CST-250 Activity 2

Contents

Activity 2.1: Chess Board Guide

Learning Objectives: The three goals of this exercise are to:

(1) Create an application that uses a 2D array game board.

(2) Create methods that manipulate the squares according to game rules.

(3) Implement the design practice of separation of back-end logic from front-end interface code.

Description:

This program will ask the user to place a chess piece on a board. The computer will then indicate all of the possible legal moves that the piece may take in its next turn. In version 1 of the program, we will design a console app that uses only text characters to display the board. In version 2, we will design a Windows forms graphical user interface app.

Table of Contents

(1) **Board and Cell Classes**. We will design the objects used in the game as well as the methods used to calculate legal moves for the chess piece.

(2) **Console Application**. We will create a playable version of the game using text-based terminal menus and displays.

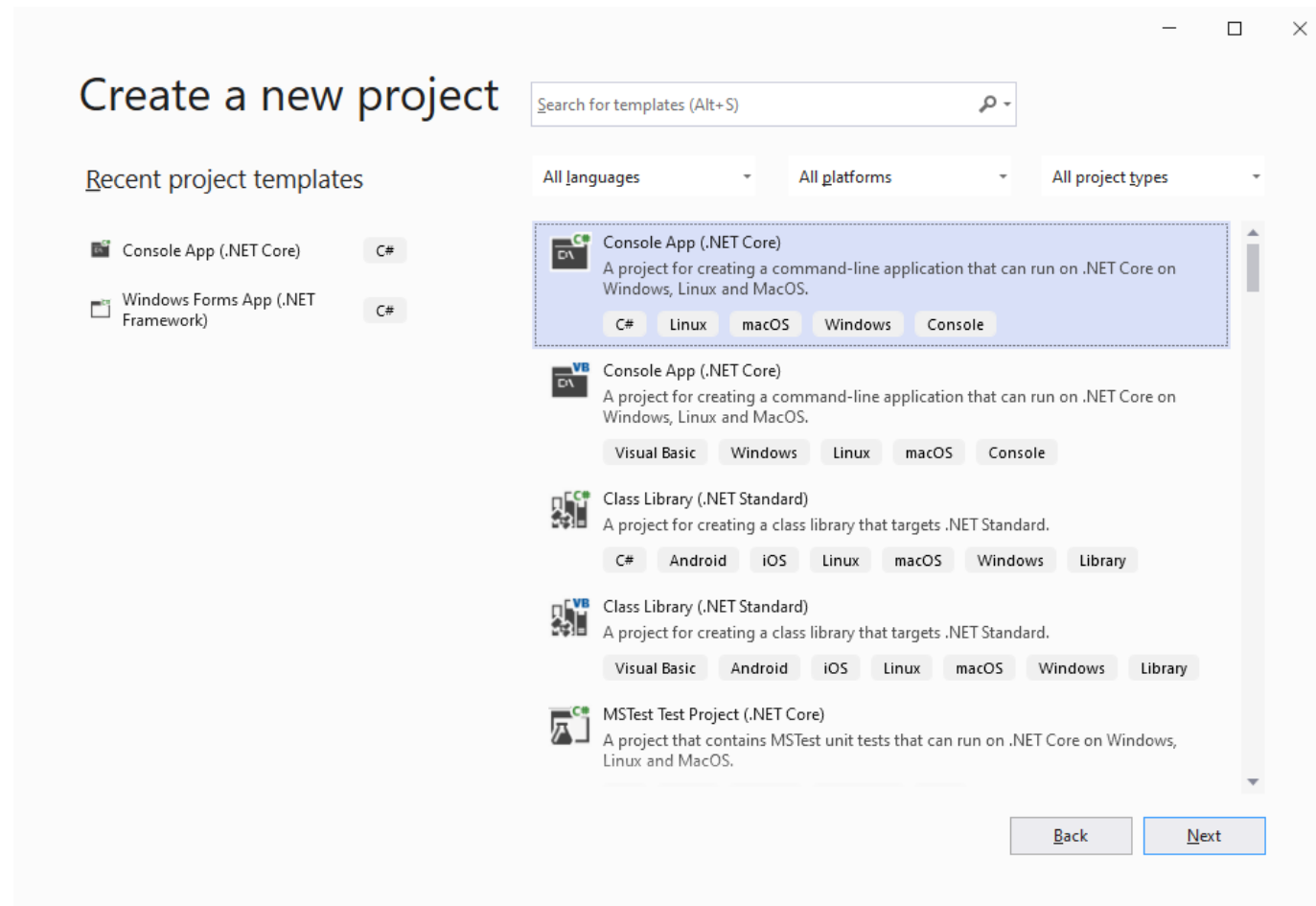(3) **Windows Application**. We will create a form with a clickable game board and a menu to choose a chess piece.
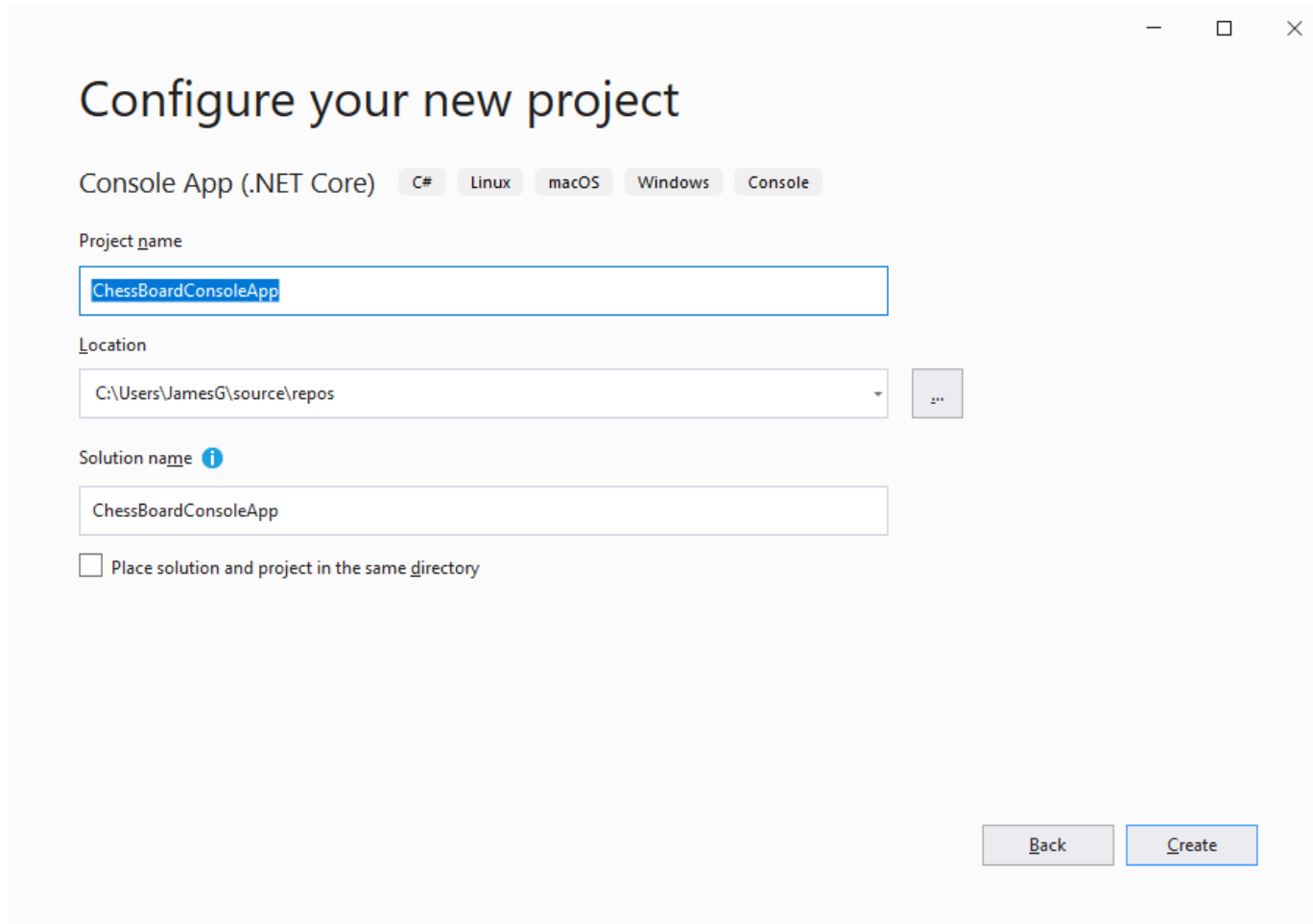
Review the Game

Read "A Quick Summary of the Rules of Chess," located in the topic materials.

Part 1 – Board and Cell Classes

Let's get started with Visual Studio. I am using version 2015. Other versions will work well with minor detail changes.

1. Start a New Project (File > New > Project)
2. Choose type Console.

3. Name it **ChessBoardConsoleApp**.

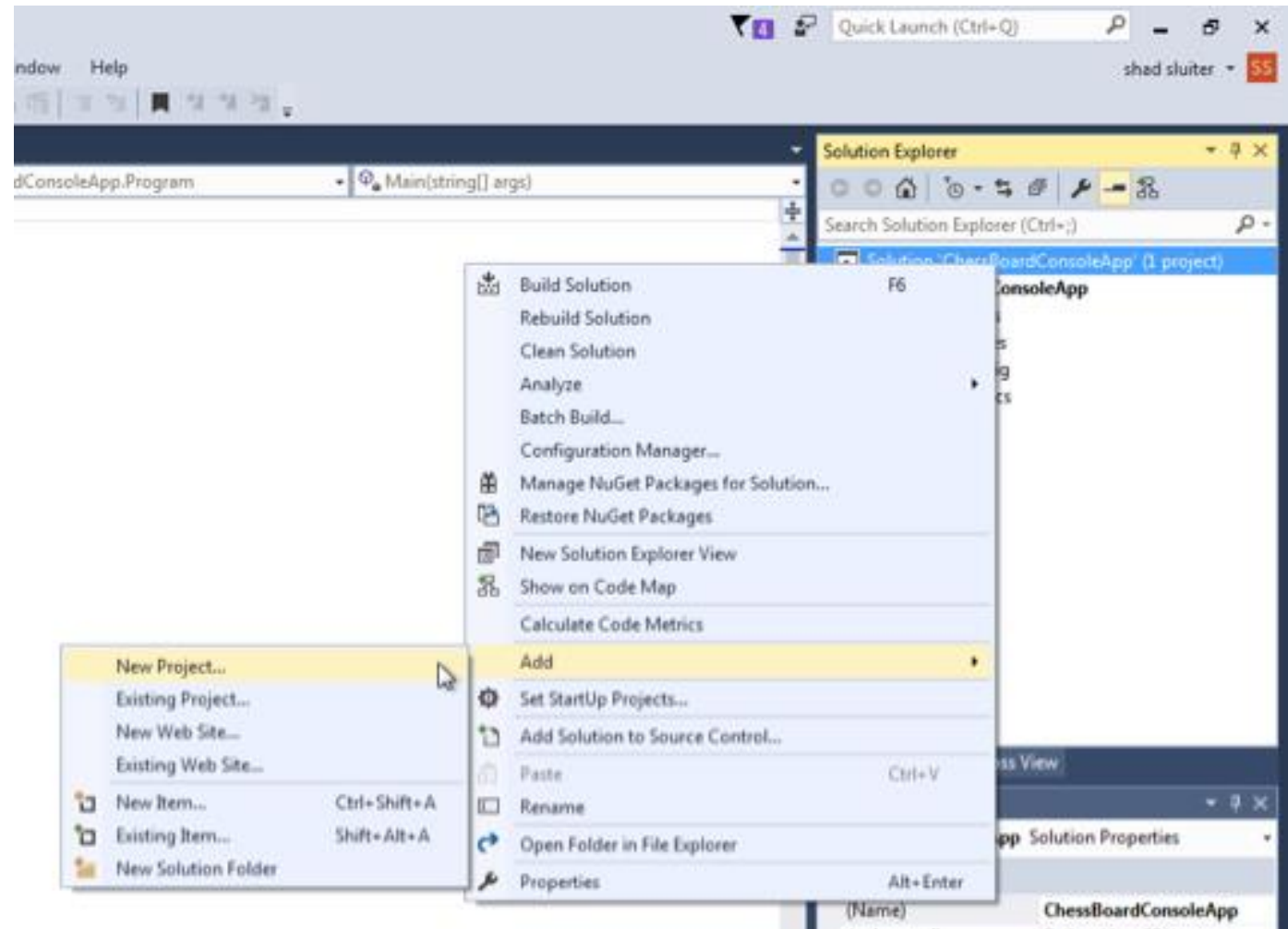You should see the start of a console application project.

This is the project that will contain the console application flow. It will be a text-driven loop and a way to print the game board in text.

4. Create a second new project within this solution.
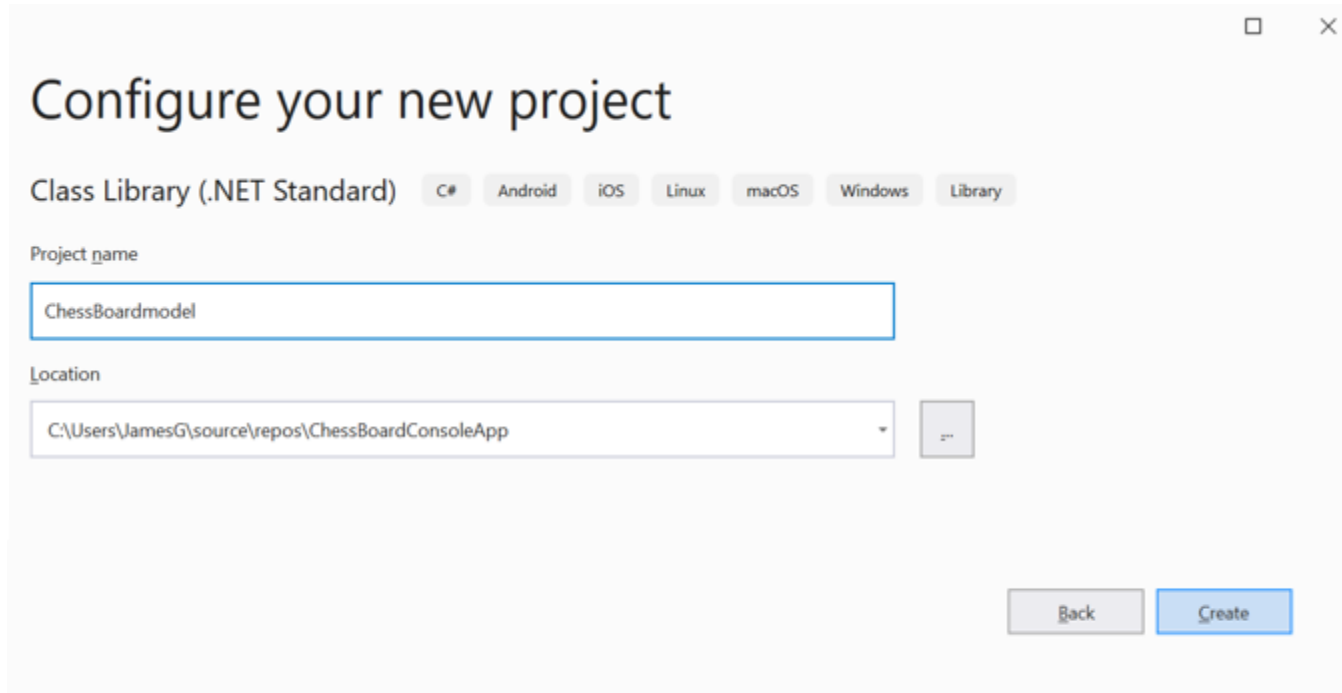
Right click on the Solution Name and choose **Add → New Project**.

