XNA 4.0 RPG Tutorials

Part 11a

Game Editors

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.

I had started a new tutorial that I was adding in editors for the game. I realized when I was about half done that I needed to revamp a few things. They are minor things but changing them now means that they won't have to be changed down the road. After changing a few things I will then work on the editors.

First thing I think is it will be better to be able to read in character classes at run time rather than have static character classes. The ability is already there, I just need to make a few quick adjustments. First, right click the **Fighter**, **Thief**, **Priest**, and **Wizard** classes in the **CharacterClasses** folder of the **RpgLibrary** project and select **Delete**. You are not going to need them. I am going to add a manager class to the **RpgLibrary** to manage the entity types. Right click the **RpgLibrary**, select **Add** and then **Class**. Name this class **EntityDataManager**. This is the code for the **EntityDataManager** class.

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

namespace RpgLibrary.CharacterClasses
{
    public class EntityDataManager
    {
        #region Field Region

        readonly Dictionary<string, EntityData> entityData = new Dictionary<string,
EntityData>();

        #endregion

        #region Property Region

        public Dictionary<string, EntityData> EntityData
        {
            get { return entityData; }
        }

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion
    }
```

```
}
```

A fairly straight forward class. There is one field, **entityData**, that is marked readonly to hold all of the **EntityData** objects in the game. This modifier is like the **const** modifier. The difference is you can assign it a value in the class declaration or in the constructor of the class. This doesn't mean that the field can't be modified though. It just means that there can't be an assignment to the field. You can access the field using methods and properties, like adding an item to the list using the **Add** method. The **entityData** field is a **Dictionary<string, EntityData>**. I used a dictionary as **EntityData** instances should have unique names, as they are names for character classes. There is a property to expose the field.

I'm going to make an update to the **EntityData** class. I'm going to remove the static methods that I added and add in an override of the **ToString** method. I'm also going to add in a **Clone** method to clone an **EntityData** class. The way I'm going to do the editors you are not going to need the static methods. I'm going to do it in such a way that content will be written to an XML file. You can then read in this XML file using the **Content Pipeline** and reflection. Change the **EntityData** class to the following.

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

namespace RpgLibrary.CharacterClasses
{
    public class EntityData
    {
        #region Field Region

        public string EntityName;

        public int Strength;
        public int Dexterity;
        public int Cunning;
        public int Willpower;
        public int Magic;
        public int Constitution;

        public string HealthFormula;
        public string StaminaFormula;
        public string MagicFormula;

        #endregion

        #region Constructor Region

        private EntityData()
        {
        }

        public EntityData(
            string entityName,
            int strength,
            int dexterity,
            int cunning,
            int willpower,
            int magic,
            int constitution,
            string health,
            string stamina,
            string mana)
        {
            EntityName = entityName;
            Strength = strength;
```

```
            Dexterity = dexterity;
            Cunning = cunning;
            Willpower = willpower;
            Cunning = cunning;
            Willpower = willpower;
            Magic = magic;
            Constitution = constitution;
            HealthFormula = health;
            StaminaFormula = stamina;
            MagicFormula = mana;
        }

        #endregion

        #region Method Region

        public override string ToString()
        {
            string toString = "Name = " + EntityName + ", ";
            toString += "Strength = " + Strength.ToString() + ", ";
            toString += "Dexterity = " + Dexterity.ToString() + ", ";
            toString += "Cunning = " + Cunning.ToString() + ", ";
            toString += "Willpower = " + Willpower.ToString() + ", ";
            toString += "Magic = " + Magic.ToString() + ", ";
            toString += "Constitution = " + Constitution.ToString() + ", ";
            toString += "Health Formula = " + HealthFormula + ", ";
            toString += "Stamina Formula = " + StaminaFormula + ", ";
            toString += "Magic Formula = " + MagicFormula;

            return toString;
        }

        public object Clone()
        {
            EntityData data = new EntityData();

            data.EntityName = this.EntityName;
            data.Strength = this.Strength;
            data.Dexterity = this.Dexterity;
            data.Cunning = this.Cunning;
            data.Willpower = this.Willpower;
            data.Magic = this.Magic;
            data.Constitution = this.Constitution;
            data.HealthFormula = this.HealthFormula;
            data.StaminaFormula = this.StaminaFormula;
            data.MagicFormula = this.MagicFormula;

            return data;
        }

        #endregion
    }
}
```

The **ToString** and an equals sign, another space and then the field. For all fields except the last I also add a comma and a space. I didn't implement the **ICloneable** interface this time. You will also want to change your **Animation** class to not use the **ICloneable** interface. In the .NET Compact version, which the Xbox 360 uses, you will get a compile time error that **ICloneable** is not available due to its protection level. You will still want to have a **Clone** method though, just don't use the interface to do it. The **Clone** method uses the private constructor to create a new instance of **EntityData**. It then sets all of the fields of the instance using the current object.

