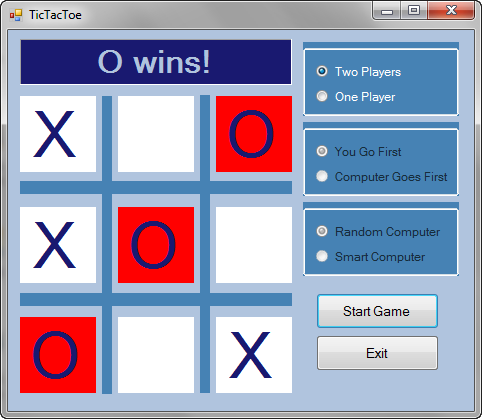
**Tic Tac Toe**

The next project we build is the classic **Tic Tac Toe** game, where you try to line up 3 X’s or 3 O’s in a 3 by 3 grid. We’ll develop a two player version and one where you can play against the computer.



**Tic Tac Toe Project**

On the left is the grid for playing **Tic Tac Toe**. Label controls abound. A label is used to tell you whose turn it is. Labels are used to mark X’s and O’s in the grid (although only X’s are shown in this design mode). Skinny label controls are used to form the dark blue grid. On the right are three group box controls and two button controls. Each group box holds two radio button controls used to establish game options. The two button controls are used to start and stop the game and to exit the program.

Run the project (press <**F5**>). The game will appear in its ‘stopped’ state, the grid is cleared and waiting for you to choose game options (one or two players and, if one player, who goes first and how smart you want the computer to be). The grid is disabled – no marks can be made (by clicking the grid).

Click the **Start Game** button to start playing. Its caption will change (now reading **Stop Game**) and the options group boxes and **Exit** button will become disabled. The label at the top of the grid says **X’s Turn**. X always goes first in this game (whether it’s you, the human player, or the computer).

In this state, you make a mark in the grid by clicking on the desired square. The computer will then place its mark, making it your turn again. After each mark, the board is examined for a win or a draw. You keep alternating turns until there is a win, the grid is full or until you press **Stop Game**. The game works the same way for two players, with the two players alternating turns marking the grid.

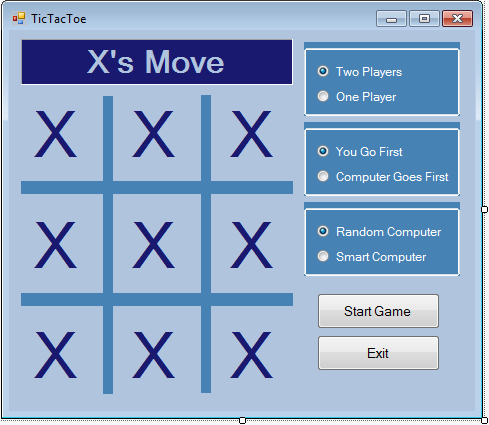
You will now build this project in stages. As you build Visual C# Express projects, it is recommend taking a slow, step-by-step process. It minimizes programming errors and helps build your confidence as things come together in a complete project.

**Tic Tac Toe Form Design**

Start a new project in Visual C# Express. We need 14 label controls (set **AutoSize** to **False** to allow resizing; set **BorderStyle** temporarily to **FixedSingle** so we can see where they fit) on the left side of the form. Place three group boxes on the right side, along with two button controls. Place two radio button controls in each group box. The form should look like this:

The label controls are used for messaging, to display the marks (X and O) and to form the grid. The radio buttons establish game options. The buttons (one to start and stop the game and one to exit the project) are used to control operation of the game.

