XNA 4.0 RPG Tutorials

Part 11c

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.

This is the third part of the tutorial on adding a game editor to the project. This tutorial will be more about coding than designing forms. You will want to make the editor the start up project for the duration of this tutorial. Right click the **RpgEditor** project in the solution explorer and select **Set As StartUp Project**.

I want to make a quick change to the forms that hold all of a specific class in the game. To start remove all code from **FormWeapon**, **FormArmor**, **FormShield**, and **FormClasses** that has to do with menu items. For example the code for **FormWeapon** looks like this now.

```
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;

namespace RpgEditor
{

    public partial class FormWeapon : FormDetails
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public FormWeapon()
        {
            InitializeComponent();

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

        #endregion

        #region Button Event Handler Region

        void btnAdd_Click(object sender, EventArgs e)
        {
        }

        void btnEdit_Click(object sender, EventArgs e)
        {
        }

        void btnDelete_Click(object sender, EventArgs e)
        {
        }

        #endregion
    }
}
```

Go to the designer view of **FormDetails** by right clicking it and selecting **View Designer**. Right click on the **Menu Item Strip** and select **Delete**. Now click on the **List Box** and move its top border up a little.

You don't want the forms that inherit from **FormDetails** to close when the user tries to close the form. You only want to close them when the parent MDI form is closed. Luckily enough there is an event that you can wire and stop that from happening. Right click **FormDetails** in the solution explorer and select **View Code**. Change the **Constructor** region to the following and add this new region.

```
#region Constructor Region

public FormDetails()
{
    InitializeComponent();

    if (FormDetails.ItemManager == null)
        ItemManager = new ItemManager();

    if (FormDetails.EntityDataManager == null)
        EntityDataManager = new EntityDataManager();

    this.FormClosing += new FormClosingEventHandler(FormDetails_FormClosing);
}

#endregion

#region Event Handler Region

void FormDetails_FormClosing(object sender, FormClosingEventArgs e)
{
    if (e.CloseReason == CloseReason.UserClosing)
    {
        e.Cancel = true;
        this.Hide();
    }

    if (e.CloseReason == CloseReason.MdiFormClosing)
    {
        e.Cancel = false;
        this.Close();
    }
}

#endregion
```

The **FormClosing** event is an event that can be canceled. The **FormClosingEventArgs** argument has a property, **CloseReason**, that holds the reason why the form is being closed. If the value is **UserClosing**, the user tried to close the form, I set the **Cancel** property to true so the event will be canceled and I call the **Hide** method to hide the form. If the **CloseReason** is **MdiFormClosing** then the main form is closing and you want to close the form. I set the **Cancel** property to false and call the **Close** method of the form. Right click the **FormDetails** form again in the solution explorer and this time select **View Designer**. Set the **MinimizeBox** property to false so the user can't minimize the form.

When a menu item is selected you want to bring that form to the front. Lucky enough there is a method you can call to do just that, **BringToFront**. Right click **FormMain** in the solution explorer and select **View Code** to open the code for that form. Change the event handlers for the **Menu Item Event Handler** region to the following.

```
#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;
            itemsToolStripMenuItem.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();
    frmClasses.BringToFront();
}

void armorToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (frmArmor == null)
    {
        frmArmor = new FormArmor();
        frmArmor.MdiParent = this;
    }

    frmArmor.Show();
    frmArmor.BringToFront();
}

void shieldToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{
    if (frmShield == null)
    {
        frmShield = new FormShield();
        frmShield.MdiParent = this;
    }

    frmShield.Show();
    frmShield.BringToFront();
}

void weaponToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (frmWeapon == null)
    {
        frmWeapon = new FormWeapon();
        frmWeapon.MdiParent = this;
    }

    frmWeapon.Show();
    frmWeapon.BringToFront();
}

#endregion
```

The new code just calls the **BringToFront** method after the **Show** method so that form will be on top of all other forms. What I'm going to do next is handle creating a new game from the main form. I want to add a couple static fields and get only properties to expose their values. Also add a using statement for the **System.IO** name space.

```
using System.IO;

static string gamePath = "";
static string classPath = "";
static string itemPath = "";
```

