XNA 4.0 RPG Tutorials

Part 18A

Finding Loot Part 2

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.

In the last tutorial I worked on adding chests for the player to interact with to find loot. In this tutorial I going to continue on with that. The first thing I want to do is to add in two classes for keys for unlocking locks. Like you can find on chests, doors, and other objects. Keys are just another type of item so they belong in the **ItemClasses** folder of the **RpgLibrary**. Like other types of items you will want a data class for use in editors and reading in data and a class for the keys themselves that inherits from the **BaseItem** class. Right click the **ItemClasses** folder, select **Add** and then **Class**. Name this new class **KeyData**. Repeat the process and name the new class **Key**. The code for both of those classes follows next.

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

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

        public string Name;
        public string Type;

        #endregion

        #region Constructor Region

        public KeyData()
        {
        }

        #endregion

        #region Method Region

        public override string ToString()
        {
            string toString = Name + ", ";
            toString += Type;

            return toString;
        }

        #endregion
    }
}
```

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

namespace RpgLibrary.ItemClasses
{
    public class Key : BaseItem
    {
        #region Field region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public Key(string name, string type)
            : base(name, type, 0, 0, null)
        {
        }

        #endregion

        #region Virtual Method Region

        public override object Clone()
        {
            Key key = new Key(this.Name, this.Type);

            return key;
        }

        #endregion
    }
}
```

Not much you haven't seen before. The **KeyData** class has two public fields. The **Name** of the key and the **Type** of the key. I debated about including other fields like a price and a weight. I didn't think they were necessary. The **Type** field can be of great use. If you've played the **Fable** games they have special silver keys that can be used to open special chests. Each chest requires a certain number of these keys. Using the **Type** field you can easily add this functionality into your games. It will require a minor tweak that I will make later. The **ToString** method of the **KeyData** class just combines the name of the key and the type. The **Key** class is also simplistic. It inherits from the **BaseItem** class. It takes just two string parameters for the name and the type. In the call to the base constructor I pass in the parameters passed to the **Key** class and 0 for the price and weight and null for **allowableClasses**. The **Clone** method just creates a new key and returns it.

The first tweak for handling multiple keys of the same type is to add in two fields to the **ChestData** class. They are **KeyType** which is a string and **KeysRequired** which is an integer. I'm going to also associate a **DifficultyLevel** with a chest from the **SkillClasses** folder for the lock on the chest. In the **ToString** method you add in these three new fields, calling their **ToString** methods. I also removed the **TextureName** field from the class. I will be handling associating an image with a chest when I get to the level editor as it applies more to levels. Change the **ChestData** method to the following.

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

using RpgLibrary.SkillClasses;
```

```
namespace RpgLibrary.ItemClasses
{
    public class ChestData
    {
        public string Name;
        public DifficultyLevel DifficultyLevel;
        public bool IsLocked;
        public bool IsTrapped;
        public string TrapName;
        public string KeyName;
        public string KeyType;
        public int KeysRequired;
        public int MinGold;
        public int MaxGold;
        public Dictionary<string, string> ItemCollection;

        public ChestData()
        {
            ItemCollection = new Dictionary<string, string>();
        }

        public override string ToString()
        {
            string toString = Name + ", ";
            toString += DifficultyLevel.ToString() + ", ";
            toString += IsLocked.ToString() + ", ";
            toString += IsTrapped.ToString() + ", ";
            toString += TrapName + ", ";
            toString += KeyName + ", ";
            toString += KeyType + ", ";
            toString += KeysRequired.ToString() + ", ";
            toString += MinGold.ToString() + ", ";
            toString += MaxGold.ToString();

            foreach (KeyValuePair<string, string> pair in ItemCollection)
            {
                toString += ", " + pair.Key + "+" + pair.Value;
            }

            return toString;
        }
    }
}
```

You also need update the **Chest** class, specifically the **Clone** method. Change the **Clone** of the **Chest** class to the following.