When done setting properties, my form looks like this:

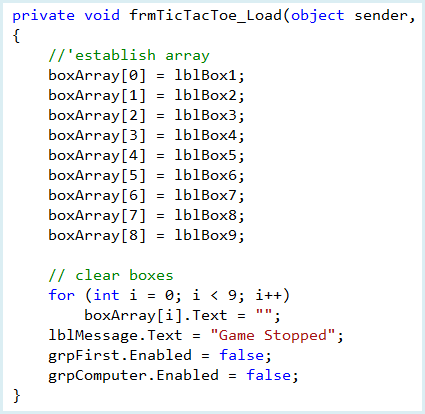


**Code Design – Initializing Stopped State**

To make our programming job easier, we will establish a control array (**boxArray**) for the nine label controls used to display the X and O marks in the grid. This array is defined below the form constructor in the code window:

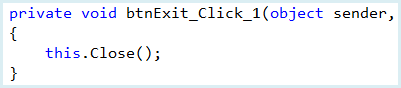
****

The assignments are made in the **frmTicTacToe Load** event method. We also set a message in the **lblMessage** control and blank out the label controls that display the marks. Finally, we want the game to initially be in **Two Players** mode so we disable the two group boxes associated with options for a one player game:

****

Even though we have written minimal code, we can run the project to make sure the form is properly initialized. As desired, the game initializes in **Two Players** mode. We have two choices at this point: click **btnStartStop** (the button with **Start Game**) or click **btnExit** (the button with **Exit**). We could also change to **One Player** mode – we will address this option in some detail later.

The code for exiting is simple. The **btnExit Click** event:

****

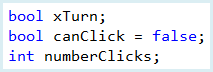
The code for the **btnStartStop** button is much more complicated. We will build it in several steps. First, we look at switching the game from stopped to playing state.

**Code Design –Stopped to Playing State**

When the user clicks the **Start Game** button in the ‘stopped’ state, several things must happen to switch the **Tic Tac Toe** game to ‘playing’ state:

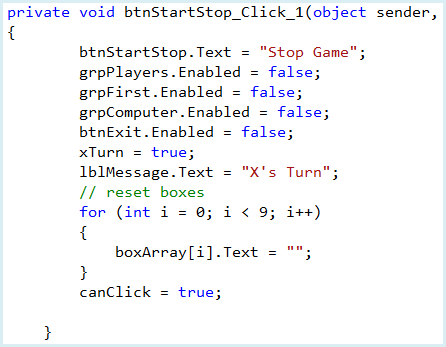
* Change the **Text** property of **btnStartStop** to **Stop Game**.
* Disable group boxes (don’t allow selection of options while playing).
* Disable **btnExit**.
* Establish this as X’s turn (since X always goes first).
* Blank out label controls displaying marks.
* Allow player to input a mark on the grid.

We establish three variables in the form level declarations to help keep track of where we are in the game:

****

If **xTurn** is **true**, it is X’s turn, otherwise it is O’s turn. **canClick** is used to determine if it’s okay to click on the label controls. It is **true** when playing, **false** (the initial value) when stopped. **numberClicks** keeps track of how many of the grid labels have been clicked on (9 maximum).

The code for the **btnStartStop Click** event that implements the steps in the above method and initializes the new variables is:

****

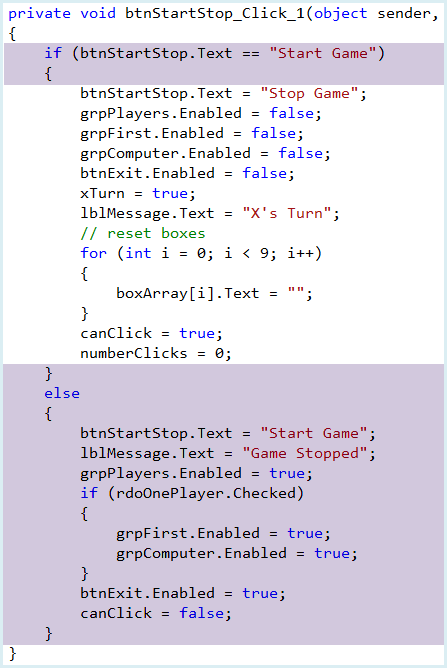
Save and run the project. Click **Start Game** and the game should switch to two player ‘playing’ state:

**Code Design –Playing to Stopped State**

When the user clicks the **Stop Game** button in the two player ‘playing’ state, several things must happen to switch the **Tic Tac Toe** game to ‘stopped’ state:

* Change the **Text** property of **btnStartStop** to **Start Game.**
* Display **Game Stopped** message.
* Enable **grpPlayers**.
* Enable **grpFirst** and **grpComputer** if one player game.
* Enable **btnExit**.
* Set **canClick** to **false.**

The code that does these steps is (modifications to the current **btnStartStop Click** event code are shaded):



Save and run the project. You should be able to now move from ‘stopped’ to ‘playing’ state and back. When stopped, make sure the proper group boxes appear for the one and two player options. Let’s write the code for the two player game, first looking at how to mark the grid.

**Code Design –Marking Grid**

In the **Tic Tac Toe** two player game, when a player clicks a box in the grid, we want the proper mark (X or O) to appear. After each mark, we then need to see if anyone won. If there is no win, and there are still empty locations, we switch to the next player.

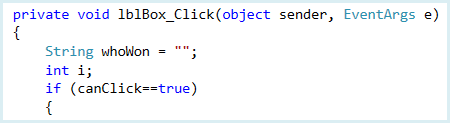
So, when a grid box is clicked, we follow these steps:

Make sure there is not a mark there already.

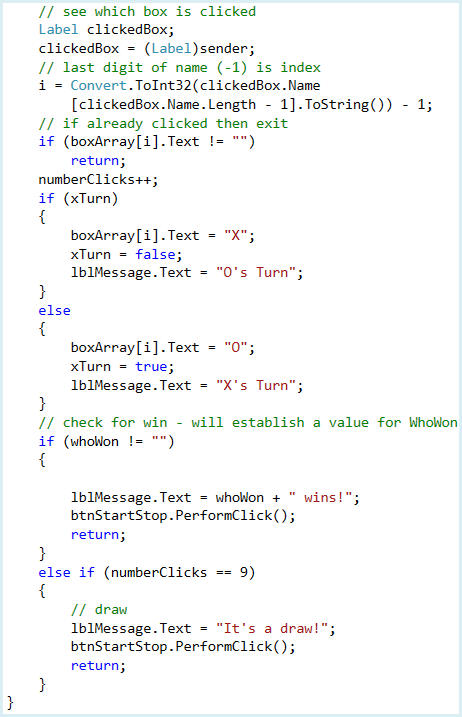
* + - Increment **numberClicks**.
    - Place proper mark in corresponding label control (**X** if **xTurn** is **true**, otherwise **O**).
    - Switch to next player (not needed if there is a win or draw).
    - Check for win. If there is a win, declare the winner and stop the game.
    - Check if **numberClicks = 9** (board is full with no win). If so, declare the game a draw and stop.

The code to perform these steps will be placed in a method named **lblBox\_Click**. This method should be set to handle the **Click** event for all 9 label controls used in the **Tic Tac Toe** grid (**lblBox1**, **lblBox2**, **lblBox3**, **lblBox4**, **lblBox5**, **lblBox6**, **lblBox7**, **lblBox8**, **lblBox9**).

Add this method (**lblBox\_Click**) to your project. (the same as the safecracker project buttons) This implements the needed steps to accept the player’s marking of the grid



Code continued below



Let’s look at this code. Clicking any of the nine label controls used in the grid will invoke this method. The first part of the code determines the index of the element of label control array (**boxArray**) that was clicked (the variable **i**). The index is one less than the last character of the label name converted to a number. If there is already a mark there (not blank), the method is exited. If blank, the proper mark is placed in **boxArray[i]**. After this, we check for a win (right now just in a comment line – we’ll fix that soon). If the variable **whoWon** is not blank (will be established by the check win logic), we declare the winner. Otherwise, we keep accepting clicks until the grid is full, declaring a draw.