Before I get to the editors I want to update the **Entity** class. Instead of it being an abstract class I want it to be a sealed class. A sealed class is a class that can't be inherited from. I also want to add in an enumeration for different types of entities. Even though there is a lot of code I don't think there is

anything that truly needs to be explained. Things that were protected were made private. A few fields were added and properties to expose them. The public constructor now takes a string for the name, an **EntityData** that defines the type of entity, an **EntityGender** for the gender of the entity, and an **EntityType** for the type of entity. When I talk about types of entities there will be four distinct groups. There are characters, like the player's characters, NPCs for the player to interact with, monsters and creatures. I could have lumped all enemies the player will face into one category, monster, but I though there may be instances where it will be useful to have animals, like giant spiders or wolves. This is the updated **Entity** class.

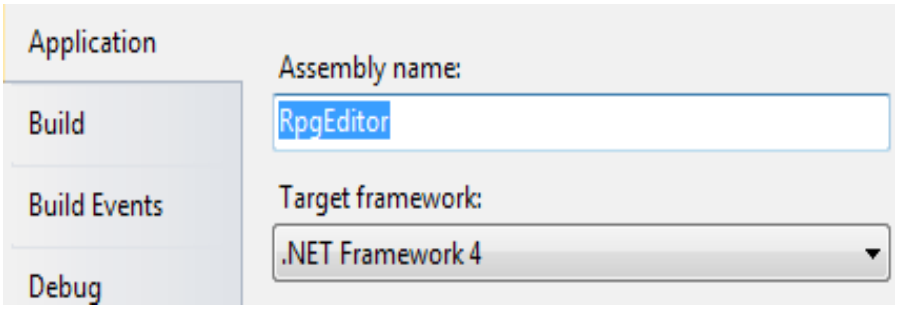
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RpgLibrary.CharacterClasses
{
    public enum EntityGender { Male, Female, Unknown }
    public enum EntityType { Character, NPC, Monster, Creature }

    public sealed class Entity
    {
        #region Vital Field and Property Region

        private string entityName;
        private string entityClass;
        private EntityType entityType;
        private EntityGender gender;

        public string EntityName
        {
            get { return entityName; }
            private set { entityName = value; }
        }

        public string EntityClass
        {
            get { return entityClass; }
            private set { entityClass = value; }
        }

        public EntityType EntityType
        {
            get { return entityType; }
            private set { entityType = value; }
        }

        public EntityGender Gender
        {
            get { return gender; }
            private set { gender = value; }
        }

        #endregion

        #region Basic Attribute and Property Region

        private int strength;
        private int dexterity;
        private int cunning;
        private int willpower;
        private int magic;
        private int constitution;

        private int strengthModifier;
        private int dexterityModifier;
        private int cunningModifier;
```

```
        private int willpowerModifier;
        private int magicModifier;
        private int constitutionModifier;

        public int Strength
        {
            get { return strength + strengthModifier; }
            private set { strength = value; }
        }

        public int Dexterity
        {
            get { return dexterity + dexterityModifier; }
            private set { dexterity = value; }
        }

        public int Cunning
        {
            get { return cunning + cunningModifier; }
            private set { cunning = value; }
        }


        public int Willpower
        {
            get { return willpower + willpowerModifier; }
            private set { willpower = value; }
        }

        public int Magic
        {
            get { return magic + magicModifier; }
            private set { magic = value; }
        }

        public int Constitution
        {
            get { return constitution + constitutionModifier; }
            private set { constitution = value; }
        }

        #endregion

        #region Calculated Attribute Field and Property Region

        private AttributePair health;
        private AttributePair stamina;
        private AttributePair mana;

        public AttributePair Health
        {
            get { return health; }
        }

        public AttributePair Stamina
        {
            get { return stamina; }
        }

        public AttributePair Mana
        {
            get { return mana; }
        }

        private int attack;
        private int damage;
        private int defense;

        #endregion
```

```
        #region Level Field and Property Region

        private int level;
        private long experience;

        public int Level
        {
            get { return level; }
            private set { level = value; }
        }

        public long Experience
        {
            get { return experience; }
            private set { experience = value; }
        }


        #endregion

        #region Constructor Region

        private Entity()
        {
            Strength = 0;
            Dexterity = 0;
            Cunning = 0;
            Willpower = 0;
            Magic = 0;
            Constitution = 0;

            health = new AttributePair(0);
            stamina = new AttributePair(0);
            mana = new AttributePair(0);
        }

        public Entity(string name, EntityData entityData, EntityGender gender, EntityType
entityType)
        {
            EntityName = name;
            EntityClass = entityData.EntityName;
            Gender = gender;
            EntityType = entityType;
            Strength = entityData.Strength;
            Dexterity = entityData.Dexterity;
            Cunning = entityData.Cunning;
            Willpower = entityData.Willpower;
            Magic = entityData.Magic;
            Constitution = entityData.Constitution;

            health = new AttributePair(0);
            stamina = new AttributePair(0);
            mana = new AttributePair(0);
        }