```
public override object Clone()
{
    ChestData data = new ChestData();
    data.Name = chestData.Name;
    data.DifficultyLevel = chestData.DifficultyLevel;
    data.IsLocked = chestData.IsLocked;
    data.IsTrapped = chestData.IsTrapped;
    data.TrapName = chestData.TrapName;
    data.KeyName = chestData.KeyName;
    data.KeyType = chestData.KeyType;
    data.KeysRequired = chestData.KeysRequired;
    data.MinGold = chestData.MinGold;
    data.MaxGold = chestData.MaxGold;

    foreach (KeyValuePair<string, string> pair in chestData.ItemCollection)
        data.ItemCollection.Add(pair.Key, pair.Value);

    Chest chest = new Chest(data);
    return chest;
}
```

I will handle opening chests with keys in a future tutorial. I just wanted to add in the functionality to this tutorial so it will be available later when it is needed. I'm going to work on the editor a bit on creating keys and chests. To do that I'm first going to update the **ItemDataManager** class to hold dictionaries of **<string, ItemDataType>** where **ItemDataType** is the data class associated with the item. Change the **ItemDataManager** to the following.

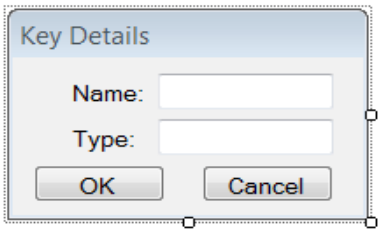
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

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>();
        readonly Dictionary<string, ReagentData> reagentData = new Dictionary<string,
ReagentData>();
        readonly Dictionary<string, KeyData> keyData = new Dictionary<string, KeyData>();
        readonly Dictionary<string, ChestData> chestData = new Dictionary<string, ChestData>();

        #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; }
        }

        public Dictionary<string, ReagentData> ReagentData
        {
            get { return reagentData; }
        }

        public Dictionary<string, KeyData> KeyData
        {
            get { return keyData; }
        }

        public Dictionary<string, ChestData> ChestData
        {
            get { return chestData; }
        }

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
```

```
        #endregion
    }
}
```

Nothing there that you haven't seen before. Build your project to make sure it builds correctly. Right click the **RpgEditor** project in the solution explorer, select **Add** and then **Windows Form**. Name this new form **FormKey**. Set the **Size** property of the form to be the **Size** property of your **FormDetails**. Set the **MinimizeBox** property to false and the **Text** property to **Keys**. You will now want to have **FormKey** inherit from **FormDetails** instead of **Form**. Right click **FormKey** 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;

namespace RpgEditor
{
    public partial class FormKey : FormDetails
    {
        public FormKey()
        {
            InitializeComponent();
        }
    }
}
```

I'm going to add another form before I get to the code for **FormKey**. Right click **RpgEditor** in the solution explorer, select **Add** and then **Windows Form**. Name this new form **FormKeyDetails**. I'm going to add a couple controls and set some properties for **FormKeyDetails**. Your finished form in the designer should resemble the one below.



Start by changing the size of the form so it is a bigger than what you need. Drag a **Label** onto the form near the top then a **Text Box** and position it to the right of the **Label**. Drag another **Label** onto the form positioning it below the first. Drag a second **Text Box** onto the form and position it below the first **Text Box**. Now drag two **Buttons** onto the form. Position the first to the left and the second to the right. Set the **Text** property of the form to **Key Details**, the **StartPosition** to **CenterParent**, the **ControlBox** to **False**, and the **FormBorderStyle** to **FixedDialog**. Set the **Name** property of the first **Text Box** to **tbName** and the second to **tbType**. Set the **Text** property of the first **Label** to **Name:** and the second to **Type:**. For the buttons the one of the left has **Name** and **Text** properties of **btnOK** and **OK**. The second **btnCancel** and **Cancel** for the **Name** and **Text** properties.

Now lets add the logic to **FormKeyDetails**. Right click **FormKeyDetails** 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.ItemClasses;

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

        KeyData key;

        #endregion

        #region Property Region

        public KeyData Key
        {
            get { return key; }
            set { key = value; }
        }

        #endregion

        #region Constructor Region

        public FormKeyDetails()
        {
            InitializeComponent();

            this.Load += new EventHandler(FormKeyDetails_Load);
            this.FormClosing += new FormClosingEventHandler(FormKeyDetails_FormClosing);

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

        #endregion

        #region Event Handler Region

        void FormKeyDetails_Load(object sender, EventArgs e)
        {
            if (key != null)
            {
                tbName.Text = key.Name;
                tbType.Text = key.Type;
            }
        }

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

        void btnOK_Click(object sender, EventArgs e)
        {
```

```
            if (string.IsNullOrEmpty(tbName.Text))
            {
                MessageBox.Show("You must enter a name for the item.");
                return;
            }

            key = new KeyData();
            key.Name = tbName.Text;
            key.Type = tbType.Text;

            this.FormClosing -= FormKeyDetails_FormClosing;
            this.Close();
        }

        void btnCancel_Click(object sender, EventArgs e)
        {
            key = null;
            this.FormClosing -= FormKeyDetails_FormClosing;
            this.Close();
        }

        #endregion
    }
}
```

This should look familiar to you. It follows the other detail forms that I created. There is a using statement to bring the **ItemClasses** name space from the **RpgLibrary** into scope. There is a field of type **KeyData** for the key that is being created or edited. If the key is being edited you can use the property to set the fields of the key. The constructor wires handlers for the **Load** and **FormClosing** events of the form. It also wires handlers for the **Click** event of **btnOK** and **btnCancel**.

