C#/WinForms Project 15 Puzzle

Create a version of the 15 puzzle game using C# and Windows Forms.

The puzzle is a 4-by-4 grid with 15 tiles numbered 1 through 15, and one empty position called the empty tile. The puzzle begins with the numbers in random order, with some restrictions on the definition of “random,” discussed below. The goal is to rearrange the tiles by sliding a tile adjacent to the empty tile into the empty slot, and so on, until all the tiles are in sorted row-major order (1-4 on row 1, 5-8 on row 2, 9-12 on row 3, 13-15 on row 4).

The GUI is a series of individual images loaded into a .NET Control called PictureBox. A PictureBox displaying a puzzle image we will call a “tile”. This is analogous to card images displayed by a JPanel in a Swing application. The enclosing Form container will have no layout manager, tiles will be added to the form and given a value for their Location property based on a simple x-y layout calculation. This is called absolute positioning in lieu of a layout manager. You will write a derived class of class PictureBox called Tile.

Use a MouseDown delegate for each Tile, implemented as an override of OnMouseDown for the Tile class. To process the event, a Tile that has been touched will check to see if it is adjacent to the empty Tile, and if so, it will swap positions with the empty position.

Tiles are attached or added to the Form during initialization and never need to be removed, only their Location property gets updated. A Tile needs to know the reference to the enclosing container so it can calculate where it is relative to the empty tile.

To begin a puzzle solving session, the tiles must be randomized, but there are some constraints. Either (1) place the tiles into the solved state and then make a large number of random but legal moves, or (2) place the tiles into the solved state and then make a large but even number of tile swaps. Initial positioning of the tiles in totally random order may result in a puzzle that can’t be solved without snapping a tile out of the plane of the puzzle and snapping it back in.
Tiles
You can create your own tile images in PowerPoint or PhotoShop, or directly in WinForms, or just use mine. A popular variation of the game is to take a single arbitrary image of the user’s choice and break it into tiles to use as the basis for the puzzle. In Windows Forms, an image can be loaded from a file and displayed by a `PictureBox` control. The `PictureBox` class is defined in the `System.Windows.Forms` namespace, and the `Image` class is defined in the `System.Drawing` namespace.

To load an `Image` object from a file use the static `FromFile()` method:

```csharp
Image img = Image.FromFile("someimg.png");
```

To assign an `Image` to a `PictureBox`, assign a reference to its `Image` property:

```csharp
PictureBox b = new PictureBox;
b.Image  = img;
b.Height = img.Height;
b.Width  = img.Width;
```

The `PictureBox` size derives directly from the size of the `Image` that it displays.

Consider defining class `Tile` as a derived class of `PictureBox`:

```csharp
public class Tile : PictureBox { ... }
```

The `Tile` class introduces a couple of additional fields/properties:

```csharp
private int numericvalue; // numeric value of this tile
private Form form; // form that the tile belongs to,
                    // can be a derived class of Form
```

However it doesn’t need to define the `Image` property, it can use the property inherited from `PictureBox`. The constructor should initialize the `Tile` object with values for its form, numeric value, and `Image` (loaded from a file). Provide public properties for numeric value and image, these should be get-only, there is no need to change these values after the object is instantiated. Finally, override the `OnMouseDown()` method so that each `Tile` responds to mouse events. When a `Tile` is pressed, the method checks to see if it is adjacent to the empty tile, and if so, swaps its location with the empty tile. This is why the `Tile` object requires a reference to the form object, methods to check relationships between `Tile` objects will be defined as part of the `Form` derived class.

An extra feature to consider is to allow multiple tiles in the same row or column as the empty tile to be moved as a group. If the user touches a tile that is not adjacent to the empty tile but is in the same row or column as the empty tile, then all tiles between the touched tile and the empty tile move as a group.
The Form Derived Class
Create a Form derived class that is the top level container for the app.

```csharp
public class PuzzleForm : Form { … }
```

This class will maintain an array of Tile objects. Since there are only 15 tiles representing values 1 through 15, you might feel that a hash map for managing them is overkill. If so, it’s okay to just maintain them in a 1-D array of Tile objects. I suggest you make the array size 16 and maintain a null pointer in the position that represents the empty tile. You could also create a 4-by-4 2-D rectangular array, but if you use a simple convention like

```csharp
int numcols = 4;
int numrows = 4;
int index = …;   // position of a tile in a 1-D array
int row = index/numcols;
int col = index%numcols;
```

you can switch between a 1-D index and the tile’s actual row/col position easily when needed. Location information for the tile in pixels can also be easily calculated from its row/col values. You have a choice to either remember the current location of the empty tile in a separate index, or to just search the array as needed to determine the index of the null reference in the array.