Save and run the project. You should be able to click on each of the grid locations, placing X’s and O’s there in alternate turns. Once you have filled all the grid locations, the game will be returned to stopped state. Wins will not be recognized.

**Using General Methods in Projects**

Many projects have tasks that need to be repeated or that compute a certain result, such as determining the winner of a **Tic Tac Toe** game. Such tasks are usually coded in a general **method**, using provided information to return some value.

Using general methods can help divide a complex application into more manageable units of code. This helps meet goals of readability and reusability. As you build projects, it will be obvious where such a method is needed. Look for areas in your application where code is repeated in different places. It would be best (shorter code and easier maintenance) to put this repeated code in a method. And, look for places in your application where you want to do some long, detailed task – this is another great use for a general method. It makes your code much easier to follow.

The form for a general method named **myMethod** is:

**private type MyMethod(arguments) // definition header**

**{**

**[Method code] return(returnedValue);**

**}**

The definition header names the **method**, specifies its **type** (the type of the returned value – if no value is returned, use the keyword **void**) and defines any input **arguments** passed to the method. The keyword **private** indicates the method will have form level scope.

**arguments** are a comma-delimited list of variables passed to the method. If there are arguments, we need to take care in how they are declared in the header statement. In particular, we need to be concerned with:

* + - Number of arguments
    - Order of arguments
    - Type of arguments

We will address each point separately.

The **number** of arguments is dictated by how many variables the method needs to do its job. You need a variable for each piece of input information. You then place these variables in a particular **order** for the argument list.

Each variable in the argument list will be a particular **data type**. This must be known for each variable. In Visual C#, all variables are passed by value, meaning their value cannot be changed in the method. Variables are declared in the argument list using standard notation:

**type variableName**

The variable name (**variableName**) is treated as a local variable in the method. Arrays can also be used as input arguments. To declare an array as an argument,

use:

**type[] arrayName**

The brackets indicate an array is being passed.

To use a general method, simply refer to it, by name, in code (with appropriate arguments). Wherever it is used, it will be replaced by the computed value. A function can be used to return a value:

**rtnValue = MyMethod(arguments);**

or in an expression:

**thisNumber = 7 \* MyMethod(arguments) / anotherNumber;**

Let’s build a quick example computes the area of a rectangle. Here’s such a method:

**public double RectArea(double width, double height)**

**{**

**double area;**

**area = width \* height; return(area);**

**}**

The method is named **RectArea**. It has two arguments, **width** and **height**, of type **double**. It returns a **double** data type. This code segment finds the area of a rectangle 5.2 units wide and 10.7 units high:

**double a;**

**.**

**.**

**a = RectArea(5.2, 10.7);**

After this, **a** will have a value of **55.64**.

To put a general method in a Visual C# application, simply type it among the event methods associated with controls. Type the header, the opening left brace, the code and the closing right brace. Define arguments and returned value type. In the code, make sure to include the **return** line that establishes the returned value.

**Code Design – Checking For Win**

We will build a general method (named **CheckForWin**) that examines the playing grid and determines if there is a winner. If so, the win will be identified and the method will return a string variable holding the marker (**X** or **O**) for the winner or a blank if there is no winner. Let’s establish a strategy for doing this.