        #endregion
    }
}
```

The next step is to add in the editor. Make sure that all open files are saved for this step. The next step is to add a project for the editors. The editors will do the serializing and deserializing of your content. You can deserialize your objects from the editor so that you don't have to read in your files and parse them manually. Right click the solution in the solution explorer. Select **Add** and then **New Project**. Select a Windows Forms project from the C# list and name this new project **RpgEditor**. Now make sure all open files are saved. Right click the **RpgEditor** project and select **Properties** to bring up the

properties dialog for the project. Under the **Application** tab set the **Target framework** to **.NET Framework 4** and not **.NET Framework 4 Client Profile** like below. Right click the **RpgEditor** and select **Add Reference**. Find the **Microsoft.Xna.Framework.Content.Pipeline** reference and add it. Also add a reference to the **Microsoft.Xna.Framework** name space.



Now you need to reference your two libraries. Right click the **RpgEditor** project and select **Add Reference**. From the **Projects** tab select the **RpgLibrary** and **XRpgLibrary** entries. Right click the **Form1** class in the solution explorer and select **Delete** to rename. Rename it to **FormMain** and in the dialog box that pops up choose to rename the code references from **Form1** to **FormMain**. Right click the **RpgEditor** again, select **Add** and then **Class**. Name this new class **XnaSerializer**. The code for the class follows next.

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

using Microsoft.Xna.Framework.Content.Pipeline.Serialization.Intermediate;

namespace RpgEditor
{
    static class XnaSerializer
    {
        public static void Serialize<T>(string filename, T data)
        {
            XmlWriterSettings settings = new XmlWriterSettings();
            settings.Indent = true;

            using (XmlWriter writer = XmlWriter.Create(filename, settings))
            {
                IntermediateSerializer.Serialize<T>(writer, data, null);
            }
        }

        public static T Deserialize<T>(string filename)
        {
            T data;

            using (FileStream stream = new FileStream(filename, FileMode.Open))
            {
                using (XmlReader reader = XmlReader.Create(stream))
                {
                    data = IntermediateSerializer.Deserialize<T>(reader, null);
                }
            }

            return data;
        }
    }
}
```

That looks like a whole pile of ugly but once it is explained I don't think you will find it that bad. First off I added using statements for the **System.IO** and **System.Xml** name spaces because I use class from both of them. There is also a using statement for an XNA name space. That is the reason I added the reference to the **Content Pipeline**. I needed access to the **IntermediateSerializer** class. With this class you can write out XML content in a way that the **Content Pipeline** can read it in using reflection as I mentioned earlier. This class is a static class. You never need to create an instance of this class. You use the two static methods to serialize and deserialize objects. They are generic methods, like the **Load** method of the **ContentManager** class. You can specify what type you want to use. This will prevent the need to write methods to serialize and deserialize every class you want to write out and read in. You can just use the generic method and pass in the type you want to use. You will see it in action in the editor.

The first method, **Serialize<T>**, is used to serialize the object into an XML document. It basically writes out the object in XML format. There maybe times when you don't want a field or property written. You can flag it with an attribute that will prevent it from being serialized. An example would be if you had a **Texture2D** field. You don't want to serialize that, just your other data.

The first step in serializing using the **IntermediateSerializer** class is to create an **XmlWriterSettings** object. You use this object to set the **Indent** property to true as you want items indented. Unlike **using** directives for name spaces a **using** statement that you find inside code automatically disposes of any disposable objects with the statement ends. In this case the **XmlWriter** will be disposed when the statement ends. Inside the using statement is why I ended up deciding to create generic methods. The **Serialize** method of the **IntermediateSerializer** class requires a type, **T**, like the **Load** method of the **ContentManager**. I can use the **T** from the generic method in the call to **Serialize**. The parameters that I pass to the **Serialize** method are an XML stream, the **XmlWriter**, the object to be serialized, and a reference path. You can pass in null for this if you don't want to use one.

The **Deserialize<T>** method returns an object of type T. It first has a local variable of type T that it can return. To use the **XmlReader** class to read in an XML file you need a stream for input. In a using statement I create a stream to open the file passed in as a parameter. Inside that code block there is another using statement that creates an **XmlReader** to read in the XML file. I then call the **Deserialize** method of the **IntermediateSerializer** class to deserialize the document. I then return the object that was deserialized. It would be a good idea to check for exceptions such as the XML file was of the wrong type or the file didn't exist or the file could not be created. That would work well in the editor though and I will do that there.

