XNA 4.0 RPG Tutorials

Part 23A

Level Editor

I'm writing these tutorials for the new XNA 4.0 framework. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the XNA 4.0 RPG tutorials page of my web site. I will be making my version of the project available for download at the end of each tutorial. It will be included on the page that links to the tutorials.

You all knew that this was coming at some point. You will want to be able to create your world at design time and read it in at run time. For that you will need a world, or level, editor. To accomplish that I'm going to add a new project that will be used to create your levels and other content that is XNA related. The **RpgEditor** is great for creating data to be used with the game but you are going to want to associated XNA related content to NPCs as an example. You will want a sprite to represent the NPC in the game.

To get started right click the **EyesOfTheDragon** solution, not the project, in the solution explorer. Select **Add** and then **New Project**. Choose a **Windows Game (4.0)** from the list of XNA projects. Name this new project **XLevelEditor**. You might be wondering why I'm adding a new game project and not a Windows Forms project. The reason has to do with Windows 7 and it not finding the XNA assemblies at run time. The alternative is to create an XNA Windows game and add a reference to **System.Windows.Forms**.

Do that now by right clicking **XLevelEditor** in the solution explorer and selected **Add Reference**. From the .NET tab select **System.Windows.Forms**. You are also going to want references to the libraries we created. Right click **XRpgEditor** again and select **Add Reference**. From the **Project** tab select the **RpgLibrary** and **XRpgLibrary** projects.

There is a problem however. This project is set to launch an XNA Windows game and not Windows Forms application. You are first going to want to add a Windows Form to the **XRpgEditor** project. Right click the **XRpgEditor**, select **Add** and then **Windows Form**. Name this new form **FormMain**. Now, open the **Program.cs** file in the **XRpgEditor** project and chance it to the following.

```
using System;
using System.Windows.Forms;

namespace XRpgEditor
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
```

```
            using (FormMain frmMain = new FormMain())
            {
                Application.Run(frmMain);
            }
        }
    }
}
```

What this code is doing is similar to the **Program.cs** file in the **RpgEditor**. There is a using statement to bring the **System.Windows.Forms** name space into scope for the **Application** class. Inside the **Main** method, the first method called by C# when a program is executed, calls a few methods of the **Application** class to set some styles for the form. Then in a using statement I create a new instance of **FormMain** and inside of the using statement call **Application.Run** to start the form. When the form closes all disposable resources held by the form will automatically be released instead of waiting for the garbage collector to release them. I had to mark **Main** with an attribute, **STAThread**, to allow certain thread related tasks to work.

There are a few more things to do to prepare the project. One is that you want to change the project so that it uses the Reach profile. I'm doing this because sometimes I will be using my laptop for the tutorials and it doesn't support the HiDef profile. Right click the **XRpgEditor** and select **Properties**. On the **XNA Game Studio** tab set the **Game Profile** to **Use Reach**. While you have the properties for the properties select the **Application** tab. Change the **Target framework** to be **.NET Framework 4** instead of **.NET Framework 4 Client Profile**. Reply **Yes** to the dialog that pops up. You can close the **Properties** now. The last step is to add a reference to the project for the **IntermediateSerializer** so you can read and write your data from the editor. Right click the **XRpgEditor** project and select **Add Reference**. From the .NET tab add **Microsoft.Xna.Framework.Content.Pipeline** and make sure you add version 4.0.0.0 for XNA 4.0.

You are going to want to host XNA inside of the editor. For that you will want a few classes from the **WinForms Series 1: Graphics** sample from App Hub. You can find the sample from the following link: http://create.msdn.com/en-US/education/catalog/sample/winforms_series_1 Download and extract the files to a directory. Navigate to that directory and drag the following files from windows explorer onto the **XRpgEditor** project: **GraphicsDeviceControl.cs**, **GraphicsDeviceService.cs**, and **ServiceContainer.cs**.

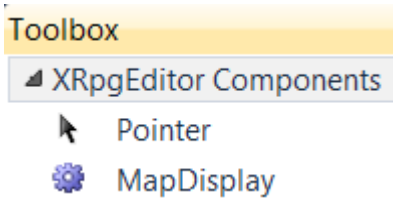
The next step is to create a class for drawing the map in. Right click the **XRpgEditor** project, select **Add** and then **Class**. Name this new class **MapDisplay**. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace XLevelEditor
{
    class MapDisplay : WinFormsGraphicsDevice.GraphicsDeviceControl
    {
        public event EventHandler OnInitialize;
        public event EventHandler OnDraw;

        protected override void Initialize()
        {
            if (OnInitialize != null)
                OnInitialize(this, null);
        }
```