5. Choose **Class Library** as the project type.

6. Name the project **ChessBoardModel**.
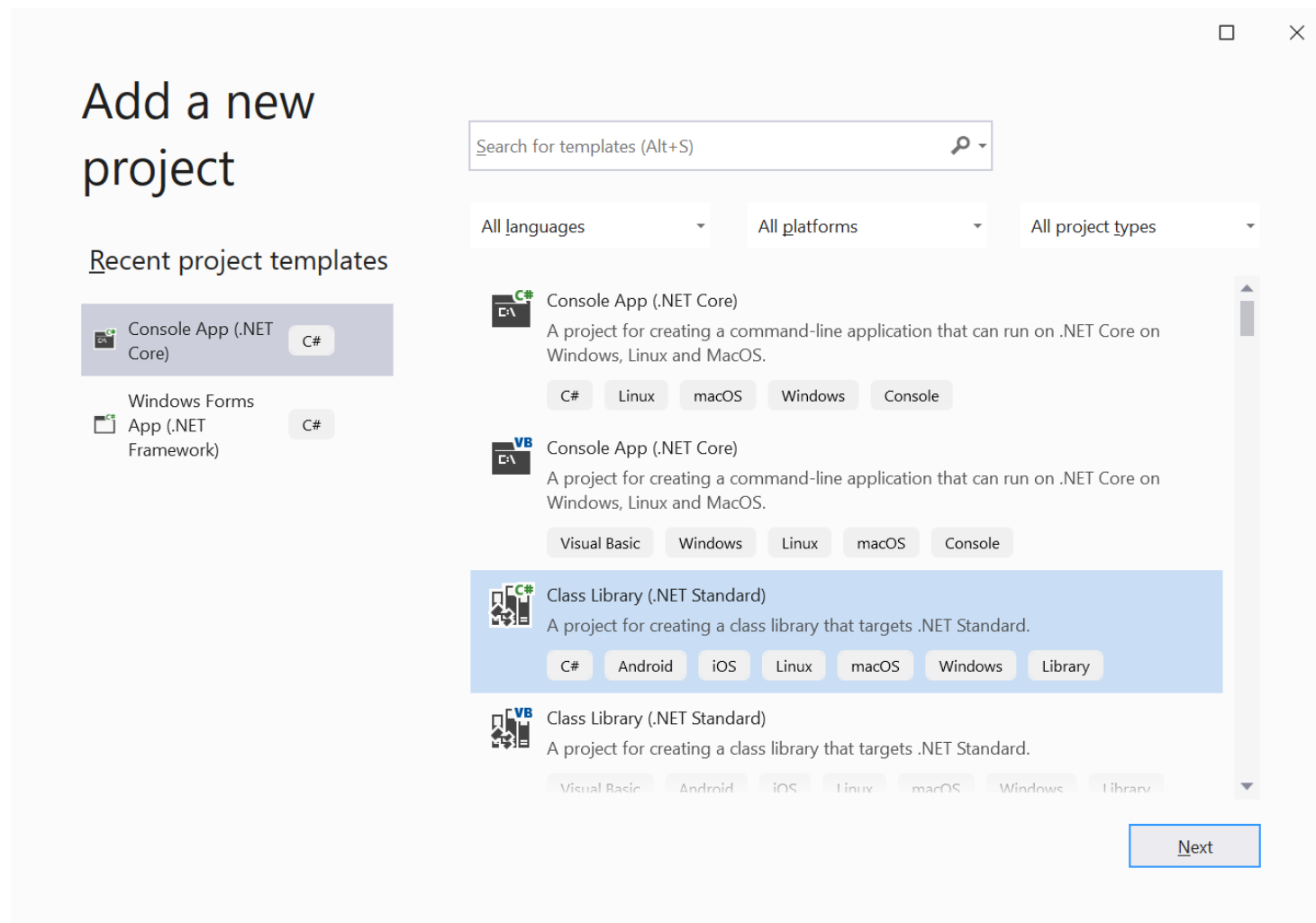
Configure your new project

Class Library (.NET Standard)   C#   Android   iOS   Linux   macOS   Windows   Library

Project name

ChessBoardmodel

Location

C:\Users\JamesG\source\repos\ChessBoardConsoleApp

Back   Create

This is the project that will contain the game pieces Cell and Board. It will also contain the methods that control the logic of the game.

7. You may delete the file called Class1.cs.

8. **Add a new class** to the ChessBoardModel project. Right-click on the project title. Choose **Add → Class**.
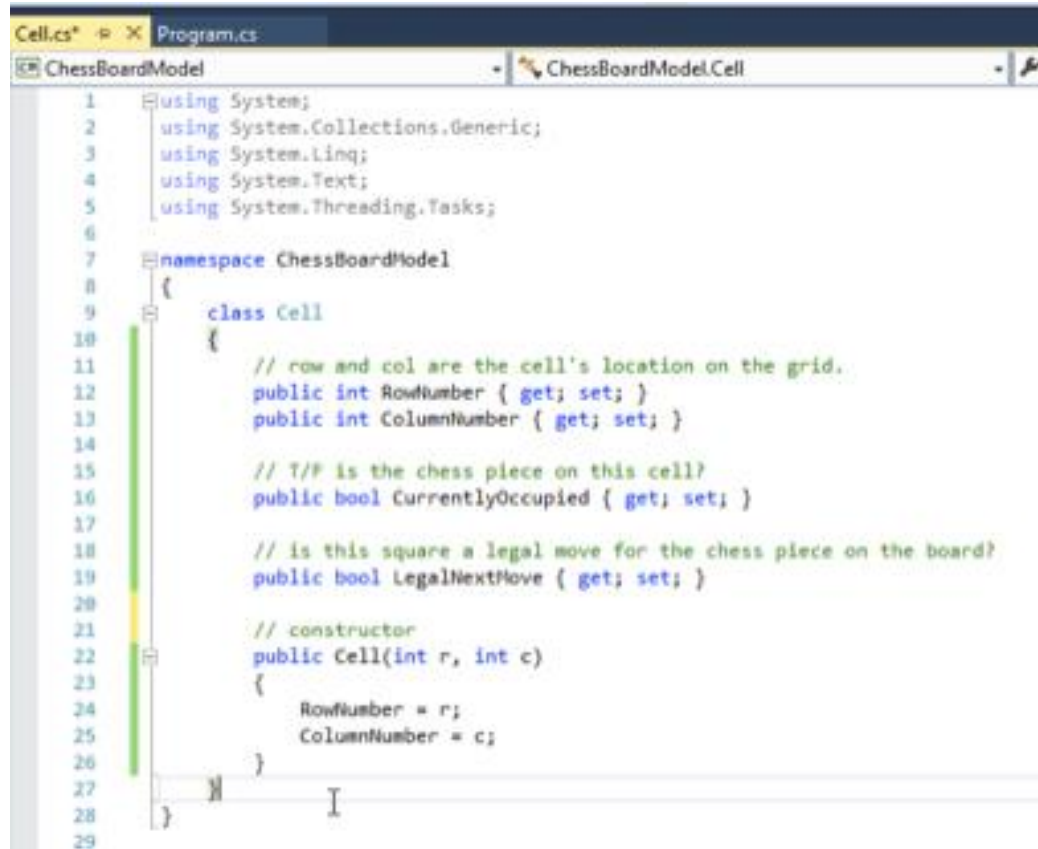
9. Use **Cell** for the class name.

Create the properties and constructor for the cell class.
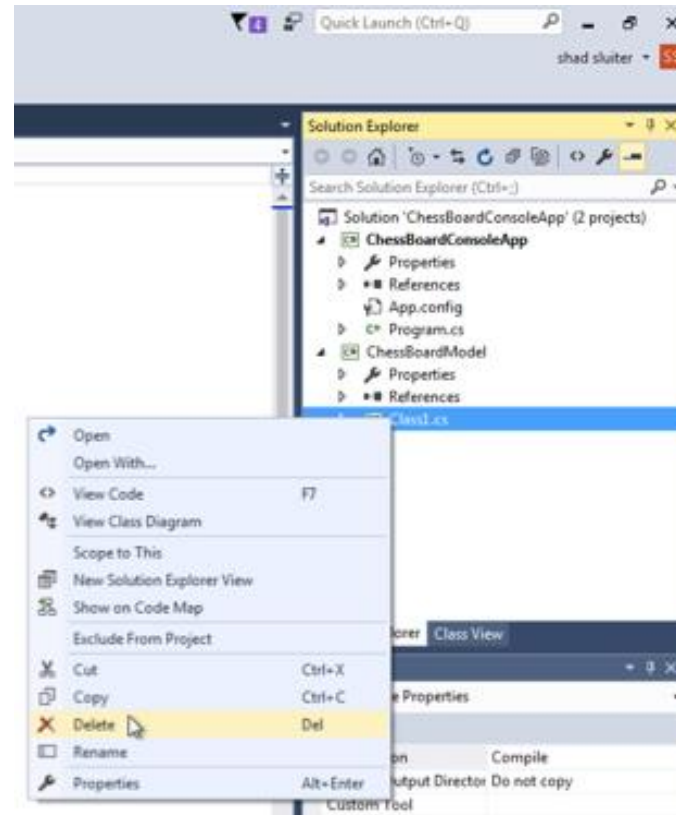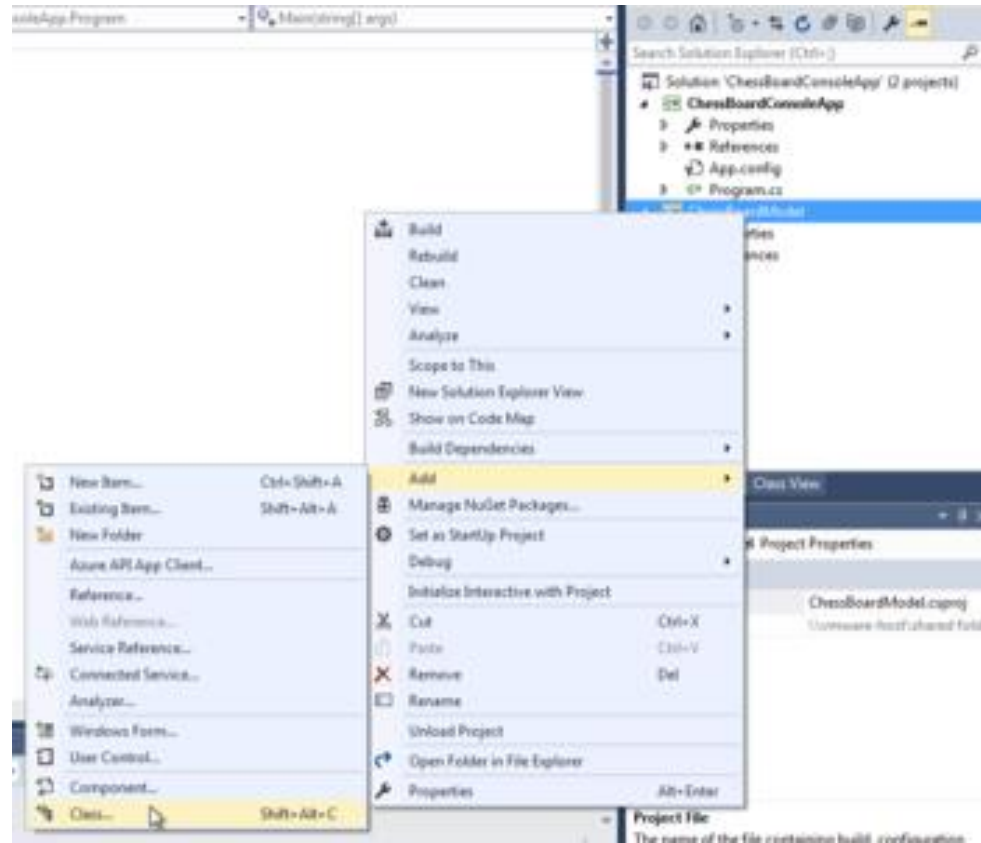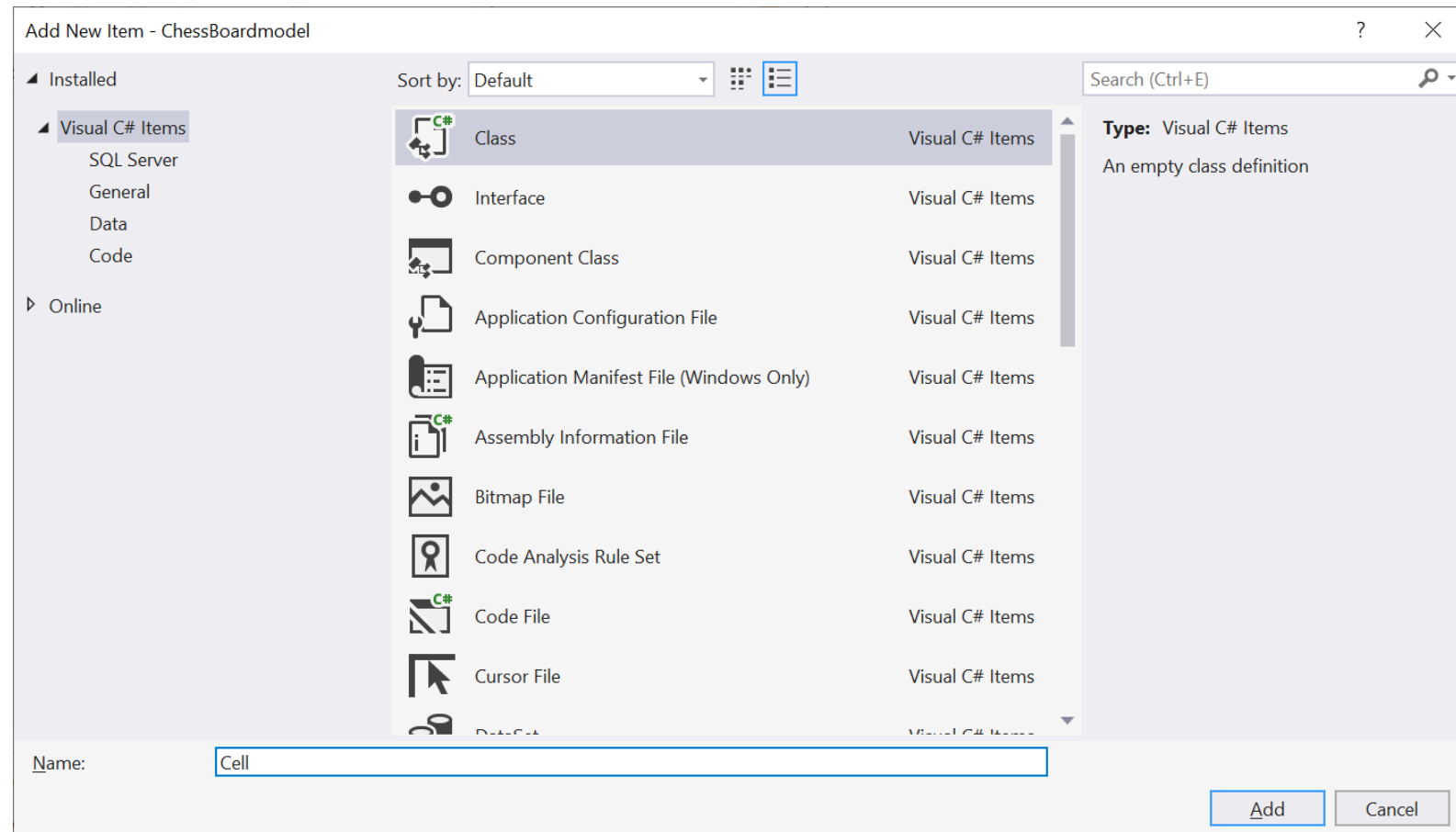
Each cell should have these properties:

RowNumber, ColumnNumber, CurrentlyOccupied and LegalNextMove

```
Cell.cs*  ₽ ×  Program.cs
ChessBoardModel                              ▼  ChessBoardModel.Cell                    ▼
1      using System;
2      using System.Collections.Generic;
3      using System.Linq;
4      using System.Text;
5      using System.Threading.Tasks;
6
7      namespace ChessBoardModel
8      {
9          class Cell
10         {
11             // row and col are the cell's location on the grid.
12             public int RowNumber { get; set; }
13             public int ColumnNumber { get; set; }
14
15             // T/F is the chess piece on this cell?
16             public bool CurrentlyOccupied { get; set; }
17
18             // is this square a legal move for the chess piece on the board?
19             public bool LegalNextMove { get; set; }
20
21             // constructor
22             public Cell(int r, int c)
23             {
24                 RowNumber = r;
25                 ColumnNumber = c;
26             }
27         }
28     }
29
```

10. Create another new class. Name it **Board.cs**
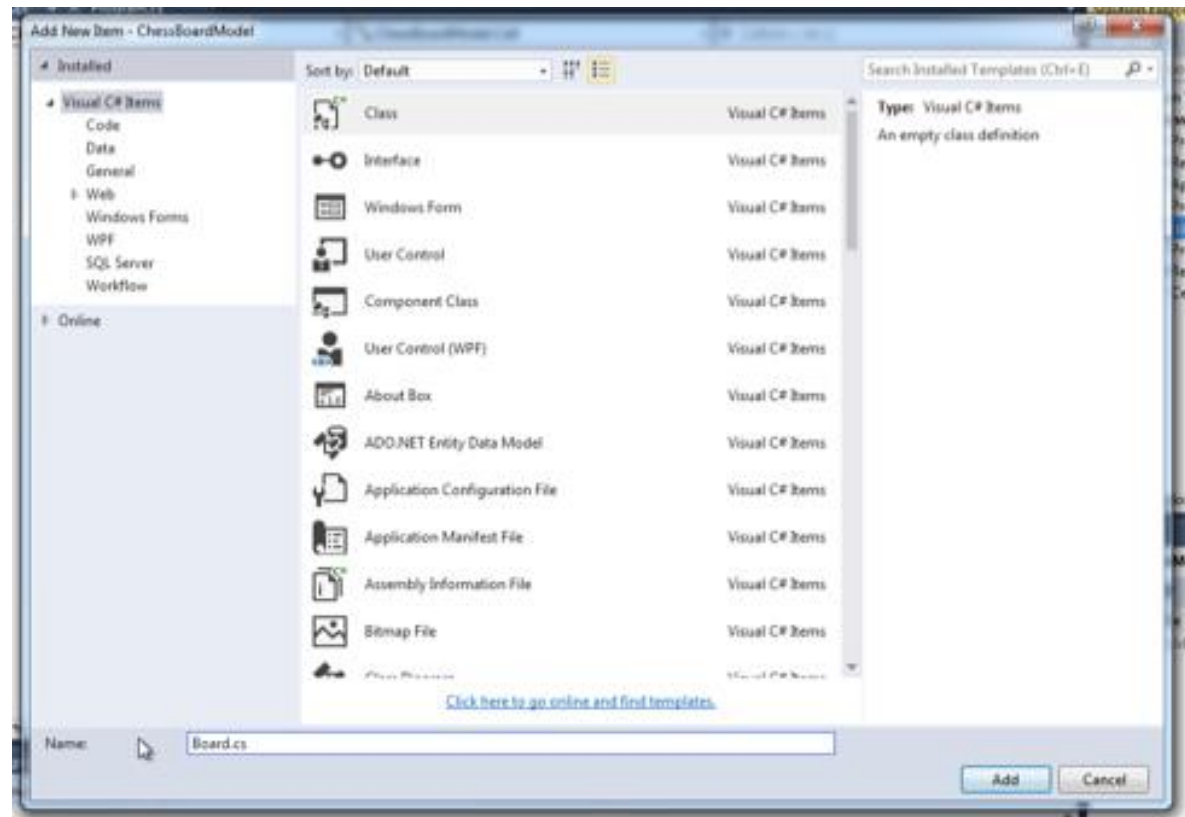
11. Create the properties **Size** and **theGrid**.