The event handler for the **Load** event of the form checks to see if the field **key** is not null. If it isn't it sets the **Text** properties of **tbName** and **tbType** to be the **Name** and **Type** fields of **key**. The event handler for **FormClosing** is the same as before. If the reason for closing the form is the user is trying to close the form the event is canceled.

In the **Click** event handler for **btnOK** I check to see if the **Text** property of **tbName** is null or empty. If it is I display a message box stating that you need to give the key a name and then exit the method. I'm not enforcing that a key have a type associated with it. If you want to add the functionality in you can do the same as I did for **tbName** with **tbType**. If **tbName** had a value I assign the **key** field a new instance of **KeyData**. I then set the **Name** and **Type** fields to be the **Text** property of **tbName** and **tbType** respectively. To allow the form to close I unsubscribe the **FormClosing** event handler and call the **Close** method to close the form.

The **Click** event handler for **btnCancel** sets the **key** field to null. It then unsubscribes the **FormClosing** event and calls the **Close** method of the form. Just like in the other forms for a specific type of item.

Time to code the logic for **FormKey**. Right click **FormKey** in the solution explorer and select **View Code**. Change the code for **FormKey** 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 System.IO;
```

```
using RpgLibrary.CharacterClasses;
using RpgLibrary.ItemClasses;

namespace RpgEditor
{
    public partial class FormKey : FormDetails
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public FormKey()
        {
            InitializeComponent();

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

        #endregion

        #region Event Handler Region

        void btnAdd_Click(object sender, EventArgs e)
        {
            using (FormKeyDetails frmKeyDetails = new FormKeyDetails())
            {
                frmKeyDetails.ShowDialog();

                if (frmKeyDetails.Key != null)
                {
                    AddKey(frmKeyDetails.Key);
                }
            }
        }

        void btnEdit_Click(object sender, EventArgs e)
        {
            if (lbDetails.SelectedItem != null)
            {
                string detail = lbDetails.SelectedItem.ToString();
                string[] parts = detail.Split(',');
                string entity = parts[0].Trim();

                KeyData data = itemManager.KeyData[entity];
                KeyData newData = null;

                using (FormKeyDetails frmKeyData = new FormKeyDetails())
                {
                    frmKeyData.Key = data;
                    frmKeyData.ShowDialog();

                    if (frmKeyData.Key == null)
                        return;

                    if (frmKeyData.Key.Name == entity)
                    {
                        itemManager.KeyData[entity] = frmKeyData.Key;
                        FillListBox();
                        return;
                    }

                    newData = frmKeyData.Key;
                }
```

```
            DialogResult result = MessageBox.Show(
                "Name has changed. Do you want to add a new entry?",
                "New Entry",
                MessageBoxButtons.YesNo);

            if (result == DialogResult.No)
                return;

            if (itemManager.KeyData.ContainsKey(newData.Name))
            {
                MessageBox.Show("Entry already exists. Use Edit to modify the entry.");
                return;
            }

            lbDetails.Items.Add(newData);
            itemManager.KeyData.Add(newData.Name, newData);
        }
    }

    void btnDelete_Click(object sender, EventArgs e)
    {
        if (lbDetails.SelectedItem != null)
        {
            string detail = (string)lbDetails.SelectedItem;
            string[] parts = detail.Split(',');
            string entity = parts[0].Trim();

            DialogResult result = MessageBox.Show(
                "Are you sure you want to delete " + entity + "?",
                "Delete",
                MessageBoxButtons.YesNo);

            if (result == DialogResult.Yes)
            {
                lbDetails.Items.RemoveAt(lbDetails.SelectedIndex);
                itemManager.KeyData.Remove(entity);

                if (File.Exists(FormMain.ItemPath + @"\Key\" + entity + ".xml"))
                    File.Delete(FormMain.ItemPath + @"\Key\" + entity + ".xml");
            }
        }
    }

