Creating a Role Playing Game with XNA Game Studio
Part 49
Picking Up Items - Part 3

To follow along with this tutorial you will have to have read the previous tutorials to understand much of what it going on. You can find a list of tutorials here: XNA Role Playing Game Tutorials You will also find the latest version of the project on the web site on that page. If you want to follow along and type in the code from this PDF as you go, you can find the previous project at this link: http://www.jtmbooks.com/rpgtutorials/New2DRPG48.zip You can download the graphics from this link: Graphics.zip

This is the third part of the tutorial dealing with picking up items. In this tutorial I will be adding the base classes for items in the game. Since C# is a good object-oriented programming language, I will be using polymorphism again in this tutorial. I am a big fan of polymorphism, especially when it comes to game programming. To be honest, it would be better to say I'm a big fan of object-oriented programming. If you still don't understand polymorphism, it is the ability of a base class to act as an inherited class at run time. I will be writing a tutorial on understanding polymorphism better, and I will also explain how I'm using it in this tutorial.

To get started, go a head and load your last solution. I am going to try and keep all of the classes related to items in the **ItemClasses** folder of the game. To get started, go a head and add a new class to the **ItemClasses** folder called **BaseItem**. This class will hold information that is common to all items. There will also be an **enum** in the class called **ItemSize**. This will be used later down the road when I add containers that can hold other items like bags and chests. This is the code for the **BaseItem** class.

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

namespace New2DRPG.ItemClasses
{
    public enum ItemSize { Tiny, Small, Medium, Large };

    public abstract class BaseItem
    {
        string name;
        int price;
        int weight;
        ItemSize itemSize;

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

        public int Price
```

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

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

        public ItemSize Size
        {
            get { return itemSize; }
            protected set { itemSize = value; }
        }

        public BaseItem(string name, int price, int weight, ItemSize size)
        {
            Name = name;
            Price = price;
            Weight = weight;
            Size = size;
        }
    }
}
```

As you can see, there are four sizes: **Tiny, Small, Medium,** and **Large**. The next think you will see is that **BaseItem** is an abstract class. This means that you can not create an instance of **BaseItem** directly. It can however, hold instances of classes that inherit from **BaseItem**. So each type of item in the game will inherit from **BaseItem** and a variable of type **BaseClass** can hold a variable of class that inherited from **BaseClass**.

There are four fields in the class: **name**, **price**, **weight**, and **itemSize**. The fields will hold the name, price, weight, and size of the item respectively. There are four public properties to expose the private variables. These may be a little confusing to some. The **get** part you are used to, it just returns the field associated with the property. The next part is what may be confusing. The **set** part has the **protected** access modifier before it. This allows the class that the property is in, and its immediate descendants to use the **set** part of the property but not other classes. I will show you an example when I discuss one of the classes that inherits from **BaseItem**. The **Name**, **Price**, **Weight**, and **Size** properties work with the **name**, **price**, **weight**, and **size** fields respectively. The constructor just assigns the fields with the appropriate parameter. I used the **set** properties to assign the parameters instead of having to use **this**.

For now I'm just going to add three basic items: **Weapons**, **Armor**, and **Shields**. As things progress and I get further into the game, I will be adding other items. I will start with weapons. Go a head and add a new class to the **ItemClasses** folder called **Weapon**. This is the code for the **Weapon** class.

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

```
namespace New2DRPG.ItemClasses
{
    public enum Hands { One, Two }

    public class Weapon : BaseItem
    {
        Hands hands;
        int attackValue;
        int attackBonus;

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

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

        public int AttackBonus
        {
            get { return attackBonus; }
            protected set { attackBonus = value; }
        }

        public Weapon(
                string weaponName,
                int price,
                int weight,
                ItemSize size,
                Hands hands,
                int attackValue,
                int attackBonus)
            : base(weaponName, price, weight, size)
        {
            NumberHands = hands;
            AttackValue = attackValue;
            AttackBonus = attackBonus;
        }
    }
}
```

The **Weapon** class is similar to the **BaseItem** class. It holds fields that will be common to all weapons. There is an **enum** here that holds the number of hands required to use a weapon. This will be helpful for determining if a character can use a shield with a weapon.