12. Create a constructor to **initialize** theGrid.

Size is a public integer.

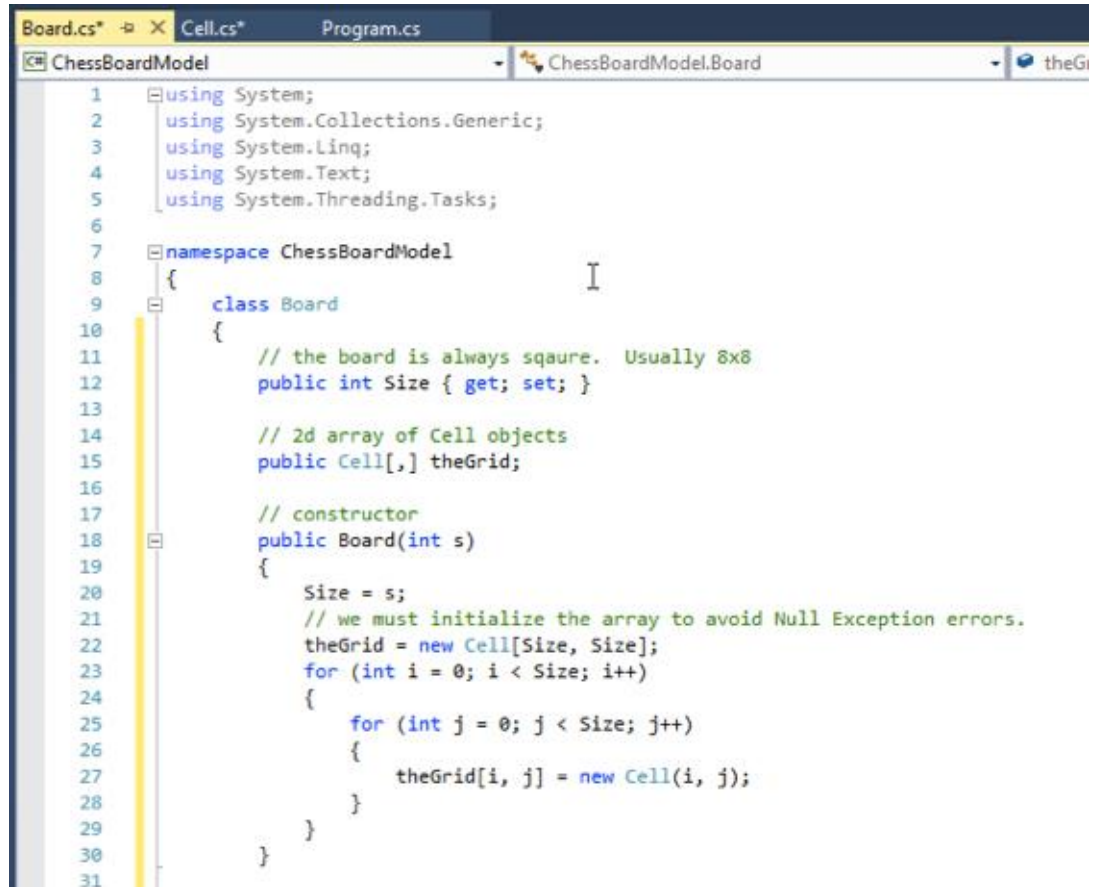**theGrid** is a 2D array that models the chessboard. Usually the board will be 8x8, but other sizes will work as well.

Mark the Board class and Cell as **public** so that other projects can view it. See lines 15 and 18 in the code for where to mark the class as public.

```
Board.cs*  ⊕ ✕  Cell.cs*        Program.cs
C# ChessBoardModel                          ▼  ⚙ ChessBoardModel.Board          ▼  ⬤ theG

 1   ⊟using System;
 2    using System.Collections.Generic;
 3    using System.Linq;
 4    using System.Text;
 5    using System.Threading.Tasks;
 6
 7   ⊟namespace ChessBoardModel
 8    {
 9   ⊟    class Board
10        {
11            // the board is always sqaure.  Usually 8x8
12            public int Size { get; set; }
13
14            // 2d array of Cell objects
15            public Cell[,] theGrid;
16
17            // constructor
18   ⊟        public Board(int s)
19            {
20                Size = s;
21                // we must initialize the array to avoid Null Exception errors.
22                theGrid = new Cell[Size, Size];
23                for (int i = 0; i < Size; i++)
24                {
25                    for (int j = 0; j < Size; j++)
26                    {
27                        theGrid[i, j] = new Cell(i, j);
28                    }
29                }
30            }
31
```
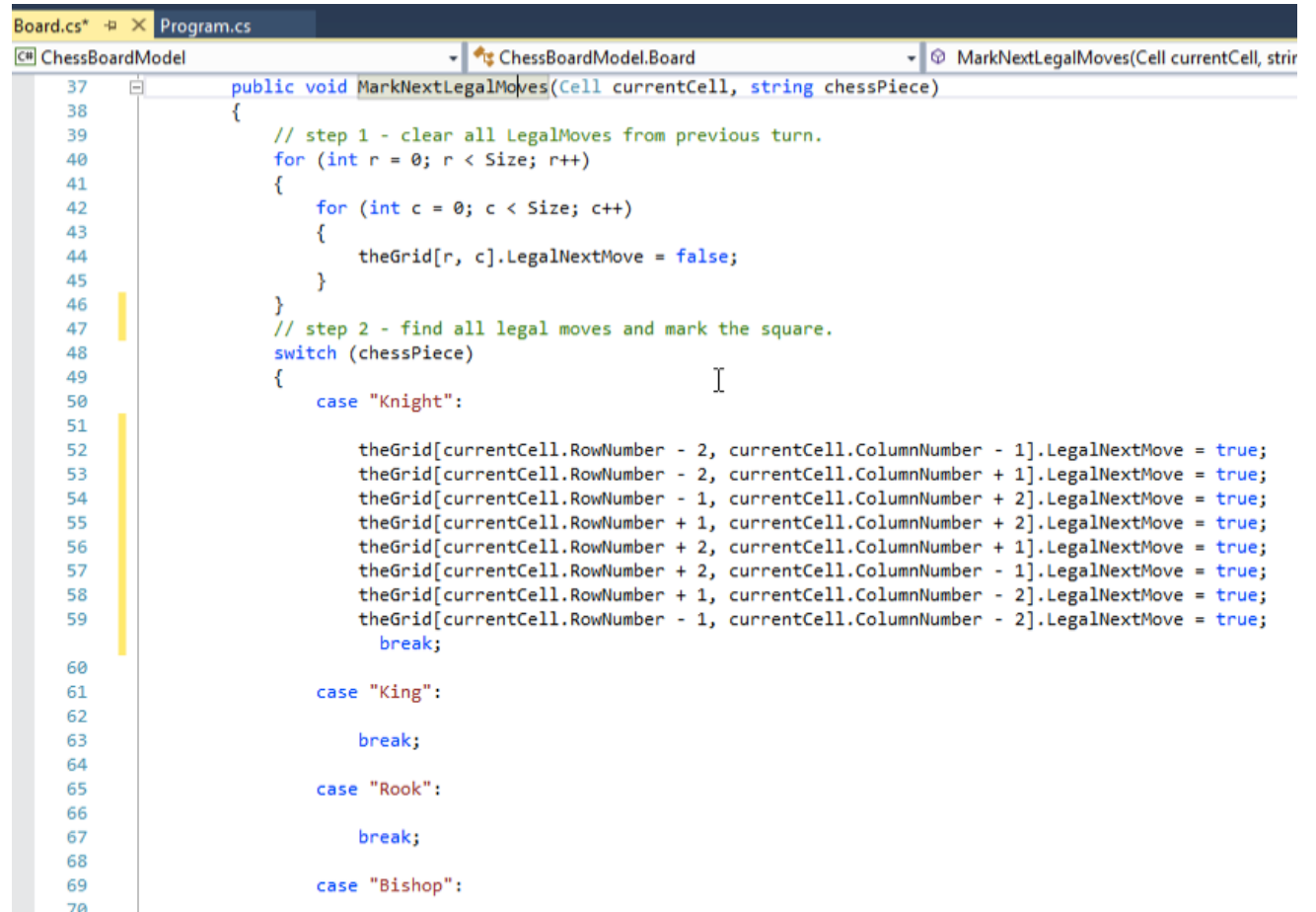
13. Create a method called MarkNextLegalMoves.

This is used to find all possible next squares that the piece may be placed.

Clear the Board -

In case the board already contains information about legal moves from previous turns in the game, reset the "LegalMove" property for all cells on the board.

Mark New Legal Moves -

Lines 51 to 59 show the math required to calculate all of the possible legal positions that a knight could move to. You should ask yourself what happens if any of the next legal moves are off the edge of the playing board. For example, if you try to place a Knight in cell (0,0), some of these statements that recommend moving the knight up or to the left are going to cause an "Out of Range" exception error with the grid array. You will need to create another helper function to check to see if a row and column pair is a valid square on the board before trying to reference that cell. That job will be left to you in one of the **challenges** at the end of the assignment.

```
Board.cs*  ⊕ ✕  Program.cs
C# ChessBoardModel                              ▾ ᵗᵍ ChessBoardModel.Board              ▾ ⊕ MarkNextLegalMoves(Cell currentCell, strin
    37    ⊟         public void MarkNextLegalMoves(Cell currentCell, string chessPiece)
    38              {
    39                  // step 1 - clear all LegalMoves from previous turn.
    40                  for (int r = 0; r < Size; r++)
    41                  {
    42                      for (int c = 0; c < Size; c++)
    43                      {
    44                          theGrid[r, c].LegalNextMove = false;
    45                      }
    46                  }
    47                  // step 2 - find all legal moves and mark the square.
    48                  switch (chessPiece)
    49                  {
    50                      case "Knight":
    51
    52                          theGrid[currentCell.RowNumber - 2, currentCell.ColumnNumber - 1].LegalNextMove = true;
    53                          theGrid[currentCell.RowNumber - 2, currentCell.ColumnNumber + 1].LegalNextMove = true;
    54                          theGrid[currentCell.RowNumber - 1, currentCell.ColumnNumber + 2].LegalNextMove = true;
    55                          theGrid[currentCell.RowNumber + 1, currentCell.ColumnNumber + 2].LegalNextMove = true;
    56                          theGrid[currentCell.RowNumber + 2, currentCell.ColumnNumber + 1].LegalNextMove = true;
    57                          theGrid[currentCell.RowNumber + 2, currentCell.ColumnNumber - 1].LegalNextMove = true;
    58                          theGrid[currentCell.RowNumber + 1, currentCell.ColumnNumber - 2].LegalNextMove = true;
    59                          theGrid[currentCell.RowNumber - 1, currentCell.ColumnNumber - 2].LegalNextMove = true;
    60                              break;
    61
    62                      case "King":
    63
    64                              break;
    65
    66                      case "Rook":
    67
    68                              break;
    69
    70                      case "Bishop":
```
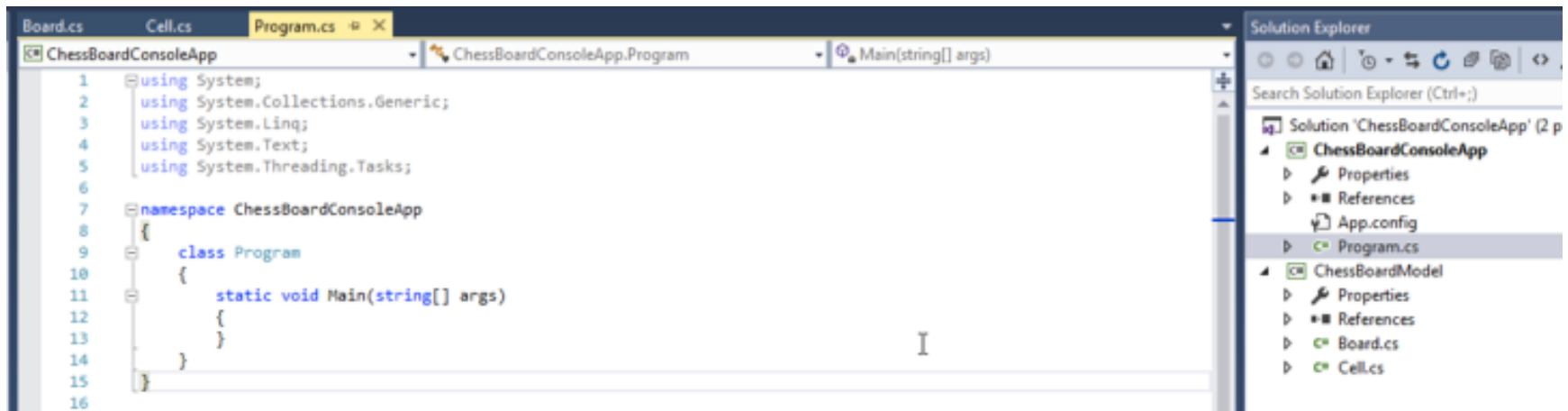
Switch Case "Knight" –

The first case for only one chess piece has been shown (line 50 above). You will have to complete the code for the other four cases: **King, Rook, Bishop, and Queen** (shown below). The Knight is perhaps the most mystifying piece on the board for a non-chess player, so the other four cases should be easier for you to code.

```
67            theGrid[currentCell.RowNumber - 1, currentCell.ColumnNumber - 2].LegalNextMove = true;
68
69                break;
70
71        case "King":
72
73                break;
74
75        case "Rook":
76
77                break;
78
79        case "Bishop":
80
81                break;
82
83        case "Queen":
84
85                break;
86
87        default:
88                break;
89            }
90
91
92        }
93
94    }
95 }
96
97
```

Part 2 - The Console App

In this section, we will create a console text-only interface to play the game. The user will be asked column and row coordinates for the location of the chess piece. The program will then print a board with all of the possible legal destinations for that piece.