There are a few things I need to do with the **RpgLibrary**. In order to deserialize the objects you need a constructor that takes no parameters. Instead of adding constructors that take no parameters to the item classes I instead created data classes with just public fields that match. For the **RolePlayingGame** class I did just add in a constructor that took no parameters. Add the following constructor to the constructor region of the **RolePlayingGame** class.

```
public RolePlayingGame()
{
}
```

Now, right click the **ItemClasses** folder in the **RpgLibrary** project, select **Add** and then **Class**. Name the class **ArmorData**. Repeat the process twice and name the classes **ShieldData** and **WeaponData**. The code for those three classes follows next.

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

namespace RpgLibrary.ItemClasses
{
    public class ArmorData
    {
        public string Name;
        public string Type;
        public int Price;
        public float Weight;
        public bool Equipped;
        public ArmorLocation ArmorLocation;
        public int DefenseValue;
        public int DefenseModifier;
        public string[] AllowableClasses;
    }
}

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

namespace RpgLibrary.ItemClasses
{
    public class ShieldData
    {
        public string Name;
        public string Type;
        public int Price;
        public float Weight;
        public bool Equipped;
        public int DefenseValue;
        public int DefenseModifier;
        public string[] AllowableClasses;
    }
}

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

namespace RpgLibrary.ItemClasses
{
    public class WeaponData
    {
        public string Name;
        public string Type;
        public int Price;
        public float Weight;
        public bool Equipped;
        public Hands NumberHands;
        public int AttackValue;
        public int AttackModifier;
        public int DamageValue;
        public int DamageModifier;
        public string[] AllowableClasses;
    }
}
```

I'm going to add in a class to manage all of these item data classes. Right click the **ItemClasses** folder in the **RpgLibrary** project, select **Add** and then **Class**. Name this new class **ItemDataManager**. This is the code for that class.

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

namespace RpgLibrary.ItemClasses
{
    public class ItemDataManager
    {
        #region Field Region

        readonly Dictionary<string,ArmorData> armorData = new Dictionary<string, ArmorData>();
        readonly Dictionary<string,ShieldData> shieldData = new Dictionary<string, ShieldData>();
        readonly Dictionary<string,WeaponData> weaponData = new Dictionary<string, WeaponData>();

        #endregion

        #region Property Region

        public Dictionary<string, ArmorData> ArmorData
        {
            get { return armorData; }
        }

        public Dictionary<string, ShieldData> ShieldData
        {
            get { return shieldData; }
        }

        public Dictionary<string, WeaponData> WeaponData
        {
            get { return weaponData; }
        }

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion
    }
}
```

It works the same way as the **EntityDataManager** class works except there are three dictionaries instead of the one. I want to change the **ItemManager** field in **FormDetails** to be **ItemDataManager** instead. I also want to expose the **itemManager** and **entityDataManager** field to other forms. Change the **Field**, **Property**, and **Constructor** regions in **FormDetails** to the following.

```
#region Field Region

protected static ItemDataManager itemManager;
protected static EntityDataManager entityDataManager;

#endregion

#region Property Region

public static ItemDataManager ItemManager
{
    get { return itemManager; }
    private set { itemManager = value; }
}

public static EntityDataManager EntityDataManager
{
    get { return entityDataManager; }
    private set { entityDataManager = value; }
}

#endregion

#region Constructor Region

public FormDetails()
{
    InitializeComponent();

    if (FormDetails.ItemManager == null)
        ItemManager = new ItemDataManager();
    if (FormDetails.EntityDataManager == null)
        EntityDataManager = new EntityDataManager();

    this.FormClosing += new FormClosingEventHandler(FormDetails_FormClosing);
}

#endregion
```

I want to add a method to **FormClasses**, **FormArmor**, **FormShield**, and **FormWeapon**. This method will fill the list box of the form with the appropriate data. The code for the method of each form follows next.

