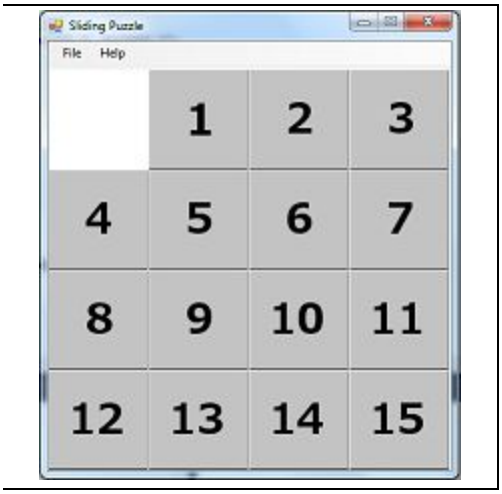
C# - Sliding Puzzle

In this exercise, you will make a sliding puzzle game. The object of the game is to get the numbers in order by sliding pieces into the blank space.



The board will be a 2 dimensional array of integers. Each cell in the array will have a unique value which corresponds to a picture to draw. You will need to track mouse and up mouse down events to know when to switch squares.

Review of nested loops

Getting this program to work is going to require nested loops (a loop within a loop) on multiple occasions. Here is an example of a nested loop which will scan every cell in the board.

```
for (int i = 0; i < BOARD_SIZE; i++)
{
     for (int j = 0; j < BOARD_SIZE; j++)
     {
        //do something to/with array[i,j]
     }
}
```

Getting Started

Create a new Windows Form Application called Sliding Puzzle.

The image resources have already been created for you. They are in the folder T:\Game Assets\Sliding Puzzle. Import them into your project.

Add a menu strip to your form with File and Help menus.

Add a 400 x 400 PictureBox control to your form.

Review the Class Diagram

The last of this packet contains a class diagram of Form1 which shows every variable and function Form1 needs to have in order for this project to work.

Class Diagram

Form1 Class	
Member variables	
const int BOARD_SIZE = 4;	The length of a side of the board.
const int TILE_SIZE = 100;	The size of the tile image used to draw the board.
Const int EMPTY = 0;	Empty should be the index of the empty tile
int [,]grid = new int [BOARD_SIZE, BOARD_SIZE]	2 dimension array to store the board. You'll call the function to Initialize the array in the constructor.
Image []images = new Image[16];	An array of the images used to draw the board. Storing them in an array will simplify the drawing code.
Point mouseDown;	A Point structure used to store the i,j coordinate of the picture cell where the mouse was clicked down. This should be between 0 and BOARD_SIZE.
Point mouseUp;	A Point structure used to store the i,j coordinate of the picture cell where the mouse was released. This should be between 0 and BOARD_SIZE.
Random r = new Random();	A random number generator
Functions	
Form1() { }	This is the default constructor which is provided for you automatically Call InitializeImages() and InitializeGrid()
void InitializeImages() { }	See the notes at the end
void InitializeGrid() { }	Sets the value in each cell in *grid* to its row number * BOARD_SIZE plus its column number.

void ScrambleBoard() { }	Switch two elements in *grid* some number of times. Ten would be fine.
void Draw(Graphics g) { }	Uses a nested loop to draw the tile for each cell in the board. Look at the value in each cell (i,j) then retrieve that image from your image array at (i* TILE_SIZE, j * TILE_SIZE)
bool IsValidMove() { }	Returns true if player is making a legal move based on the values in mouseUp and mouseDown. The code is provided for you.
bool IsSolved() { }	Returns true if all the tiles on the board are in the correct position (in order from lowest to highest). If you just return false, the game should still work fine, but I'm sure you can figure out how to implement this function.
void pictureBox1_Paint(...) { }	**Create this function using the lightning bolt.** In the body, call pictureBox1.Invalidate() to force the control to repaint.
void pictureBox1_MouseDown(...) {}	**Create this function using the lightning bolt.** Convert the mouse coordinates to array indexes and stores them in mouseDown.X and mouseDown.Y. To make the conversion, divide the X and Y coordinates by TILE_SIZE.
void pictureBox1_MouseUp(...) { }	**Create this function using the lightning bolt.** Convert the mouse coordinates to array indexes and store them in mouseUp.X and mouseUp.Y. To make the conversion, divide the X and Y coordinates by TILE_SIZE. Call IsValidMove() to see if the player is making a legal move. If so, switch the values in the grid at mouseUp and mouseDown.