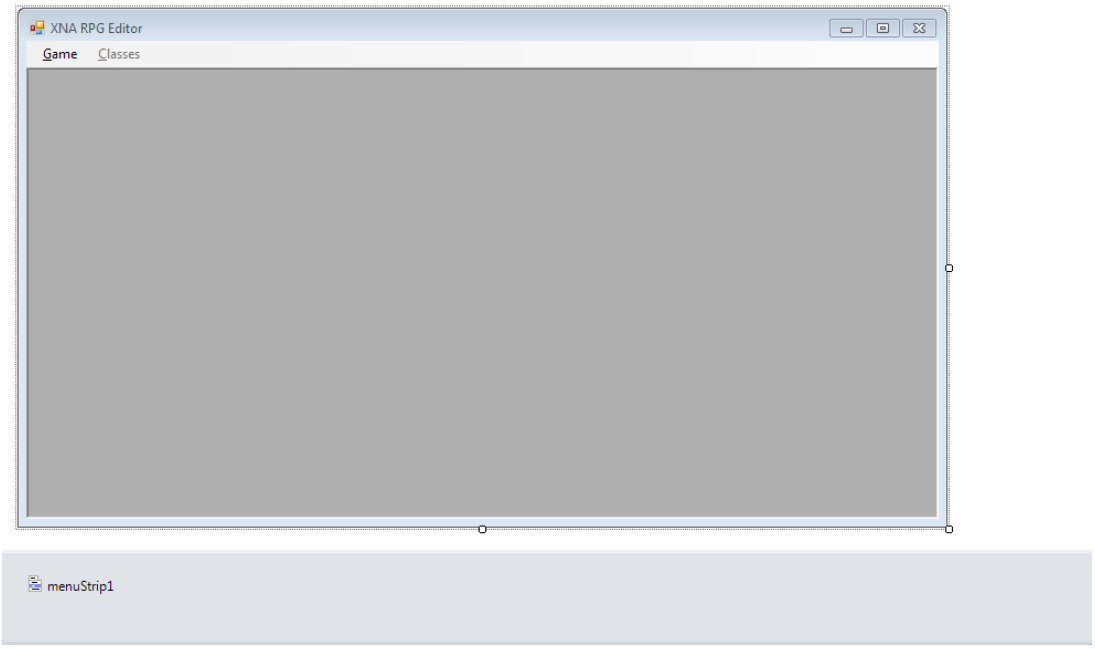
I want to add a class to the **RpgLibrary** before moving on to the editor. This class will describe your games. It is used more for the editor than anything. Right click the **RpgLibrary** project, select **Add** and then **Class**. Name this class **RolePlayingGame**. This is the code.

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

namespace RpgLibrary
{
    public class RolePlayingGame
    {
        #region Field Region
```

```
        string name;
        string description;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        public string Description
        {
            get { return description; }
            set { description = value; }
        }

        #endregion

        #region Constructor Region

        public RolePlayingGame(string name, string description)
        {
            Name = name;
            Description = description;
        }

