4 Solitaire

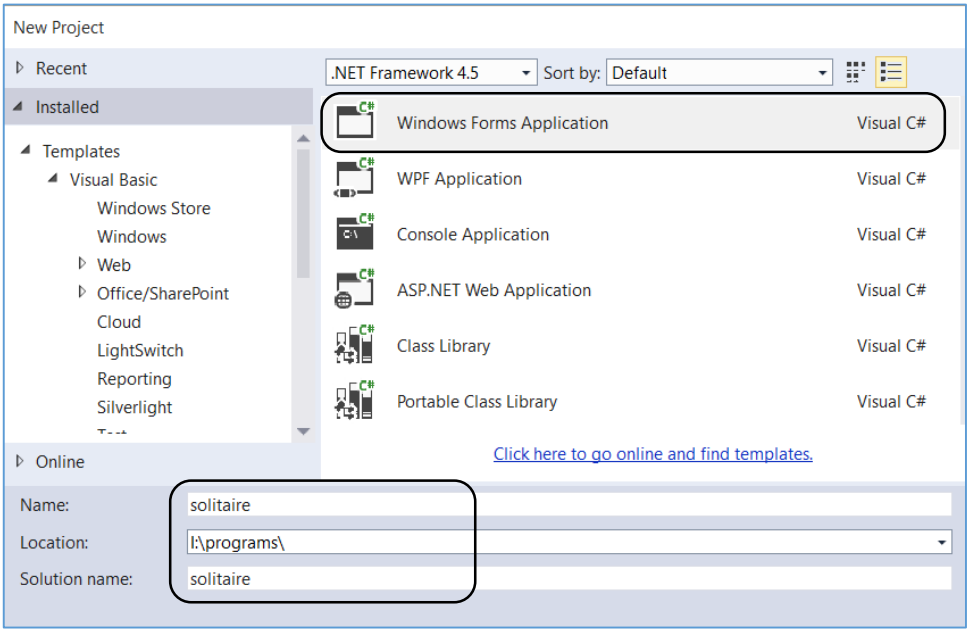
For our next application, we will produce an on-screen version of the game Solitaire.

Solitaire is a game for one player. The board consists of a cross-shaped array of holes, into which pegs or other playing pieces are inserted. At the start of the game, all holes are occupied except for the one central position.

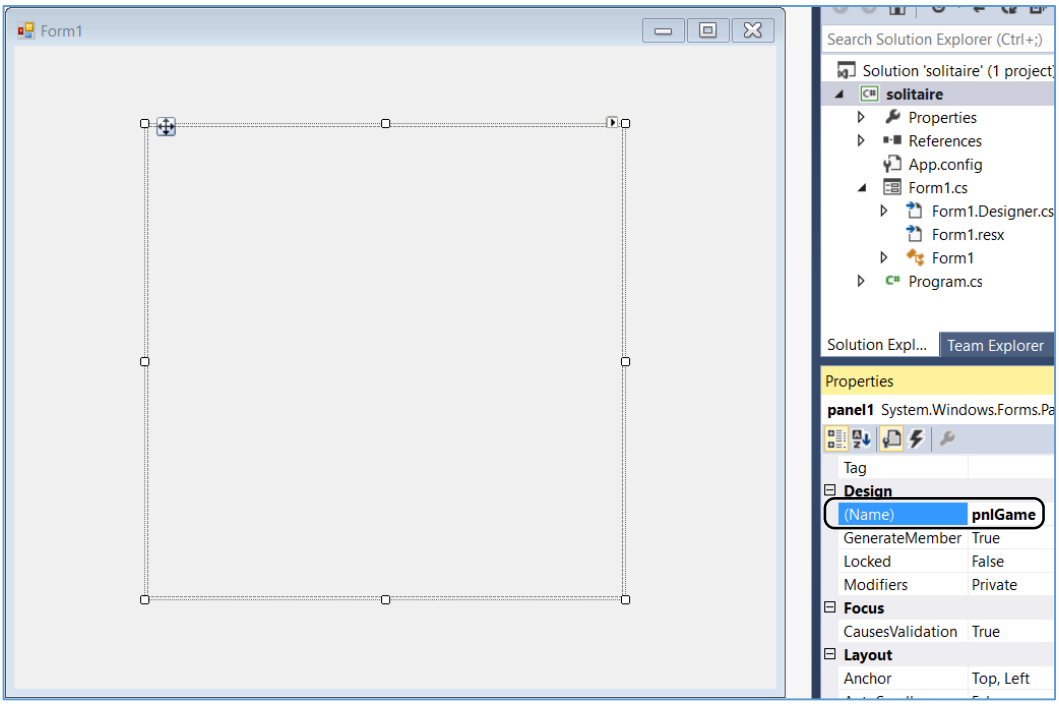
The objective of the game is to remove all but one of the playing pieces, leaving this single piece in the central hole.

A move is made by jumping a playing piece over the top of another piece, to land in a hole. The jumped piece is removed from the board. Moves may occur along the horizontal or vertical lines of the grid, but not diagonally.

In this project, we will produce an interactive graphical program to allow the user to play the game of Solitaire on screen using the mouse. The program should remove jumped pieces from the board automatically, but should only allow valid moves to be made. Begin by setting up a Windows Forms Application. Give this the name '***solitaire***':



We will begin by putting a ***Panel*** component onto Form1. Drag the mouse to produce an outline for the playing area. Give the Panel the name '***pnlGame***'



Click-right on Form1 and select '***View Code***'.

Create an empty method called ***initialiseBoard( )***. We will add code to this later to display the playing pieces.

Add a line to the ***Form1( )*** method to call ***initialiseBoard( )*** when the program starts:

```
namespace solitaire
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
            initialiseBoard();
        }

        private void initialiseBoard()
        {

        }
    }
}
```

A simple way to make the interactive screen for the game is to create a pattern of Button components to represent the grid of playing pieces and holes.