There is no `paintComponent()` method for class `Form`. In both WinForms and WPF, the need to repaint is treated like just another event. So there is a `Paint` event and a handler method named `OnPaint()`. For a Form object that requires custom painting, you write an override of `OnPaint()`. But for this project, `OnPaint()` is not needed since there is no custom drawing. Since it is composed once a PictureBox control is attached to the form, it’s an example of a GUI that knows how to draw itself, no custom code is required.

Some Suggested Methods

- `void LoadTiles() { }`: load image files and create array of Tile objects
- `void ShuffleTiles() { }`: randomize tiles to prepare for play. Initialize tiles in solution order, then randomize by making random valid moves, don’t simply place tiles in random positions.
- `void LayoutTiles() { }`: set the Location property of each tile and attach it to the parent form.
- `bool IsAdjacentToEmptyTile(Tile t) { }`: check to see if t is adjacent to the empty tile.
- `void SwapLocationWithEmptyTile(Tile t) { }`: swap positions for t and the empty tile.
Top-Level Skeleton

```csharp
public class PuzzleForm : Form {
    ...

    public static void Build() {
        PuzzleForm f = new PuzzleForm();
        Application.EnableVisualStyles();
        Application.Run(f);
    }

    [STAThread]
    public static void Main() {
        PuzzleForm.Build();
    }
}
```

C# Threads and Windows Forms
If you want to create an animation effect to show tile moves, you will need to create a thread to control the redraw of the tile in different positions with a delay as it’s being moved. C# thread-related classes are defined in the `System.Threading` namespace, you must add this namespace to your source file, you don’t get it for free as in Java.

In Java, the standard way to define a thread is as follows:
- Define a subclass of class Thread
- Override the run() method, which defines the task that the thread will control
- Instantiate an object of the class
- Call the start() method of the object.

The process is similar in C#:
- Define a method that satisfies the ThreadStart delegate: `delegate void ThreadStart();`
- Create a Thread object with a ThreadStart object as parameter (note that the Thread class is sealed).
- Call the Start() method of the object.

```csharp
void DoSomething() { // method to be run by thread
    ...
}

void x() { // actually running it
    Thread t = new Thread(DoSomething);
    t.Start();
}
```

This is a shortcut for the more formal

```csharp
void x() {
    Thread t = new Thread(new ThreadStart(DoSomething));
    t.Start();
}
```
C# doesn’t support anonymous classes but it does support anonymous delegates, so another possibility:

```csharp
Thread t = new Thread(delegate() {
    ...
});
t.Start();
```

In Windows Forms, the rule about threads is that a Control should be modified only by the thread that created it. If you place a time-consuming task on a separate thread, and during the task you need to modify a Control, you will have to call the Invoke() method to move the modification back to the thread that created the Control, on behalf of the current thread.

.NET Controls have a convenient bool property `InvokeRequired` that indicates if the Control needs a call to Invoke() or not. The property is basically doing a convenience check to let you know if the current thread is the same as the thread that created the control. As an example, suppose we create an instance of PictureBox during GUI construction. Then later on, we create a separate thread for a time-consuming text and during the thread we need to move the location of the PictureBox object.

In the constructor, we build a PictureBox and attach it to the Form:

```csharp
class MyForm : Form {
    private PictureBox b;

    public MyForm() {
        b = new PictureBox();
        b.Image = ...;
        b.Parent = this;
        b.Location = new Point(50,75);
        ...
    }
}
```

We also define a convenience method to move a PictureBox to some location:

```csharp
private void MovePictureBox(PictureBox b, Point loc) {
    b.Location = loc;
    Invalidate();
}
```

This method can be called from any thread. During the method the PictureBox's location is changed and the GUI is repainted.
We define a ThreadStart delegate object for the long running task.

```csharp
private void LongTask() {
    while (...) {
        Thread.Sleep(1000);
        Point p = new Point(…);

        MovePictureBox(b, p);
    }
}
```

We start the thread from another method:

```csharp
Thread t = new Thread(LongTask);
t.Start();
```

The problem is now, back in LongTask(), the PictureBox is being modified by a custom thread. We modify LongTask() to fix the problem like this. First we define a delegate to represent the MovePictureBox() method:

```csharp
delegate void MoveDelegate(PictureBox b, Point p);
```

Then we modify LongTask() to call MovePictureBox() indirectly if necessary:

```csharp
private void LongTask() {
    while (...) {
        Thread.Sleep(1000);
        Point p = new Point(…);

        if (b.InvokeRequired) {
            MoveDelegate d = MovePictureBox;
            Invoke(d, new object[]{ b, p });
        }
        else {
            MovePictureBox(b, p);
        }
    }
}
```

The Invoke() method takes two params, the first is the method to be called, the second is an array of objects that are the parameters to the method.

A general and somewhat prepackaged solution is available from the BackgroundWorker class.