There are eight possible ways to win (3 horizontal, 3 vertical, 2 diagonal). Notice the indices of the label control array (**boxArray**) used in the playing grid are laid out in this manner:

|  |  |  |
| --- | --- | --- |
| **0** | **1** | **2** |
| **3** | **4** | **5** |
| **6** | **7** | **8** |

If we declare a string array named **possibleWins**, its 8 elements would be:

**possibleWins[0] = "012"**

**possibleWins[1] = "345"**

**possibleWins[2] = "678"**

**possibleWins[3] = "036"**

**possibleWins[4] = "147"**

**possibleWins[5] = "258"**

**possibleWins[6] = "048"**

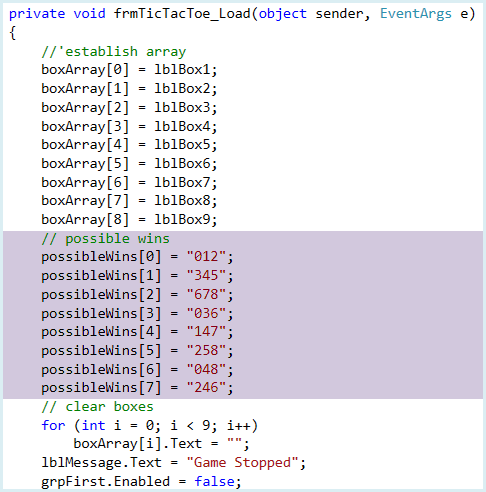
**possibleWins[7] = "246"**

So our win logic would be to go through each possible win and see if the corresponding elements of **boxArray** all contain the same mark (**X** or **O**, but not blank). If so, a winner is declared.

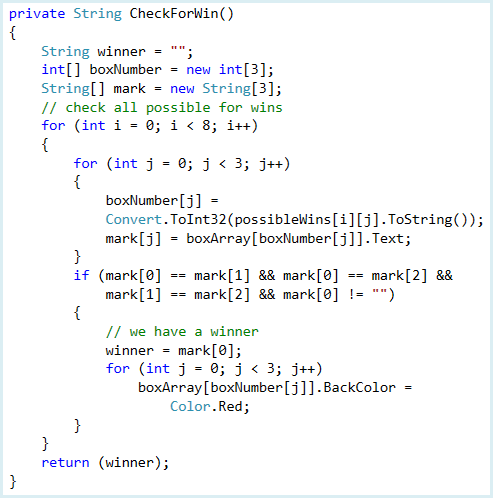
Add this form level declaration to the code window:

****

Modify the **frmTicTacToe Load** method to establish values for the array (changes are shaded):

****

Add the general method **CheckForWin** to your project.



**Q1:** In the first two “for loops”, the upper bounds are i < 8, and j < 3. What do these two numbers represent? (2 points)

This code goes through all the possible wins. If all the marks in a particular horizontal, vertical or diagonal row match, the background color of that row is changed to red and the corresponding winner returned. Study the code to see how it works. The **boxNumber** array holds the indices of the **boxArray** label array for each possible win.

You may be confused by the notation:

**possibleWins[ i ][ j ]**

It’s really fairly simple. **possibleWins[ i ]** is the array of **String** variables (length of 3) with the possible wins. **possibleWins[ i ][ j ]** is the jth element (a **char** type) in that string. As an example:

**possibleWins[2] = "678"**

and

**possibleWins[2][0] = "6"**

**possibleWins[2][1] = "7"**

**possibleWins[2][2] = "8"**

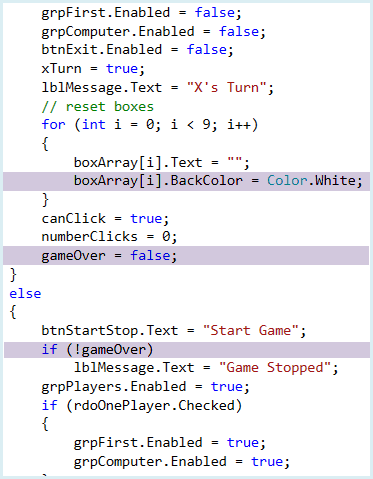
We can now modify the **lblBox Click** routine to check for wins, but we need one other modification. Notice when a win or draw is declared, we stop the game by performing a click on **btnStartStop**. We need some way to distinguish between this method of stopping the game and just having a player click the stop button (which stops the game before a win or draw is declared). Add this variable to the form level declarations:

****

If **gameOver** is **true**, there has been a win or draw. If **false**, the **Stop Game**

button was clicked before the game ended.