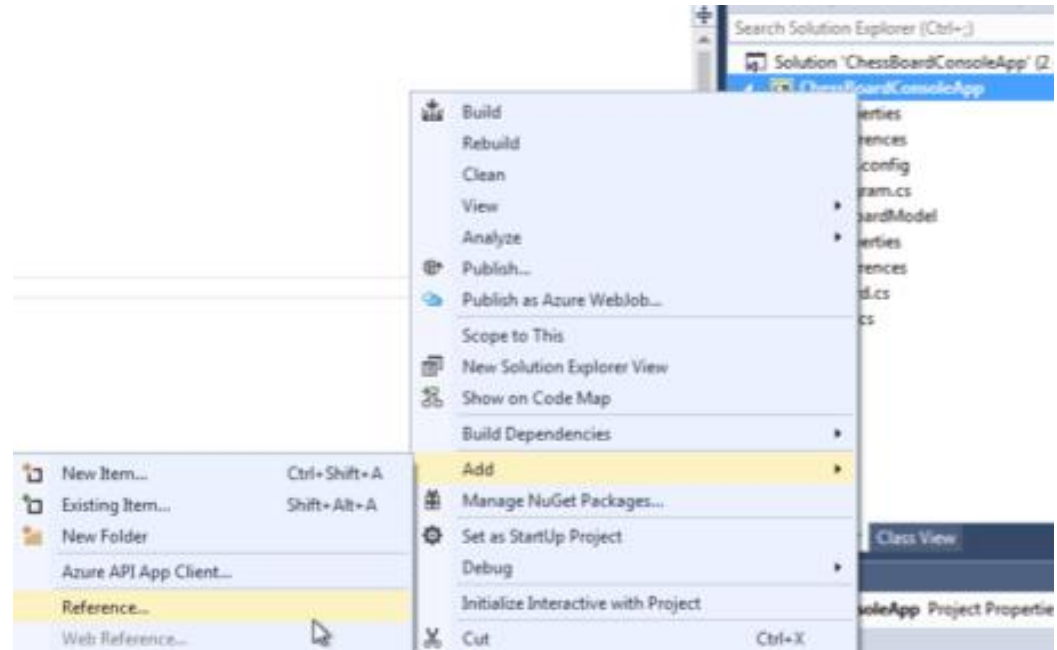1.   In the **ChessBoardConsoleApp** project, open the **Program.cs** file.

2.   In the **ChessBoardConsoleApp** project, add a reference to the **ChessBoardModel**.

3.   Open the **Program.cs** file in the **ChessBoardConsoleApp** project.
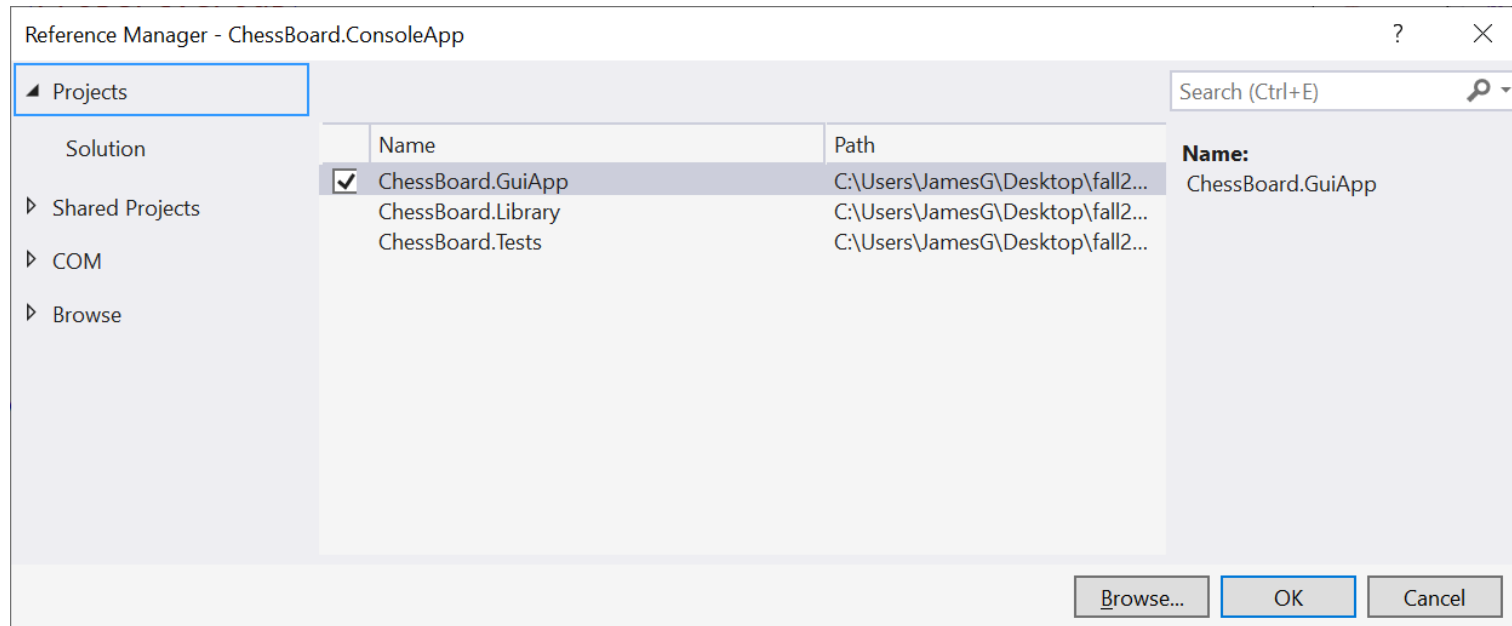
4.   Write out comments to guide us in creating steps in our program (image below).

a) show the empty chessboard

b) get the location of the chess piece

c) calculate and mark the cells where legal moves are possible.

 d) show the chess board. Use . for an empty square, X for the piece location and + for a possible legal
    move

e) wait for another return key to end the program

File Edit View Project Build Debug Test Analyze Tools Extensions Window Help   Search (Ctrl+Q)   ChessBoard

Debug   Any CPU   ChessBoard.Library   ▶ ChessBoard.Library

Live Share

Program.cs*

ChessBoard.ConsoleApp   ChessBoardConsoleApp.Program   Main(string[] args)

```csharp
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6
7   namespace ChessBoardConsoleApp
8   {
        0 references
9       class Program
10      {
            0 references
11          static void Main(string[] args)
12          {
13              // show the empty chessboard
14              // get the location of the chess piece
15              // calculate and mark the cells where leagal moves are possible
16              // Show the chessboard and use "." for an empty square, "X" for the piece location
17              // and "+" for a possible legal move.
18              // Wait for another return key to exit the program
19          }
20      }
21  }
22
```

100 %   No issues found   Ln: 21   Ch: 2   SPC   CRLF

Team Explorer - Connect

Connect | Offline

Manage Connections ▾

▲ Hosted Service Providers

Azure DevOps
Microsoft Corporation

Services to help you ship high quality software. On time, every time. Focus on your code. We'll simplify the rest.

Connect...   Get started for free

▲ Local Git Repositories

Git features have moved. Use the Git menu to access your existing local

Properties

Output

Show output from: Package Manager

Error List   Output

Ready   ↑ Add to Source Control ▲

5. Create an instance of the Board class (see line 11).

6. Invent some function names that will accomplish the tasks listed in the comments plan. These function names will cause IDE warnings but ignore them for now. We will create these functions in the next steps.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ChessBoardConsoleApp
{
    class Program
    {
        static Board myBoard = new Board(8);

        static void Main(string[] args)
        {
            // show the empty chess board
            printGrid(myBoard);

            // get the location of the chess piece
            Cell myLocation = setCurrentCell();

            // calculate and mark the cells where legal moves are possible.
            myBoard.MarkNextLegalMoves(myLocation, "Knight");

            // show the chess board.  Use . for an empty square, X for the piece location and + for a possible
            legal move
            printGrid(myBoard);

            // wait for another return key to end the program
            Console.ReadLine();
        }
    }
}
```

7.   In lines 11 and 17, you can see that the data types "Board" and "Cell" are unrecognized. The Cell data type will not work until we insert the "using" statement in the header of the file. Hover over the word "cell" and click on the "suggested fixes".

Cell and Board classes are now usable in our project.

```csharp
ChessBoardConsoleApp          ▼    ChessBoardConsoleApp.Program          ▼   Main(st

1   using ChessBoardModel;
2   using System;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Text;
6   using System.Threading.Tasks;
7
8   namespace ChessBoardConsoleApp
9   {
10      class Program
11      {
12          static Board myBoard = new Board(8);
13
14          static void Main(string[] args)
15          {
16              // show the empty chess board
17              printGrid(myBoard);
18
19              // get the location of the chess piece
20              Cell myLocation = setCurrentCell();
21
22              // calculate and mark the cells where legal moves are possible.
23              myBoard.MarkNextLegalMoves(myLocation, "Knight");
24
25              // show the chess board.  Use . for an empty square, X for the piece l
                   legal move
26              printGrid(myBoard);
27
28              // wait for another return key to end the program
29              Console.ReadLine();
30          }
31      }
32   }
33
```

8. Create the function **printBoard()**. You can see that I am using a single letter to represent each square. In a challenge at the end of this assignment, you are going to create a more elaborate-looking board.

```
Miscellaneous Files          ChessBoardConsole.Program          Main(string[] args)
30
31
32              }
33      static public void printGrid(Board myBoard)
34      {
35      // print the board on the console.  Use "X" for current location, "+" for legal move and "." for an
        empty square.
36          for (int i = 0; i < myBoard.Size; i++)
37          {
38              for (int j = 0; j < myBoard.Size; j++)
39              {
40                  if (myBoard.theGrid[i, j].CurrentlyOccupied )
41                  {
42                      Console.Write("X");
43                  }
44                  else if (myBoard.theGrid[i, j].LegalNextMove)
45                  {
46                      Console.Write("+");
47                  }
48                  else
49                  {
50                      Console.Write(".");
51                  }
52              }
53              Console.WriteLine();
54          }
55          Console.WriteLine("=================================");
56      }
57
```

9. Create the function

   setCurrentCell(). This function's job is to get the row and column number of the piece on the board. It also sets the "Occupied" property to **true** for the cell on the board where the piece is placed.

Notice that the program does not ask which kind of piece the player wants to use (Knight, Queen, King, Bishop, or Rook). That feature is left to you in a challenge at the end of the assignment.

You should be able to run the app now.

```
53                    Console.WriteLine();
54                }
55                Console.WriteLine("=================================");
56            }
57
58        static public Cell setCurrentCell()      I
59        {
60            Console.Out.Write("Enter your current row number ");
61            int currentRow = int.Parse(Console.ReadLine());
62
63            Console.Out.Write("Enter your current column number ");
64            int currentCol = int.Parse(Console.ReadLine());
65
66            myBoard.theGrid[currentRow, currentCol].CurrentlyOccupied = true;
67
68            return myBoard.theGrid[currentRow, currentCol];
69        }
70    }
71 }
72
```

GRAND CANYON
U N I V E R S I T Y
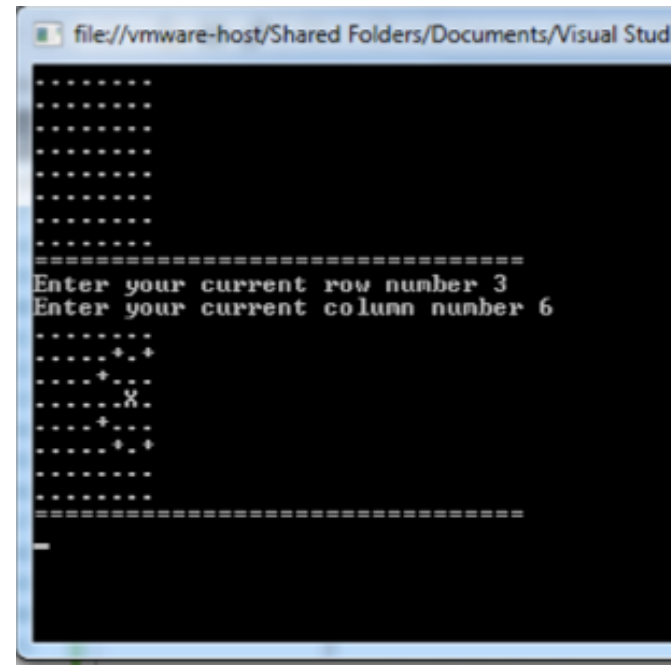
Challenges

The program works but is not yet finished. Here are some things you need to complete.

1. **Fix Out-of-Bounds Errors.**

   Your job is to check to see if an item in the grid array actually exists before you try to reference it.

2. **Fix InputEerrors**

   The user is supposed to type in a number between 0 and 7 when asked to place a piece on the board. If he/she tries to put in letters or numbers outside of the acceptable range, the program will crash. Verify the input from the user before proceeding with the rest of the program.

3. **Multiple Pieces**

   The program is only designed to work with the knight chess piece. Modify the program so that it asks which piece should be placed on the board and then gives the correct legal moves for that piece.

4. **Board Printing Upgrade**

   The program shows only a single character for each board cell. Modify the output to print a board outline like the example shown here.

```
You have chosen to put a Knight at (3, 4).
Here are the legal moves you can now make.
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   | + |   | + |   |   |
+---+---+---+---+---+---+---+---+
|   |   | + |   |   |   | + |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   | X |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   | + |   |   |   | + |   |
+---+---+---+---+---+---+---+---+
|   |   |   | + |   | + |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

Part 3 – Windows Application

In this section, we are going to repurpose the **ChessBoardModel** project by combining it with a GUI front end. The function of the application will be the same, but the visual appeal will be much higher.

You will need to have completed Part 2 of this assignment to successfully complete Part 3.

1.   Choose **File → New → Project**

2.   Select the **Windows Forms Application** option.
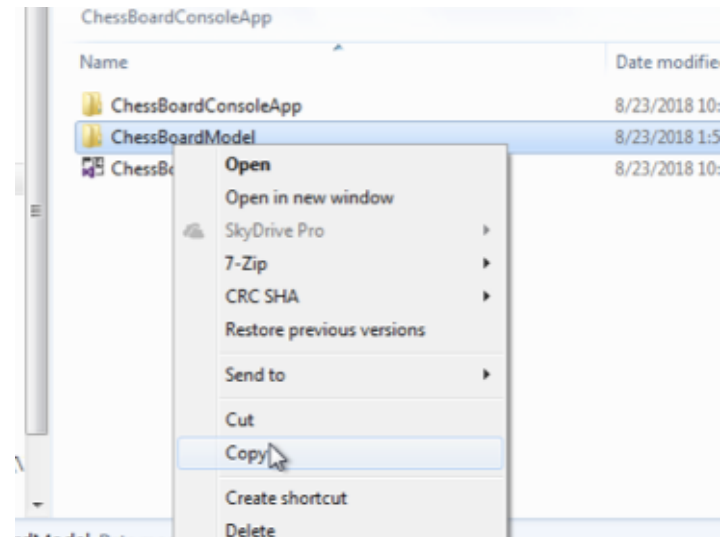
3.   Name the project **ChessBoardGUIApp**.