    #endregion

    #region Method Region

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

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

    private void AddKey(KeyData keyData)
    {
        if (FormDetails.ItemManager.KeyData.ContainsKey(keyData.Name))
        {
            DialogResult result = MessageBox.Show(
                keyData.Name + " already exists. Overwrite it?",
                "Existing key",
                MessageBoxButtons.YesNo);

            if (result == DialogResult.No)
                return;

            itemManager.KeyData[keyData.Name] = keyData;
            FillListBox();
```

```
                return;
            }

            itemManager.KeyData.Add(keyData.Name, keyData);
            lbDetails.Items.Add(keyData);
        }

        #endregion
    }
}
```

Again, this should look familiar as other forms use the same code. The difference is that instead of working with armor, shields, or weapons, I'm working with keys. There is the using statement for the **System.IO** name space. Event handlers for the click event of the buttons are wired in the constructors. The **Click** event handler of **btnAdd** creates a form in a using block. I show the form. If the **Key** property of the form is not null I call the **AddKey** method passing in the **Key** property. The **Click** event handler of **btnEdit** checks to see if the **SelectedItem** of **lbDetails** is not null. It parses the string to get the name of the key. It then gets the **KeyData** for the selected item and sets **newData** to null. In a using statement a form is created. The **Key** property of the form is set to the **KeyData** of **SelectedItem.** I call the **ShowDialog** method to display the form. If the **Key** property of form is null I exit the method. If the name is the same as before I assign the entry in the item manager to be the new key, call **FillListBox** to update the key and exit the method. I then set **newData** to be the **Key** property of the form. The name of the key changed so I display a message box asking if the new key should be added. If the result is no I exit the method. If there is a key with that name already I display a message box and exit. If there wasn't I add the new key to the list box and the item manager. The **Click** event handler for **btnDelete** checks to make sure that the **SelectedItem** of the list box is not null. It parses the selected item and displays a message box asking if the key should be deleted. If the result of the message box is Yes I remove the key from the list box and I remove if from the item manager as well. I then delete the file, if it exists.

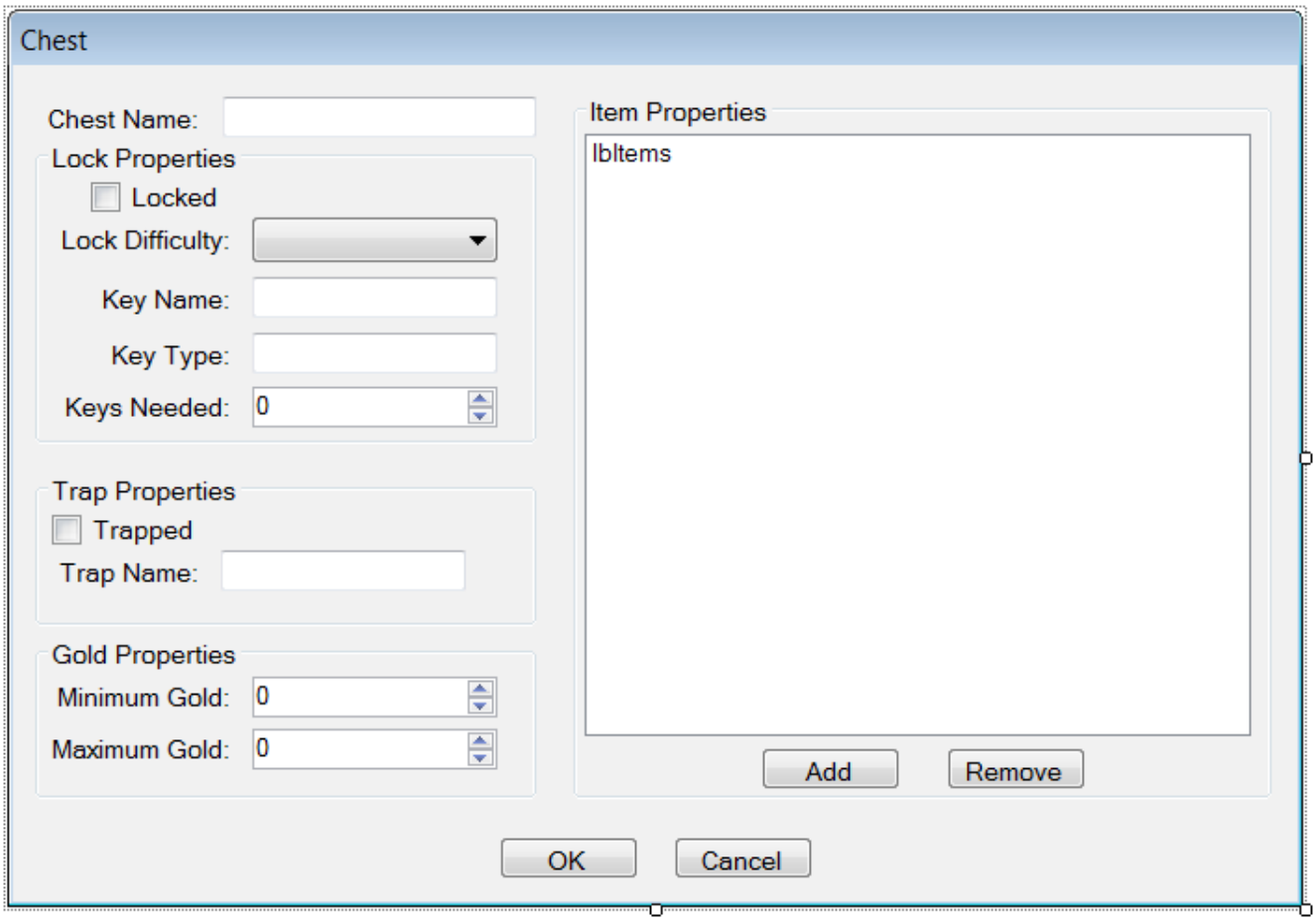
I noticed something in my code that really should be fixed. In the code of **FormWeapon** I have as the class name **Weapons** instead of **FormWeapon**. If you have that problem as well this is a good time to fix it. Right click **FormWeapon** in the solution explorer and select **View Code**. Place your cursor over **Weapons** and press **F2** to rename it. In the dialog box that pops up replace **Weapons** with **FormWeapon** and press OK. In the second dialog box that pops up just select **OK**.

Before I get to adding these forms to the editor I want to add in forms that work with **ChestData**. Right click **RpgEditor** in the solution explorer, select **Add** and then **Windows Form**. Name this new form **FormChest**. Set the **Size** property of the form to be the **Size** property of your **FormDetails**. Set the **MinimizeBox** property to false and the **Text** property to **Chests**. You will now want to have **FormChest** inherit from **FormDetails** instead of **Form**. Right click **FormChest** 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;

namespace RpgEditor
{
    public partial class FormChest : FormDetails
    {
```

```
        public FormChest()
        {
            InitializeComponent();
        }
    }
}
```