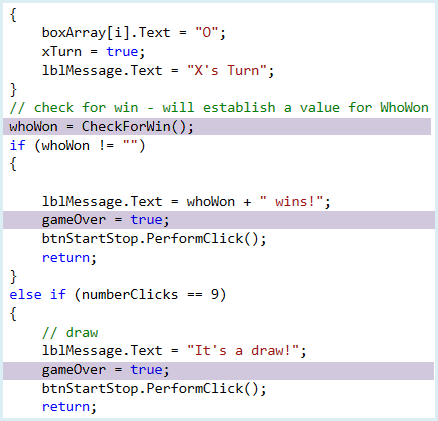
Make the shaded changes to the **btnStartStop Click** method:

****

These changes do several things. The grid label controls’ **BackColor** is restored to **White**, in case a win was displayed. The **gameOver** variable is set to **false** when starting a new game. And, the **Game Stopped** message is not displayed if **gameOver** is **true** (meaning a winner or draw is declared).

Now, modify (changes are shaded) the **lblBox Click** method to incorporate the

**CheckForWin** method and the new **gameOver** variable:

****

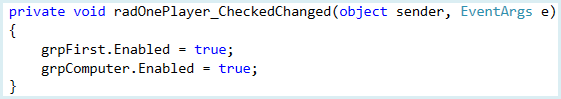
Save and run the project. You and a friend should be able to compete with wins and draws properly determined.

The two player game is now complete. Let’s start looking at how to implement a one player game versus a computer opponent. We’ll start easy, just having the computer make random moves (no brains!). This will help us establish the logic of switching players when the computer is one of them.

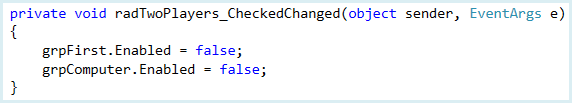
**Code Design – Number of Players Selection**

Different options are available to the player of **Tic Tac Toe** depending on the number of players selected. We have seen for the two player game that the choice of who goes first and determining how smart the computer is are not available as options. Now that we are considering the one player game (against the computer), this must change.

Place this short piece of code in the **radOnePlayer Click** event method. This enables the **grpFirst** and **grpComputer** group boxes to allow additional options available for the one player game:

****

Likewise, place this code in the **radTwoPlayers Click** event method to disable the **grpFirst** and **grpComputer** group boxes for the two player game:

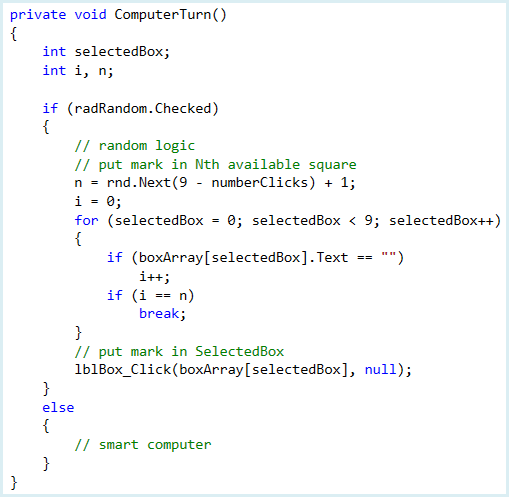
****

Save and run the project to make sure the player choices work as desired.