Now we need to make a copy of the previous Board and Cell class project and include it in our newest version of the chess board app.
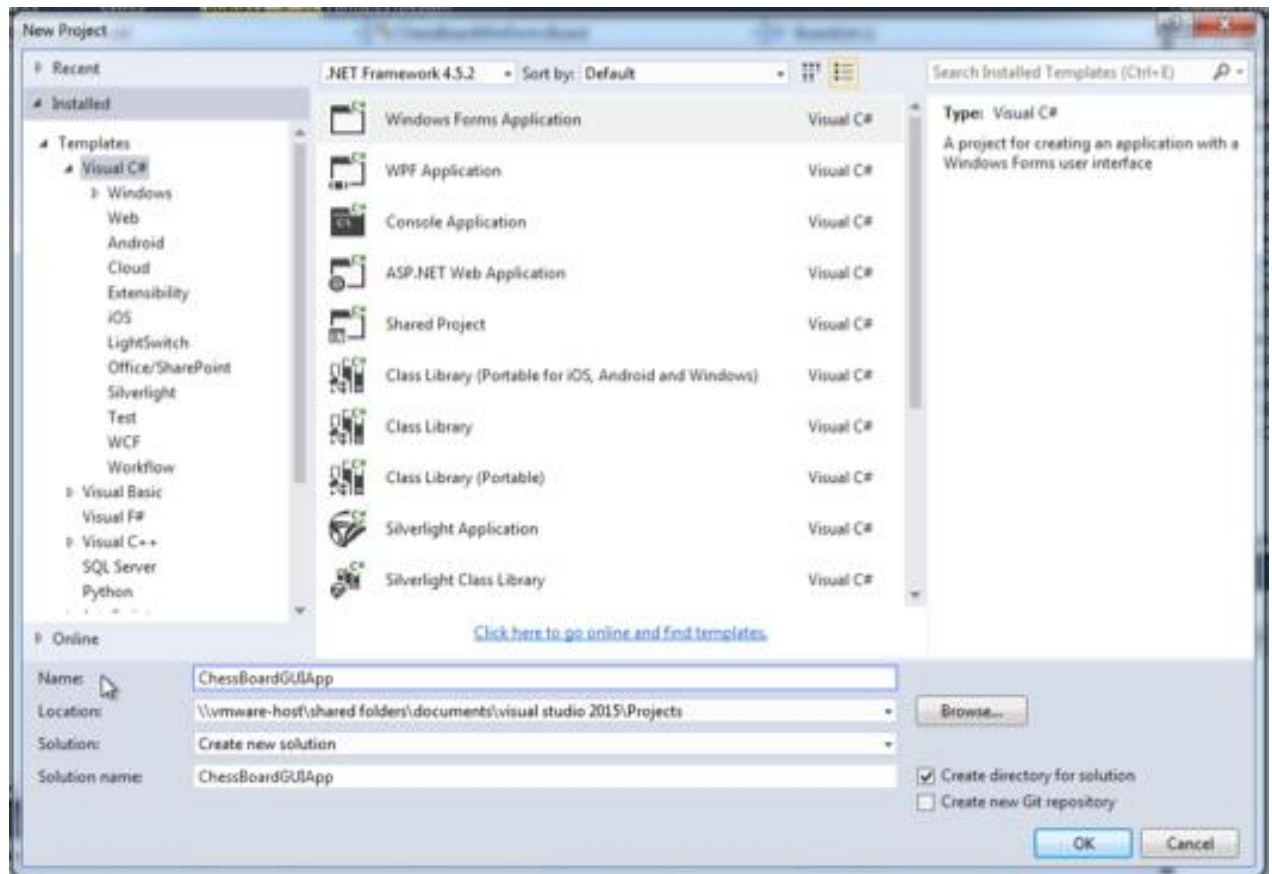
4. In the Windows file explorer, locate the previous project folder. It is probably located at **c:\users\yourname\documents\Visual Studio 2015\Projects\ChessBoardConsoleApp**
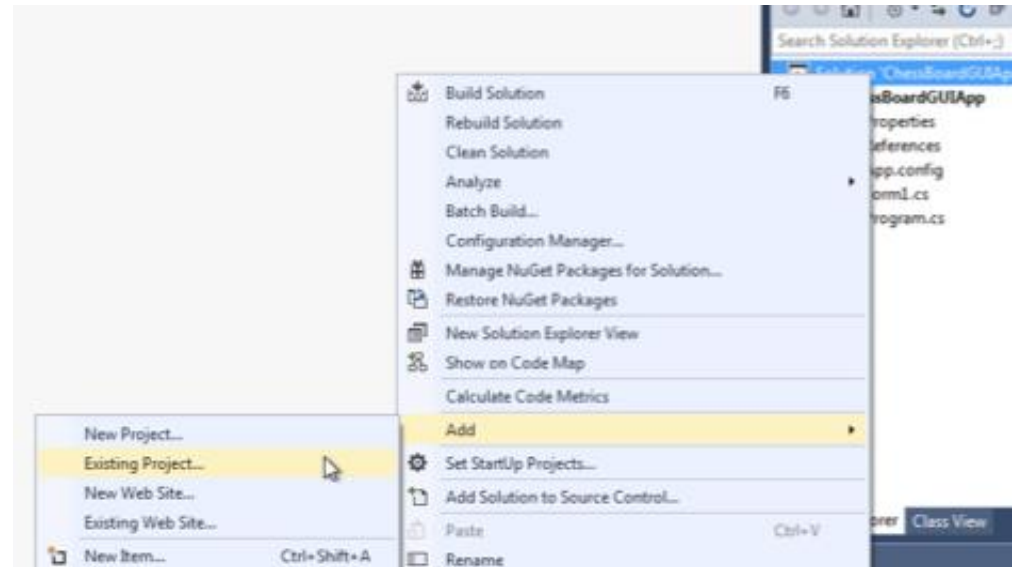

5. Copy the folder ChessBoardModel.

6. Navigate to the new project folder you just created. It is probably located at c:\users\yourname\documents\visual studio 2015\projects\chessBoardGUIApp.

7. Paste the folder into this new project folder. Right-click in the white space and choose Paste.
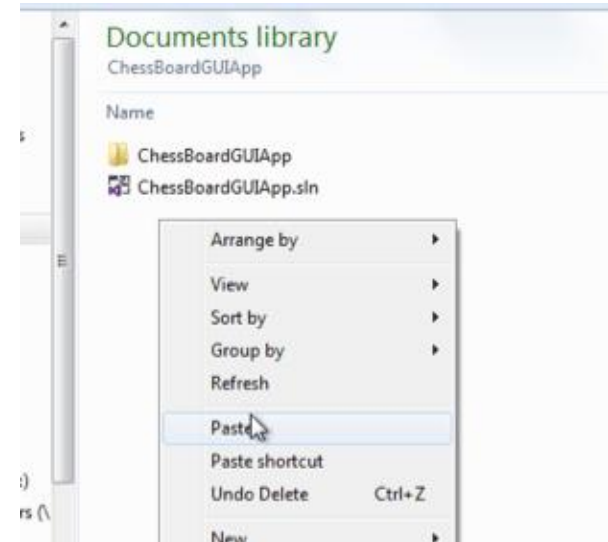
The ChessBoardGUIApp should now have two folders inside it
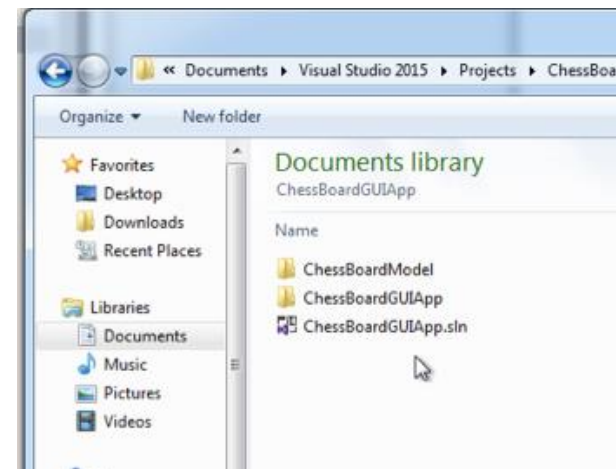
8. Return to Visual Studio.

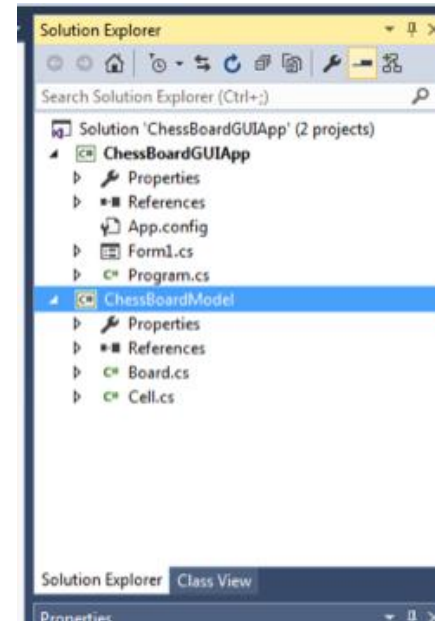9. **Right-click** on the solution title.

   Choose **Add → Existing Project**

10. Navigate to the folder:
     c:\users\yourname\documents\visualstudio2015\projects\ChessBoardGUIApp\ChessBoardModel.

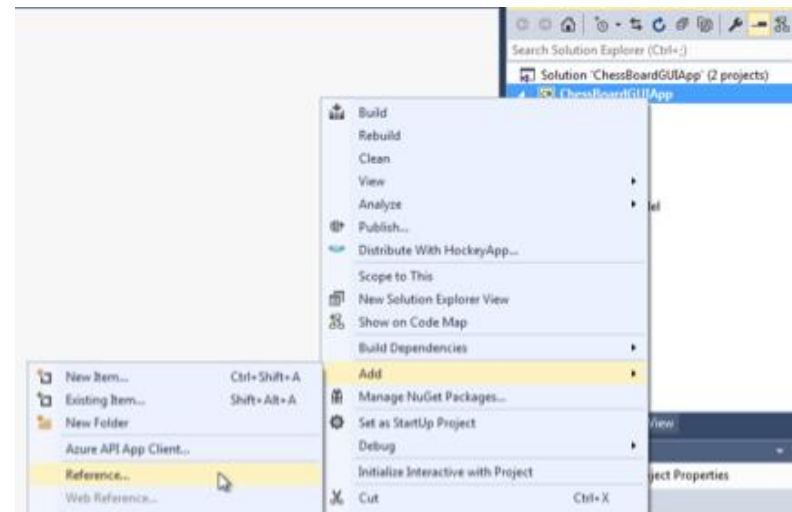11. Select the file **ChessBoardModel**.cproj and click the **Open** button.
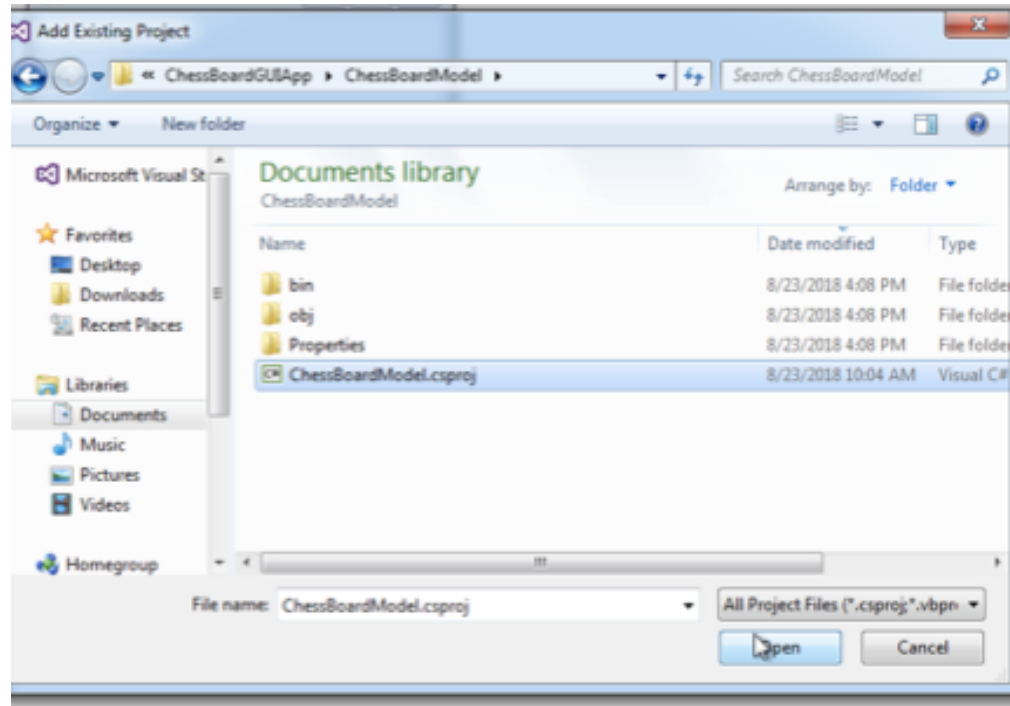
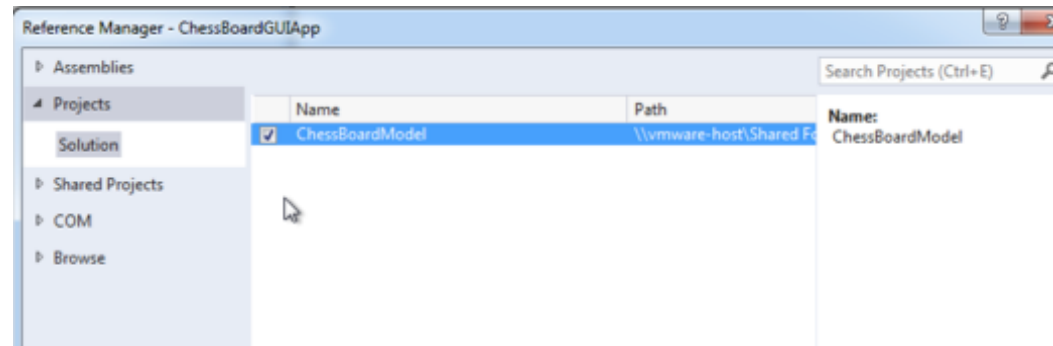You should see two projects in your Solution Explorer window.

12.   Add a reference to the new project.

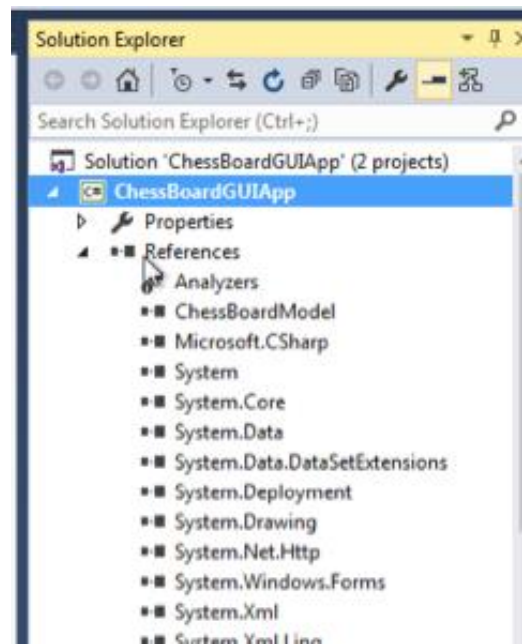Right-click on the **ChessBoardGUIApp** name.

Choose **Add → Reference**.

13. In the Reference Manager window, **check** the box for **ChessBoardModel**.
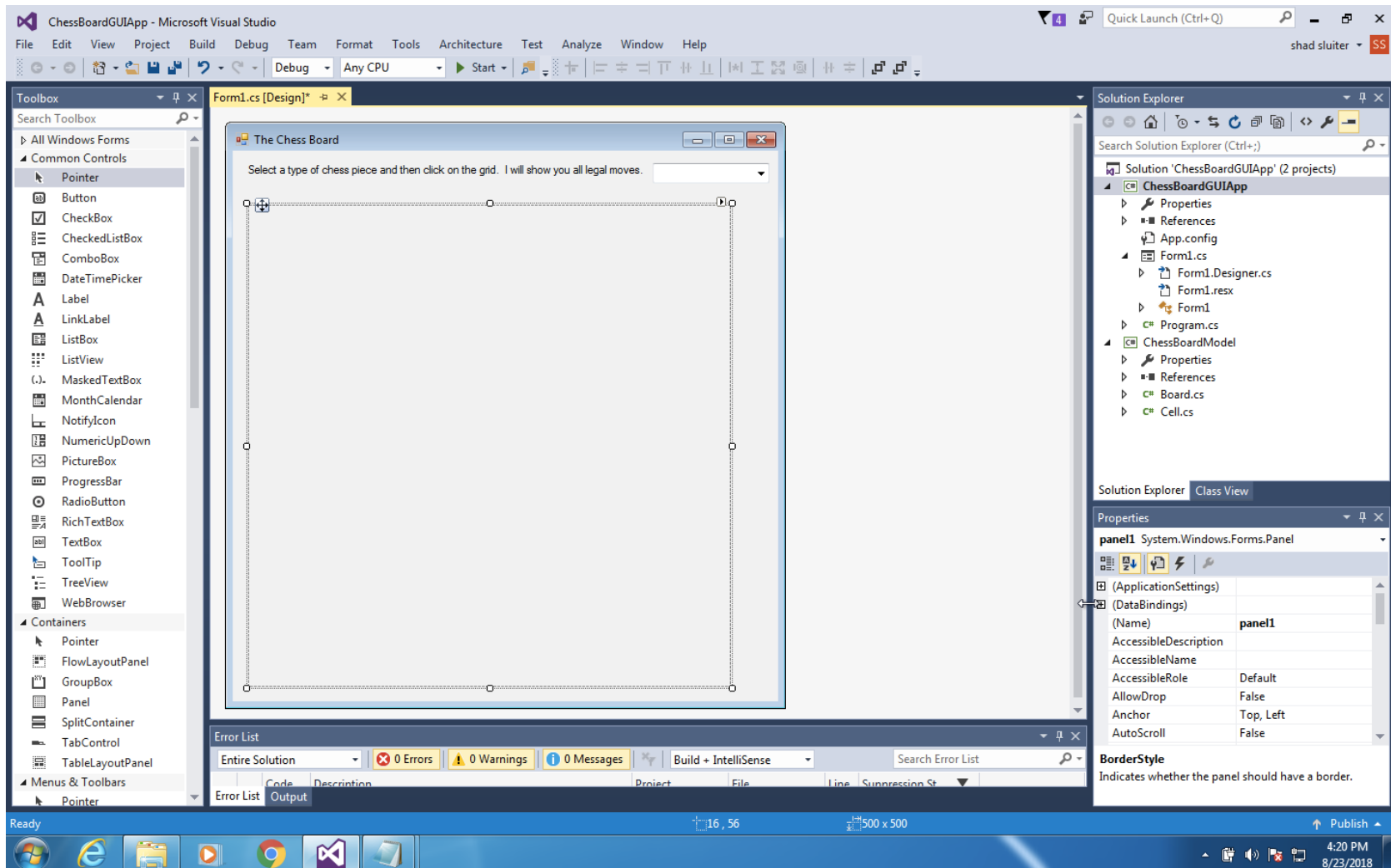

You should see the **ChessBoardModel** in the list of **references** for this project.
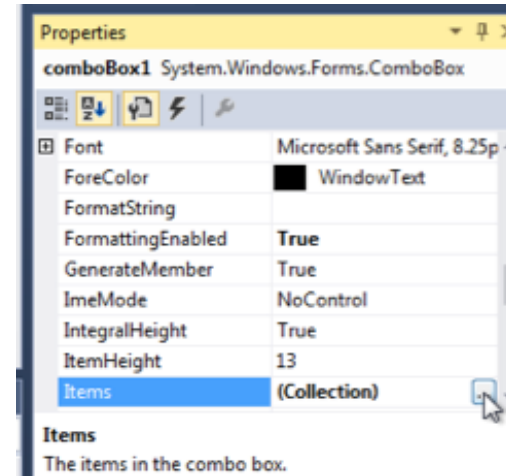
14. Expand the size of Form1 to about 600 x 600 pixels.