        #endregion
    }
}
```

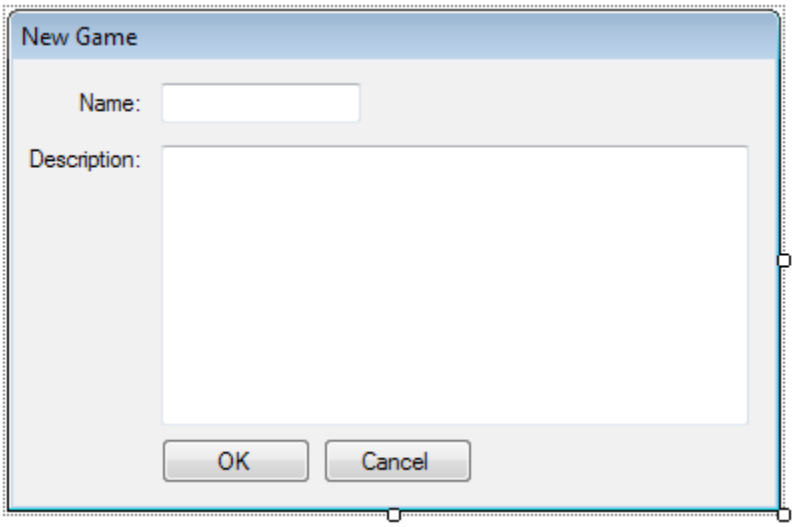
There are two fields in the class. One for the name of the game and the other for a description of the game. There are public properties to expose the fields. The constructor of the class takes a name and description and sets the fields using the properties. This class will be fleshed out more as we go.

Designing forms is never easily done using text. I will show you a picture of what my finished forms look like in Visual C# and what controls were added. Open the designer view of **FormMain** by right clicking it and selecting **View Designer**. This is what my finished form looked like in the designer.



I first made the form a little bigger. I then I dragged on was a **Menu Strip** control. In the first entry of the **Menu Strip** type **&Game**. The & before Game means that if the user presses ALT+G it will open that menu. Under the **Game** add in four other entries: **&New Game**, **&Open Game**, **&Save Game**, -, and **E&xit Game**. Entering the -, minus sign, gives you a separator in your menu. Beside **&Game** you want to add **&Classes**. Set the **Enabled** property for this item to false. You now want to set a few property of **FormMain**. Set the **IsMdiContainer** property to true, the **Text** property to **XNA Rpg Editor**, and the **StartUpPosition** to **CenterScreen**. MDI is a way of having multiple children forms of a parent form. This way you can have multiple forms open like the form for creating character classes and a form for creating weapons and switch back and forth between them easily.

The next form I want to create is a form for creating a new game. Right click the **RpgEditor** project, select **Add** and then **Windows Form**. Name this new form **FormNewGame**. This is what my **FormNewGame** looked like in the designer.



First, make the form a little bigger to house all of the controls. Drag on a **Label** and set its **Text** property to **Name:**. Dragon on a second **Label** and set its **Text** property to **Description:**. Drag on a **Text Box** and position it beside the **Name: Label**. Set its **Name** property to **tbName**. Drag a second **Text Box** onto the form. Set its **Name** property to **tbDescription** and its **Multiline** property to true. Make it a little bigger to give the user more of an area to work with. Drag two buttons onto the form and position them under **tbDescription**. Set the **Name** of the first to **btnOK**, the **Text** property to **OK** and the **DialogResult** property to **OK**. Set the **Name** of the second to **btnCancel**, **Text** to **Cancel**, and **DialogResult** to **Cancel**. On the form itself set the **Text** property to **New Game**, **Control Box** to false, and **FormBorderStyle** to **FixedDialog**.

While this form is open might as well code the logic for it. Right click **FormNewGame** in the solution explorer and select **View Code**. Change the code for **FormNewGame** to the following.

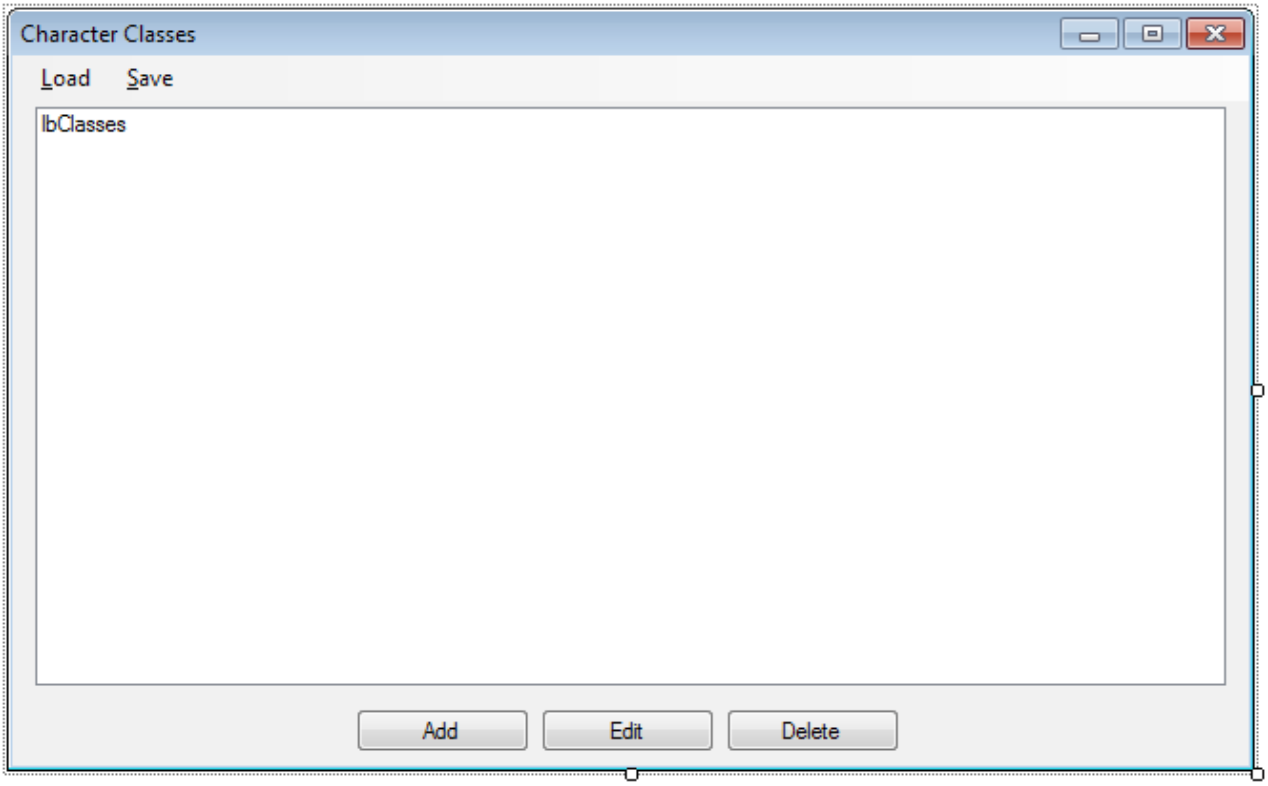
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

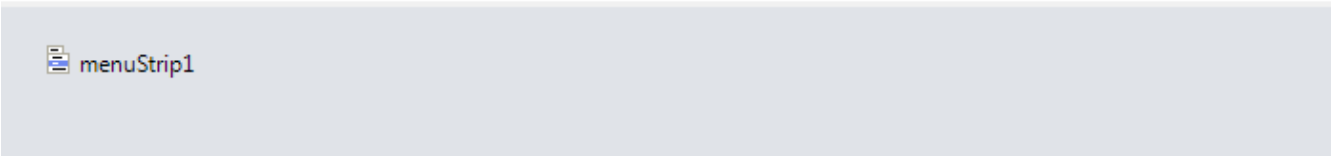
```
using RpgLibrary;

namespace RpgEditor
{
    public partial class FormNewGame : Form
    {
        #region Field Region

        RolePlayingGame rpg;

        #endregion

        #region Property Region

        public RolePlayingGame RolePlayingGame
        {
            get { return rpg; }
        }