FormClasses

```
public void FillListBox()
{
    lbDetails.Items.Clear();

    foreach (string s in FormDetails.EntityDataManager.EntityData.Keys)
        lbDetails.Items.Add(FormDetails.EntityDataManager.EntityData[s]);
}
```

FormArmor

```
public void FillListBox()
{
    lbDetails.Items.Clear();

    foreach (string s in FormDetails.ItemManager.ArmorData.Keys)
        lbDetails.Items.Add(FormDetails.ItemManager.ArmorData[s]);
}
```

FormShield

```
public void FillListBox()
{
    lbDetails.Items.Clear();

    foreach (string s in FormDetails.ItemManager.ShieldData.Keys)
        lbDetails.Items.Add(FormDetails.ItemManager.ShieldData[s]);
}
```

FormWeapon

```
public void FillListBox()
{
    lbDetails.Items.Clear();

    foreach (string s in FormDetails.ItemManager.WeaponData.Keys)
        lbDetails.Items.Add(FormDetails.ItemManager.WeaponData[s]);
}
```

In the **Click** event handler of the **New Game** menu item is where I will handle creating a new game. Change the **newGameMenuToolStripItem_Click** method to the following.

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

        if (result == DialogResult.OK && frmNewGame.RolePlayingGame != null)
        {
            FolderBrowserDialog folderDialog = new FolderBrowserDialog();

            folderDialog.Description = "Select folder to create game in.";
            folderDialog.SelectedPath = Application.StartupPath;

            DialogResult folderResult = folderDialog.ShowDialog();

            if (folderResult == DialogResult.OK)
            {
                try
                {

                    gamePath = Path.Combine(folderDialog.SelectedPath, "Game");
                    classPath = Path.Combine(gamePath, "Classes");
                    itemPath = Path.Combine(gamePath, "Items");

                    if (Directory.Exists(gamePath))
                        throw new Exception("Selected directory already exists.");

                    Directory.CreateDirectory(gamePath);
                    Directory.CreateDirectory(classPath);
                    Directory.CreateDirectory(itemPath + @"\Armor");
                    Directory.CreateDirectory(itemPath + @"\Shield");
                    Directory.CreateDirectory(itemPath + @"\Weapon");

                    rolePlayingGame = frmNewGame.RolePlayingGame;
                    XnaSerializer.Serialize<RolePlayingGame>(gamePath + @"\Game.xml",
rolePlayingGame);
                }
                catch (Exception ex)
                {
                    MessageBox.Show(ex.ToString());
                    return;
                }

                classesToolStripMenuItem.Enabled = true;
                itemsToolStripMenuItem.Enabled = true;

            }
        }
    }
}
```

The new code is in the if statement that checks to see if the result of the dialog was **DialogResult.OK** and that the **RolePlayingGame** property of the form is not null. If those were both true I create a **FolderBrowserDialog** object. I set the **SelectedPath** property to be the path where the editor started from. I also set the **Description** property to let the user know what folder they are browsing for. I capture the result of showing that dialog. If the result was **DialogResult.OK** there is a **try-catch** block where I attempt to make directories to save the game in. I use the **Combine** method of the **Path** class to create the paths to save to. The path for the game, **gamePath**, is the selected path from the dialog and **Game**. The **classPath** is the **gamePath** and **Classes**. The **itemPath** is the **gamePath** and **Items**. I then check to see if **gamePath** exists. If it does I throw an exception saying that a game already exists in that directory. I only want to have one game per directory. I then call the **CreateDirectory** method of the **Directory** class to create the directories. Each sub-item type will be stored in a directory of its own under the **itemPath** directory. The **rolePlayingGame** field is set to the **RolePlayingGame** property of the new game dialog. I then call the **Serialize<T>** method to serialize the **rolePlayingGame** field. As you can see I specify **RolePlayingGame** for **T**. For the file name I use the **gamePath** and **\Game.xml**. If there was an exception I catch it and display it in a message box and exit the method. If the game was created successfully I set the **Enabled** property of the class and item menu items.

There is a bit of a dependency here. Items require that you have the classes that are allowed to use the item. So in order to code the item forms you need to code the forms dealing with character classes first. Right click **FormClasses** and select **View Code**. I'm going to update the event handler for the click event of **btnAdd**. Change the code for **btnAdd_Click** to the following. Add the **AddEntity** method to the **Method Region**.

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

        if (frmEntityData.EntityData != null)
        {
            AddEntity(frmEntityData.EntityData);
        }
    }
}

private void AddEntity(EntityData entityData)
{
    if (FormDetails.EntityDataManager.EntityData.ContainsKey(entityData.EntityName))
    {
        DialogResult result = MessageBox.Show(
            entityData.EntityName + " already exists. Do you want to overwrite it?",
            "Existing Character Class",
            MessageBoxButtons.YesNo);

        if (result == DialogResult.No)
            return;

        FormDetails.EntityDataManager.EntityData[entityData.EntityName] = entityData;

        FillListBox();
        return;
    }

    lbDetails.Items.Add(entityData.ToString());

    FormDetails.EntityDataManager.EntityData.Add(
        entityData.EntityName,
        entityData);
}
```