15. Place the following controls on the form: Label1, ComboBox1 and Panel1. I set the size of Panel1 to 500 x 500 pixels. I changed Label1.Text to give brief instructions to the user.

16.   Select ComboBox1.

17.   In the Properties list, select the line called "items."

18.   Click the ellipsis button next to the word (collection). This will open a dialog box where you can fill in the contents of the control.

19.   Create a list of the names of the chess pieces. Write one name per line.

20.   Click OK to close the dialog box.
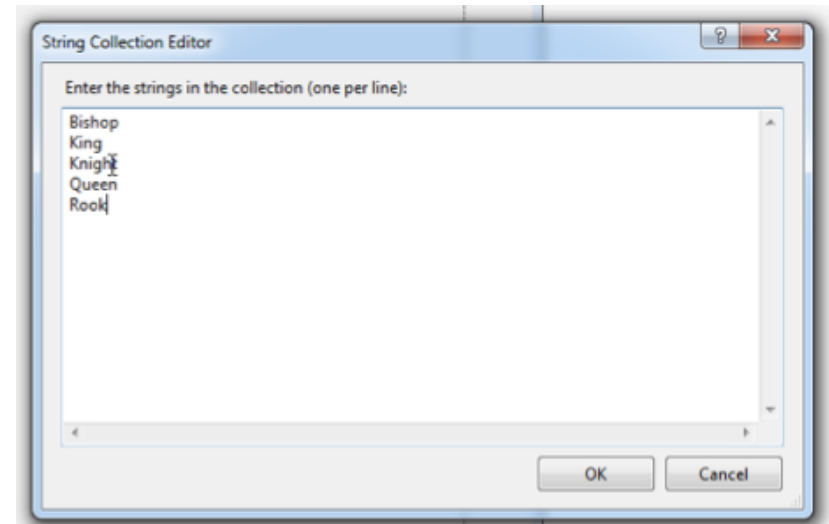
21.   Change the **DropDownStyle** property to Drop Down List.

Now let's do some programming.
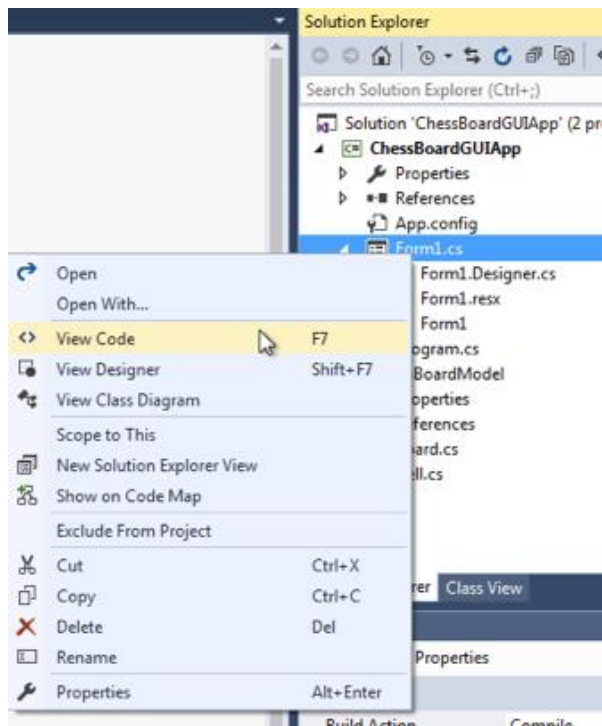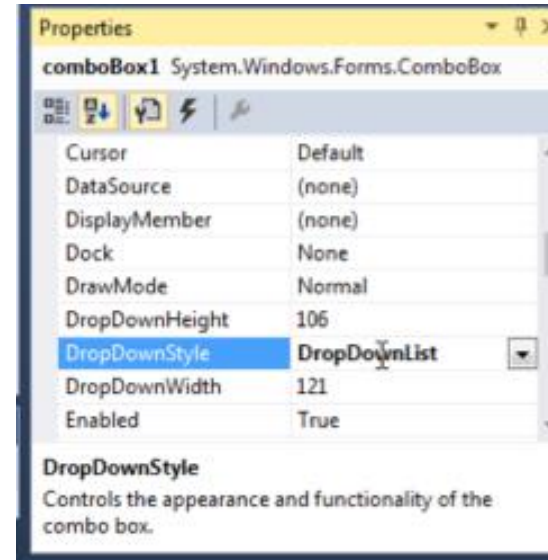
22.   Right-click on the Form1.cs line and choose View Code.

23. Create a new instance of the Board class with size 8 on line 15.

24. Create a 2D array of buttons on line 16.

25. Add the **using ChessBoardModel** statement to the top of the project. You can do this automatically by hovering over the word **Board** and choosing the **auto fix suggestion**.

26. Inside the Form1() constructor function, we will add a new helper function called **populateGrid**(). The function will be unknown to the compiler, but we will create the function in the next step.

27. Create the **populateGrid** function as shown here.

```
11
12      namespace ChessBoardGUIApp
13      {
14  ⊟      public partial class Form1 : Form
15          {
16              static public Board myBoard = new Board(8);
17              public Button[,] btnGrid = new Button[myBoard.Size, myBoard.Size];
18
19  ⊟          public Form1()
20              {
21                  InitializeComponent();
22                  populateGrid();
23              }
24      }
25
```

Code Explanation:

Study the comments in the code. They explain what is going on.

First, we need to some calculations. We want all the buttons to be the same size. We divide the width of Panel1 by the number of buttons we plan to place there. This value becomes **buttonSize**.

To ensure that the Panel is square, set the **Height** equal to the **Width**.

Run a loop through each row and then column to place a new button on the Panel. We are *programmatically* creating button controls on the Form instead of placing each button on the form designer by hand.

We will need to create a function to handle the button clicks so you should see an error on line 42 where the code mentions **Grid_Button_Click**. This is not a keyword name so we could have named it *Handle_Cell_Click* or something similar.
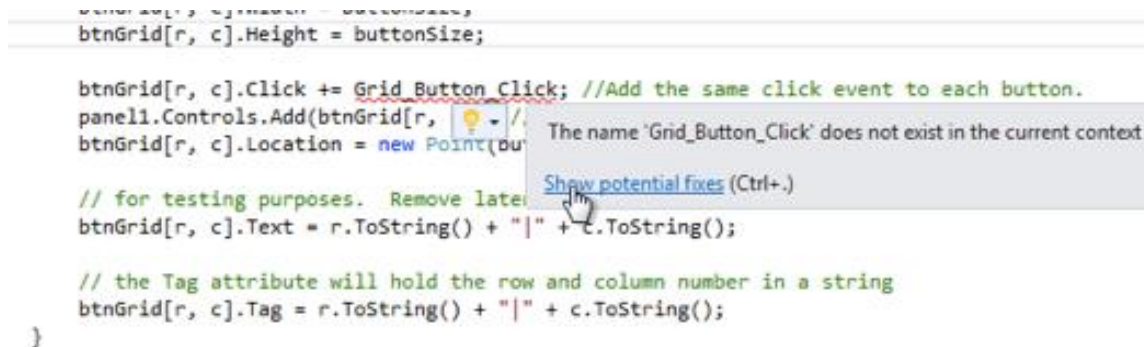
```
1.cs*        Board.cs        Form1.cs*  ⊟ ✕  Form1.cs [Design]*
hessBoardGUIApp                    ▾  ⁑ ChessBoardGUIApp.Form1            ▾  ⊕ populateGrid()
  18
  19    ⊟        public Form1()
  20             {
  21                 InitializeComponent();
  22                 populateGrid();
  23             }
  24
  25    ⊟        public void populateGrid()
  26             {
  27                 // this function will fill the panel1 control with buttons.
  28                 int buttonSize = panel1.Width / myBoard.Size;  // calculate the width of each button on the Grid
  29                 panel1.Height = panel1.Width;    // set the grid to be square.
  30
  31                 // nested loop. Create buttons and place them in the Panel
  32                 for (int r = 0; r < myBoard.Size; r++)
  33                 {
  34                     for (int c = 0; c < myBoard.Size; c++)
  35                     {
  36                         btnGrid[r, c] = new Button();
  37
  38                         // make each button square
  39                         btnGrid[r, c].Width = buttonSize;
  40                         btnGrid[r, c].Height = buttonSize;
  41
  42                         btnGrid[r, c].Click += Grid_Button_Click; //Add the same click event to each button.
  43                         panel1.Controls.Add(btnGrid[r, c]); // place the button on the Panel
  44                         btnGrid[r, c].Location = new Point(buttonSize * r, buttonSize * c); // position it in x,y
  45
  46                         // for testing purposes.  Remove later.
  47                         btnGrid[r, c].Text = r.ToString() + "|" + c.ToString();
  48
  49                         // the Tag attribute will hold the row and column number in a string
  50                         btnGrid[r, c].Tag = r.ToString() + "|" + c.ToString();
  51                     }
  52                 }
  53             }
  54         }
```
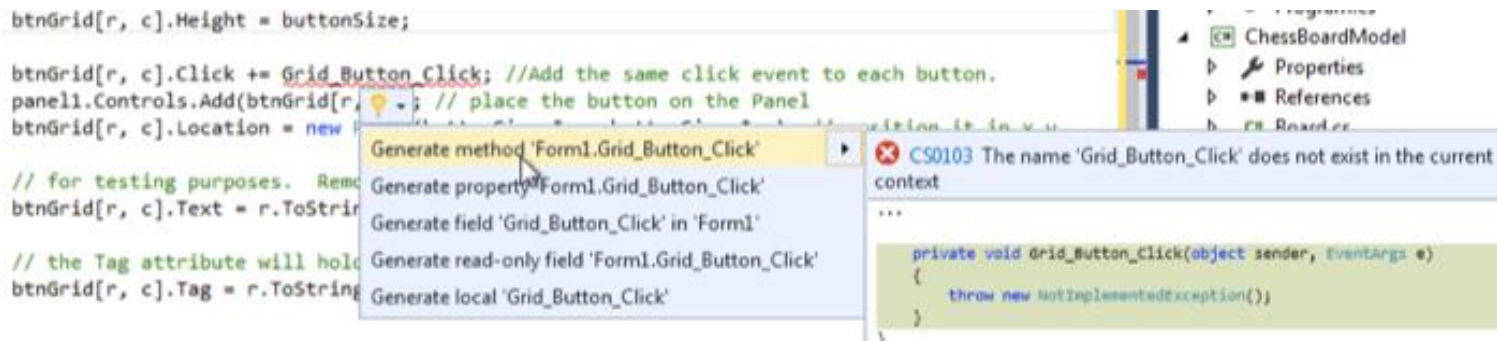
For testing purposes, we are going to print the row and column number on each button. We will remove line 47 at a later time.

The **Tag** attribute is like an **invisible Text property**. You can store a string on each control. This will be useful later when we determine the row and column of a button.

28.   Add a function for the **Grid_Button_Click** to temporarily handle any button clicks. Every button should be connected to this click action.

The IDE suggests you add a new function when you hover over the offending text.

29.   Choose **Genenerate method 'Form1.Grid_Button_Click'**

30. Put a Messagebox command in the new function.

```
49                    // the Tag attribute will hold the row and colum
50                    btnGrid[r, c].Tag = r.ToString() + "|" + c.ToStr:
51                }
52            }
53        }
54
55        private void Grid_Button_Click(object sender, EventArgs e)
56        {
57            MessageBox.Show("you clicked a button!");
58        }
59    }
60 }
61
```

31. Run the program. You should notice several things:
   a. The buttons are programmatically created and positioned in a grid.
   b. The buttons all have their row and column numbers printed on them.
   c. Every button is connected to the same click event.

32.   Replace the code in the **Grid_Button_Click** function to match the following image.

Code Explanation.

The method has an important parameter called object sender. This refers to the control that caused this method to be called. We can refer to this parameter later as (sender as Button).

On lines 75 to 78, the program gets the row and column number from the Tag value of the button that was clicked.

On lines 80 to 83, we call a function from the back-end classes to identify all legal moves. We hard-coded the word "Knight" into the function call. However, we will change this in one of the challenges at the end of the assignment.

On lines 85 to 91, we reset the background color of all the buttons.

On line 95, we set the clicked button background to another color.

```
72
73          private void Grid_Button_Click(object sender, EventArgs e)
74          {
75              // get the row and column number of the button just clicked.
76              string[] strArr = (sender as Button).Tag.ToString().Split('|');
77              int r = int.Parse(strArr[0]);
78              int c = int.Parse(strArr[1]);
79
80              // run a helper function to label all legal moves for this piece.
81              Cell currentCell = myBoard.theGrid[r, c];
82              myBoard.MarkNextLegalMoves(currentCell, "Knight");
83              updateButtonLabels();
84
85              // reset the background color of all buttons to the default (original) color.
86              for (int i=0; i<myBoard.Size; i++)
87              {
88                  for (int j=0; j<myBoard.Size; j++)
89                  {
90                      btnGrid[i, j].BackColor = default(Color);
91                  }
92              }
93
94              // set the background color of the clicked button to something different.
95              (sender as Button).BackColor = Color.Cornsilk;
96          }
```

33. Next, we need to create the function **updateButtonLabels**(), which will assign a Text property to each button according to the data that is stored in the myBoard object.

```
80
81        public void updateButtonLabels()
82        {
83            for (int r = 0; r < myBoard.Size; r++)
84            {
85                for (int c = 0; c < myBoard.Size; c++)
86                {
87                    btnGrid[r, c].Text = "";
88                    if (myBoard.theGrid[r, c].CurrentlyOccupied) btnGrid[r, c].Text = "Knight";
89                    if (myBoard.theGrid[r, c].LegalNextMove) btnGrid[r, c].Text = "Legal";
90                }
91            }
92        }
93    }
94 }
95
```