```
        protected override void Draw()
        {
            if (OnDraw != null)
                OnDraw(this, null);
        }
    }
}
```

The first step is that I inherit the class from **WinFormsGraphicsDevice.GraphicsDeviceControl** that is from the **WinForms** series from App Hub that represents a control that can be rendered to using XNA. This is an abstract class and you need to implement two methods: **Initialize** and **Draw**. I also added in two event handlers called **OnInitialize** and **OnDraw**. In the **Initialize** method I check to see if **OnIntialize** is not null and if it isn't I call the **OnInitialize** method. I do something similar for the **Draw** method but with **OnDraw** instead of **OnInitialize**.

Build your project now. After building it bring up the design view of **FormMain** by right clicking it and selecting **View Designer**. If you expand the Toolbox you should see something similar to the following at the top of the Toolbox.



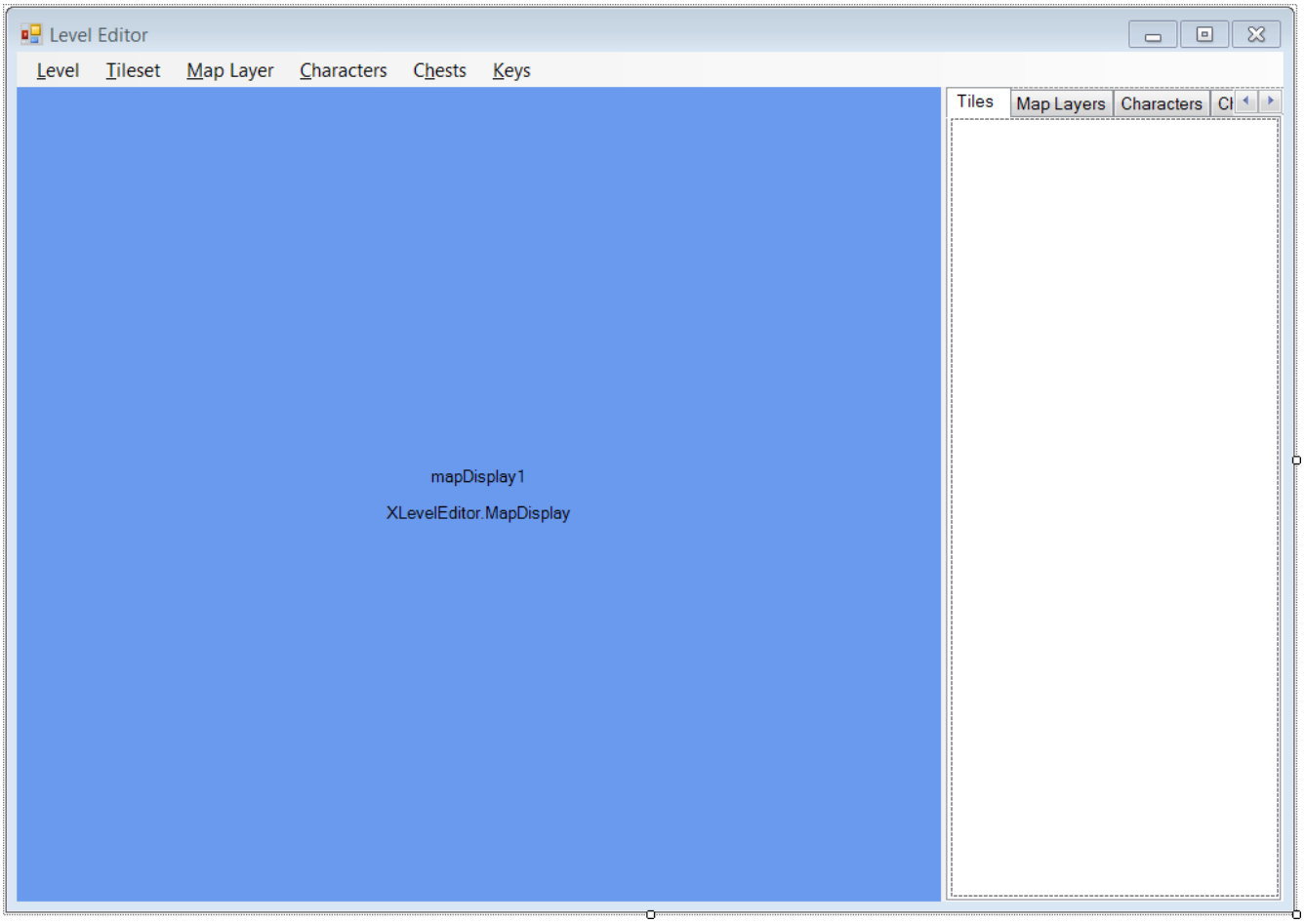
The next step is going to be to design some forms. The main form will be the actual editor. The child forms will be for adding items to a level. I found using tables for designing forms is much nicer than just having a bunch of scrawling text. Set the following properties for **FormMain**.

FormMain	
Size	1024, 720
StartPosition	CenterParent

Now, drag a **Menu Strip** onto **FormMain**. The items in the first row are the menu items along the top. Any items in a column under item are the menu items below it.

&Level	**&Tileset**	**&Map Layer**	**&Characters**	**C&hests**	**&Keys**
&New Level	&New Tileset	&New Layer			
&Open Level	&Open Tileset	&Open Layer			
&Save Level	&Save Tileset	&Save Layer			
-	&Remove Tileset				
E&xit					

The next control that I added to the form was a **Split Container**. The **Split Container** has two panels that can hold controls. In the one panel I added in a **MapDisplay** control that I created earlier. The other panel holds a **Tab Control** that has tabs for the different parts of the map, tiles, map layers, characters, etc.



Drag a **Split Container** onto **FormMain**. Set the following properties for the **Split Container**. The default values for the other properties are fine.

Property	Value
(Name)	splitContainer1
Dock	Fill
SplitterDistance	800

Onto **Panel1** of the **Split Container**, the one of the left, drag on a **Map Display** control that you created earlier. Set the following properties for the **Map Display** control.

Property	Value
(Name)	mapDisplay
Dock	Fill
TabIndex	0

Onto **Panel2** of the **Split Container**, drag on a **Tab Control**. The properties you want to set for the **Tab Control** are next.

Property	Value
(Name)	tabProperties
Dock	Fill
TabIndex	1

In the properties window for the **Tab Control** there is an entry **TabPages**. Click the **(Collection)** part will bring up a dialog for the pages. Select the default pages and click the **Remove** button to remove them. You will now add in pages for the tabs. There are 5 tabs in total. Set the following properties for the tabs.

#	(Name)	Text
0	tabTilesets	Tiles
1	tabLayers	Map Layers
2	tabCharacters	Characters
3	tabChests	Chests
4	tabKeys	Keys

That takes up more room but I think that it looks a little nicer and is a little more precise when it comes to designing forms. The next step is to design the tabs. Bring up the **Tiles** tab in the editor. You can do that by clicking the **Tiles** tab in the **Tab Control**. If that tab is not visible clicking the arrow buttons will move between tabs. What my **Tiles** tab looks like is on the next page.

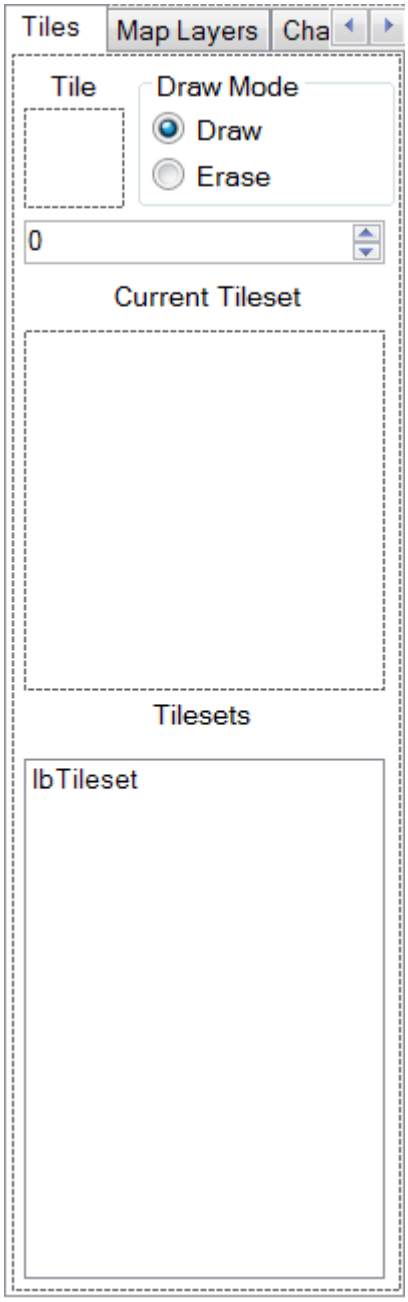
The tab is made up of three **Labels**, one **Group Box**, two **Picture Boxes**, two **Radio Buttons**, one **List Box**, and a **Numeric Up Down** control. The first control to drag onto the tab is a **Label**. Set the following properties of the **Label**.

Property	Value
(Name)	lblTile
AutoSize	FALSE