You will also want a form for creating chests. Right click the **RpgEditor** project in the solution explorer, select **Add** and then **Windows Form**. Name this new form **FormChestDetails**. My finished form in the designer is next.



I set a few properties for the form. I set **ControlBox** to **False**, **FormBorderStyle** to **FixedDialog** and **Text** to **Chest**. Set **StartPosition** to **CenterParent** as well.

There are a lot of controls on the form and I tried to group them logically. First, make your form a lot bigger to house all of the controls. The first control I dragged onto the form was a **Label** and set its **Text** property to **Chest Name:**. I then dragged a **Text Box** beside the **Label** and set its **Name** property to **tbName**. I also made the **Text Box** a little wider. Below the **Text Box** I dragged a **Group Box** to group the items related to the lock on the chest together. I set that **Group Box**'s **Text** property to **Lock Properties**. I dragged a **Check Box** onto that **Group Box** and set its **Name** to **cbLock** and its **Text** to **Locked**. Under the **Check Box** I dragged a **Label** and **Combo Box**. Set the **Label**'s **Text** property to **Lock Difficulty:** and the **Combo Box**'s **Name** property to **cboDifficulty**. I then dragged a **Label** and set its **Text** property to **Key Name:** and a **Text Box** beside that and set its **Name** property to **tbKeyName**. I dragged another **Label** and **Text Box** under those two. The **Label**'s **Text** property was

set to **Key Type:** and the **Text Box**'s **Name** was set to **tbKeyType**. I then dragged a **Label** and **Numeric Up Down** onto the **Group Box**. I set the **Text** property of the **Label** to **Keys Needed:** and the **Name** property of the **Numeric Up Down** to **nudKeys**. I also set the **Enabled** property of the **Combo Box**, **Text Boxes**, and **Numeric Up Down** to **False** initially. The will be enabled if the user checks the **Check Box** to be checked.

I dragged a second **Group Box** onto the form below the **Lock Properties Group Box** and set the width to be the same width. I set the **Text** property of the second **Group Box** to **Trap Properties**. I dragged a **Check Box** onto that **Group Box** and set its **Name** property to be **cbTrap**. I set then dragged a **Label** and **Text Box** onto the second **Group Box**. I set the **Text** property of the **Label** to **Trap Name:** and the **Name** property of the **Text Box** to **tbTrap**. I set the **Enabled** property of **tbTrap** to **False** as well.