The event handler creates a new form inside of a using statement so it will be disposed of when the code exits the block of code. I call the **ShowDialog** method of the form to display it. If the **EntityData** property of the form is not null I call the **AddEntity** method passing in the **EntityData** property of the form.

The **AddEntity** data method will add the **EntityData** adds the new **EntityData** object to **lbDetails** and the the **EntityDataManager** on **FormDetails**. If there exists an **EntityData** object with the same name as the new one and you try and add it an exception will be thrown. So, there is an if statement that checks to see if the key is in the dictionary. If it is I display a message box stating that there is already an entry and if the user wants to overwrite it. If they select No I exit the method. If they want to overwrite it I assign the new **EntityData** object to that key value. I call the method **FillListBox** to refill the list box with the **EntityData** objects. I then exit the method. If there wasn't an existing **EntityData** object I add it to the list box and add it to the **EntityDataManager**.

I'm going to code saving games next. I believe that you should be able to write out data before trying to read it in. The first step is to add code to the **saveGameMenuToolStripItem_Click** event handler. Change the code for that method to the following.

```
void saveGameToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{
    if (rolePlayingGame != null)
    {
        try
        {
            XnaSerializer.Serialize<RolePlayingGame>(gamePath + @"\Game.xml", rolePlayingGame);
            FormDetails.WriteEntityData();
            FormDetails.WriteItemData();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.ToString(), "Error saving game.");
        }
    }
}
```

I check to see if **rolePlayingGame** is not null. If it is null then there is nothing to save. If it is not then there is a try-catch block that tries to save the game. I first use the **Serialize<T>** method to write out the **rolePlayingGame** field. I then call static methods that I added to **FormDetails** for writing out the values in the **EntityDataManager** and the **ItemManager**. If an exception was thrown I display a message box with the error.

The **WriteEntityData** and **WriteItemData** methods write out the data for the appropriate type. Add the following methods to the **Method Region** of **FormDetails**.

```
public static void WriteEntityData()
{
    foreach (string s in EntityDataManager.EntityData.Keys)
    {
        XnaSerializer.Serialize<EntityData>(
            FormMain.ClassPath + @"\" + s + ".xml",
            EntityDataManager.EntityData[s]);
    }
}

public static void WriteItemData()
{
    foreach (string s in ItemManager.ArmorData.Keys)
    {
        XnaSerializer.Serialize<ArmorData>(
            FormMain.ItemPath + @"\Armor\" + s + ".xml",
            ItemManager.ArmorData[s]);
    }

    foreach (string s in ItemManager.ShieldData.Keys)
    {
        XnaSerializer.Serialize<ShieldData>(
            FormMain.ItemPath + @"\Shield\" + s + ".xml",
            ItemManager.ShieldData[s]);
    }

    foreach (string s in ItemManager.WeaponData.Keys)
    {
        XnaSerializer.Serialize<WeaponData>(
            FormMain.ItemPath + @"\Weapon\" + s + ".xml",
            ItemManager.WeaponData[s]);
    }
}
```