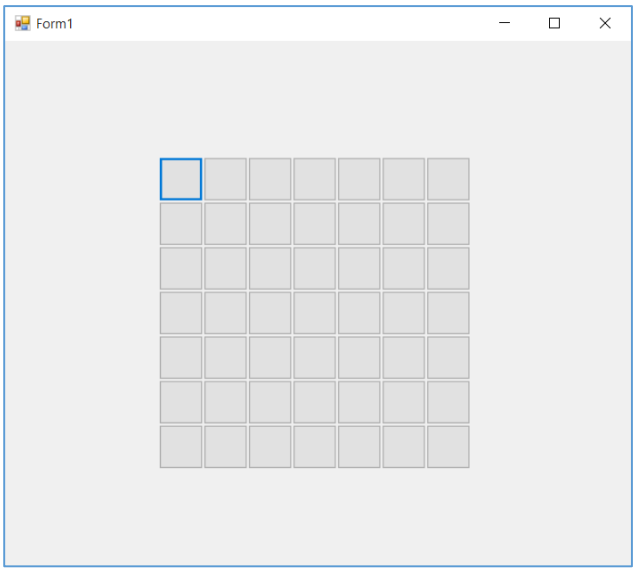
Rather than adding each button by hand, which would take a lot of time and would be inaccurate, we can write program code to set up the buttons automatically when the program runs.

Begin by creating a ***two dimensional array*** of ***Button*** components by adding the line of code above the ***initialiseBoard( )*** method.

Within ***initialiseBoard( )*** we can use two nested loops to produce a rectangular grid of 7 rows and 7 columns of buttons. Each button will have a width and height of 40 pixels, and we change the top left corner position of each button so that it fits correctly into the grid pattern.

```
public Form1()
{
    InitializeComponent();
    initialiseBoard();
}

Button[,] btnGame = new Button[8, 8];

private void initialiseBoard()
{
    for (int i = 1; i <= 7; i++)
    {
        for (int j = 1; j <= 7; j++)
        {
            btnGame[i, j] = new Button();
            btnGame[i, j].Width = 40;
            btnGame[i, j].Height = 40;
            btnGame[i, j].Left = (40 * i);
            btnGame[i, j].Top = (40 * j);
            pnlGame.Controls.Add(btnGame[i, j]);
        }
    }
}
```

Run the program, and a grid of buttons should be created:



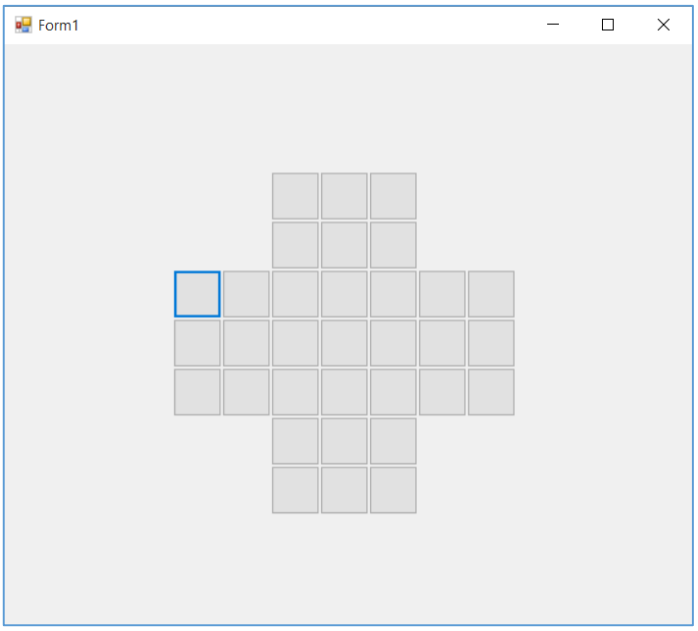
This is a good start, but the playing area is actually in the shape of a cross, with the four corner areas omitted. We can modify the ***initialiseBoard( )*** method to allow for this.

Set up a ***Boolean*** variable called '***present***'. For any grid position, we will set 'present' to ***TRUE*** if a button is required, but ***FALSE*** if that position is to be left blank.

Add the nested ***IF..*** conditions which will set 'present' to FALSE for each of the corner areas of the cross.

```
private void initialiseBoard()
{
    bool present;

    for (int i = 1; i <= 7; i++)
    {
        for (int j = 1; j <= 7; j++)
        {
            present = true;
            if (i <= 2 || i >= 6)
            {
                if (j <= 2 || j >= 6)
                {
                    present = false;
                }
            }

            if (present == true)
            {
                btnGame[i, j] = new Button();
                btnGame[i, j].Width = 40;
                btnGame[i, j].Height = 40;
                btnGame[i, j].Left = (40 * i);
                btnGame[i, j].Top = (40 * j);
                pnlGame.Controls.Add(btnGame[i, j]);
            }
        }
    }
}
```

Run the program, and we should now have the correct pattern of buttons for the playing area:



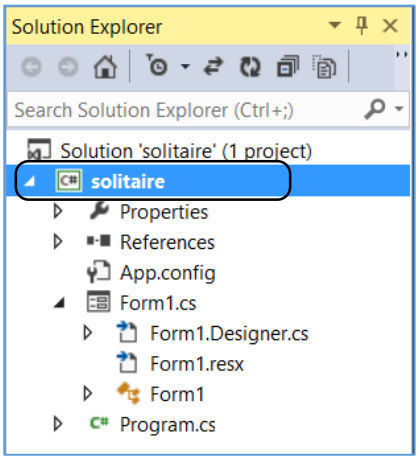
The next step is to make the board look more realistic by adding graphics to represent the playing pieces and holes. Two small graphics images will be needed, which can be displayed on the buttons as appropriate:

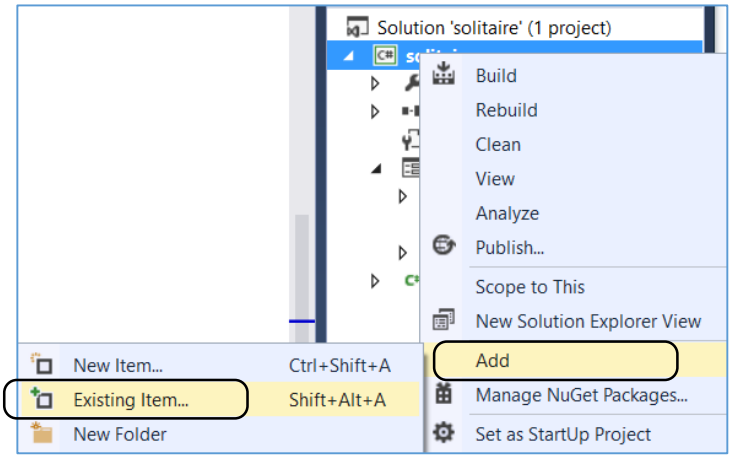
peg.png

hole.png

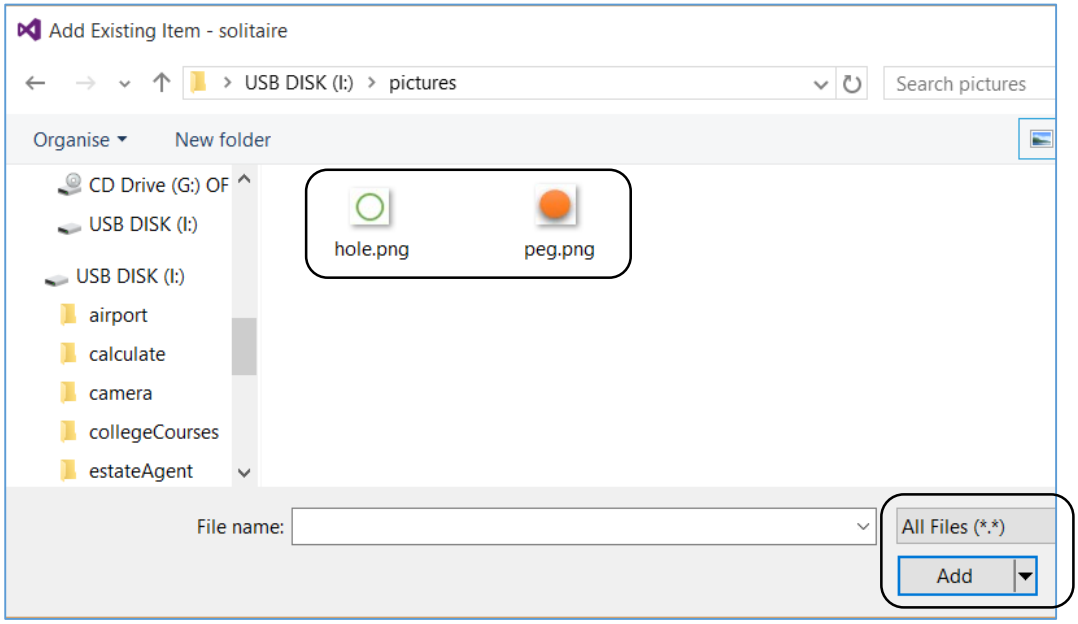
Create or copy these images, and store them somewhere on your computer.

We will now import the images into the C# project. To do this, go to the Solution Explorer window and right-click on the '***solitaire***' program icon. Select '***Add*** / ***Existing Item***'.





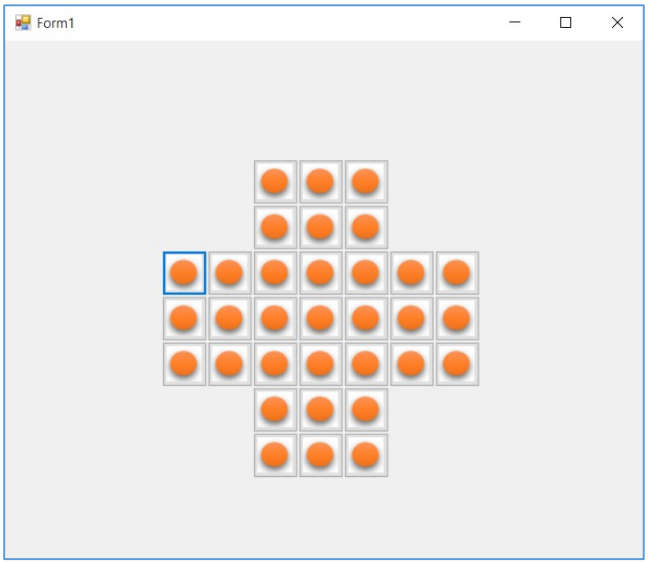
Set the file type to '***All Files***' and navigate to where the graphics images are saved. Click '***Add***' for each image file:



We can now return to the ***initialiseBoard( )*** method, and add a line of code to display the '***peg.png***' image on each button:

```
    if (present == true)
    {
        btnGame[i, j] = new Button();
        btnGame[i, j].Width = 40;
        btnGame[i, j].Height = 40;
        btnGame[i, j].Left = (40 * i);
        btnGame[i, j].Top = (40 * j);

        btnGame[i, j].Image = Image.FromFile("../../peg.png");

        pnlGame.Controls.Add(btnGame[i, j]);
    }
```

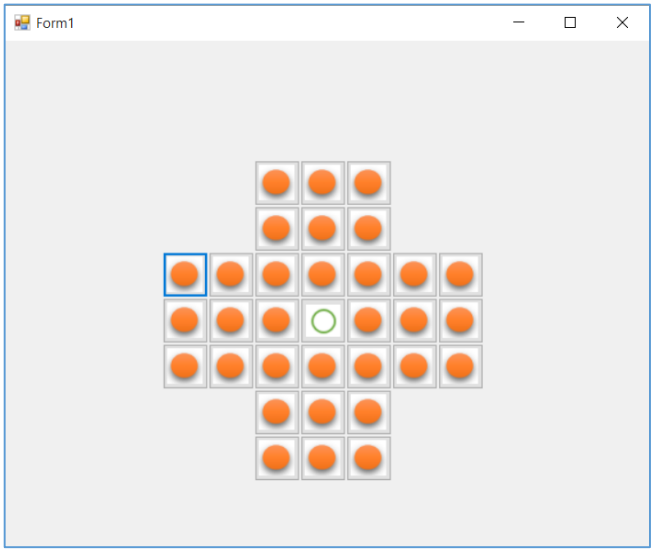
Run the program, and the pattern of playing pieces should appear:



This is almost correct, but we need to begin the game with an empty hole at the centre of the board. Add code to the ***initialiseBoard( )*** method to treat this as a special case:

```
if (present == true)
 {
     btnGame[i, j] = new Button();
     btnGame[i, j].Width = 40;
     btnGame[i, j].Height = 40;
     btnGame[i, j].Left = (40 * i);
     btnGame[i, j].Top = (40 * j);

     btnGame[i, j].Image = Image.FromFile("../../peg.png");

     if (i == 4 && j == 4)
     {
         btnGame[i, j].Image = Image.FromFile("../../hole.png");
     }

     pnlGame.Controls.Add(btnGame[i, j]);
 }
```

Run the program and check that the pattern is now correct:



The next stage is to play the game, but before that we must provide the program with a way of recording the positions of the playing pieces and the empty holes. This will be necessary if the computer is going to check for valid moves.