        #endregion

        #region Constructor Region

        public FormNewGame()
        {
            InitializeComponent();
            btnOK.Click += new EventHandler(btnOK_Click);
        }

        #endregion

        #region Event Handler Region

        void btnOK_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(tbName.Text) || string.IsNullOrEmpty(tbDescription.Text))
            {
                MessageBox.Show("You must enter a name and a description.", "Error");
                return;
            }

            rpg = new RolePlayingGame(tbName.Text, tbDescription.Text);

            this.Close();
        }

        #endregion
    }
}
```

There is a field of type **RolePlayingGame** to be constructed using the form and a public property to expose it as read only, get only. The constructor wires an event handler for the **Click** event of **btnOK**. The event handler checks to see if **tbName** or **tbDescription** have text in them. If they don't a message box is shown and the method exits. Because I set the **DialogResult** of **btnOK** to OK the form will close when **btnOK** is pressed. If both text boxes had values then I create a new **RolePlayingGame** object with their values and then close the form.

I'm going to add two forms for dealing with entity data, or character classes. The one form holds all of the character classes in the game. The second is use for creating new classes and editing character classes. Right click the **RpgEditor** project, select **Add** and then **Windows Form**. Name this new form **FormClasses**. What my form looked like in the designer is on the next page. To get started drag a **Menu Strip** onto the form and making the form a little bigger. You can set the **Text** property of the form to **Character Classes**. For the **Menu Strip** you can set the first item to **&Load** and the second

item to **&Save**. Drag a **List Box** onto form and size it similarly to mine. You can also drag three **Buttons** onto the form positioning them under the **List Box**.



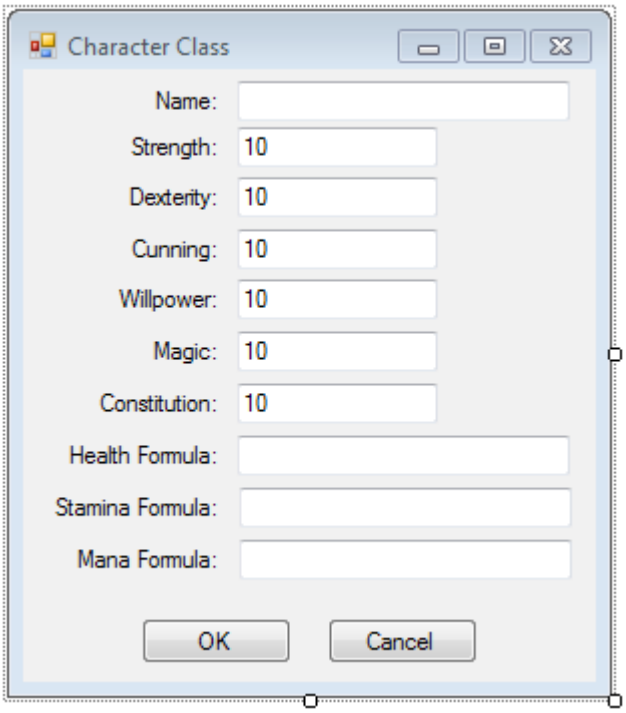


For the **List Box** set the following properties. **Name** is **lbClasses**, and **Anchor** property to **Top**, **Bottom**, **Left**, **Right**. Setting the **Anchor** property to that will have the list box change its size as you change the size of the form. For the first button set its **Name** property to **btnAdd**, **Anchor** property to **Bottom**, and its **Text** property to **Add**. The second button's **Name**, **Anchor**, and **Text** properties are **btnEdit**, **Bottom**, and **Edit** respectively. Set those same properties for the last to **btnDelete**, **Bottom**, and **Delete**.

The last form I'm going to add is the form for creating a new character class or editing an existing one. Right click the **RpgEditor** project, select **Add** and then **Windows Form**. Name this new form **FormEntityData**. What my form looked like is on the next page.

Make the form bigger so there is room for all of the controls. I then dragged 10 **Label** controls onto the form. Set the text properties of the labels to **Name:**, **Strength:**, **Dexterity:**, **Cunning:**, **Willpower:**, **Magic:**, **Constitution:**, **Health Formula:**, **Stamina Formula:**, and **Mana Formula:**. You will then

drag a **Text Box** onto the form and position it beside the **Name:** label. Make it a little bigger and set the **Name** property to **tbName**.



I next dragged a **Masked Text Box** onto the form. Set the **Mask** property to 00 which means the text box will only accept 2 digits. Set the **Text** property to 10 as well. Instead of having to set those properties 5 more times you can replicate the control. Select the control and while holding the **Ctrl** key drag the **Masked Text Box** below the other. Repeat the process until there is a masked text box beside the other attributes. Once you have all six set their **Name** properties to match the text of the label that they are across from: **mtbStrength**, **mtbDexterity**, **mtbCunning**, **mtbWillpower**, **mtbMagic**, and **mtbConstitution**.

You will want to drag a **Text Box** beside the remaining labels. Name them according to the label they are beside: **tbHealth**, **tbStamina**, and **tbMana**. The last two controls to drag onto the form are the buttons. Set the **Name** of the one to **btnOK**, and the **Text** property to **OK**. For the second button set the **Name** to **btnCancel**, the text to **Cancel**, and the **DialogResult** to **Cancel**. Click on the title bar of the form to bring up the properties of the form. Set the **AcceptButton** property to **btnOK**, the **CancelButton** property to **btnCancel**, **FormBorderStyle** to **FixedDialog**, and the **Text** property to **Character Class**.

I'm going to add in the logic for the form as there isn't a lot of code to it. Right click **FormEntityData** and select **View Code**. This is the code for that form.

