XNA 4.0 RPG Tutorials

Part 9

Item Classes

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'm going to take a break from XNA in this tutorial and get started with classes related to items in the game. Load up your solution from last time. Right click the **RpgLibrary** solution, select **Add** and then **New Folder**. Name this new folder **ItemClasses**. This folder is going to hold all of the classes related to items as the name applies. Now, right click the **ItemClasses** folder, select **Add** and then **Class**. Name this new class **BaseItem**. The code for that class follows next.

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

namespace RpgLibrary.ItemClasses
{
    public enum Hands { One, Two }

    public enum ArmorLocation { Body, Head, Hands, Feet }

    public abstract class BaseItem
    {
        #region Field Region

        protected List<Type> allowableClasses = new List<Type>();

        string name;
        string type;
        int price;
        float weight;
        bool equipped;

        #endregion

        #region Property Region

        public List<Type> AllowableClasses
        {
            get { return allowableClasses; }
            protected set { allowableClasses = value; }
        }

        public string Type
        {
            get { return type; }
            protected set { type = value; }
        }

        public string Name
        {
```

```
            get { return name; }
            protected set { name = value; }
        }

        public int Price
        {
            get { return price; }
            protected set { price = value; }
        }

        public float Weight
        {
            get { return weight; }
            protected set { weight = value; }
        }

        public bool IsEquiped
        {
            get { return equipped; }
            protected set { equipped = value; }
        }

        #endregion

        #region Constructor Region

        public BaseItem(string name, string type, int price, float weight, params Type[]
allowableClasses)
        {
            foreach (Type t in allowableClasses)
                AllowableClasses.Add(t);

            Name = name;
            Type = type;
            Price = price;
            Weight = weight;
            IsEquiped = false;
        }

        #endregion

        #region Abstract Method Region

        public abstract object Clone();

        public virtual bool CanEquip(Type characterType)
        {
            return allowableClasses.Contains(characterType);
        }

        public override string ToString()
        {
            string itemString = "";

            itemString += Name + ", ";
            itemString += Type + ", ";
            itemString += Price.ToString() + ", ";
            itemString += Weight.ToString();