A simple method is to use a two dimensional array of code numbers which correspond to the buttons on the game board. For each button displaying a '***peg***' , the equivalent ***code value will be 1***. For each button displaying a '***hole***', the equivalent ***code value will be 0***.

Start by setting up the array:

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        initialiseBoard();
    }

    Button[,] btnGame = new Button[8, 8];

    int[,] play = new int[8, 8];
```

Go to the ***initialiseBoard( )*** method.

At the point where a button is set to display the '***peg***' image, we will set the corresponding value in the '***play***' array to ***1***.

At the point where the central button is set to display the '***hole***' image, we will set the corresponding value in the '***play***' array to ***0***:

```
         if (present == true)

        {
            btnGame[i, j] = new Button();
            btnGame[i, j].Width = 40;
            btnGame[i, j].Height = 40;
            btnGame[i, j].Left = (40 * i);
            btnGame[i, j].Top = (40 * j);

            btnGame[i, j].Image = Image.FromFile("../../peg.png");
            play[i, j] = 1;

            if (i == 4 && j == 4)
            {
                btnGame[i, j].Image = Image.FromFile("../../hole.png");
                play[i, j] = 0;
            }

            pnlGame.Controls.Add(btnGame[i, j]);
        }
```

We can now consider the way that the user will play the game, by clicking on the piece which they wish to move, then clicking on the hole to which it should be moved. For this to work, the program must be able to:

- identify which buttons have been clicked, and
- have a method which will process the move.

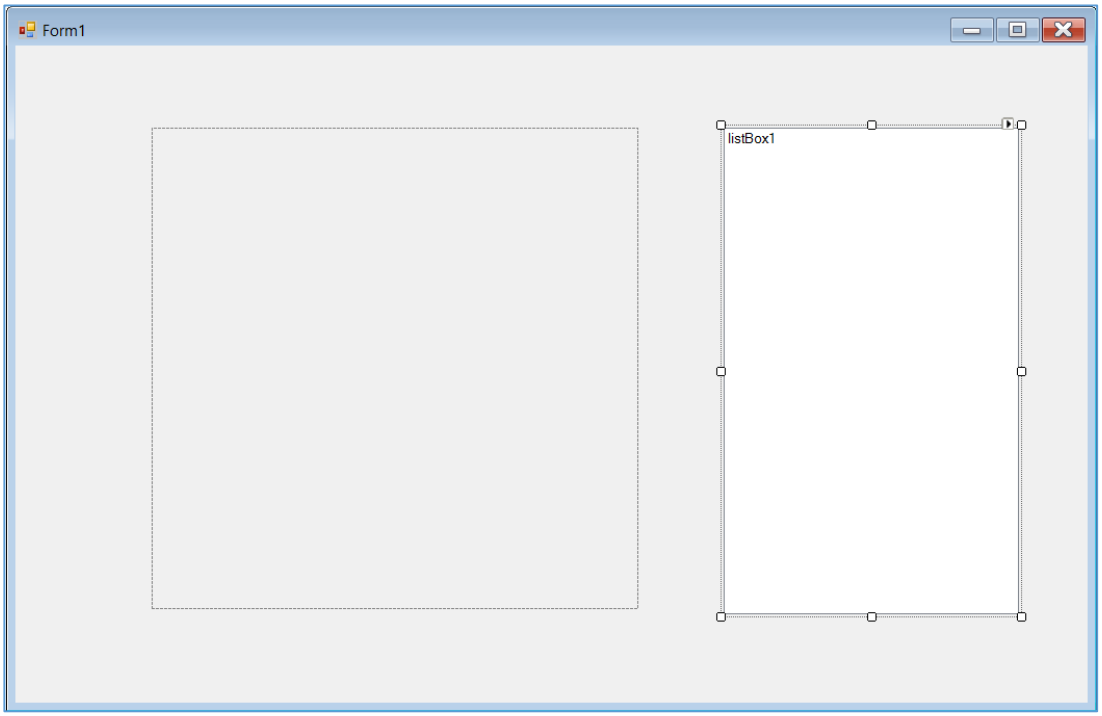
We will add some code to do these things…

We can allocate names to each button, made up from its column and row position. For example, the button in the middle of the top row will be: '***btn41***', and the button in the centre of the board will be '***btn44***'. The button names are created using the loop counter values **i** and **j**

For the button to respond when it is clicked, we need to create an event handler, and set up an empty button-click method:

```
                if (i == 4 && j == 4)
                {
                    btnGame[i, j].Image = Image.FromFile("../../hole.png");
                    play[i, j] = 0;
                }

                String buttonName = "btn" + i + j;
                btnGame[i, j].Name = buttonName;
                btnGame[i, j].Click += new EventHandler(game_Click);