```
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.CharacterClasses;
```

```
namespace RpgEditor
{
    public partial class FormEntityData : Form
    {
        #region Field Region

        EntityData entityData = null;

        #endregion

        #region Property Region

        public EntityData EntityData
        {
            get { return entityData; }
            set { entityData = value; }
        }

        #endregion

        #region Constructor Region

        public FormEntityData()
        {
            InitializeComponent();
            this.Load += new EventHandler(FormEntityData_Load);
            btnOK.Click += new EventHandler(btnOK_Click);
            btnCancel.Click += new EventHandler(btnCancel_Click);
        }

        #endregion

        #region Event Handler Region

        void FormEntityData_Load(object sender, EventArgs e)
        {
            if (entityData != null)
            {
                tbName.Text = entityData.EntityName;
                mtbStrength.Text = entityData.Strength.ToString();
                mtbDexterity.Text = entityData.Dexterity.ToString();
                mtbCunning.Text = entityData.Cunning.ToString();
                mtbWillpower.Text = entityData.Willpower.ToString();
                mtbConstitution.Text = entityData.Constitution.ToString();
                tbHealth.Text = entityData.HealthFormula;
                tbStamina.Text = entityData.StaminaFormula;
                tbMana.Text = entityData.MagicFormula;
            }
        }

        void btnOK_Click(object sender, EventArgs e)
        {
            if (string.IsNullOrEmpty(tbName.Text) || string.IsNullOrEmpty(tbHealth.Text) ||
                string.IsNullOrEmpty(tbStamina.Text) || string.IsNullOrEmpty(tbMana.Text))
            {
                MessageBox.Show("Name, Health Formula, Stamina Formula and Mana Formula must have
values.");
                return;
            }

            int str = 0;
            int dex = 0;
            int cun = 0;
            int wil = 0;
            int mag = 0;
            int con = 0;

            if (!int.TryParse(mtbStrength.Text, out str))
            {
```

```
                MessageBox.Show("Strength must be numeric.");
                return;
            }

            if (!int.TryParse(mtbDexterity.Text, out dex))
            {
                MessageBox.Show("Dexterity must be numeric.");
                return;
            }

            if (!int.TryParse(mtbCunning.Text, out cun))
            {
                MessageBox.Show("Cunning must be numeric.");
                return;
            }

            if (!int.TryParse(mtbWillpower.Text, out wil))
            {
                MessageBox.Show("Willpower must be numeric.");
                return;
            }

            if (!int.TryParse(mtbMagic.Text, out mag))
            {
                MessageBox.Show("Magic must be numeric.");
                return;
            }

            if (!int.TryParse(mtbConstitution.Text, out con))
            {
                MessageBox.Show("Constitution must be numeric.");
                return;
            }

            entityData = new EntityData(
                tbName.Text,
                str,
                dex,
                cun,
                wil,
                mag,
                con,
                tbHealth.Text,
                tbStamina.Text,
                tbHealth.Text);

            this.Close();
        }

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

        #endregion
    }
}
```

I added in a using statement for the **CharacterClasses** name space of the **RpgLibrary**. I have a field **entityData** of type **EntityData** for the entity data being entered. There is also a public property to expose it to other forms.

In the constructor of the form I wire event handlers for the **Load** event of the form and the **Click** events of **btnOK** and **btnCancel**. In the **Load** event of the form I check to see **entityData** is not null. If it is not null I fill out the text boxes and masked text boxes with the appropriate fields from **entityData**.

The click event of **btnOK** checks to make sure that the **Text** properties of **tbName**, **tbHealth**, **tbStamina**, and **tbMana** have values. If they don't I display a message box and exit the method. There are then six local integer variables for the stats of **EntityData**. Next follows a series of if statements where I use the **TryParse** method of the int class to parse the **Text** property of the masked text boxes for the stats. Even though the mask only allows numeric values they can be empty and if you just use the **Parse** method that will generate an exception and your program will crash. If any of the **TryParse** calls fail I display a message box and exit the method. I then set the **entityData** field to a new instance using the values of the text boxes and local integer variables. Finally I close the form. In the **Click** event handler for **btnCancel** I set the **entityData** field to null. I then close the form.

I'm going to add in some basic logic for the **FormClasses** now. Right click **FormClasses** in the **RpgEditor** project and select **View Code**. Add the following code for **FormClasses**.