We should be able to run the app now and see the legal moves for a knight.
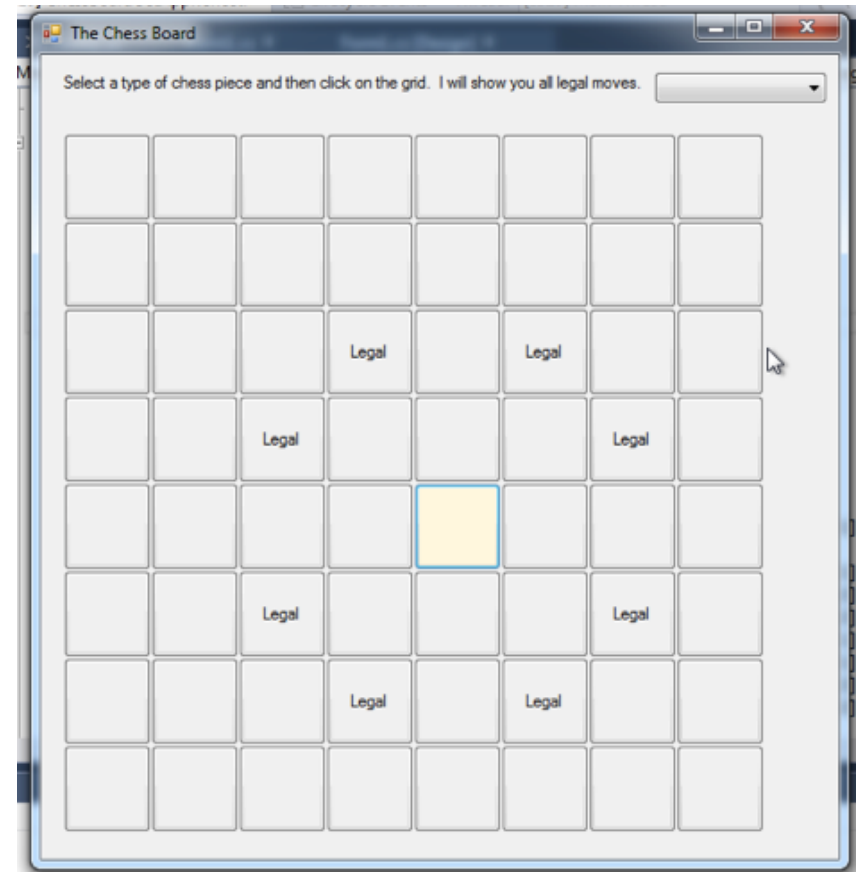
Final Results for Each Piece
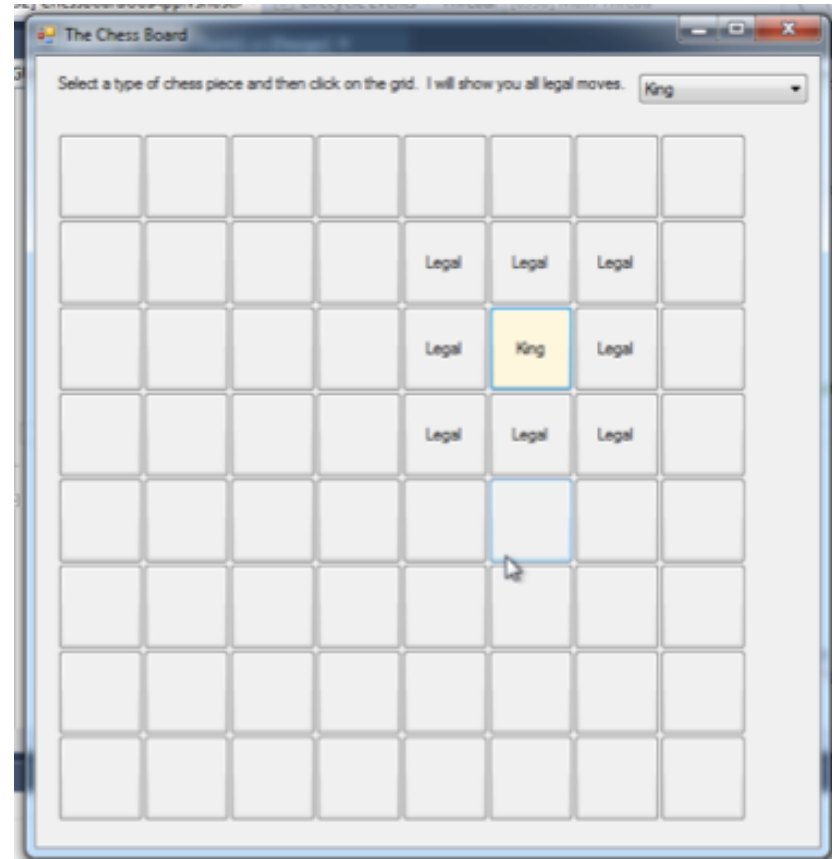
Challenges

Congratulations. Your program should be working (mostly). However, there is some unfinished business.

1.   **Error Checking**

       Double-check to make sure all cases work as designed. Check for out-of-bounds errors.

2.   **Multiple Pieces**

Utilize the ComboBox1 control values to select all five different chess pieces. Currently, the word "Knight" is hard-coded into program in several places. You will have to create a new event handler for the ComboBox1 control, store the selected chess piece name in a variable, and pass this to the function **MarkNextLegalMoves**.

3.　　**Center Button Text**

Currently, the clicked button changes color, but the button text does not show what kind of piece is placed on the board. Return to the **Board.cs** class and look at the **MarkNextLegalMoves** method. You will see that the method updates the **LegalNextMove** for each cell, but nowhere does it set the property value for **CurrentlyOccupied**.

Activity 2.1 Deliverables:

1. Zip file containing all source code.

2. Microsoft Word document containing screenshots of the application being run. Be sure to demonstrate all features that were created in the tutorial, as well as the challenges.

Activity 2.2 Animal Classes

Purpose: Demonstrate the use of polymorphism and inheritance in object-oriented programming design.

Create an Animal Class

1. Create a new Console Application in Visual Studio.

2. Add a new class to the project. Right-click on the project title and choose **Add → Class**
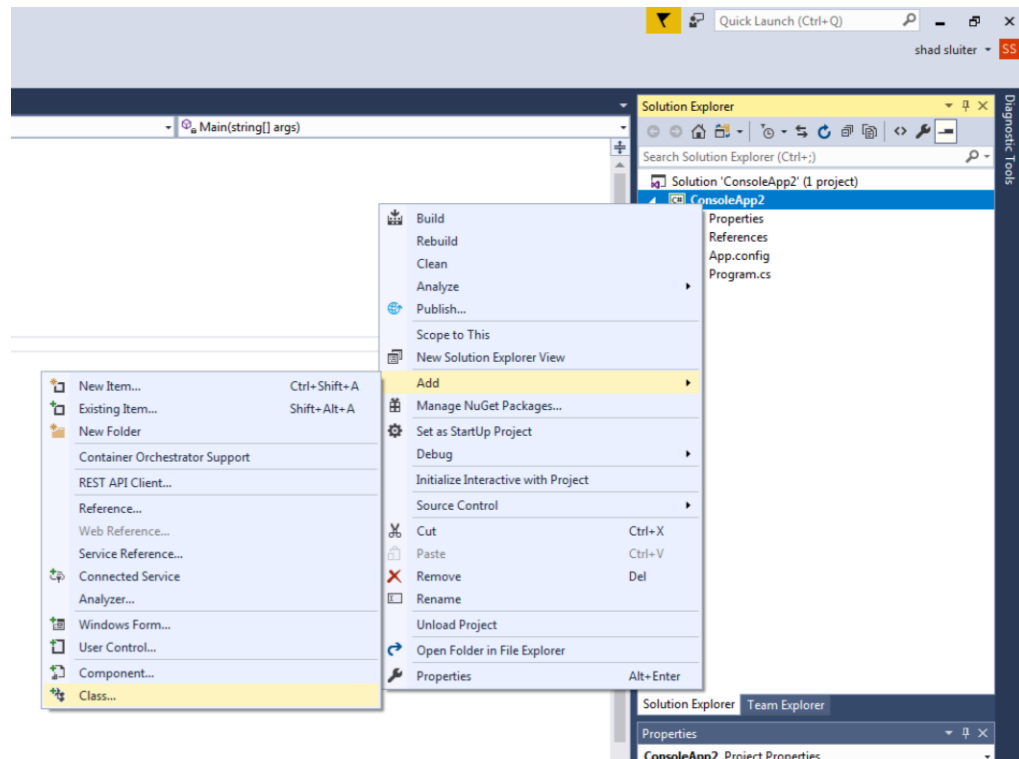
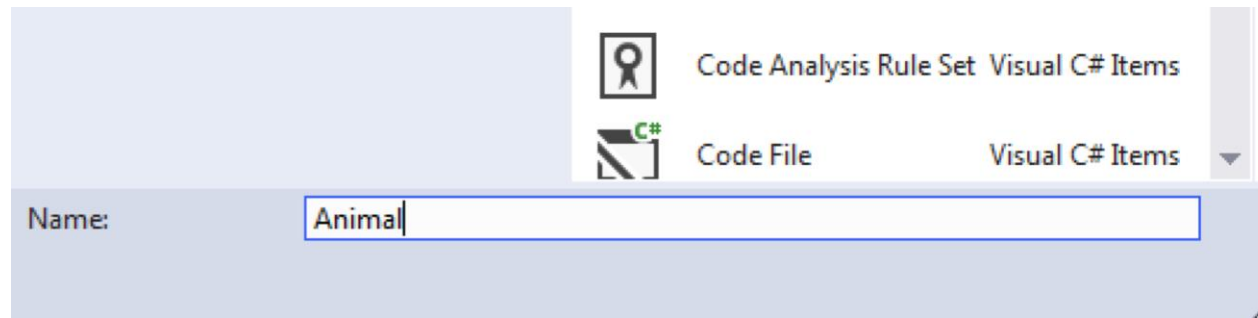3. Name the class **Animal**.
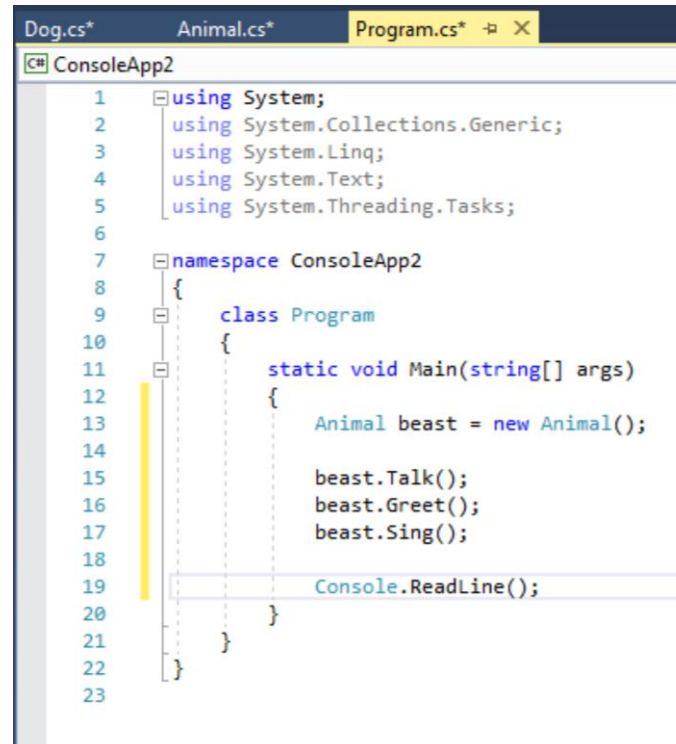
4. Add the following code for the animal class.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp2
{
    class Animal
    {
        public Animal()
        {
            Console.WriteLine("Animal constructor");
        }
        public void Greet()
        {
            Console.WriteLine("Animal says Hello");
        }
        public void Talk()
        {
            Console.WriteLine("Animal talking");
        }
        public virtual void Sing()
        {
            Console.WriteLine("Animal song");
        }
    };
}
```

5. In the Program.cs file, create a new instance of the Animal class. Name it **beast** or something else generic and perform the three methods we have programmed.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Animal beast = new Animal();

            beast.Talk();
            beast.Greet();
            beast.Sing();

            Console.ReadLine();
        }
    }
}
```

6. Run the program. You should see the animal methods print like this:

```
C:\Users\shad sluiter\source\r

Animal constructor
Animal talking
Animal says Hello
Animal song
```

7. Create another class called **Dog**. Add the following code. Notice that Dog **extends** animal. The original animal methods sing, greet, and talk are **inherited** and **overridden**. There is a new method called fetch that is unique to dog. Not all animals will like to play fetch.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp2
{
    class Dog : Animal
    {
        public Dog()
        {
            Console.WriteLine("Dog constructor.  Good puppy.");
        }
        public new void Talk()
        {
            Console.WriteLine("Bark Bark Bark");
        }
        public override void Sing()
        {
            Console.WriteLine("Hooowwwl!");
        }

        public void Fetch(String thing)
        {
            Console.WriteLine("Oh boy.  Here is your " + thing + ".  Let's do it again!");
        }
    };
}
```
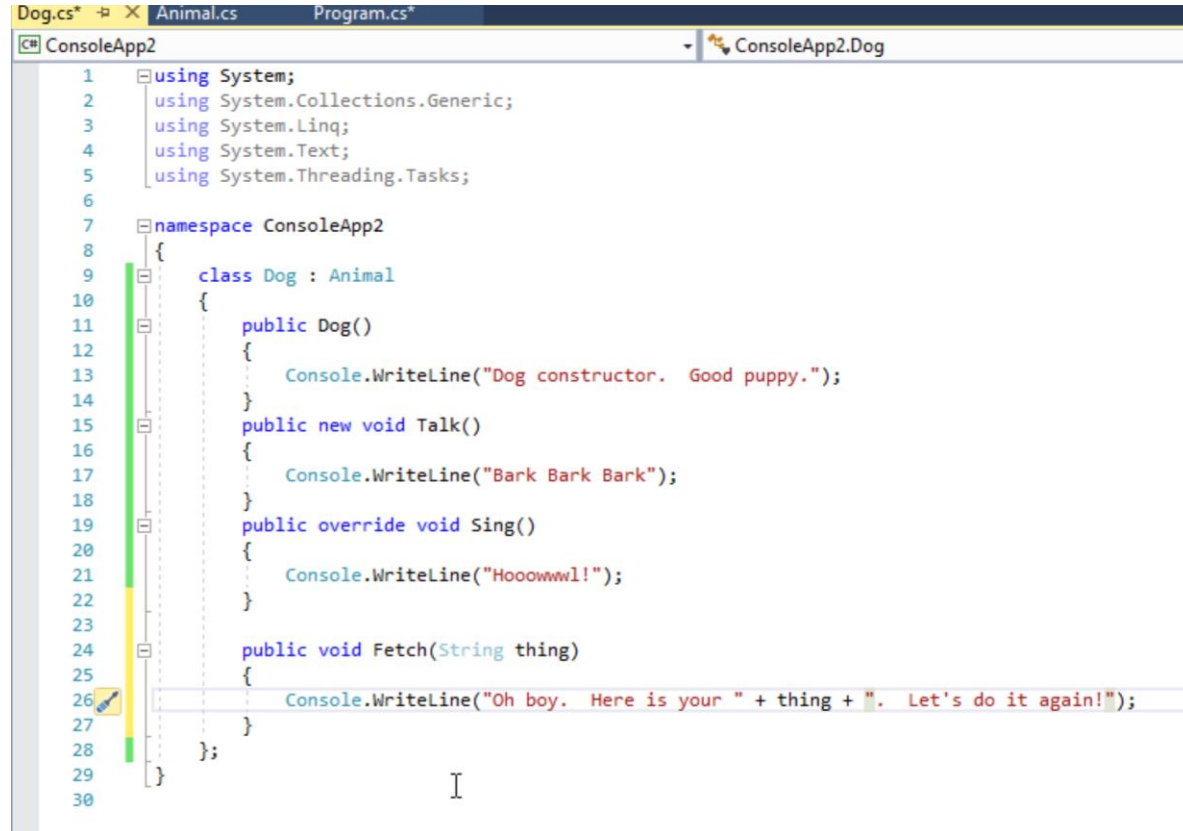
Interfaces

Now let's add some wild animals to the mix.

1. Create a **new Item**. Create an interface and name it **IDomesticated**. Let's say that a domesticated animal can be touched and will accept food from a human. Define two methods called **TouchMe** and **FeedMe**. Notice that in an interface, the methods have no code. An Interface is simply a contract that says, "Any class that implements IDomesticated must have these two methods."

2.   Return to the Dog class and add **IDomesticated** to the list of classes that are implemented.