Location	7, 7
Size	50, 17
Text	Tile
TextAlign	TopCenter

Now, drag on a **Picture Box** under **lblTile**. Set these properties of the **Picture Box**.

Property	Value
(Name)	pbTilePreview
Location	7, 27
Size	50, 50

Now, drag a **Group Box** to the right of **lblTile**. Set the following properties for the **Group Box**.



Property	Value
(Name)	gbDrawMode
Location	63, 7
Size	128, 70
Text	Draw Mode

Onto the **Group Box** you are going to drag on two **Radio Buttons**. Set the following properties for the **Radio Buttons**.

Property	Value
(Name)	rbDraw
AutoSize	TRUE
Checked	TRUE
Location	7, 20
Text	Draw

Property	Value
(Name)	rbErase
AutoSize	TRUE
Location	7, 43
Text	Erase

Under the other controls drag on a **Numeric Up Down** control. Set the following properties for that control.

Property	Value
(Name)	nudCurrentTile
Location	7, 83
Size	180, 22

Under the **Numeric Up Down** control drag a **Label** control. Set the following properties for the **Label**.

Property	Value
(Name)	lblCurrentTileset
AutoSize	FALSE
Location	7, 112

Size	180, 23
Text	Current Tileset
TextAlign	TopCenter

Under that you will drag on a **Picture Box**. Set the following properties for the **Picture Box**.

Property	Value
(Name)	pbTilesetPreview
Location	7, 138
Size	180, 180

Under the **Picture Box** control drag a **Label** control. Set the following properties for the **Label**.

Property	Value
(Name)	lblTilesets
AutoSize	FALSE
Location	7, 321
Size	180, 23
Text	Tilesets
TextAlign	TopCenter

The last control to drag on is a **List Box**. Set the following properties for the **List Box**.

Property	Value
(Name)	lbTileset
Location	7, 352
Size	180, 260

The last thing I'm going to design is the **Map Layers** tab. There is just the one control on the **Map Layers** tab, a **Checked List Box**. I used a **Checked List Box** rather than a **List Box** to control how the layers are drawn. If you have a layer in the list box checked it will be drawn, otherwise it won't be drawn. Select the **Map Layers** tab and drag a **Checked List Box** onto the tab. Set the following properties of the **Checked List Box**.

Property	Value
(Name)	clbLayers
Dock	Fill

To be able to use this feature I need to make a quick change to the **TileMap** and **MapLayer** classes. What I'm going to do is instead of drawing the tiles in the **TileMap** class, have the **MapLayer** class

draw the tiles. I will start with the **MapLayer** class. You will want to add using statements for the basic XNA framework and the XNA framework **Graphics** classes. Add the following using statements and method to the **MapLayer** class.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

public void Draw(SpriteBatch spriteBatch, Camera camera, List<Tileset> tilesets)
{
    Point cameraPoint = Engine.VectorToCell(camera.Position * (1 / camera.Zoom));
    Point viewPoint = Engine.VectorToCell(
        new Vector2(
            (camera.Position.X + camera.ViewportRectangle.Width) * (1 / camera.Zoom),
            (camera.Position.Y + camera.ViewportRectangle.Height) * (1 / camera.Zoom)));

    Point min = new Point();
    Point max = new Point();

    min.X = Math.Max(0, cameraPoint.X - 1);
    min.Y = Math.Max(0, cameraPoint.Y - 1);
    max.X = Math.Min(viewPoint.X + 1, Width);
    max.Y = Math.Min(viewPoint.Y + 1, Height);

    Rectangle destination = new Rectangle(0, 0, Engine.TileWidth, Engine.TileHeight);
    Tile tile;

    for (int y = min.Y; y < max.Y; y++)
    {
        destination.Y = y * Engine.TileHeight;

        for (int x = min.X; x < max.X; x++)
        {
            tile = GetTile(x, y);

            if (tile.TileIndex == -1 || tile.Tileset == -1)
                continue;

            destination.X = x * Engine.TileWidth;

            spriteBatch.Draw(
                tilesets[tile.Tileset].Texture,
                destination,
                tilesets[tile.Tileset].SourceRectangles[tile.TileIndex],
                Color.White);
        }
    }
}
```