```
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.CharacterClasses;

namespace RpgEditor
{
    public partial class FormClasses : Form
    {
        #region Field Region

        EntityDataManager entityDataManager = new EntityDataManager();

        #endregion

        #region Constructor Region

        public FormClasses()
        {
            InitializeComponent();

            loadToolStripMenuItem.Click += new EventHandler(loadToolStripMenuItem_Click);
            saveToolStripMenuItem.Click += new EventHandler(saveToolStripMenuItem_Click);

            btnAdd.Click += new EventHandler(btnAdd_Click);
            btnEdit.Click += new EventHandler(btnEdit_Click);
            btnDelete.Click += new EventHandler(btnDelete_Click);
        }

        #endregion

        #region Menu Item Event Handler Region

        void loadToolStripMenuItem_Click(object sender, EventArgs e)
        {
        }

        void saveToolStripMenuItem_Click(object sender, EventArgs e)
        {
        }

        #endregion

        #region Button Event Handler Region
```

```
        void btnAdd_Click(object sender, EventArgs e)
        {
            using (FormEntityData frmEntityData = new FormEntityData())
            {
                frmEntityData.ShowDialog();

                if (frmEntityData.EntityData != null)
                {
                    lbClasses.Items.Add(frmEntityData.EntityData.ToString());
                }
            }
        }

        void btnEdit_Click(object sender, EventArgs e)
        {
        }

        void btnDelete_Click(object sender, EventArgs e)
        {
        }

        #endregion
    }
}
```

There is an **EntityDataManager** field in this class because it will hold the different **EntityData** objects. In the constructor I wire the event handlers for the menu items and the buttons. I added some basic code to the **btnAdd_Click** event handler. There is a using statement that creates a new instance of **FormEntityData** so that when you are done with the form it will be disposed of. Inside I call the **ShowDialog** method of the form to display it as a modal form. That required the form to be closed before you can use the calling form again. I check to see if the **EntityData** property of the form is not null. If it is I add the **EntityData** object to the list box of **EntityData** objects. I will add in more functionality in another tutorial.

I'm also going to add a little logic to the **FromMain**. Right click **FormMain** and select **View Code**. Update the code for **FormMain** 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;
using RpgLibrary.CharacterClasses;
using RpgLibrary.ItemClasses;

namespace RpgEditor
{
    public partial class FormMain : Form
    {
        #region Field Region

        RolePlayingGame rolePlayingGame;
        FormClasses frmClasses;

        #endregion

        #region Property Region
        #endregion
```

```
        #region Constructor Region

        public FormMain()
        {
            InitializeComponent();

            newGameToolStripMenuItem.Click += new EventHandler(newGameToolStripMenuItem_Click);
            openGameToolStripMenuItem.Click += new EventHandler(openGameToolStripMenuItem_Click);
            saveGameToolStripMenuItem.Click += new EventHandler(saveGameToolStripMenuItem_Click);
            exitToolStripMenuItem.Click += new EventHandler(exitToolStripMenuItem_Click);

            classesToolStripMenuItem.Click += new EventHandler(classesToolStripMenuItem_Click);
        }

        #endregion

        #region Menu Item Event Handler Region

        void newGameToolStripMenuItem_Click(object sender, EventArgs e)
        {
            using (FormNewGame frmNewGame = new FormNewGame())
            {
                DialogResult result = frmNewGame.ShowDialog();

                if (result == DialogResult.OK && frmNewGame.RolePlayingGame != null)
                {
                    classesToolStripMenuItem.Enabled = true;
                    rolePlayingGame = frmNewGame.RolePlayingGame;
                }
            }
        }

        void openGameToolStripMenuItem_Click(object sender, EventArgs e)
        {
        }

        void saveGameToolStripMenuItem_Click(object sender, EventArgs e)
        {
        }

        void exitToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        void classesToolStripMenuItem_Click(object sender, EventArgs e)
        {
            if (frmClasses == null)
            {
                frmClasses = new FormClasses();
                frmClasses.MdiParent = this;
            }

            frmClasses.Show();
        }

        #endregion

        #region Method Region
        #endregion
    }
}
```

There is a field of type **RolePlayingGame** to hold the **RolePlayingGame** object associated with the editor. There is also a field **FormClasses** for the form that holds all of the **EntityData** objects. The constructor wires the event handlers for all of the menu items.

The handler for the **New Game** menu item has a using statement that creates an instance of **FormNewGame**. Inside the using statement I call the **ShowDialog** method of the form and capture the result in the variable **result**. If **result** is **OK**, meaning the form was closed by clicking **OK**, and the **RolePlayingGame** property of the form is not null I set the **Classes** menu item to **Enabled** and the **rolePlayingGame** field to the **RolePlayingGame** property of **FormNewGame**. In the **Exit** menu item handler I just call **Close** on the form to close the form. Later I will add in code to make sure that data is saved before closing the form.

The handler for the **Classes** menu item checks to see if **frmClasses** is null. If it is then it has not been created yet so I create it. I also set the **MdiParent** property of the form to be the current instance of **FormMain** using **this**. After creating the form if necessary I call the **Show** method rather than **ShowDialog**. This allows you to flip between child forms of the parent form easily.

I'm going to end this tutorial here as it is rather on the long side. I want to try and keep them to a reasonable length so that you don't have too much to digest at once. 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