	Then, call pictureBox1.Invalidate() to redraw the board. After switching the squares, call IsSolved() to see if the game is over. It returns *true*, print a victory message.

Initializing the Images

There are 16 tiles, and loading them one at a time would be a pain. Fortunately, the ResourceManager class can load them for us as long as we know the names of the resources we want to retrieve. In this project, the tiles all follow the same naming convention so we can take advantage of the resource manager.

```
//put this code in a loop to load all the images
images[i] = (Image) Properties.Resources.ResourceManager.GetObject("Tile" + i);
```

Note the explicit cast in the code above.

Initializing the Board

In the constructor or Form1_Load event, you need to set the initial values in the 2d array. Use a nested loop as shown in the example above. Set the value of each cell to:

```
grid[i, j] = j * BOARD_SIZE + i;
```

This puts values in the *grid* array in ascending order

Scrambling the Board

You will need a function to scramble the board. To do this you need to switch two elements in the 2d array several times. This is just like switching elements in a regular array each that you need two random numbers to define a particular cell because a two dimensional array has two indexes (not just one).

Here's the jist of what will go into your function which scrambles the board...

1. Pick two random numbers within the array boundaries (0,3). These will represent a row and column. We'll call them x1 and y1.

2. Save the value of the grid at x1,y1 in a temp variable.

3. Pick another two random numbers within the array boundaries (0,3). These will represent a row and column. We'll call them x2 and y2.

4. Assign the value of the cell at x2, y2 into the cell at x1, y1.

5. Assign the value of the temp variable to the cell at x2, y2.

6. Enclose steps 1 through 5 in a for loop which repeats 10 times.

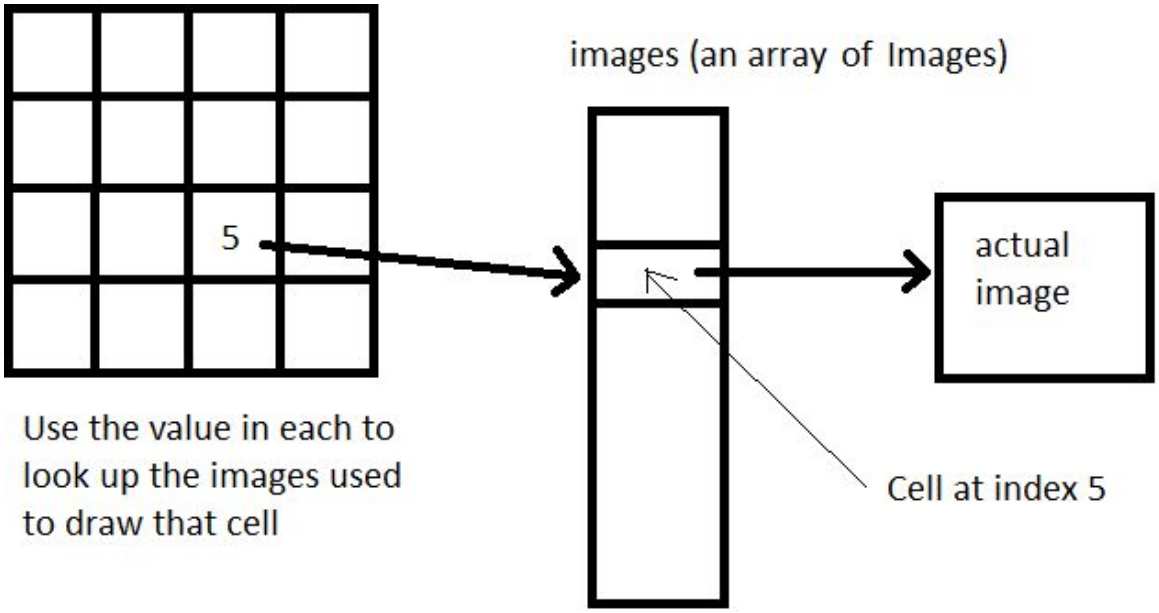
Now when this function runs, it will scramble the board by repeatedly switching two random cells.

Drawing the Board

Create a function called Draw() which accepts a Graphics object as its parameter. This function will be called after the player slides a piece and also by paint events which are automatically generated when the screen needs to be refreshed.

In your Draw() function, draw the grid as explained below.