```
IDomesticated.cs*        Dog.cs*  ⏸ ✕ Animal.cs        Program.cs*
C# ConsoleApp2                                              ▾  🔧 ConsoleApp2.Dog
    1        ⊟using System;
    2         │ using System.Collections.Generic;
    3         │ using System.Linq;
    4         │ using System.Text;
    5        └ using System.Threading.Tasks;
    6
    7        ⊟namespace ConsoleApp2
    8         │ {
    9      ⊟       class Dog : Animal, IDomesticated
   10             {
   11      ⊟         public Dog()        🔧 ▾      •⚲ interface ConsoleApp2.IDomesticated
   12                {
   13                    Console.WriteLine("Dog constructor.   Good puppy.");       'Dog' does not implement interface member 'IDomesticated.TouchMe()'
   14                }                                                              'Dog' does not implement interface member 'IDomesticated.FeedMe()'
   15      ⊟         public new void Talk()
   16                {                                Show potential fixes (Alt+Enter or Ctrl+.)
   17                    Console.WriteLine("Bark Bark Bark");
   18                }
   19      ⊟         public override void Sing()
   20                {
   21                    Console.WriteLine("Hooowwwl!");
   22                }
   23
   24      ⊟         public void Fetch(String thing)
   25                {
   26                    Console.WriteLine("Oh boy.  Here is your " + thing + ".  Let's do it again!");
   27                }
   28             };
   29         }
   30
```
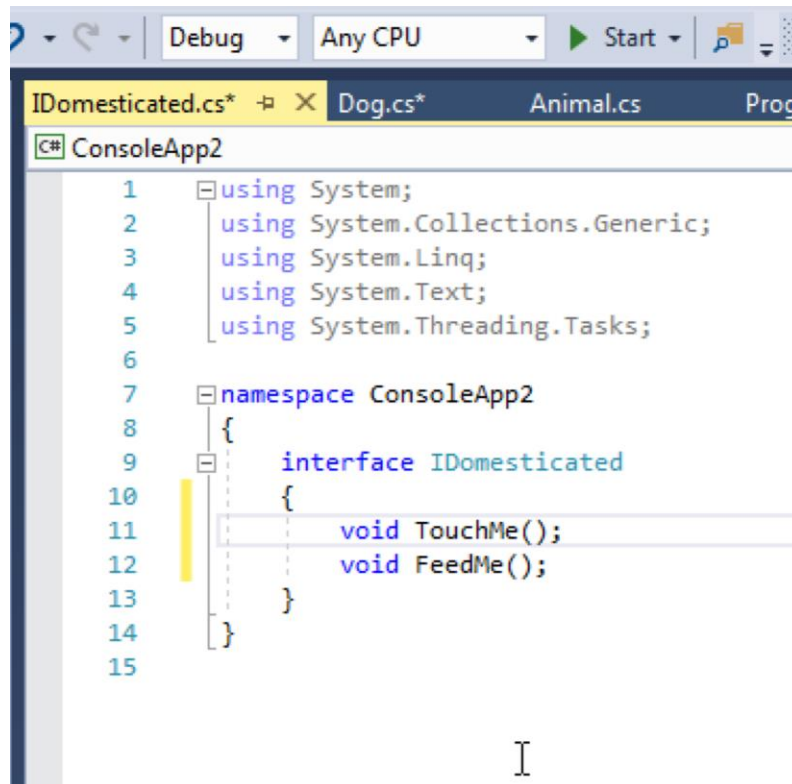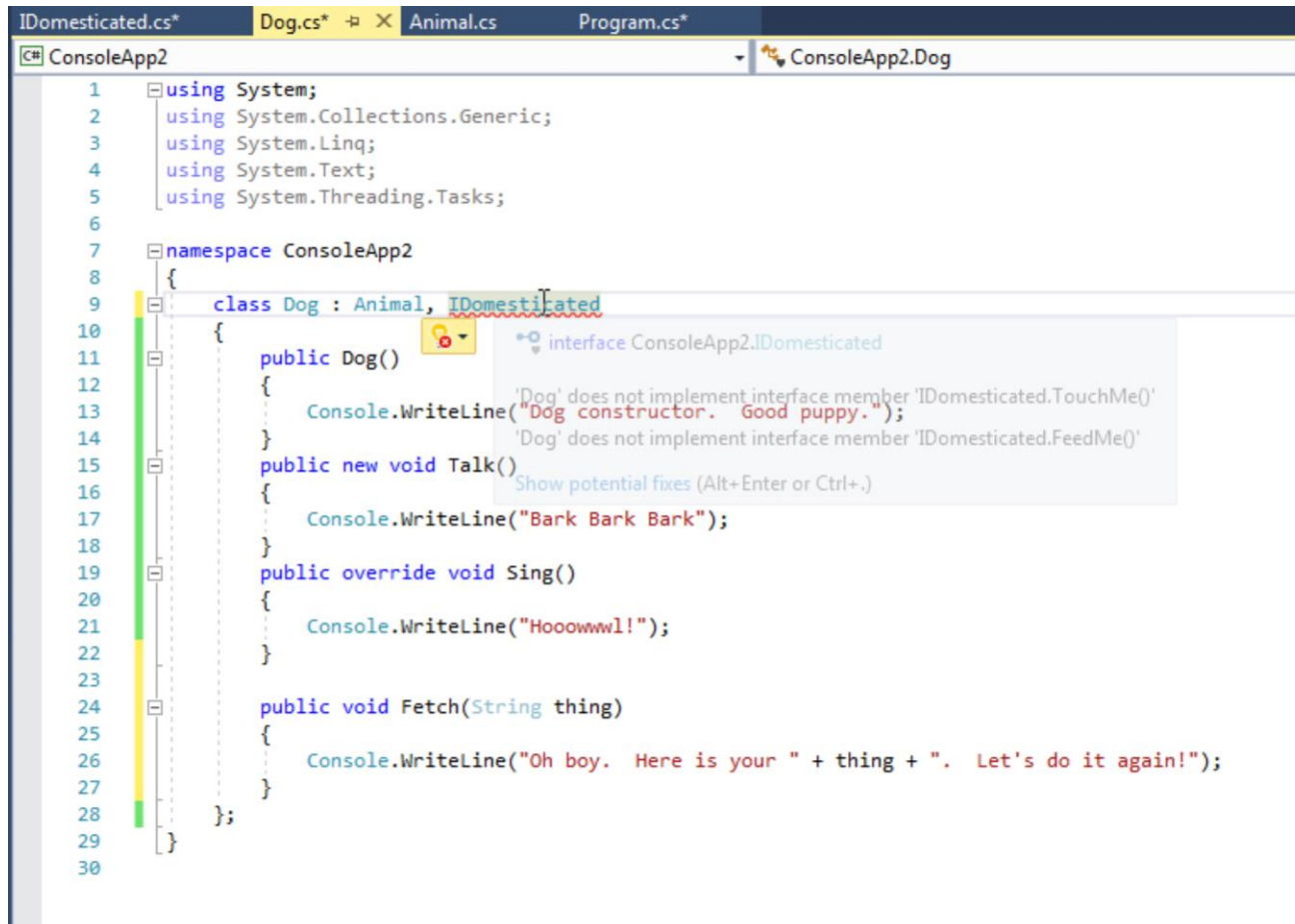
3. You should see a new warning after adding the new implementation. If you select "Show Potential Fixes," you can automatically add new methods to the class.

```
ConsoleApp2                                    ConsoleApp2.Dog                           Fetch(string thing)
 1   using System;
 2   using System.Collections.Generic;
 3   using System.Linq;
 4   using System.Text;
 5   using System.Threading.Tasks;
 6
 7   namespace ConsoleApp2
 8   {
 9       class Dog : Animal, IDomesticated
10       {
11           public Dog()          Implement interface          ▸    ❌ CS0535 'Dog' does not implement interface member
12           {                     Implement interface explicitly     'IDomesticated.TouchMe()'
13               Console.WriteLine("Dog cons                      ...
14           }                                                       }
15           public new void Talk()
16           {                                                   public void TouchMe()
17               Console.WriteLine("Bark Bar                     {
18           }                                                       throw new NotImplementedException();
19           public override void Sing()                         }
20           {
21               Console.WriteLine("Hooowwwl                     public void FeedMe()
22           }                                                   {
23                                                                   throw new NotImplementedException();
24           public void Fetch(String thing)                     }
25           {                                                   };
26               Console.WriteLine("Oh boy.  Here is your " + thing + ".  Let's do it again!");
27           }                     Preview changes
28       };                        Fix all occurrences in: Document | Project | Solution
29   }
30
```

4. Modify the new methods to do something appropriate for a Dog class.

```csharp
namespace ConsoleApp2
{
    class Dog : Animal, IDomesticated
    {
        public Dog()
        {
            Console.WriteLine("Dog constructor.  Good puppy.");
        }
        public new void Talk()
        {
            Console.WriteLine("Bark Bark Bark");
        }
        public override void Sing()
        {
            Console.WriteLine("Hooowwwl!");
        }

        public void Fetch(String thing)
        {
            Console.WriteLine("Oh boy.  Here is your " + thing + ".  Let's do it again!");
        }

        public void TouchMe()
        {
            Console.WriteLine("Please scratch behind my ears.");
        }

        public void FeedMe()
        {
            Console.WriteLine("It's suppertime. The very best time of day!!!");
        }
    };
}
```

5. Let's create an instance of Dog. I named my dog "bowser" after a family pet. Notice that bowser can do all of the methods that were included in the animal class (greet, talk, sing), as well as what is unique to the Dog class (fetch), and finally, the two methods that are common to IDomesticated (feedMe and touchMe). Since the Greet method was not overridden in the Dog class, bowser uses the "Animal says hello" method that was inherited from Animal.

6. Let's add a nondomesticated animal to the program. Create a **new class** called **Robin** and extend Animal. In this case, I only choose to override the Sing method as shown (right).

```csharp
Robin.cs*  ⊕ ✕  IDomesticated.cs      Dog.cs        Animal.cs        Program.cs
C# ConsoleApp2                                          ▾  ⚡ ConsoleApp2.Robin
1    using System;
2    using System.Collections.Generic;
3    using System.Linq;
4    using System.Text;
5    using System.Threading.Tasks;
6
7    namespace ConsoleApp2
8    {
9        class Robin : Animal
10       {
11           public virtual void Sing()
12           {
13               Console.WriteLine("Chirp Chirp");
14           }
15       }
16   }
17
```

```csharp
6
7    namespace ConsoleApp2
8    {
9        class Program
10       {
11           static void Main(string[] args)
12           {
13               Animal beast = new Animal();
14
15               beast.Talk();
16               beast.Greet();
17               beast.Sing();
18
19               Dog bowser = new Dog();
20
21               bowser.Talk();
22               bowser.Greet();
23               bowser.Sing();
24               bowser.Fetch("stick");
25               bowser.FeedMe();
26               bowser.TouchMe();
27
28               Robin red = new Robin();
29
30               red.Talk();
31               red.Greet();
32               red.Sing();
33               red.Fetch("worm");
34               red.FeedMe();
35               red.TouchMe();
36
37               Console.ReadLine();
38           }
39       }
100 %  ▾
```
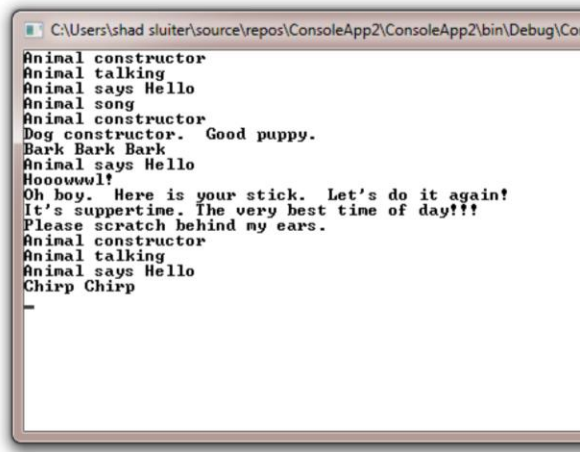
7. Create a new instance of Robin in the Main( ) method of the Program.cs file. Notice that I cannot use the methods from Dog or from IDomesticated. The methods TouchMe, Fetch, and FeedMe are not part of the inheritance of Robin (left).

8. If I comment out the lines with errors, the program will run. Notice that Robin uses the Animal methods for the constructor, talk and greet. Only the sing method was overridden in the Robin class.

```
7   namespace ConsoleApp2
8   {
9       class Program
10      {
11          static void Main(string[] args)
12          {
13              Animal beast = new Animal();
14
15              beast.Talk();
16              beast.Greet();
17              beast.Sing();
18
19              Dog bowser = new Dog();
20
21              bowser.Talk();
22              bowser.Greet();
23              bowser.Sing();
24              bowser.Fetch("stick");
25              bowser.FeedMe();
26              bowser.TouchMe();
27
28              Robin red = new Robin();
29
30              red.Talk();
31              red.Greet();
32              red.Sing();
33              //red.Fetch("worm");
34              //red.FeedMe();
35              //red.TouchMe();
36
37              Console.ReadLine();
38          }
```

```
C:\Users\shad sluiter\source\repos\ConsoleApp2\ConsoleApp2\bin\Debug\Co
Animal constructor
Animal talking
Animal says Hello
Animal song
Animal constructor
Dog constructor.   Good puppy.
Bark Bark Bark
Animal says Hello
Hooowwwl!
Oh boy.   Here is your stick.   Let's do it again!
It's suppertime. The very best time of day!!!
Please scratch behind my ears.
Animal constructor
Animal talking
Animal says Hello
Chirp Chirp
```
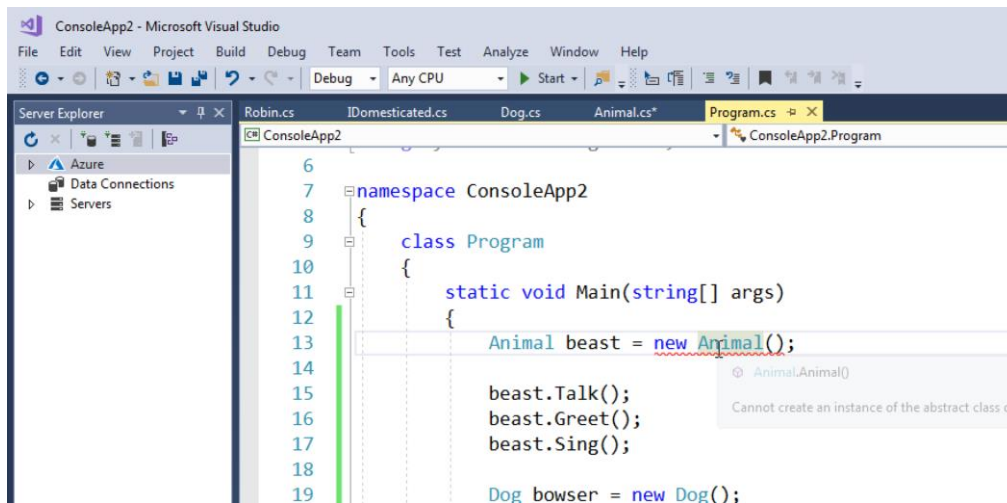
Abstract Classes

Now that we have several specific types of animals, let's eliminate the ability to create a generic "beast." In the real world, "animal" is an abstract term. Every animal has a specific type and classification. If we change the Animal class to abstract, we will not be able create an instance of animal. In the code below, the only change is to line 9.

Notice that in Main ( ) we now get an error message. Since Animal is an abstract class, we are not allowed to create an instance called "beast" anymore.
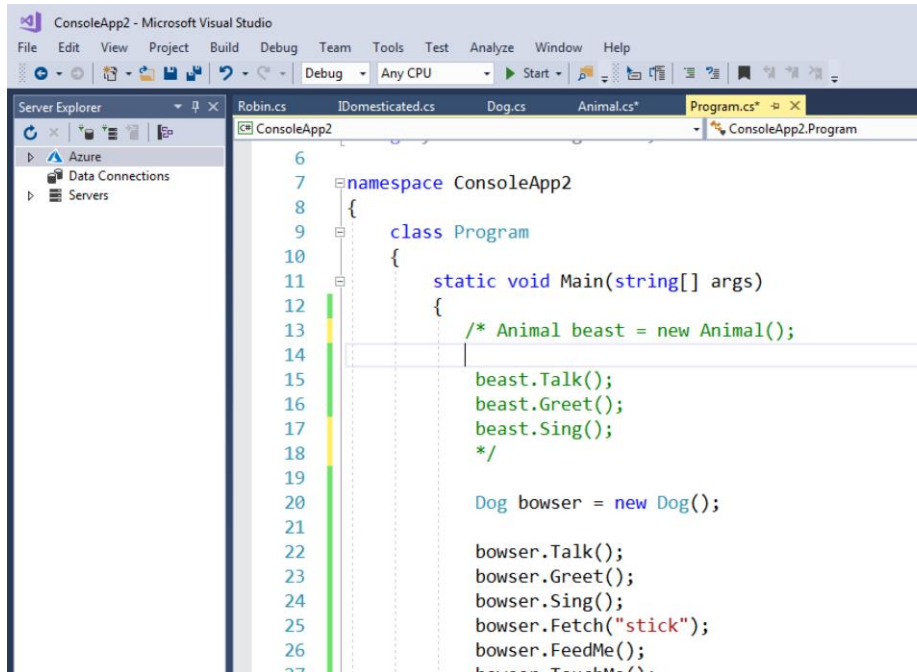
```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp2
{
    abstract class Animal
    {
        public Animal()
        {
            Console.WriteLine("Animal constructor");
        }
        public void Greet()
        {
            Console.WriteLine("Animal says Hello");
        }
        public void Talk()
        {
            Console.WriteLine("Animal talking");
        }
        public virtual void Sing()
        {
            Console.WriteLine("Animal song");
        }
    };
}
```

```
ConsoleApp2 - Microsoft Visual Studio
File   Edit   View   Project   Build   Debug   Team   Tools   Test   Analyze   Window   Help
```

```csharp
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            Animal beast = new Animal();

            beast.Talk();
            beast.Greet();
            beast.Sing();

            Dog bowser = new Dog();
```

Cannot create an instance of the abstract class or...

Comment out the beast section.

Challenge

Demonstrate your understanding of abstract, interfaces, inheritance, and overriding by completing the following challenges:
1. Create two new Interface classes appropriate for animals. Some examples for specialty methods include IRidable (horses, donkeys, etc.), IMilkable (cows, goats), IFlyable, IPreditor, INocturnal, ISwimmable, etc.
2. In the new interface, define at least one new method to make this interface different from the generic animal class.
3. Create three new types of animal class that inherit from the Animal class, as well one or more of the interfaces you defined. You may override some of the Animal class methods (greet, talk, or sing).
4. Demonstrate new instances of the animals in the main() method of Program.cs

Activity 2.2 Deliverables:
1. Submit a ZIP file of the project folder's source code.
2. Submit a text file with a URL to a video that demonstrates the application being run. Show the code in the video and explain what your new animal classes and interfaces do.