I then dragged a third **Group Box** onto the form and sized the width to be the same as the other two. I set its **Text** property to **Gold Properties**. I dragged a **Label** and **Numeric Up Down** onto this **Group Box**. I set the **Text Property** of the **Label** to **Minimum Gold:** and the **Name** property of the **Numeric Up Down** to **nudMinGold**. I dragged another **Label** and **Numeric Up Down** onto this **Group Box**. I set the **Text** property of the **Label** to **Maximum Gold:** and the **Name** property of the **Numeric Up Down** to **nudMaxGold**. I set the **Maximum** property of **nudMinGold** and **nudMaxGold** to be 10000. You will want to tweak this number as you test your game to make sure that it is not unbalanced. If the player accumulates too much gold and can buy really expensive and powerful items then your balance goes out the window. The same is true if you give the player too little gold. They won't be able to defend themselves against more powerful monsters.

I then dragged on another **Group Box** and set its **Text** property to **Item Properties**. I dragged a **List Box** onto the **Group Box** and made it wider and taller. I set its **Name** property to **lbItems**. I then dragged two **Buttons** below the **List Box**. The one on the left I named **btnAdd** and set its **Text** property to **Add**. The one on the right I named **btnRemove** and set its **Text** property to **Remove**.

The last two controls I dragged onto the form were two **Buttons**. The first button, the one on the left, I set its **Name** to **btnOK** and its **Text** to **OK**. The second I set its **Name** property to **btnCancel** and its **Text** property to **Cancel**.

I'm going to add some logic to the form now. I'm going to skip the items for now and add that in a future tutorial. Right click **FormChestDetails** in the solution explorer and select **View Code**. Change the code for **FormChestDetails** 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.ItemClasses;
using RpgLibrary.TrapClasses;
using RpgLibrary.SkillClasses;

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

```
        ChestData chest;

        #endregion

        #region Property Region

        public ChestData Chest
        {
            get { return chest; }
            set { chest = value; }
        }

        #endregion

        #region Constructor Region

        public FormChestDetails()
        {
            InitializeComponent();

            this.Load += new EventHandler(FormChestDetails_Load);
            this.FormClosing += new FormClosingEventHandler(FormChestDetails_FormClosing);

            foreach (string s in Enum.GetNames(typeof(DifficultyLevel)))
            {
                cboDifficulty.Items.Add(s);
            }

            cboDifficulty.SelectedIndex = 0;

            cbLock.CheckedChanged += new EventHandler(cbLock_CheckedChanged);
            cbTrap.CheckedChanged += new EventHandler(cbTrap_CheckedChanged);

            btnAdd.Click += new EventHandler(btnAdd_Click);
            btnRemove.Click += new EventHandler(btnRemove_Click);

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

        #endregion

        #region Form Event Handler Region

        void FormChestDetails_Load(object sender, EventArgs e)
        {
            if (chest != null)
            {
                tbName.Text = chest.Name;

                cbLock.Checked = chest.IsLocked;
                tbKeyName.Text = chest.KeyName;
                tbKeyType.Text = chest.KeyType;
                nudKeys.Value = (decimal)chest.KeysRequired;

                tbKeyName.Enabled = chest.IsLocked;
                tbKeyType.Enabled = chest.IsLocked;
                nudKeys.Enabled = chest.IsLocked;

                cbTrap.Checked = chest.IsTrapped;
                tbTrap.Text = chest.TrapName;

                tbTrap.Enabled = chest.IsTrapped;

                nudMinGold.Value = (decimal)chest.MinGold;
                nudMaxGold.Value = (decimal)chest.MaxGold;
            }
        }
```

```
        void FormChestDetails_FormClosing(object sender, FormClosingEventArgs e)
        {
            if (e.CloseReason == CloseReason.UserClosing)
            {
                e.Cancel = true;
            }
        }

        #endregion

        #region Check Box Event Handler Region

        void cbLock_CheckedChanged(object sender, EventArgs e)
        {
            cboDifficulty.Enabled = cbLock.Checked;
            tbKeyName.Enabled = cbLock.Checked;
            tbKeyType.Enabled = cbLock.Checked;
            nudKeys.Enabled = cbLock.Checked;
        }

        void cbTrap_CheckedChanged(object sender, EventArgs e)
        {
            tbTrap.Enabled = cbTrap.Checked;
        }

        #endregion

        #region Button Event Handler Region

        void btnAdd_Click(object sender, EventArgs e)
        {
        }

        void btnRemove_Click(object sender, EventArgs e)
        {
        }

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

            if (cbTrap.Checked && string.IsNullOrEmpty(tbTrap.Text))
            {
                MessageBox.Show("You must supply a name for the trap on the chest.");
                return;
            }

            if (nudMaxGold.Value < nudMinGold.Value)
            {
                MessageBox.Show("Maximum gold in chest must be greater or equal to minimum
gold.");
                return;
            }

            ChestData data = new ChestData();

            data.Name = tbName.Text;
            data.IsLocked = cbLock.Checked;

            if (cbLock.Checked)
            {
                data.DifficultyLevel = (DifficultyLevel)cboDifficulty.SelectedIndex;
                data.KeyName = tbKeyName.Text;
                data.KeyType = tbKeyType.Text;
                data.KeysRequired = (int)nudKeys.Value;
            }
```

```
            data.IsTrapped = cbTrap.Checked;

            if (cbTrap.Checked)
            {
                data.TrapName = tbTrap.Text;
            }

            data.MinGold = (int)nudMinGold.Value;
            data.MaxGold = (int)nudMaxGold.Value;

            chest = data;
            this.FormClosing -= FormChestDetails_FormClosing;
            this.Close();
        }

        void btnCancel_Click(object sender, EventArgs e)
        {
            chest = null;
            this.FormClosing -= FormChestDetails_FormClosing;
            this.Close();
        }

        #endregion
    }
}
```