The board is represented as a 4x4 (two dimensional) array of *ints*. You just need nested loops to draw each cell at *i,j* in the array. Then, get the value in that cell and use it to lookup the image to draw it with. Lastly, draw the image at i * 100, j * 100 because each tile is 100 pixels x 100 pixels.



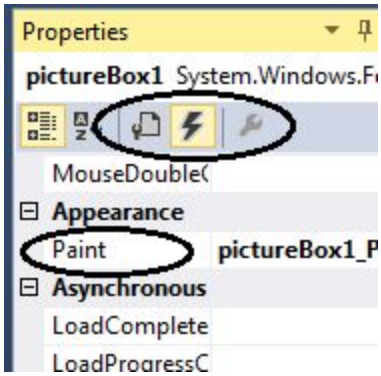
Creating a paint event

Your form is not going to know to call your Draw() function when the picture box needs to be repainted.

1. Create a Paint event handler.
2. Call your Draw() function from the Paint event handler

Here's how to do it...

1. Go to the Design view for your form and single click on the picture box control.
2. Click on the lightning bolt. This brings up a list of events you can listen for.
3. Double click on **Paint**. Visual Studio will create a function for you.



4. Tell this paint function to call your paint function.

```
private void pictureBox1_Paint(object sender, PaintEventArgs e)
{
    Draw(e.Graphics); //call your Draw function
}
```

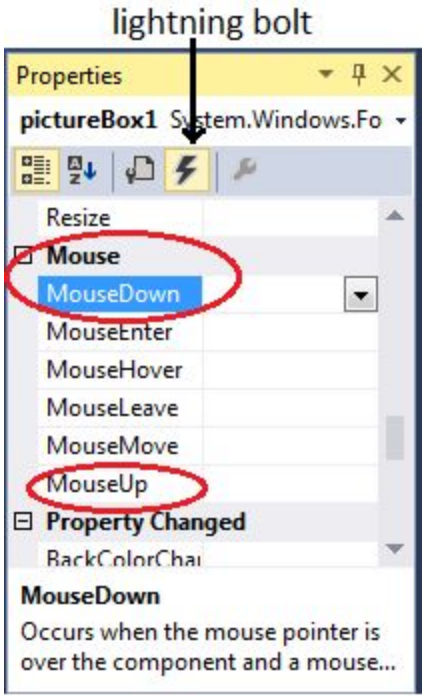
Try to get your board initialized and drawn properly before you go on to the mouse event functions.

Mouse up and mouse down events

To slide a puzzle piece, the player will click the mouse down in on one square, then release the mouse where that piece should go. This means we need to listen to both mouse up and mouse down events. This is going be just like what we did to create the Paint event handler.

To add functions to handle these types of events...

1. Find the properties dialog box for the PictureBox
2. Click on the lightning bolt. This shows a list of events
3. Double click on MouseDown and MouseUp



This will create two functions for you to handle mouse up and mouse down events on the PictureBox control.

Which square did the player click on?

You'll have to keep track of which square the player click and released on. So, how do you convert from screen coordinates to grid coordinates? The solution is easy. Just divide the X and Y in the MouseEventArgs by 100. This works because each tile is 100 pixels wide.

Here's a freebie...

```
private void pictureBox1_MouseDown(object sender, MouseEventArgs e)
{
     mouseDown.X = e.X / TILE_SIZE;
     mouseDown.Y = e.Y / TILE_SIZE;
}
```

Determining valid moves

When the mouse is released, you'll need to see the player has made a valid move.

If the mouse was released in an empty square that was only one cell up/down or left/right from the start square, you have a valid move. Diagonal moves are not allowed, the x move amount can be 1 or the y move amount can be 1 but not both. Also, the direction of movement doesn't matter, so can use the Math.Abs() function to remove the sign.

Determining when the player has won

This function is almost exactly like the function which initializes the board. The only difference is that rather than setting the value of a cell[i,j], you are checking it.

1. Create a function called IsSolved() which returns a *bool.*
2. Add a nested loop to the function which examines each cell.
3. Inside the inner loop, check to see if the cell at *i,j* has the correct value. If it doesn't return *false.*

```
if (grid[i, j] != j * BOARD_SIZE + i)
{
     return false; //this piece is not in its correct position
}
```

4. At the bottom, after the loops, *return true.* If the program gets to this point, all the values in *grid* must have been correct.

Alternate approach:

You could also loop through the array, looking at each cell. If the value of a cell is less than a previous cell, the game is not over.