Basically what I did was move the code for drawing a map to the layer and fixed up a couple syntax errors caused by moving things. I needed to pass the **List<Tileset>** to the method as the tilesets are part of the map. I could have moved the **List<Tileset>** to the layer but that would make the changes more difficult to do.

The change to the **TileMap** class was much simpler. There is now just a foreach loop that loops through all of the layers calling the **Draw** method of each layer passing in the appropriate values. Change the **Draw** method of the **TileMap** class to the following.

```
public void Draw(SpriteBatch spriteBatch, Camera camera)
{
    foreach (MapLayer layer in mapLayers)
    {
        layer.Draw(spriteBatch, camera, tilesets);
    }
}
```

Rather than reading and writing the classes in the **XRpgLibrary** directly, I'm going to add some data classes to the **RpgLibrary**. You read and write the data classes from the editor. When the player unlocks an area of your world you read in the data using the **Content Pipeline**. Since they are changing the world as they explore it you will be using a different mechanism for saving and loading games.

To get started, right click the **RpgLibrary** and select **New Folder**. Name the new folder **WorldClasses**. Right click the **WorldClasses** folder, select **Add** and then **Class**. Name the new class **TilesetData**. The code for that class follows next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RpgLibrary.WorldClasses
{
    public class TilesetData
    {
        public string TilesetName;
        public string TilesetImageName;
        public int TileWidthInPixels;
        public int TileHeightInPixels;
        public int TilesWide;
        public int TilesHigh;
    }
}
```

Just a basic public class that has enough data to create a **Tileset** from the **XRpgLibrary**. There are two string fields, **TilesetName** and **TilesetImageName**. **TilesetName** is the name of the tileset and will be used in a manager class. **TilesetImageName** is the name of the file that holds the image for the tileset. There are then four integer fields that describe the tiles in a tileset. There are fields for the width and height of the tiles in the tileset, **TileWidthInPixels** and **TileHeightInPixels**. There are also fields for the number of tiles wide the tile set is and how many tiles high, **TilesWide** and **TilesHigh**. With those values you have enough information to pass to the constructor of the **Tileset** class to create a **Tileset**.

The next class to add is a class to represent a map layer as maps are made up of lists of **Tileset** and **MapLayer** objects. Right click the **WorldClasses** folder in the **RpgLibrary** folder, select **Add** and then **Class**. Name this new class **MapLayerData**. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RpgLibrary.WorldClasses
{
    public struct Tile
    {
        public int TileIndex;
        public int TileSetIndex;

        public Tile(int tileIndex, int tileSetIndex)
        {
            TileIndex = tileIndex;
            TileSetIndex = tileSetIndex;
        }
    }

    public class MapLayerData
```

```
    {
        public string MapLayerName;
        public int Width;
        public int Height;
        public Tile[] Layer;

        private MapLayerData()
        {
        }

        public MapLayerData(string mapLayerName, int width, int height)
        {
            MapLayerName = mapLayerName;
            Width = width;
            Height = height;

            Layer = new Tile[height * width];
        }

        public void SetTile(int x, int y, Tile tile)
        {
            Layer[y * Width + x] = tile;
        }

        public Tile GetTile(int x, int y)
        {
            return Layer[y * Width + x];
        }
    }
}
```

I added a structure to this class for tiles. It has public integer fields for the index of the tile and the index of the tileset. I included a constructor that requires two parameters: **tileIndex** and **tileSetIndex**. They are for the index of the tile and the index of the tileset respectively. Making it a structure rather than a class makes it a value type rather than a reference type. So, when I create the array for tiles in the **MapLayerData** the tiles automatically have values.

The **MapLayerData** class has four public fields: **Layer** that is a single dimension array of **Tile**, **MapLayerName** that is a string for the name of the layer, **Width** and **Height** that are integers and are the width and height of the map. Why did I choose a single dimension array rather than a two dimension array like in the tile engine? You can't serialize arrays with more than one dimension. You can simulate a multidimension array in a one dimension array though.

There are two constructors for the **MapLayerData** class. There is a private constructor that takes no parameters that will be used in serializing and deserializing the class. The public constructor takes three paramaters: **mapLayerName** that is the name of the map layer, **width** and **height** that are the width and height of the map. The public constructor first sets the **MapLayerName**, **Width**, and **Height** fields to the values passed in. I then create a single dimension array of **Tile** that is **height** times **width**. To find out how many elements are in a 2D array you multiple the width of the array by the height of the array so to represent a 2D array in a 1D array you need an array that is that size.

There are two public methods in this class. The first, **SetTile**, sets that tile that is represented at coordinates **x** and **y** to the **Tile** passed in. To set the element, and retrieve the element, you need to be consistent in determining the index. I used the calculation **y * Width + x**. Lets think about this a bit. Take a small 2D array, [4, 3]. You can represent it using a 1D array [12]. You have **y** between 0 and 3 and **x** between 0 and 2 in loops like this.

```
for (y = 0; y < 4; y++)
```

```
    for (x = 0; x < 3; x++)
```

When you **y** at 0 the calculation for the 1D array will be 0 * 3 + 0 = 0, 0 * 3 + 1 = 1, and 0 * 3 + 2 = 2. Then if you move to **y** at 1 you will get 1 * 3 + 0 = 3, 1 * 3 + 1 = 4, and 1 * 3 + 2 = 5. When you then move to **y** at 2 you will get 2 * 3 + 0 = 6, 2 * 3 + 1 = 7, and 2 * 3 + 2 = 8. Finally, when you move **y** to 3 you will get 3 * 3 + 0 = 9, 3 * 3 + 1 = 10, and 3 * 3 + 2 = 11. You can see that using the calculation **y * WidthOfArray + x** you can determine the position of a 2D element at (x, y) in 1D array. This is pretty much what the computer does when you declare a 2D array. It sets aside an area of memory the size of the array. It uses calculations like above to decide where the element resides in memory.

Now that you have classes that represent a tileset and a map layer you can create a class that can represent an entire map. Right click the **WorldClasses** in the **RpgLibrary** project, select **Add** and then **Class**. Name this new class **MapData**. The code for that class follows next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RpgLibrary.WorldClasses
{
    public class MapData
    {
        public string MapName;
        public MapLayerData[] Layers;
        public TilesetData[] Tilesets;