            return itemString;
        }

        #endregion
    }
}
```

There are two enums included in the code for this class, at the name space scope. Having enums at the name space level makes them accessible with out having to use a class name to reference them. The

Hands enum is for the number of hands it takes to use a weapon. The **ArmorLocation** enum is for the location of armor. Instead of having classes for helms, gloves, armor, and boots, I will wrap them all in one class. The **BaseItem** class is an abstract class. Other classes will inherit from this class.

There are a number of things common to all items. Only certain classes should be able to use some items. A fighter shouldn't be able to use a wizard's wand for example. So there is a field **List<Type>** , **allowableClasses**, that will hold the classes that can use the item. The class **Type** holds information about a class. It can be compared to see if two **Type** fields are equal. This way you can use the **typeof** operator to find out what type an object is.

All items in the game will also have a name so there is a field for that. The next field **type** is a string. This is a field that can be used to assign a type to an item. An example of a type would be sword. Sword describes many similar types of items. There are short swords, long swords, bastard swords, etc. These are all swords and have similar characteristics. Items also have a price and weight so there are fields for that as well. In previous RPG tutorials I used an integer for weight, this time around I went for a float so you can have items that don't weigh a lot. For example, a ring won't weight 1 pound and technically doesn't have a weight of 0 pounds. The last field is to determine if an item is equipped or not. You can't equip all items though. You don't have to equip a potion to use it for example, you just use the potion. I will explain how I'm handling that in a bit.

There are properties to expose all of the fields. They are all public properties and the read, or get, parts are public. The write, or set, parts of the properties are all protected. This means that they can be used for assigning values in classes that inherit from **BaseItem**.

Instead of requiring that you create and pass in a **List<Type>** to the constructor I used the **params** keyword. The **params** keyword allows you to pass in zero or more items of the type specified. The format is `params parameterType[] parameterName.` It has to be the last parameter of the method. The constructor takes as parameters the name of the item, the type of the item, the price of the item, the weight of the item, and the classes that can use the item. At the start of the constructor in a foreach loop, I loop through all of the **Types** that where passed in. Inside the loop I add the **Type** to the **List<Type>** using the **AllowableClasses** property. It then sets the **name**, **price**, and **weight** fields to the values passed in. It then sets the **equipped** field to false. This means that when you retrieve an item it isn't automatically equipped by a character.

There is an abstract method, **Clone**, that all classes that inherit from **BaseItem** must implement. I will be using a class for managing all items in the game. This method will return a copy of an item. Since items are represented using classes they are reference types. If you return the original and a change is made, say equipping the item, all variables that refer to that item will be changed.

The next method is a virtual method that can you can override in inherited classes, **CanEquip**. This method returns if the **allowableClasses** field contains the type passed in. In a class that inherits from **BaseItem** that you don't want to be able to be equipped you can just return false. The last method that I added was an override of the **ToString** method. It just returns a string with the item's name, type, price and weight.

With this base class I'm going to create three inherited classes. There are basic items that you find in a fantasy RPG. They are for weapons, armor, and shields. As the tutorials progress I will be adding in more and more item types.

I will start with weapons. Right click the **ItemClasses** folder, select **Add** and then **Class**. Name this new class **Weapon**. This is the code for the **Weapon** class.

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

namespace RpgLibrary.ItemClasses
{
    public class Weapon : BaseItem
    {
        #region Field Region

        Hands hands;
        int attackValue;
        int attackModifier;
        int damageValue;
        int damageModifier;

        #endregion

        #region Property Region

        public Hands NumberHands
        {
            get { return hands; }
            protected set { hands = value; }
        }

        public int AttackValue
        {
            get { return attackValue; }
            protected set { attackValue = value; }
        }

        public int AttackModifier
        {
            get { return attackModifier; }
            protected set { attackModifier = value; }
        }

        public int DamageValue
        {
            get { return damageValue; }
            protected set { damageValue = value; }
        }

        public int DamageModifier
        {
            get { return damageModifier; }
            protected set { damageModifier = value; }
        }

        #endregion

        #region Constructor Region

        public Weapon(
                string weaponName,
                string weaponType,
                int price,
                float weight,
                Hands hands,
                int attackValue,
                int attackModifier,
                int damageValue,
                int damageModifier,
                params Type[] allowableClasses)
```

```
            : base(weaponName, weaponType, price, weight, allowableClasses)
        {
            NumberHands = hands;
            AttackValue = attackValue;
            AttackModifier = attackModifier;
            DamageValue = damageValue;
            DamageModifier = damageModifier;
        }

        #endregion

        #region Abstract Method Region

        public override object Clone()
        {
            Type[] allowedClasses = new Type[allowableClasses.Count];

            for (int i = 0; i < allowableClasses.Count; i++)
                allowedClasses[i] = allowableClasses[i];

            Weapon weapon = new Weapon(
                Name,
                Type,
                Price,
                Weight,
                NumberHands,
                AttackValue,
                AttackModifier,
                DamageValue,
                DamageModifier,
                allowedClasses);

            return weapon;
        }

        public override string ToString()
        {
            string weaponString = base.ToString() + ", ";
            weaponString += NumberHands.ToString() + ", ";
            weaponString += AttackValue.ToString() + ", ";
            weaponString += AttackModifier.ToString() + ", ";
            weaponString += DamageValue.ToString() + ", ";
            weaponString += DamageModifier.ToString();

            foreach (Type t in allowableClasses)
                weaponString += ", " + t.Name;

            return base.ToString();
        }

        #endregion
    }
}
```

There are four additional fields and properties in the class. The new fields are **attackValue**, **attackModifier**, **damageValue**, and **damageModifier**. They are exposed by the corresponding properties: **AttackValue**, **AttackModifier**, **DamageValue**, and **DamageModifier**. The **attackValue** and **attackModifier** fields are for the attack value and bonus of the weapon. The **attackModifier** is added to the **attackValue** or the weapon to get its total attack value. The modifier can be negative if you have a cursed weapon for example. This value helps to determine if an attack by the weapon is successful. The **damageValue** and **damageModifier** fields are used to determine the damage the weapon will inflict, reduced by the defense value of the armor worn.

The constructor takes a few parameters in addition to what the **BaseItem** class requires. There is a parameter for the number of hands needed to use the weapon. There are also parameters for the attack

and damage fields. The constructor sets the fields added to the **Weapon** class using the properties that were created.

The **Clone** method creates a new **Weapon** using the fields of the current weapon. It then returns the new instance. The interesting part was that you want to pass in the **allowableClasses** field to the constructor. To handle this, I created an array of **Type** called **allowedClasses** the same size as the **Count** property of **allowableClasses**. In a for loop I populate the **allowedClasses** array with the **allowableClasses** array's values. You could probably also have used the **Clone** method of the array class to make a clone of the array.