The code should look some what familiar. I've used the same basic code in all of the forms for creating a specific item. There is some new stuff though. As usual there is a field and property for the type of item that is being created, **ChestData** in this case.

The constructor wires several event handlers. The **Load** and **FormClosing** events are wired first. I also fill the **Combo Box** with the names from the **DifficultyLevel** enumeration and set the **SelectedIndex** of the **Combo Box** to 0. I then wire the handlers for the **Check** event of the two **Check Boxes**. I also wire the handlers for all four of the buttons.

In the **Load** event handler I check to see if the **chest** field is not null. If it has a value I fill the form with values. I set the **Enabled** properties of controls associated with a chest being locked to be the **IsLocked** field of the **chest** field. So, if the chest is locked the controls will be enabled and disabled if the chest is not locked. I do the same with the **IsTrapped** field and the controls associated with a chest being trapped. I'm not worrying about items at the moment. That will be added in down the road.

There is nothing you haven't seen before in the **FormClosing** event handler. It just cancels the event if the reason for closing the form is **UserClosing**. The event will be unsubscribed from if a chest is successfully created or the user cancels the changes.

In the **CheckChanged** handlers I set the **Enabled** property of the controls associated with the **Check Box** to the **Checked** property of the **Check Box**. If the **Check Box** is checked then the controls will be enabled and disabled if the **Check Box** is not checked.

The **Click** event handler for **btnOK** does a little validation of the form. It checks to see if the chest has a name. If **cbTrap** is checked and the **Text** property of **tbTrap** is null or empty then the user must supply a name for the trap. Ideally you will want to confirm that the trap actually exists or in game ignore the **IsTrapped** property if no trap exists. I also check to make sure that the value of max gold is not less than min gold. I then create a new chest using the values on the form. I set the field to be the new chest, unsubscribe from the **FormClosing** event handler and then close the form.

The event handler for the **Click** event of **btnCancel** holds nothing new or interesting. It just assigns the **chest** field to be null, unsubscribes the **FormClosing** event handler and closes the form.

I'm now going to add the code to **FormChest**. Right click **FormChest** in the solution explorer and select **View Code** to bring up the code. This is the code for **FormChest**.

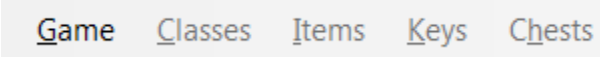
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using System.IO;

using RpgLibrary.ItemClasses;

namespace RpgEditor
{
    public partial class FormChest : FormDetails
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public FormChest()
        {
            InitializeComponent();

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

        #endregion

        #region Event Handler Region

        void btnAdd_Click(object sender, EventArgs e)
        {
            using (FormChestDetails frmChestDetails = new FormChestDetails())
            {
                frmChestDetails.ShowDialog();

                if (frmChestDetails.Chest != null)
                {
                    AddChest(frmChestDetails.Chest);
                }
            }
        }

        void btnEdit_Click(object sender, EventArgs e)
        {
            if (lbDetails.SelectedItem != null)
            {
                string detail = lbDetails.SelectedItem.ToString();
                string[] parts = detail.Split(',');
                string entity = parts[0].Trim();

                ChestData data = itemManager.ChestData[entity];
                ChestData newData = null;
```