**Code Design – Random Computer Moves**

A big part of allowing the computer to play against a human (you, the player) is to decide what “thought processes” to give the computer. We’re allowing two choices: a **random computer** and a **smart computer**. We’ll start by coding up the random computer. All of the logic behind a computer move will be implemented in a general method named **ComputerTurn**. For random moves, it will simply choose (at random) one of the empty boxes in the grid and place a marker in that box (X if computer goes first, O if human goes first). Add a random object to the form level declarations area of the code window:

Place this general method **ComputerTurn** in your project:

****

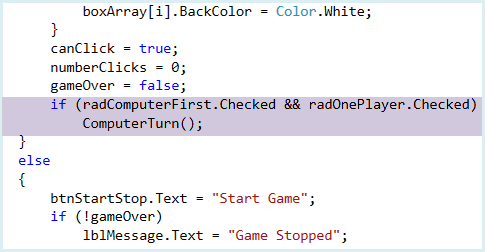
**Q2:** Explain how the program randomly decides which box to place a mark in (explain the variables i, n, selectedBox, and the for loop)

To mark the identified box, **boxArray[selectedBox]**, we simulate a click on that box by calling the **lblBox\_Click** method with the appropriate **sender** argument (the first argument):

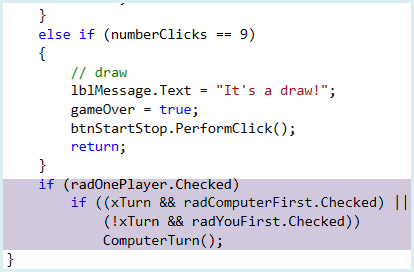
**lblBox\_Click(boxArray[selectedBox], null)**

The program will know whether to place an X or O in the box based on previously implemented logic.

Now, let’s implement the **ComputerTurn** method to allow the computer to play. We need to modify two methods. First, when we start a game, if the computer moves first, we need to invoke **ComputerTurn**. Make the shaded changes to the **btnStartStop Click** method:

****

And, after a mark is placed by the human player, the computer needs to take a turn. This logic is in the **lblBox Click** method. The needed changes are shaded:

****

With the added code, the computer takes a turn when it goes first and it’s X’s turn or takes a turn when the human goes first and it’s O’s turn.

Save and run the game. Choose the one player option and make sure **Random Computer** is selected. Try playing a few games (you go first, computer going first). Make sure things work properly; you should see the computer is pretty easy to beat!

**Code Design – Smart Computer Moves**

We’ve come to one of the more fun and more challenging parts of this game project. How can we make our computer a smarter opponent? In any game where a computer is playing against a human, we must be able to write down some rules that give the computer the appearance of intelligence. Many computer opponents are unbeatable. For example, it is very hard for a human to beat a computer at the game of chess.

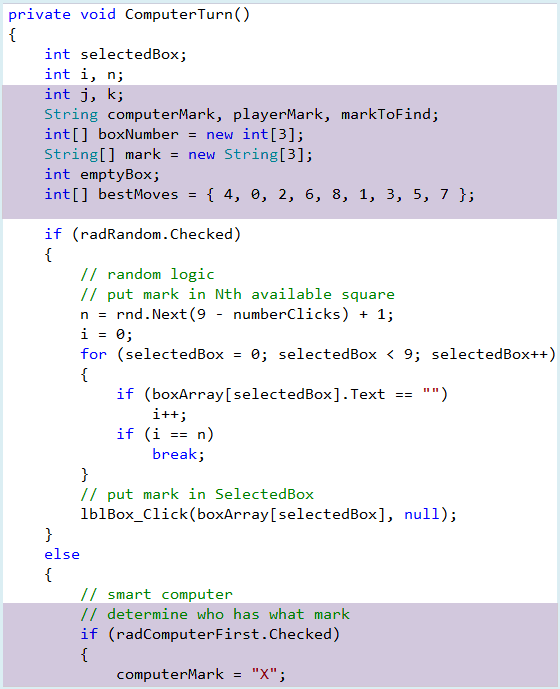
So how do we make our computer a better **Tic Tac Toe** player? We try to imbed choices we would make if we were playing the game. So for the computer to be a smart player, the programmer needs to be a smart player. This usually takes practice and study. For the game of **Tic Tac Toe**, we can develop a fairly simple, yet very intelligent strategy. Let’s do it.