        private MapData()
        {
        }

        public MapData(string mapName, List<TilesetData> tilesets, List<MapLayerData> layers)
        {
            MapName = mapName;
            Tilesets = tilesets.ToArray();
            Layers = layers.ToArray();
        }
    }
}
```

Not a very complex class. There are just three fields. A string for the name of the map, an array of **MapLayerData** for the layers in the map, and an array of **TilesetData** for the tilesets in the map. There is a private constructor that will be used in serializing and deserializing the map and a public constructor that takes three parameters: the name of the map, a **List<TilesetData>** for the tilesets the maps uses, and a **List<MapLayerData>** for the layers in the map. It sets the fields using the values that are passed to the constructor. I use the **ToArray** method of the **List<T>** class to convert the lists to arrays.

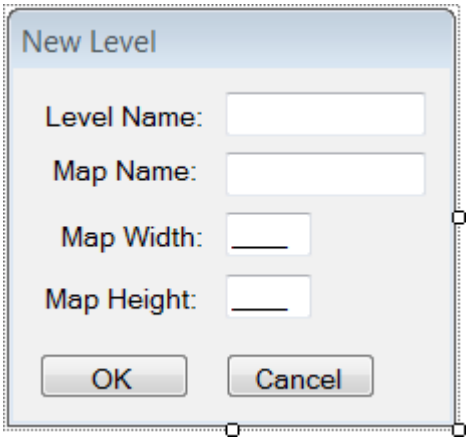
I want to add another data class, for levels. Right click the **WorldClasses** in the **RpgLibrary** project, select **Add** and then **Class**. This is the code for that class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RpgLibrary.WorldClasses
{
    public class LevelData
```

```
    {
        public string LevelName;
        public string MapName;
        public int MapWidth;
        public int MapHeight;
        public string[] CharacterNames;
        public string[] ChestNames;
        public string[] TrapNames;

        private LevelData()
        {
        }

        public LevelData(
            string levelName,
            string mapName,
            int mapWidth,
            int mapHeight,
            List<string> characterNames,
            List<string> chestNames,
            List<string> trapNames)
        {
            LevelName = levelName;
            MapName = mapName;
            MapWidth = mapWidth;
            MapHeight = mapHeight;
            CharacterNames = characterNames.ToArray();
            ChestNames = chestNames.ToArray();
            TrapNames = trapNames.ToArray();
        }
    }
}
```

Another rather straight forward class. There are fields for the name of the level, **LevelName**, the map for the level, **MapName**, the width and height of the map**, MapWidth** and **MapHeight**, the names of characters on the map, **CharacterNames**, the name of chests on the map**, ChestNames**, and the names of traps on the map, **TrapNames**. The private constructor will be used in serializing and deserializing objects. The public constructor takes a parameter for each of the fields. For the arrays of strings I pass in **List<string>** and use the **ToArray** method of **List<T>**.

I'm going to move back to the editor now. I want to add in a form for creating a new level. Right click the **XLevelEditor** project in the solution explorer, select **Add** and then **Windows Form**. Name this new form **FormNewLevel**. My finished form looked like this.



To start you are going to want to set some of the properties of the form itself. They are in the following table.

Property	Value
ControlBox	FALSE
FormBorderStyle	FixedDialog
Size	225, 210
StartUpPosition	CenterParent
Text	New Level

Onto the form I dragged four **Labels**, two **Text Boxes**, two **Masked Text Boxes** and two **Buttons**. I will do the controls in pairs that are in the same row of the form. Drag on a **Label** and **Text Box** onto the form. Set the following properties.

Label

Property	Value
(Name)	lblLevelName
Location	13, 15
Text	Level Name:

Text Box

Property	Value
(Name)	tbLevelName
Location	106, 12

Drag on another **Label** and **Text Box** setting the following properties for them.

Label

Property	Value
(Name)	lblMapName
Location	17, 42
Text	Map Name:

Text Box

Property	Value
(Name)	tbMapName
Location	106, 42

The next pair of controls are a **Label** and a **Masked Text Box**. Drag them on and set these properties.

Label

Property	Value
(Name)	lblMapWidth

Location	21, 75
Text	Map Width:

Masked Text Box

Property	Value
(Name)	mtbWidth
Location	106, 72
Mask	0000 (four zeros)
Size	45, 22

The next pair of controls are a **Label** and a **Masked Text Box**. Drag them on and set these properties.

Label

Property	Value
(Name)	lblMapHeight
Location	13, 106
Text	Map Height:

Masked Text Box

Property	Value
(Name)	mtbHeight
Location	106, 103
Mask	0000 (four zeros)
Size	45, 22

The last two controls to add are two **Button** controls. Add them and set these properties.

(Name)	Location	Size	Text
btnOK	13, 142	75, 23	OK
btnCancel	106, 142	75, 23	Cancel

Now that the form has been designed it is time to code its logic. Right click **FormNewLevel** in the **XLevelEditor** project and select **View Code**. Change the code for the form to the following.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

```
using RpgLibrary.WorldClasses;

namespace XLevelEditor
{
    public partial class FormNewLevel : Form
    {
        #region Field Region

        bool okPressed;
        LevelData levelData;

        #endregion

        #region Property Region

        public bool OKPressed
        {
            get { return okPressed; }
        }

        public LevelData LevelData
        {
            get { return levelData; }
        }

        #endregion

        #region Constructor Region

        public FormNewLevel()
        {
            InitializeComponent();

            btnOK.Click += new EventHandler(btnOK_Click);
            btnCancel.Click += new EventHandler(btnCancel_Click);
        }

        #endregion

        #region Button Event Handler Region

        void btnOK_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(tbLevelName.Text))
            {
                MessageBox.Show("You must enter a name for the level.", "Missing Level Name");
                return;
            }

            if (string.IsNullOrEmpty(tbMapName.Text))
            {
                MessageBox.Show("You must enter a name for the map of the level.", "Missing Map
Name");
                return;
            }

            int mapWidth = 0;
            int mapHeight = 0;

            if (!int.TryParse(mtbWidth.Text, out mapWidth) || mapWidth < 1)
            {
                MessageBox.Show("The width of the map must be greater than or equal to one.",
"Map Size Error");
                return;
            }

            if (!int.TryParse(mtbHeight.Text, out mapHeight) || mapHeight < 1)
            {
                MessageBox.Show("The height of the map must be greater than or equal to one.",
"Map Size Error");
```

```
                return;
            }

            levelData = new RpgLibrary.WorldClasses.LevelData(
                tbLevelName.Text,
                tbMapName.Text,
                mapWidth,
                mapHeight,
                new List<string>(),
                new List<string>(),
                new List<string>());

            okPressed = true;
            this.Close();
        }

        void btnCancel_Click(object sender, EventArgs e)
        {
            okPressed = false;
            this.Close();
        }

        #endregion

    }
}
```

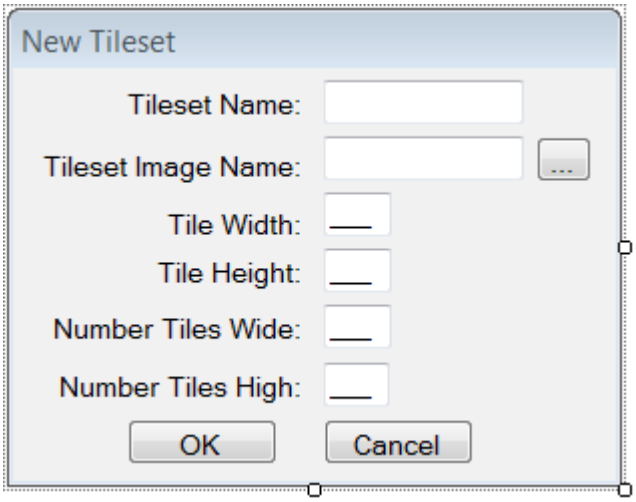
There is a using statement to bring the **LevelData** class that I added into scope. There are two fields in this class and properties to read their values, but not write them. The first, **okPressed**, determines how the form was closed, by the OK button or the Cancel button. The second, **LevelData**, holds the information off the form if the form was closed successfully. The constructor wires event handlers for the **Click** events of the buttons.