In the override of the **ToString** method there is a string, **armorString**, that I assign the value of the **ToString** method of the base class, **BaseItem**. I then append a comma and a space after that. I then add in the fields for the **Weapon** class one by one appending a comma and a space after them, except for the last one. The reason is I don't want there to be a trailing comma if no classes can equip or use the item. In a foreach loop I loop through **allowableClasses** and append a comma and **Name** property of the type.

The next class is the **Armor** class. Right click the **ItemClasses** folder, select **Add** and then **Class**. Name this new class **Armor**. This is the code for the **Armor** class.

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

namespace RpgLibrary.ItemClasses
{
    public class Armor : BaseItem
    {
        #region Field Region

        ArmorLocation location;

        int defenseValue;
        int defenseModifier;

        #endregion

        #region Property Region

        public ArmorLocation Location
        {
            get { return location; }
            protected set { location = value; }
        }

        public int DefenseValue
        {
            get { return defenseValue; }
            protected set { defenseValue = value; }
        }

        public int DefenseModifier
        {
            get { return defenseModifier; }
            protected set { defenseModifier = value; }
        }

        #endregion

        #region Constructor Region
```

```
        public Armor(
                string armorName,
                string armorType,
                int price,
                float weight,
                ArmorLocation locaion,
                int defenseValue,
                int defenseModifier,
                params Type[] allowableClasses)
            : base(armorName, armorType, price, weight, allowableClasses)
        {
            Location = location;
            DefenseValue = defenseValue;
            DefenseModifier = defenseModifier;
        }

        #endregion

        #region Abstract Method Region

        public override object Clone()
        {
            Type[] allowedClasses = new Type[allowableClasses.Count];

            for (int i = 0; i < allowableClasses.Count; i++)
                allowedClasses[i] = allowableClasses[i];

            Armor armor = new Armor(
                Name,
                Type,
                Price,
                Weight,
                Location,
                DefenseValue,
                DefenseModifier,
                allowedClasses);

            return armor;
        }

        public override string ToString()
        {
            string armorString = base.ToString() + ", ";
            armorString += Location.ToString() + ", ";
            armorString += DefenseValue.ToString() + ", ";
            armorString += DefenseModifier.ToString();

            foreach (Type t in allowableClasses)
                armorString += ", " + t.Name;

            return armorString;
        }

        #endregion
    }
}
```

Like in the **Weapon** class there are a couple new fields. There are fields for the location of the armor, the defense value of the armor, and the defense modifier of the armor. There are properties to expose these fields. The constructor takes parameters for the new fields and sets them. Like the **ToString** method of the **Weapon** class, the **ToString** method creates a string using the return value of the base class appending a comma and a space. The location, defense value, and defense modifier are appended with commas and spaces after the location and defense value. Then the allowed classes are append with a comma, space, and name of the type.

The last new item class is the shield class. It looks and works the same as the other classes. There are just two new fields in this class though. There is a defense value and a defense modifier. Since there is really not much new I won't explain the code. Right click the **ItemClasses** folder, select **Add** and then **Class**. Name this new class **Shield**. This is the code for that class.

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

namespace RpgLibrary.ItemClasses
{
    public class Shield : BaseItem
    {
        #region Field Region

        int defenseValue;
        int defenseModifier;

        #endregion

        #region Property Region

        public int DefenseValue
        {
            get { return defenseValue; }
            protected set { defenseValue = value; }
        }

        public int DefenseModifier
        {
            get { return defenseModifier; }
            protected set { defenseModifier = value; }
        }

        #endregion

        #region Constructor Region

        public Shield(
                string shieldName,
                string shieldType,
                int price,
                float weight,
                int defenseValue,
                int defenseModifier,
                params Type[] allowableClasses)
            : base(shieldName, shieldType, price, weight, allowableClasses)
        {
            DefenseValue = defenseValue;
            DefenseModifier = defenseModifier;
        }

        #endregion

        #region Abstract Method Region

        public override object Clone()
        {
            Type[] allowedClasses = new Type[allowableClasses.Count];

            for (int i = 0; i < allowableClasses.Count; i++)
                allowedClasses[i] = allowableClasses[i];

            Shield shield = new Shield(
                Name,
                Type,
                Price,
```

```
                Weight,
                DefenseValue,
                DefenseModifier,
                allowedClasses);

            return shield;
        }

        public override string ToString()
        {
            string shieldString = base.ToString() + ", ";
            shieldString += DefenseValue.ToString() + ", ";
            shieldString += DefenseModifier.ToString();

            foreach (Type t in allowableClasses)
                shieldString += ", " + t.Name;

            return shieldString;
        }

        #endregion
    }
}
```

The last class that I'm going to add in this tutorial is a class to manage the items in the game. This is not a class for inventory. That will be added in further down the line. What this class will be used for is reading in all of the items in the game, and writing them from an editor. Right click the **ItemClasses** folder, select **Add** and then **Class**. Name this class **ItemManager**. This is the code for that class.