The first method, **WriteEntityData**, loops through all of the keys in the **EntityData** dictionary in the **EntityDataManager** class. Inside the loop I call the **Serialize<T>** method of the **XnaSerializer** class. For the file name I use the static **ClassPath** property of **FormMain**, add a \ to place it in that directory, add the name of the **EntityData**, and an **xml** extension. The **WriteItemData** method works basically the same way. The difference is where they are written. Armor is written to the **Armor** sub-directory of the **Items** folder, shields to **Shield**, and weapons to **Weapon**. As more classes are added to the game you just add in methods to write them out.

Opening a game will require a little more work than writing it out. The way I decided to handle reading in a game is to display a **FolderBrowserDialog** to allow the user to browse to the folder that holds their game. From there I try and open the game. Change the **openGameToolMenuStripItem_Click** method to the following and add the methods **OpenGame** and **PrepareForms**.

```
void openGameToolStripMenuItem_Click(object sender, EventArgs e)
{
    FolderBrowserDialog folderDialog = new FolderBrowserDialog();

    folderDialog.Description = "Select Game folder";
    folderDialog.SelectedPath = Application.StartupPath;

    bool tryAgain = false;

    do
    {
        DialogResult folderResult = folderDialog.ShowDialog();
        DialogResult msgBoxResult;

        if (folderResult == DialogResult.OK)
        {
            if (File.Exists(folderDialog.SelectedPath + @"\Game\Game.xml"))
            {
                try
                {
                    OpenGame(folderDialog.SelectedPath);
                    tryAgain = false;
                }
                catch (Exception ex)
                {
                    msgBoxResult = MessageBox.Show(
                        ex.ToString(),
                        "Error opening game.",
                        MessageBoxButtons.RetryCancel);

                    if (msgBoxResult == DialogResult.Cancel)
                        tryAgain = false;
                    else if (msgBoxResult == DialogResult.Retry)
                        tryAgain = true;
                }
            }
            else
            {

                msgBoxResult = MessageBox.Show(
                    "Game not found, try again?",
                    "Game does not exist",
                    MessageBoxButtons.RetryCancel);

                if (msgBoxResult == DialogResult.Cancel)
                    tryAgain = false;
                else if (msgBoxResult == DialogResult.Retry)
                    tryAgain = true;
            }
        }
    } while (tryAgain);
}

private void OpenGame(string path)
{
    gamePath = Path.Combine(path, "Game");
    classPath = Path.Combine(gamePath, "Classes");
    itemPath = Path.Combine(gamePath, "Items");

    rolePlayingGame = XnaSerializer.Deserialize<RolePlayingGame>(
        gamePath + @"\Game.xml");

    FormDetails.ReadEntityData();
    FormDetails.ReadItemData();

    PrepareForms();
}

private void PrepareForms()
{
    if (frmClasses == null)
    {
        frmClasses = new FormClasses();
        frmClasses.MdiParent = this;
    }

    frmClasses.FillListBox();

    if (frmArmor == null)
    {
        frmArmor = new FormArmor();
        frmArmor.MdiParent = this;
    }

    frmArmor.FillListBox();

    if (frmShield == null)
    {
        frmShield = new FormShield();
        frmShield.MdiParent = this;
    }

    frmShield.FillListBox();

    if (frmWeapon == null)
    {
        frmWeapon = new FormWeapon();
        frmWeapon.MdiParent = this;
    }

    frmWeapon.FillListBox();

    classesToolStripMenuItem.Enabled = true;
    itemsToolStripMenuItem.Enabled = true;
}
```

The **openGameToolStripMenuItem** method first creates a **FolderBrowserDialog** object and sets the **Description** property to **Select Game folder**. I also set the **SelectedPath** to the folder the application started in using the **StartUpPath** property of the **Application** class. There is a bool variable that will determine if the user would like to try again if there is an error opening the game. It is set to false initially.

