ")) {
  if(board[0][0].equals(" " + player + " ") &&
   &amp;nbsp;board[2][2].equals(" " + player + " ")) {
   diagonalNumber = 1;
  }
  else if(board[0][2].equals(" " + player + " ") &&
    board[2][0].equals(" " + player + " ")) {
   diagonalNumber = 2;
  }
 }
 return diagonalNumber;
}
```

With these methods added to the code for the program, the next step is to work out how to call them to check for a win. Keep in mind that each of the methods that detects a win returns a value greater than 0 if the player whose positions are being checked has won.

Next, you would call the three methods in turn and retain the result for each check. Assume that these statements are in the body of a for loop that limits the game to nine turns using the loop variable turn.

The player array consists of 'X' and 'O', so the modulus operator gets the index of the current player's mark:

↱ EXAMPLE

```
int winningRow = checkForRowWin(board, player[turn % 2]);
int winningColumn = checkForColumnWin(board, player[turn % 2]);
int winningDiagonal = checkForDiagonalWin(board, player[turn % 2]);
```

Since the winner has to have taken one of the rows, one of the columns, or one of the diagonals, the overall process for checking for a win involves using the Boolean || (or) operator.

A selection statement like this will compare the values:

↱ EXAMPLE

```
if(winningRow > 0 || winningColumn > 0 || winningDiagonal > 0) {
 System.out.println("\n" + player[turn % 2] + " wins!\n");
 break;
}
```

It is assumed that this code is in the body of the for loop that handles the turns. The break statement is needed, since there is no point in continuing the game after a player has won.

Since there is no way for either player to win before the fifth iteration of the loop (when X has had three turns and O has had two), there is no need to check for a win until the fifth iteration.

TRY IT

**Directions**: To wrap up your work for this unit, start a new file in Replit named TicTacToeMethods.java and assemble the pieces (methods), along with the code in main(), to produce a complete game.

The overall process that is needed is to define the methods covered, with the appropriate parameters and return types:

1. drawGame()
2. isValidSelection()
3. isPositionOpen()
4. checkForRowWin()
5. checkForColumnWin()
6. checkForDiagonalWin()

With these methods set up, the next task is to get the code in place for the main() method:

STEP BY STEP

1. Declare and initialize the two-dimensional array for the board.
2. Declare and initialize the one-dimensional char array with the symbols for the players.

3. Declare and initialize integers for col and row (with initial value 0).
4. Declare a validSelection boolean with an initial value of false (so the user is prompted for #input until the entry is valid).
5. Declare a Scanner for reading the user input.
6. Draw the initial empty board.
7. Set up a for loop to run up to nine times to handle the turns. (Even turns are X's turns and odd turns are O's turns).
8. Inside the for loop that handles the turns, set up a while loop that runs until the user makes a valid move.
9. In the while loop, prompt for the chosen row and column and read the input.
10. Check if the selected position is valid.
11. If the selected position is valid, check if the selected space is open.
12. If the position is valid and open, set the character for the select square and set variable so that the while loop ends.
13. Redraw the board.
14. Check for a win if the turn is the 5th iteration of the for loop.
15. At the bottom of the for loop, set the variable for the while loop so that the loop will run during the next iteration of the for loop.

There is some room for variation, but here is a working implementation of this process:

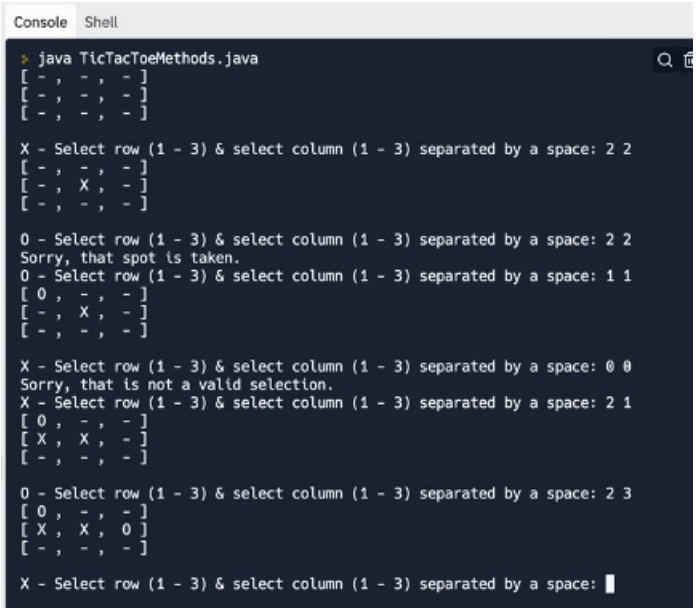
```
import java.util.Arrays;
import java.util.Scanner;

public class TicTacToeMethods {
  public static void main(String[] args) {
    String[][] board = {{" - ", " - ", " - "},
                        {" - ", " - ", " - "},
                        {" - ", " - ", " - "}};
    // Array of player marks
    char[] player = {'X', 'O'};

    int col = 0;
    int row = 0;
    // Initialized so input while loop runs until entry is valid
    boolean validSelection = false;