In the handler for the **Click** event for **btnOK** I validate the form. If the **Text** property of **tbLevelName** is null or empty I display a message box stating that a name must be entered for the level and exit the method. Similarly, I display a message and exit if the **Text** property of **tbMapName** is null or empty. There are then two local variables that will hold the conversion of the **Text** property of the **Masked Text Boxes** into integers. I use the **TrParse** method to attempt to parse the **Text** property of **mtbWidth**. If the return value is false the second half of the expression will not be evaluated. If the attempt was successful I then check if the result is less than 1. If either evaluate to true I display a message and exit the method. I do something similar for **mtbHeight** but work with height instead of width. You should really do better evaluation than the map width and height less than 1. If the form passed validation I create a new **LevelData** object using the **Text** properties of the two **Text Boxes**, the **mapWidth** and **mapHeight** local variables, and new **List<string>** instance for the **List<string>** parameters. I then set **okPressed** to true and close the form.

The handler for the **Click** event for **btnCancel** is much simpler. It just sets the **okPressed** field to false and then closes the form.

I want to create a form for creating **TilesetData** objects. Right click the **XLevelEditor**, select **Add** and then **Windows Form**. Name this new form **FormNewTileset**. My form looked like is on the next page.

There are quite a few controls on the form. There are six **Labels**, two **Text Boxes**, three **Buttons**, and four **Masked Text Boxes**. The controls are again positioned in rows of two columns, except for the row for selecting a tileset image name. This row has a third column, a **Button**, that is used to select the image file associated with the tileset.



I set several of the properties for the form itself, shown in the next table.

Property	Value
ControlBox	FALSE
FormBorderStyle	FixedDialog
Size	300, 240
StartPosition	CenterParent
Text	New Tileset

I'm going to do the controls like the last form a row at a time. The first controls to drag on are a **Label** and a **Text Box**. Set the following properties for those controls.

Label

Property	Value
(Name)	lblTilesetName
Location	54, 10
Text	Tileset Name:

Text Box

Property	Value
(Name)	tbTilesetName
Location	155, 5

For the next row you will need three controls, a **Label**, a **Text Box**, and a **Button**. Set these properties.

Label

Property	Value
(Name)	lblTilesetImageName

Location	12, 40
Text	Tileset Image Name:

Text Box

Property	Value
(Name)	tbTilesetImage
Enabled	FALSE
Location	155, 34

Button

Property	Value
(Name)	btnSelectImage
Location	261, 34
Size	28, 23
Text	...

The next four rows are sets of **Labels** and **Masked Text Boxes**. So, drag a **Label** and **Masked Text Box** onto the form and set the following properties.

Label

Property	Value
(Name)	lblTileWidth
Location	74, 69
Text	Tile Width:

Masked Text Box

Property	Value
(Name)	mtbTileWidth
Location	155, 62
Mask	000 (three zeros)
Size	34, 22

Drag another **Label** and **Masked Text Box** onto the form and set the following properties.

Label

Property	Value
(Name)	lblTileHeight
Location	74, 69
Text	Tile Height:

Masked Text Box

Property	Value
(Name)	mtbTileHeight
Location	155, 90
Mask	000 (three zeros)
Size	34, 22

Drag another **Label** and **Masked Text Box** onto the form and set the following properties.

Label

Property	Value
(Name)	lblTilesWide
Location	17, 121
Text	Number Tiles Wide:

Masked Text Box

Property	Value
(Name)	mtbTilesWide
Location	155, 118
Mask	000 (three zeros)
Size	34, 22

Drag another **Label** and **Masked Text Box** onto the form and set the following properties.

Label

Property	Value
(Name)	lblTilesHigh
Location	20, 150
Text	Number Tiles High:

Masked Text Box

Property	Value
(Name)	mtbTilesHigh
Location	155, 147
Mask	000 (three zeros)
Size	34, 22

The last two controls to add are two **Button** controls. Add them and set these properties.

(Name)	Location	Size	Text
btnOK	13, 142	75, 23	OK
btnCancel	106, 142	75, 23	Cancel

Now I'm going to add some code to **FormNewTileset**. Right click **FormNewTileset** in the solution explorer and select **View Code**. Change the code to the following.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

using RpgLibrary.WorldClasses;

namespace XLevelEditor
{
    public partial class FormNewTileset : Form
    {
        #region Field Region

        bool okPressed;

        TilesetData tilesetData;

        #endregion

        #region Property Region

        public TilesetData TilesetData
        {
            get { return tilesetData; }
        }

        public bool OKPressed
        {
            get { return okPressed; }
        }

        #endregion

        #region Constructor Region

        public FormNewTileset()
        {
            InitializeComponent();

            btnSelectImage.Click += new EventHandler(btnSelectImage_Click);
            btnOK.Click += new EventHandler(btnOK_Click);
            btnCancel.Click += new EventHandler(btnCancel_Click);
        }

        #endregion

        #region Button Event Handler Region

        void btnSelectImage_Click(object sender, EventArgs e)
        {
            OpenFileDialog ofDialog = new OpenFileDialog();
            ofDialog.Filter = "Image Files|*.BMP;*.GIF;*.JPG;*.TGA;*.PNG";
            ofDialog.CheckFileExists = true;
```

```
            ofDialog.CheckPathExists = true;
            ofDialog.Multiselect = false;