```
            using (FormChestDetails frmChestData = new FormChestDetails())
            {
                frmChestData.Chest = data;
                frmChestData.ShowDialog();

                if (frmChestData.Chest == null)
                    return;

                if (frmChestData.Chest.Name == entity)
                {
                    itemManager.ChestData[entity] = frmChestData.Chest;
                    FillListBox();
                    return;
                }

                newData = frmChestData.Chest;
            }

            DialogResult result = MessageBox.Show(
                "Name has changed. Do you want to add a new entry?",
                "New Entry",
                MessageBoxButtons.YesNo);

            if (result == DialogResult.No)
                return;

            if (itemManager.ChestData.ContainsKey(newData.Name))
            {
                MessageBox.Show("Entry already exists. Use Edit to modify the entry.");
                return;
            }

            lbDetails.Items.Add(newData);
            itemManager.ChestData.Add(newData.Name, newData);
        }
    }

    void btnDelete_Click(object sender, EventArgs e)
    {
        if (lbDetails.SelectedItem != null)
        {
            string detail = (string)lbDetails.SelectedItem;
            string[] parts = detail.Split(',');
            string entity = parts[0].Trim();

            DialogResult result = MessageBox.Show(
                "Are you sure you want to delete " + entity + "?",
                "Delete",
                MessageBoxButtons.YesNo);

            if (result == DialogResult.Yes)
            {
                lbDetails.Items.RemoveAt(lbDetails.SelectedIndex);
                itemManager.ChestData.Remove(entity);

                if (File.Exists(FormMain.ItemPath + @"\Chest\" + entity + ".xml"))
                    File.Delete(FormMain.ItemPath + @"\Chest\" + entity + ".xml");
            }
        }
    }

    #endregion

    #region Method Region

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

        foreach (string s in FormDetails.ItemManager.ChestData.Keys)
```

```
                lbDetails.Items.Add(FormDetails.ItemManager.ChestData[s]);
        }

        private void AddChest(ChestData ChestData)
        {
            if (FormDetails.ItemManager.ChestData.ContainsKey(ChestData.Name))
            {
                DialogResult result = MessageBox.Show(
                    ChestData.Name + " already exists. Overwrite it?",
                    "Existing Chest",
                    MessageBoxButtons.YesNo);

                if (result == DialogResult.No)
                    return;

                itemManager.ChestData[ChestData.Name] = ChestData;
                FillListBox();
                return;
            }

            itemManager.ChestData.Add(ChestData.Name, ChestData);
            lbDetails.Items.Add(ChestData);
        }

        #endregion
    }
}
```

Again, this should look familiar as other forms use the same code. The difference is that instead of working with armor, shields, or weapons, I'm working with chests. There is the using statement for the **System.IO** name space. Event handlers for the click event of the buttons are wired in the constructors. The **Click** event handler of **btnAdd** creates a form in a using block. I show the form. If the **Chest** property of the form is not null I call the **AddChest** method passing in the **Chest** property. The **Click** event handler of **btnEdit** checks to see if the **SelectedItem** of **lbDetails** is not null. It parses the string to get the name of the chest. It then gets the **ChestData** for the selected item and sets **newData** to null. In a using statement a form is created. The **Chest** property of the form is set to the **ChestData** of **SelectedItem.** I call the **ShowDialog** method to display the form. If the **Chest** property of form is null I exit the method. If the name is the same as before I assign the entry in the item manager to be the new key, call **FillListBox** to update the key and exit the method. I then set **newData** to be the **Chest** property of the form. The name of the chest changed so I display a message box asking if the new chest should be added. If the result is no I exit the method. If there is a chest with that name already I display a message box and exit. If there wasn't I add the new chest to the list box and the item manager. The **Click** event handler for **btnDelete** checks to make sure that the **SelectedItem** of the list box is not null. It parses the selected item and displays a message box asking if the chest should be deleted. If the result of the message box is Yes I remove the chest from the list box and I remove if from the item manager as well. I then delete the file, if it exists.

The last thing I'm going to tackle in the editor is adding the functionality for the new forms. That will be done in **FormMain**. I need to update the form a little to handle chests and keys. They will have menu entries of their own. Right click **FormMain** in the solution explorer and select **View Designer**. Click the **Menu Strip** on the form. Beside the **Items** entry add a new entry **&Keys**. Set the **Enabled** property of this item to **False**. Beside the **Keys** entry add a new entry **C&hests** and set its **Enabled** proeprty to **False** as well. Your menu bar should resemble the following.



I'm going to end this tutorial here and continue it in a part B. The plan was to add in keys and chests to the editor and read them in to the game. 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