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

namespace RpgLibrary.ItemClasses
{
    public class ItemManager
    {
        #region Fields Region

        Dictionary<string, Weapon> weapons = new Dictionary<string, Weapon>();
        Dictionary<string, Armor> armors = new Dictionary<string, Armor>();
        Dictionary<string, Shield> shields = new Dictionary<string, Shield>();

        #endregion

        #region Keys Property Region

        public Dictionary<string, Weapon>.KeyCollection WeaponKeys
        {
            get { return weapons.Keys; }
        }

        public Dictionary<string, Armor>.KeyCollection ArmorKeys
        {
            get { return armors.Keys; }
        }

        public Dictionary<string, Shield>.KeyCollection ShieldKeys
        {
            get { return shields.Keys; }
        }

        #endregion

        #region Constructor Region
```

```
        public ItemManager()
        {
        }

        #endregion

        #region Weapon Methods

        public void AddWeapon(Weapon weapon)
        {
            if (!weapons.ContainsKey(weapon.Name))
            {
                weapons.Add(weapon.Name, weapon);
            }
        }

        public Weapon GetWeapon(string name)
        {
            if (weapons.ContainsKey(name))
            {
                return (Weapon)weapons[name].Clone();
            }

            return null;
        }

        public bool ContainsWeapon(string name)
        {
            return weapons.ContainsKey(name);
        }

        #endregion

        #region Armor Methods

        public void AddArmor(Armor armor)
        {
            if (!armors.ContainsKey(armor.Name))
            {
                armors.Add(armor.Name, armor);
            }
        }

        public Armor GetArmor(string name)
        {
            if (armors.ContainsKey(name))
            {
                return (Armor)armors[name].Clone();
            }

            return null;
        }

        public bool ContainsArmor(string name)
        {
            return armors.ContainsKey(name);
        }

        #endregion

        #region Shield Methods

        public void AddShield(Shield shield)
        {
            if (!shields.ContainsKey(shield.Name))
            {
                shields.Add(shield.Name, shield);
            }
        }
```

```
        public Shield GetShield(string name)
        {
            if (shields.ContainsKey(name))
            {
                return (Shield)shields[name].Clone();
            }

            return null;
        }

        public bool ContainsShield(string name)
        {
            return shields.ContainsKey(name);
        }

        #endregion
    }
}
```

There are three **Dictionary** fields. The key for them is a string and the value is the item type. The one for the **Weapon** class is called **weapons**, the one for **Armor** is **armors**, and **shields** for the **Shield** class. There is a default constructor here but does nothing at the moment. I added it in because I will be using it down the road.

There are three get only properties that return the **KeyCollection** of the corresponding dictionary. The **WeaponKeys** property returns the **Keys** property of the **weapons** dictionary. **ArmorKeys** and **ShieldKeys** do the same for the **armor** and **shield** dictionaries. They were added more for the editor as there will be times when it will be useful to get the keys related to items.

I added in regions for the methods of the different item types. I can see there being quite a few methods added for each of the item types so it is best to try to group them. For each of the item types there are **Get** and **Add** methods. The **Add** methods are used to add new items to the **ItemManager**. The **Get** methods are used to retrieve an item from the **ItemManager** using the item's name. It doesn't return the actual item, it casts the return of the **Clone** method to the appropriate type. You have to do this because to have the **Clone** method work for all item types I had to use a type that all other objects inherited from. I used **object** because all classes inherit from the **object** type. The **Add** methods check to make sure there isn't an item with that name in the **Dictionary** already. If you try to add an item with the same key an exception will be thrown. Similarly, the **Get** methods check to see if there is an item with the key in the **Dictionary**. If you try to retrieve a value that has no key you will also generate an exception. If there is no appropriate item in a **Get** method I return null, meaning no item was found.

Things are coming together but there is still more work to be done. I think that is more than enough for this tutorial. I'd like 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