            DialogResult result = ofDialog.ShowDialog();

            if (result == DialogResult.OK)
            {
                tbTilesetImage.Text = ofDialog.FileName;
            }
        }

        void btnOK_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(tbTilesetName.Text))
            {
                MessageBox.Show("You must enter a name for the tileset.");
                return;
            }

            if (string.IsNullOrEmpty(tbTilesetImage.Text))
            {
                MessageBox.Show("You must select an image for the tileset.");
                return;
            }

            int tileWidth = 0;
            int tileHeight = 0;
            int tilesWide = 0;
            int tilesHigh = 0;

            if (!int.TryParse(mtbTileWidth.Text, out tileWidth))
            {
                MessageBox.Show("Tile width must be an integer value.");
                return;
            }
            else if (tileWidth < 0)
            {
                MessageBox.Show("Tile width must me greater than zero.");
                return;
            }

            if (!int.TryParse(mtbTileHeight.Text, out tileHeight))
            {
                MessageBox.Show("Tile height must be an integer value.");
                return;
            }
            else if (tileHeight < 0)
            {
                MessageBox.Show("Tile height must be greater than zero.");
                return;
            }

            if (!int.TryParse(mtbTilesWide.Text, out tilesWide))
            {
                MessageBox.Show("Tiles wide must be an integer value.");
                return;
            }
            else if (tilesWide < 0)
            {
                MessageBox.Show("Tiles wide must me greater than zero.");
                return;
            }

            if (!int.TryParse(mtbTilesHigh.Text, out tilesHigh))
            {
                MessageBox.Show("Tiles high must be an integer value.");
                return;
            }
            else if (tilesHigh < 0)
            {
```

```
                MessageBox.Show("Tiles high must be greater than zero.");
                return;
            }

            tilesetData = new TilesetData();

            tilesetData.TilesetName = tbTilesetName.Text;
            tilesetData.TilesetImageName = tbTilesetImage.Text;
            tilesetData.TileWidthInPixels = tileWidth;
            tilesetData.TileHeightInPixels = tileHeight;
            tilesetData.TilesWide = tilesWide;
            tilesetData.TilesHigh = tilesHigh;

            okPressed = true;
            this.Close();
        }

        void btnCancel_Click(object sender, EventArgs e)
        {
            okPressed = false;
            this.Close();
        }

        #endregion
    }
}
```

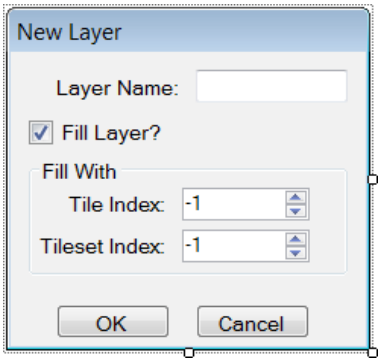
There is a using statement to bring the **TilesetData** class into scope. The first field, **okPressed**, and is a bool that will hold how the form was closed. The other field, **tilesetData**, is of type **TilesetData** will hold an object if the form is successfully closed using the OK button. There are properties to expose the values of the fields as read only, or get only. The constructor wires event handlers for the **Click** event of all three buttons.

In the handler for **btnSelectImage** I create a **OpenFileDialog** object. I set the **Filter** property so that the dialog will filter will show common image files. I set the **CheckFileExsists** and **CheckPathExists** properties to true so that we are sure the file exists. I also set **Multiselect** to false so that the user can only select one file. I capture the result of the **ShowDialog** method and compare it to **OK**. If the dialog was closed using the **OK** button I set the **Text** property of **tbTilesetImage** to the **FileName** property of the dialog.

The handler for **btnOK** does a lot of validation on the values on the form. If the **Text** property of either **tbTilesetName** or **tbTilesetImage** is null or empty I display a message box and exit the method. There are four local variables to hold the result of converting the **Text** property of the **Masked Text Boxes** to integers. I use the **TryParse** method of the **int** class try and convert the **Text** properties. If the conversion fails I display a message saying the value must be an integer value and exit the method. If it succeeded I check to see if the value is less than 1. If it is less than 1 I display a message saying the value must be greater than zero and exit the method. If the data on the form is valid I create a **TilesetData** object and set the fields to the values from the form. I set the **okPressed** field to true and then close the form. The handler for **btnCancel** sets the **okPressed** field to false and closes the form.

The last form that I want to create is a form for making new layers of the map. Right click the **XLevelEditor** project in the solution explorer, select **Add** and then **Windows Form**. Name this new form **FormNewLayer**. My finished form looked like is on the next page. Set the properties in the table on the next page for the form.

Property	Value
ControlBox	FALSE
FormBorderStyle	FixedDialog
Size	240, 230
StartPosition	CenterParent
Text	New Layer



There are a few controls on the form. There is a **Label** and a **Text Box** that are for getting the name of the layer. There is a **Check Box** that if checked the layer will be filled with a specific tile and tileset. There is a **Group Box** that I placed a pair of **Label** and **Numeric Up And Down** rows. The last controls are buttons for closing the form. The way the form will work is that if the **Check Box** is checked the map will be filled using the values from the **Numeric Up And Down** controls for the tile index and tileset. Remember when you are drawing a layer that if a tile index or tileset index are -1 that tile is skipped. So for the **Numeric Up And Down** controls I set the minimum value to -1 and the starting value to -1 as well. I also set the maximum value to 512. You may want it higher depending on the number of tiles in your tileset. It seemed to be a logical choice to me.

The first controls to drag onto the form are a **Label** and **Text Box**. Set the following properties for the controls.

Label

Property	Value
(Name)	lblLayerName
Location	27, 15
Text	Layer Name:

Text Box

Property	Value
(Name)	tbLayerName
Location	122, 12

Next drag a **Check Box** onto the form and set these properties.

Check Box

Property	Value
(Name)	cbFill
Checked	TRUE
Location	12, 44
Text	Fill Layer?

The next control to add is a **Group Box**. Set the following properties for the **Group Box**.

Group Box

Property	Value
(Name)	gbFillLayer
Location	12, 71
Size	210, 79
Text	Fill With

Onto the **Group Box** drag a **Label** and a **Numeric Up Down**. Set the following properties for those controls.

Label

Property	Value
(Name)	lblTileIndex
Location	23, 22
Text	**Tile Index**:

Numeric Up Down

Property	Value
(Name)	nudTile
Location	101, 20
Maximum	512
Minimum	-1
Size	84, 22
Value	-1

Drag a second **Label** and **Numeric Up Down** onto the form and set these properties.

Label

Property	Value
(Name)	lblTilesetIndex
Location	4, 50
Text	**Tileset Index**:

Numeric Up Down

Property	Value
(Name)	nudTileset
Location	101, 48

Maximum	512
Minimum	-1
Size	84, 22
Value	-1

The last two controls are the **Button** controls. Drag the two **Buttons** onto the form and set the following values.

Buttons

(Name)	Location	Size	Text
btnOK	30, 169	75, 23	OK
btnCancel	122, 169	75, 23	Cancel

Before I get to the code for this form I want to add another public constructor to the **MapLayerData** class. I will be adding in two more parameters. One will be a tile index and the other will be a tilset index. In that constructor I will fill the layer with tiles using the tile index and tileset index. Add the following constructor to the **MapLayerData** class.