                pnlGame.Controls.Add(btnGame[i, j]);
            }
        }
    }
}


private void game_Click(object sender, EventArgs e)
{

}
```

We can now think about the game play. This is quite complex, so we need a way of checking that the program is working correctly. For testing purposes, add a ***ListBox*** component at the side of the playing grid. This will be removed later when we are sure that the processing is correct.

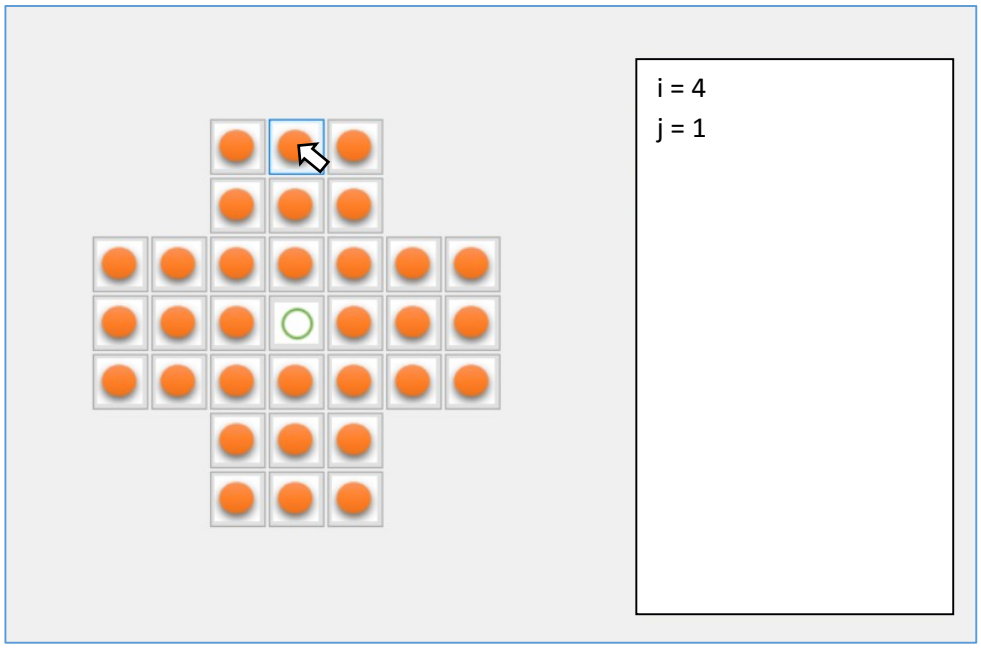


It is important that the program correctly identifies the button that is clicked.

Add code to the ***game_Click( )*** method which will check the name of the button and use this to determine its column and row number. We will output these values to the ***ListBox***, to ensure that they are correct.

```
private void game_Click(object sender, EventArgs e)
{
    Button clickedButton = (Button)sender;

    string s = clickedButton.Name;
    int Ipos = Convert.ToInt16(s.Substring(3, 1));
    int Jpos = Convert.ToInt16(s.Substring(4, 1));

    listBox1.Items.Add("i = " + Ipos);
    listBox1.Items.Add("j = " + Jpos);

}
```

Run the program. Click on different buttons, and check that the correct column and row values are given in each case:



Go to the start of the program and add variables which we will use to record the button positions clicked during a move.

```
public Form1()
{
    InitializeComponent();
    initialiseBoard();
}

Button[,] btnGame = new Button[8, 8];
int[,] play = new int[8, 8];

int startI = 0, startJ = 0, finishI = 0, finishJ = 0;
```

Two buttons must be clicked by the player in order to make a move – firstly on the ***piece they wish to move***, and then on the ***hole to which it will move***. The following lines of code record the column and row positions of each of the button clicks:

- We begin with the start position undefined, with a column value of zero. The first button_click coordinates are assigned to the '***start***' variables.
- Once the start position has been found, the next button_click coordinates are assigned to the '***finish***' variables.
- The move made by the player can then be checked against the rules of the game, and the move carried out if it is valid.
- The '***start***' column variable is then set back to zero, ready to receive the pair of button clicks during the next move.

Add the lines of code to carry out these actions, and set up an empty method ready for the code which will check and carry out the moves.

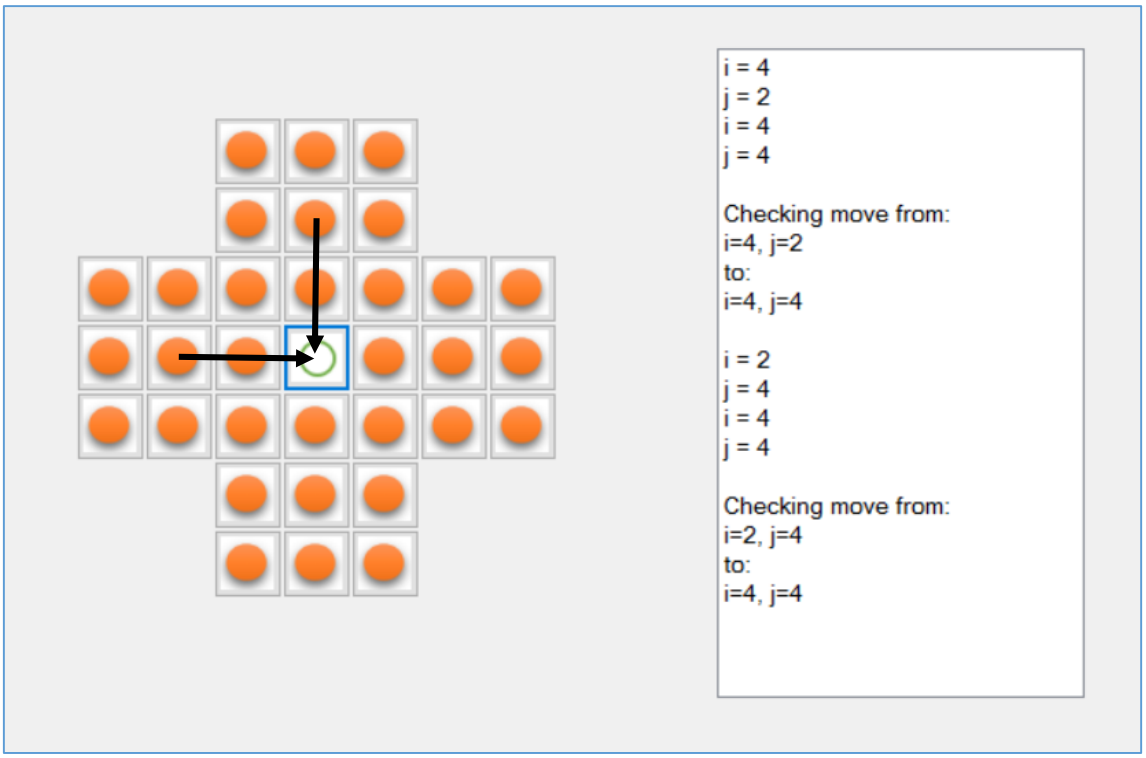
```
        listBox1.Items.Add("i = " + Ipos);
        listBox1.Items.Add("j = " + Jpos);

        if (startI == 0)
        {
            startI = Ipos;
            startJ = Jpos;
        }
        else
        {
            finishI = Ipos;
            finishJ = Jpos;
            checkMove(startI,startJ,finishI,finishJ);
            startI = 0;
        }
    }

    private void checkMove(int startI,int startJ,int finishI,int finishJ)
    {

    }
```

Once the program enters the ***checkMove( )*** method, it should know the start and finish positions for the proposed move. Let's begin by checking that these positions are being identified correctly from the mouse clicks.

Add lines of code to the ***checkMove( )*** method which will output the start and finish positions to the list box:

```
    private void checkMove(int startI,int startJ,int finishI,int finishJ)
    {
        listBox1.Items.Add("Checking move from: ");

        listBox1.Items.Add("i=" + startI + ", j=" + startJ);

        listBox1.Items.Add("to: ");
        listBox1.Items.Add("i=" + finishI + ", j=" + finishJ);

        listBox1.Items.Add("");
    }
```

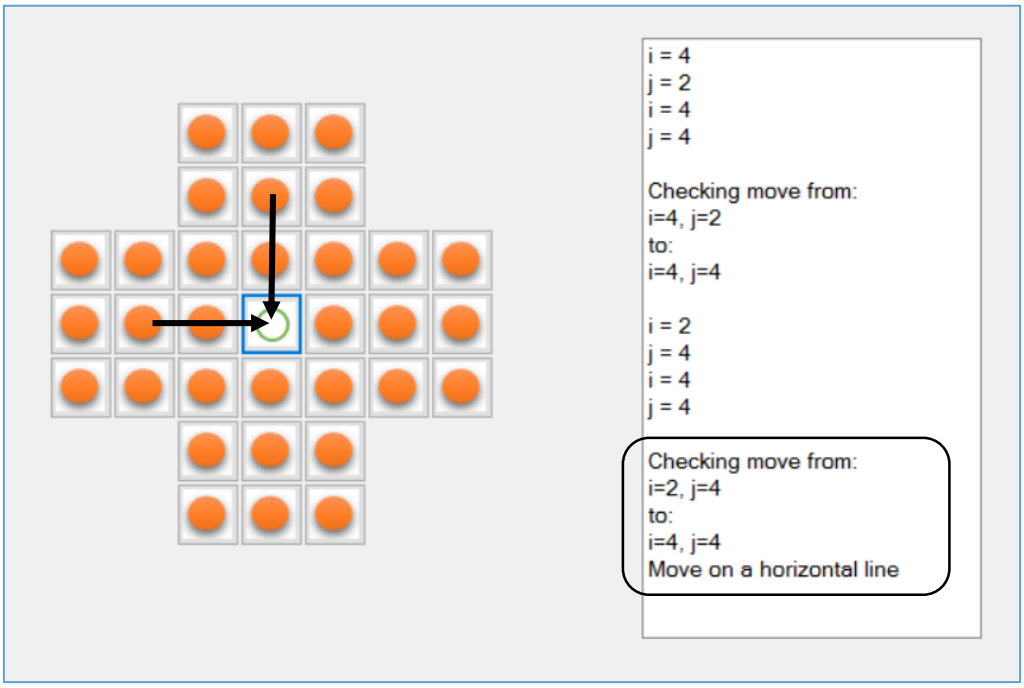
Run the program. Click on pairs of buttons, and verify that the correct column and row positions are being recorded in the list box:



Moves are only valid if the start and finish positions are either on the ***same horizontal row***, or in the ***same vertical column*** of the grid. We can easily check for a horizontal move, by checking that the start and finish j positions are the same. Add a ListBox output line to show this:

```
private void checkMove(int startI,int startJ,int finishI,int finishJ)
{
    listBox1.Items.Add("");

    listBox1.Items.Add("Checking move from: ");
    listBox1.Items.Add("i=" + startI + ", j=" + startJ);

    listBox1.Items.Add("to: ");
    listBox1.Items.Add("i=" + finishI + ", j=" + finishJ);

    if (startJ == finishJ)
    {
        listBox1.Items.Add("Move on a horizontal line");
    }

    listBox1.Items.Add("");
}
```

Run the program, and check that a horizontal move can be detected correctly:

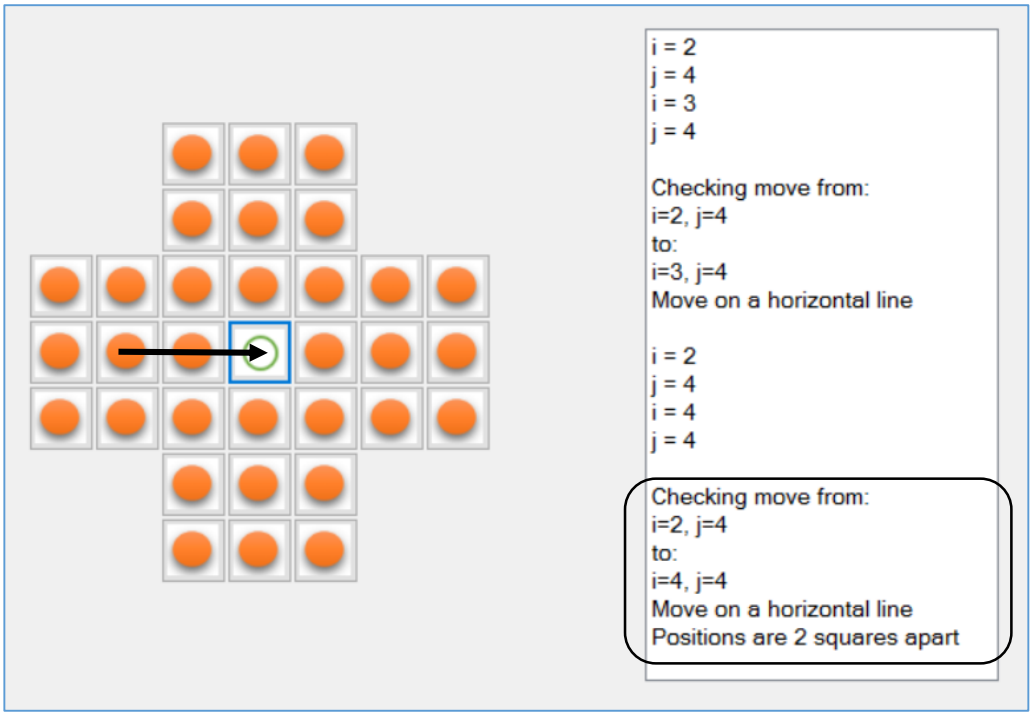


The next requirement for a valid move is that the ***start*** and ***finish*** positions must be ***two squares apart***, for example: moving along a row from column 2 to column 4, or in the opposite direction from column 4 to column 2.

We can check this by subtracting the finish column from the start column, and checking that the answer is 2 when ***any minus sign is ignored***. This can be done using the ***ABSOLUTE*** function, which shows any number, positive or negative, as its positive equivalent:

```
if (startJ == finishJ)
{
    listBox1.Items.Add("Move on a horizontal line");
    if (Math.Abs(startI - finishI) == 2)
    {
        listBox1.Items.Add("Positions are 2 squares apart");
    }
}
```

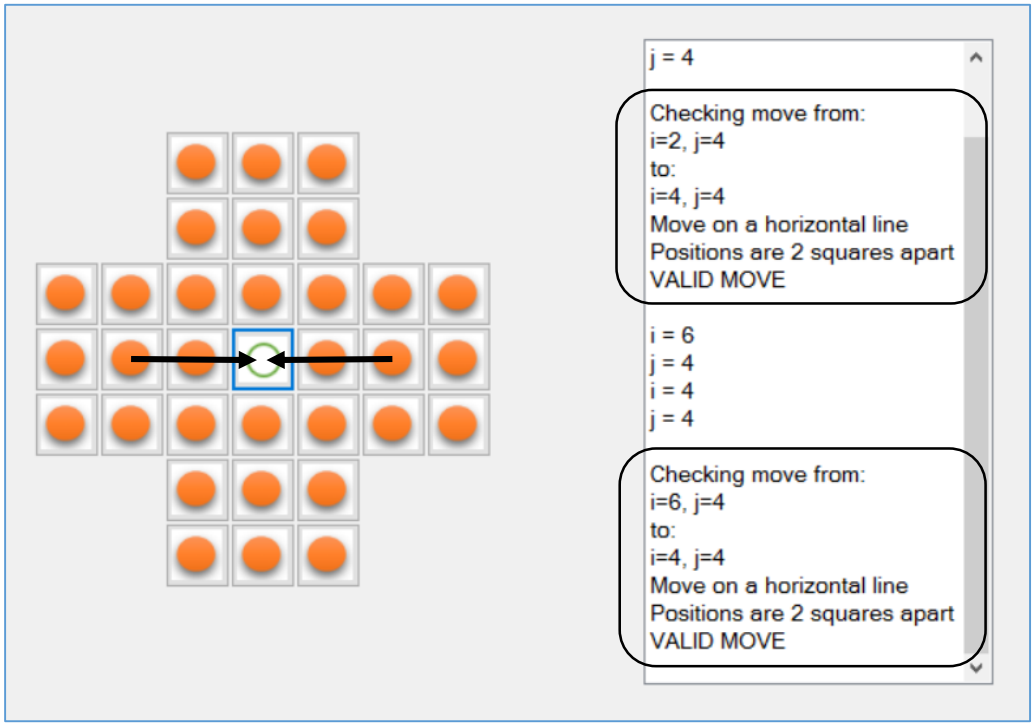
Run the program and check that the computer can detect moves between positions which are two squares apart on a horizontal line.



The final condition for a valid move is that the ***start*** square and ***middle*** square have playing pieces on them, whilst the ***finish*** square is empty. This can be checked using the ***play[ ] array*** values:

```
if (Math.Abs(startI - finishI) == 2)
{
    listBox1.Items.Add("Positions are 2 squares apart");
    int middle = (startI + finishI) / 2;

    if (play[startI, startJ] == 1 && play[middle, startJ] == 1
                                    && play[finishI, finishJ] == 0)
    {
        listBox1.Items.Add("VALID MOVE");
    }
}
```

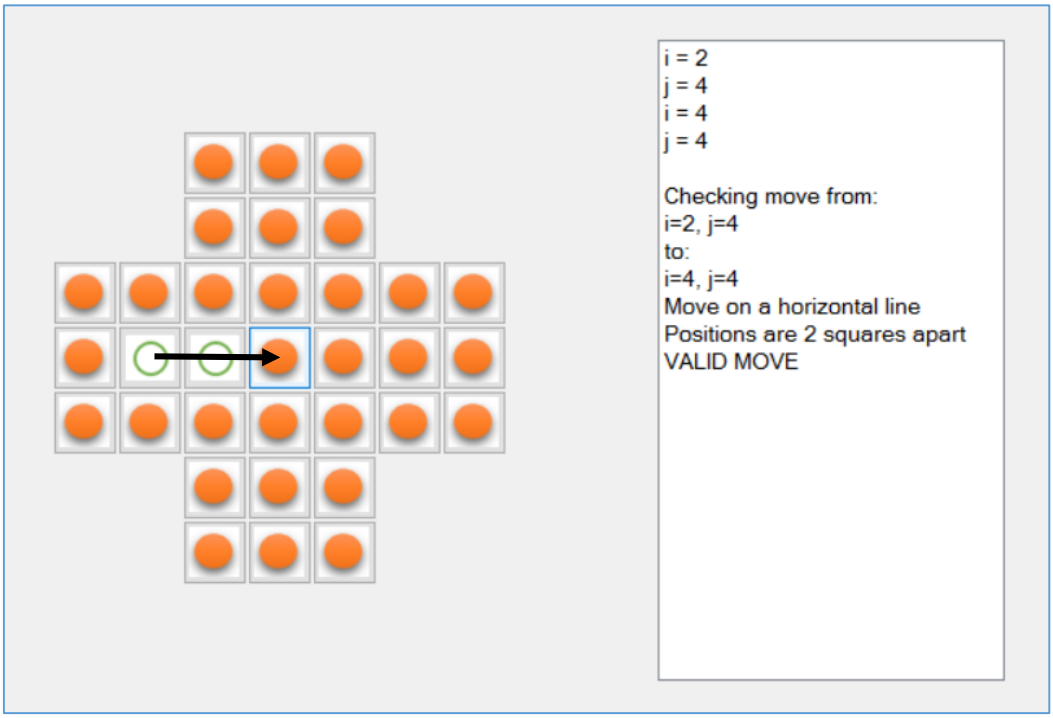
Run the program, and test that the computer can correctly detect valid horizontal moves:



Once a valid move has been made, we can change the pattern of playing pieces on the board accordingly. The ***play[ ] array*** values can be updated to reflect the new state of the board:

```
if (play[startI, startJ] == 1 && play[middle, startJ] == 1
                                                && play[finishI, finishJ] == 0)
{
     listBox1.Items.Add("VALID MOVE");

     play[startI, startJ] = 0;
     btnGame[startI, startJ].Image = Image.FromFile("../../hole.png");

     play[middle, startJ] = 0;
     btnGame[middle, startJ].Image = Image.FromFile("../../hole.png");

     play[finishI, finishJ] = 1;
     btnGame[finishI, finishJ].Image = Image.FromFile("../../peg.png");
}
```

Run the program again and test that horizontal moves are now displayed correctly:



If the procedure for horizontal moves is now working correctly, then the screen display can be tidied by removing the ListBox. The lines in the program which output text to the ListBox can also be removed:

```
private void checkMove(int startI,int startJ,int finishI,int finishJ)
{

    if (startJ == finishJ)
    {
        if (Math.Abs(startI - finishI) == 2)
        {
             int middle = (startI + finishI) / 2;

             if (play[startI, startJ] == 1 && play[middle, startJ] == 1
                                              && play[finishI, finishJ] == 0)
             {
                 play[startI, startJ] = 0;
                 btnGame[startI, startJ].Image = Image.FromFile("../../hole.png");

                 play[middle, startJ] = 0;
                 btnGame[middle, startJ].Image = Image.FromFile("../../hole.png");

                 play[finishI, finishJ] = 1;
                 btnGame[finishI, finishJ].Image = Image.FromFile("../../peg.png");
             }
         }
     }
```

Make a copy of the '***horizontal move***' code, and paste this immediately below to make an equivalent '***vertical move***' procedure:

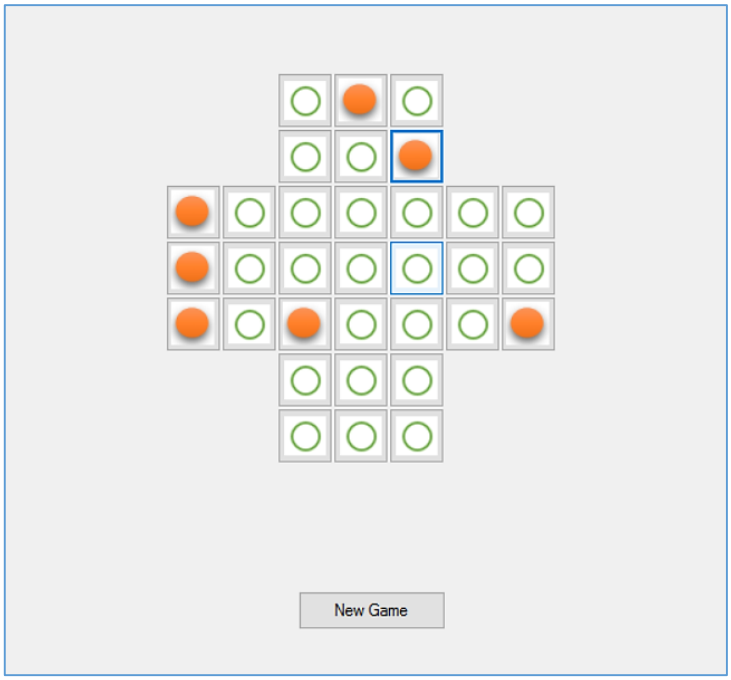
```
private void checkMove(int startI,int startJ,int finishI,int finishJ)
{
    if (startJ == finishJ)
    {
        if (Math.Abs(startI - finishI) == 2)
        {
             int middle = (startI + finishI) / 2;

             if (play[startI, startJ] == 1 && play[middle, startJ] == 1
                                           && play[finishI, finishJ] == 0)
             {
                 play[startI, startJ] = 0;
                 btnGame[startI, startJ].Image = Image.FromFile("../../hole.png");

                 play[middle, startJ] = 0;
                 btnGame[middle, startJ].Image = Image.FromFile("../../hole.png");

                 play[finishI, finishJ] = 1;
                 btnGame[finishI, finishJ].Image = Image.FromFile("../../peg.png");
             }
        }
    }
```

copy

To complete the 'vertical move' procedure, some changes to the code are necessary. These are outlined below:

```
if (startI == finishI)
{
     if (Math.Abs (startJ - finishJ) == 2)
     {
         int middle = (startJ + finishJ) / 2;

         if (play[startI, startJ] == 1 && play[startI, middle] == 1
                                       && play[finishI, finishJ] == 0)
         {
            play[startI, startJ] = 0;
            btnGame[startI, startJ].Image = Image.FromFile("../../hole.png");

           play[startI, middle] = 0;

           btnGame[startI, middle].Image = Image.FromFile("../../hole.png");

            play[finishI, finishJ] = 1;
            btnGame[finishI, finishJ].Image = Image.FromFile("../../peg.png");
         }
     }
}
```

Run the program and test both horizontal and vertical moves are made correctly. It should now be possible to play the complete game of solitaire.



One last improvement that can be made is the addition of a '***New Game***' button, to reset the playing pieces to the start position. Add a Button component to the form and give this the name ***btnNewgame***.

To restart the game, it is necessary to remove the current buttons, then call the ***initialiseBoard( )*** method to set up the playing pieces again in the start position. Add code to the button_click method to do this:

```
private void btnNewgame_Click(object sender, EventArgs e)
{
    int totalButtons = pnlGame.Controls.Count;
    for (int i = 0; i < totalButtons; i++)
    {
        pnlGame.Controls.RemoveAt(0);
    }
    initialiseBoard();
}
```

Run the program. Make some moves, then test that the '***New Game***' method operates correctly.