When it is the computer’s turn, what should its move be? The rules we will use are (in order of choice):

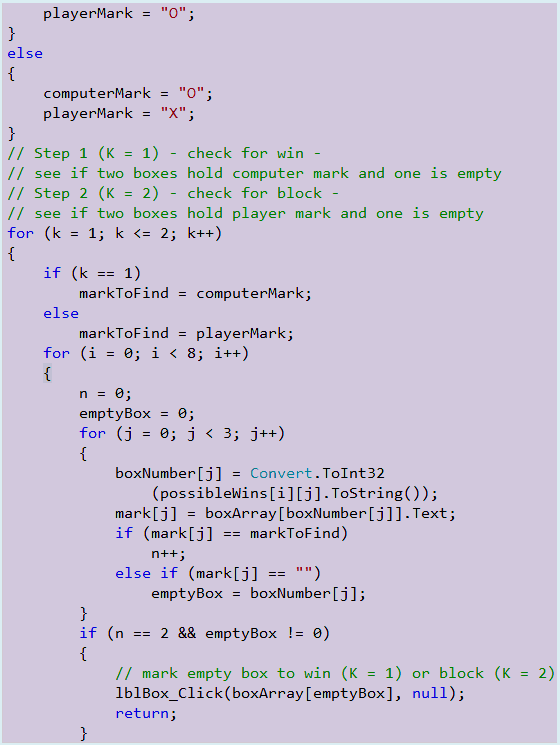
* + 1. If the computer can win with a move, make that move and the game is over. So, if there’s a row with two of the computer’s markers and an empty space, the empty space is the place to mark!
    2. If the computer can block with a move, make that move and the opponent can’t win on the next move. So, if there’s a row with two of the human player’s markers and an empty space, the empty space is the place to mark!
    3. If there is no possible win or possible block, make a move in this order: center square, one of the four corner squares, or one of the four side squares.

I think you see this logic makes sense. You may wonder about Step 3 – why we choose that particular order. Recall there are 8 possible ways to win in **Tic Tac Toe** (3 horizontal, 3 vertical, 2 diagonal). The center square is needed in 4 of these, any one corner is involved in 3 possibilities, while any side is involved in just 2 wins. Hence, the order of choices is made on basic probabilities. Let’s implement this logic in code.

Here is the modified **ComputerTurn** method that implements this ‘smart’ logic. The changes are shaded:

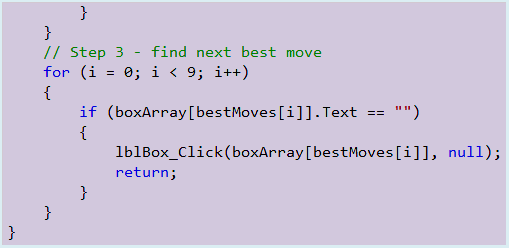


Code Continued Below



**Q3:** Place a comment beside each line of code within the *for ( i = 0 , i < 8 ; i++)* structure.

Code Continued Below



In the ‘smart’ logic, we first find out whether the computer has X or O. Steps 1 and 2 of the computer logic are done in a for loop with k as index. In that loop, we go through all the possible wins looking for a row with 2 identical marks and an empty box. For k=1, we look for the computer’s mark and an empty box – giving the computer a win on the next move. For k=2, we look for the human’s mark and an empty box – giving the computer a block on the next move. If neither Step 1 or Step 2 is successful, we move to Step 3. The next best moves are listed in desired order in the array **bestMoves**. In Step 3, we go through this array, finding the first empty box available and move there.

Save and run the project. The game is now fully functional. Try playing it against the computer. You should find it can’t be beat. The best you can do against the computer is a draw.

**Level 4+ Achievements**

* + - Use picture boxes instead of label controls in the grid. With such a change, you could use something other than X’s and O’s to mark the squares.
    - Track number of wins vs number of losses.