    Scanner input = new Scanner(System.in);

    // Draw initial empty board
    drawGame(board);

       // Start turns. Even turns are X, odd turns are O
    for(int turn = 0; turn < 9; turn++) {
      // Prompt for input until the user makes a valid move
      while(!validSelection) {
```

```
      System.out.print(player[turn % 2] + " - Select row (1 - 3) & " +
        "select column (1 - 3) separated by a space: ");
      row = input.nextInt();
      col = input.nextInt();
      if(isValidSelection(row, col)) {
        if(isPositionOpen(board, row, col)) {
          // Set player character at position
          board[row - 1][col - 1] = " " + player[turn % 2] + " ";
          // while loop will end when selected move is valid
          validSelection = true;
        }
        else { // Position is not open
          System.out.println("Sorry, that spot is taken.");
        }
      }
      // Not a valid position
      else {
        System.out.println("Sorry, that is not a valid selection.");
      }

    }
    drawGame(board);
    // Check if this is 5th or later turn (turn starts at 0)
    if(turn >= 4) {
      int winningRow = checkForRowWin(board, player[turn % 2]);
      int winningColumn = checkForColumnWin(board, player[turn % 2]);
      int winningDiagonal = checkForDiagonalWin(board, player[turn % 2]);
      if(winningRow > 0 || winningColumn > 0 || winningDiagonal > 0) {
        System.out.println("\n" + player[turn % 2] + " wins!\n");
        break;
      }
    }
    // Set so that while loop repeats until next player makes valid move
    validSelection = false;
  } // end of for loop for turns
} // end of main()

// Method to draw board. Parameter is a 2-d array with board data.
public static void drawGame(String[][] board) {
  for(int row = 0; row < board.length; row++) {
    System.out.println(Arrays.toString(board[row]));
  }
  System.out.println();
}

// Method to check if player has selected row & column within bounds
```

```
// Takes 2 int parameters for selected row & column.
// Returns true if selection is in bounds, false if not
public static boolean isValidSelection(int row, int column) {
  // ! needed to return true if selection is valid position
  return !(row < 1 || row > 3 || column < 1 || column > 3);
}

// Method to check if selected position is open.
// Parameters: 2-d array with board data, ints for row & column
// Returns true is selected position is open, otherwise false
public static boolean isPositionOpen(String[][] board, int row, int column)
{
  // Open squares marked with " - "
  return board[row - 1][column - 1].equals(" - ");
}

// Check if a player has a win in 1 of the rows.
// Parameters: 2-d array with board data, char ('X' or 'O') for player
// Returns 0 if no winning row, or 1, 2, or 3 if winning row
public static int checkForRowWin(String[][] board, char player) {
  int rowNumber = 0;
  for(int row = 0; row < 3; row++) {
    if(board[row][0].equals(" " + player + " ") &&
       board[row][1].equals(" " + player + " ") &&
       board[row][2].equals(" " + player + " ")) {
      rowNumber = row + 1;
      break;
    }
  }
  return rowNumber;
}

// Check if a player has a win in 1 of the columns.
// Parameters: 2-d array with board data, char ('X' or 'O') for player
// Returns 0 if no winning col, or 1, 2, or 3 if winning col
public static int checkForColumnWin(String[][] board, char player) {
  int columnNumber = 0;
  for(int column = 0; column < 3; column++) {
    if(board[0][columnNumber].equals(" " + player + " ") &&
       board[1][columnNumber].equals(" " + player + " ") &&
       board[2][columnNumber].equals(" " + player + " ")) {
      columnNumber = column + 1;
    }
  }
  return columnNumber;
}
```

```
// Check if a player has a win in 1 of the diagonals.
// Parameters: 2-d array with board data, char ('X' or 'O') for player
// Returns 0 if no winning diagonal, or 1 or 2 if winning diagonal

public static int checkForDiagonalWin(String[][] board, char player) {
  int diagonalNumber = 0;
  // Player must have central square to have winning diagonal
  if(board[1][1].equals(" " + player + " ")) {
    if(board[0][0].equals(" " + player + " ") &&
       board[2][2].equals(" " + player + " ")) {
      diagonalNumber = 1;
    }
    else if(board[0][2].equals(" " + player + " ") &&
       board[2][0].equals(" " + player + " ")) {
      diagonalNumber = 2;
    }
  }
  return diagonalNumber;
}

} // end of TicTacToeMethods class
```

Here is a screen shot of the first few turns in a game to show how this program works:




## SUMMARY

In this lesson, you have completed the Tic-Tac-Toe game. Building on work in previous lessons, you learned how to put together **methods to improve the board** and handle key parts of the functionality for the game. You learned that these methods draw the board with the current state of the game, validate a player's move, and **check for a win**. You have seen how to pass the relevant data to each method using appropriate parameters to get the result from the method via an appropriate return

type.

Source: This content and supplemental material has been adapted from Java, Java, Java: Object-Oriented Problem Solving. Source **cs.trincoll.edu/~ram/jjj/jjj-os-20170625.pdf**

It has also been adapted from “Python for Everybody” By Dr. Charles R. Severance. Source **py4e.com/html3/**