```
public MapLayerData(string mapLayerName, int width, int height, int tileIndex, int tileSet)
{
    MapLayerName = mapLayerName;
    Width = width;
    Height = height;

    Layer = new Tile[height * width];

    Tile tile = new Tile(tileIndex, tileSet);

    for (int y = 0; y < height; y++)
        for (int x = 0; x < width; x++)
            SetTile(x, y, tile);
}
```

There is a **Tile** variable called **tile** that is created using the parameters passed in. It is safe to do this because the **Tile** is a structure and not a class. That makes it a value type and not a reference type. So if you modify one of the **Tile** objects in the array it will not affect the others. There is then a set of nested loops added to this constructor. The outer loop is for the Y coordinate and the inner loops is for the X coordinate. I call the **SetTile** method passing in **x**, **y**, and the **Tile** I created before.

I want to extend the **MapLayer** class in the **XRpgLibrary** to include a static method that will return a **MapLayer** from a **MapLayerData**. Add the following using statement and method to the **MapLayer** class in the **XRpgLibrary**.

```
using RpgLibrary.WorldClasses;

public static MapLayer FromMapLayerData(MapLayerData data)
{
    MapLayer layer = new MapLayer(data.Width, data.Height);

    for (int y = 0; y < data.Height; y++)
        for (int x = 0; x < data.Width; x++)
        {
            layer.SetTile(
```

```
                x,
                y,
                data.GetTile(x, y).TileIndex,
                data.GetTile(x, y).TileSetIndex);
        }

    return layer;
}
```

The new method takes a **MapLayerData** parameter that is the object you want to convert. Inside the method I create a new **MapLayer** instance using the **Height** and **Width** properties of the object passed in. There is then a set of nested for loops. The outer loop will loop through the height of the layer and the inner array will loop through the width of the layer. Inside the loop I use the **SetTile** method of the **MapLayer** class to set the tile passing in values from the **GetTile** method of **MapLayerData** to get the tile at the (x, y) coordinates from the loop variables.

I'm now going to add the code for **FormNewLayer**. Right click **FormNewLayer** in the **XLevelEditor** and select **View Code**. Change the code to the following.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using RpgLibrary.WorldClasses;

namespace XLevelEditor
{
    public partial class FormNewLayer : Form
    {
        #region Field Region

        bool okPressed;

        int LayerWidth;
        int LayerHeight;

        MapLayerData mapLayerData;

        #endregion

        #region Property Region

        public bool OKPressed
        {
            get { return okPressed; }
        }

        public MapLayerData MapLayerData
        {
            get { return mapLayerData; }
        }

        #endregion

        #region Constructor Region

        public FormNewLayer(int width, int height)
        {
            InitializeComponent();

            LayerWidth = width;
```

```
            LayerHeight = height;

            btnOK.Click += new EventHandler(btnOK_Click);
            btnCancel.Click += new EventHandler(btnCancel_Click);
        }

        #endregion

        #region Button Event Handler Region

        void btnOK_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(tbLayerName.Text))
            {
                MessageBox.Show("You must enter a name for the layer.", "Missing Layer Name");
                return;
            }

            if (cbFill.Checked)
            {
                mapLayerData = new MapLayerData(
                    tbLayerName.Text,
                    LayerWidth,
                    LayerHeight,
                    (int)nudTile.Value,
                    (int)nudTileset.Value);
            }
            else
            {
                mapLayerData = new MapLayerData(
                    tbLayerName.Text,
                    LayerWidth,
                    LayerHeight);
            }

            okPressed = true;
            this.Close();
        }

        void btnCancel_Click(object sender, EventArgs e)
        {
            okPressed = false;
            this.Close();
        }

        #endregion

    }
}
```

There is a using statement to bring the **WorldClasses** from the **RpgLibrary** into scope, namely the **MapLayerData** class. There are four fields in the class. The first, **okPressed**, will be used to determine how the form closed. Then next two, **LayerWidth** and **LayerHeight**, are the width and height of the map. I'm not allowing layers of varying sizes so all layers should have the same height and width. The last field is a **MapLayerData** object. If the **OK** button is pressed I will set it to a new **MapLayerData** object. There are properties to expose the **okPressed** and **mapLayerData** fields as get only.

Forms are just classes and because they are just classes you can create new constructors for them to get values needed by the form to the form. I changed the constructor to take two integer parameters, the width and height of the map. In the constructor I set the **LayerWidth** and **LayerHeight** fields with the values passed in. I then wire the **Click** event for both of the buttons.

In the event handler for the **Click** event of **btnOK** I check to see if the **Text** property of **tbLayerName**

is null or empty. If it is I display a message box saying that a name for the layer is required and exit the method. I then check the **Checked** property of **cbFill** to see if the map should be filled with a specific tile. If that is true I create a new map using the **Text** property of **tbLayerName**, the **LayerWidth** and **LayerHeight** fields, and the **Value** property of **nudTile** and **nudTileset** cast to an integer. The **Value** property of the **Numeric Up and Down** control is a decimal so you need to convert it to an integer. If it wasn't clicked I create a new layer using the **Text** property of **tbLayerName** and the **LayerWidth** and **LayerHeight** fields. I then set **okPressed** to true and close the form.

The code for the **Click** event of **btnCancel** you've seen several times now. I set **okPressed** to false and close the form.

I'm going to stop this tutorial here and continue it in a B part. The plan for this tutorial was to get started with the level editor and we are well under way. I encourage you to visit the news page of my site, XNA Game Programming Adventures, for the latest news on my tutorials.

Good luck in your game programming adventures!

Jamie McMahon