I did it in a do-while loop instead of a while loop as you know the loop needs to go through at least once. There are two **DialogResult** variables inside of the loop. **folderResult** holds the result of the **ShowDialog** method of the **FolderBrowserDialog** object. **msgResult** holds the result of any message boxes displayed. There is an if statement to check if **folderResult** is **DialogResult.OK**. Inside that if statement there is an if statement that checks to see if **Game.xml** exists in the **Game** folder of the selected folder. Inside that if statement there is a try-catch block where I try to actually open the game. In the try part I call the **OpenGame** method and set **tryAgain** to false because opening the game worked. In the catch part I capture the result of a message box. I set the buttons for the message box so they are **Retry** and **Cancel**. If the result is **Cancel** I set **tryAgain** to false as the user doesn't want to try again. If it is **Retry** I set **tryAgain** to true.

In the else part of the if statement that checked if a game exits I capture the result of a message box similar to the previous one. I then do the same action. If **Cancel** is select **tryAgain** is set to false and true if **Retry** was selected.

The **OpenGame** method is where I actually try and read in the data for the game. I first create the paths to read data to like I did when I created a new game. I then deserialize the **Game.xml** file into the **rolePlayingGame** field. I call the **ReadEntityData** and **ReadItemData** methods of **FormDetails** to read in the entity data and item data. I then call **PrepareForms** to prepare the forms.

In the **PrepareForms** method I check if each of the forms is null. If it is null I create a new instance and set the **MdiParent** property to this, the current form. I then call the **FillListBox** method on the form to fill the list box with the data for that form.

You need to add two static methods to **FormDetails**, **ReadEntityData** and **ReadItemData**, to read in the data. You also need to add a using statement for the **System.IO** name space.

```
using System.IO;

public static void ReadEntityData()
{
    entityDataManager = new EntityDataManager();

    string[] fileNames = Directory.GetFiles(FormMain.ClassPath, "*.xml");

    foreach (string s in fileNames)
    {
        EntityData entityData = XnaSerializer.Deserialize<EntityData>(s);
        entityDataManager.EntityData.Add(entityData.EntityName, entityData);
    }
}

public static void ReadItemData()
{
    itemManager = new ItemDataManager();

    string[] fileNames = Directory.GetFiles(
        Path.Combine(FormMain.ItemPath, "Armor"),
        "*.xml");

    foreach (string s in fileNames)
    {
        ArmorData armorData = XnaSerializer.Deserialize<ArmorData>(s);
        itemManager.ArmorData.Add(armorData.Name, armorData);
    }

    fileNames = Directory.GetFiles(
        Path.Combine(FormMain.ItemPath, "Shield"),
        "*.xml");

    foreach (string s in fileNames)
    {
        ShieldData shieldData = XnaSerializer.Deserialize<ShieldData>(s);
        itemManager.ShieldData.Add(shieldData.Name, shieldData);
    }

    fileNames = Directory.GetFiles(
        Path.Combine(FormMain.ItemPath, "Weapon"),
        "*.xml");
    foreach (string s in fileNames)
    {
        WeaponData weaponData = XnaSerializer.Deserialize<WeaponData>(s);
        itemManager.WeaponData.Add(weaponData.Name, weaponData);
    }

}
```

To read in data you need a file name of the file you want to read in. I have each type of data in a folder of its own. I use the **GetFiles** method of the **Directory** class to get all of the files in that directory that returns an array of strings with the file names. In a foreach loop I iterate over all of the items in the array of tile names. Inside of the foreach loop there is a variable of the type of data I want to read in. I use the **Deserialize<T>** method of the **XnaSerializer** class to deserialize the file into the variable. I then add the object to the manager class.

I think that this is more than enough for this tutorial. The editors are coming along nicely but I think you deserve a break from them. In the next tutorial I will move back to the game instead of the editors. 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