The fields in the class are: **hands**, **attackValue**, and **attackBonus**. The represent the number of hands needed to use a weapon, the attack value of the weapon and the attack bonus of the weapon respectively. The properties **NumberHands**, **AttackValue** and **AttackBonus** are for the **hands**, **attackValue**, and **attackBonus** fields respectively. They are also public get properties and protected set properties. I've already explained the **hands** field, now I will explain the other two. The first one, **attackValue**, will represent the minimum damage the weapon will inflict. The other field, **attackBonus**, is a value that is added to **attackValue**, to determine the upper range of damage inflicted by the weapon. The are before the effect of armor is taken into consideration. You can also use the

attackBonus field to increase the damage done by magic weapon or decrease the damage done for a bad quality weapon.

The constructor of the **Weapon** class takes a number of parameters: **weaponName**, **price**, **weight**, **size**, **hands**, **attackValue**, and **attackBonus**. They represent the name of the weapon, the price of the weapon, the weight of the weapon, the size of the weapon, the attack value of the weapon and the attack bonus of the weapon. There is also a call to the constructor of the base class that requires the name, price, weight, and size of the item. The constructor then assigns the fields in the class using the properties.

The next class I will add is the base class for armor. Add a new class to the **ItemClasses** folder call **Armor**. This is the code for the **Armor** class.

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

namespace New2DRPG.ItemClasses
{
    public class Armor : BaseItem
    {
        int defenseValue;
        int defenseBonus;

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

        public int DefenseBonus
        {
            get { return defenseBonus; }
            protected set { defenseBonus = value; }
        }

        public Armor(
                string armorName,
                int price,
                int weight,
                ItemSize size,
                int defenseValue,
                int defenseBonus)
            : base(armorName, price, weight, size)
        {
            DefenseValue = defenseValue;
            DefenseBonus = defenseBonus;
        }
    }
}
```

The **Armor** class is like the **Weapon** class, it extends the **BaseItem** class. Where the **Weapon** class works for attack, the **Armor** class works for defense. The new fields in this class are **defenseValue** and **defenseBonus**. The **defenseValue** field represents the resistance of the armor.

Lighter armor will have a lower value and heavier, stronger, armor will have a higher value. Like the **attackBonus** of the **Weapon** class, the **defenseBonus** modifies the **defenseValue** of the armor. By increasing **defenseBonus** the base resistance of the armor increases. The same is true in reverse, weaker, poorer quality armor more than likely doesn't have the same resistance as regular armor. The **DefenseValue** and **DefenseBonus** properties are for the **defenseValue** and **defenseBonus** fields respectively.

The constructor for the **Armor** class takes as parameters: **armorName**, **price**, **weight**, **size**, **defenseValue**, and **defenseBonus**. They are for the name of the armor, price of the armor, weight of the armor, size of the armor, defense value of the armor, and the defense bonus of the armor. The constructor calls the constructor of **BaseItem** and sets the properties using the parameters passed in.

The last base class to add is the **Shield** class. Add a new class to the **ItemClasses** folder called **Shield**. I am not going to explain the code for the shield class. It is basically a twin of the **Armor** class. It will just be used for shields instead of armor. This is the code for the **Shield** class.

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

namespace New2DRPG.ItemClasses
{
    class Shield : BaseItem
    {
        int defenseValue;
        int defenseBonus;

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

        public int DefenseBonus
        {
            get { return defenseBonus; }
            protected set { defenseBonus = value; }
        }

        public Shield(
                string shieldName,
                int price,
                int weight,
                ItemSize size,
                int defenseValue,
                int defenseBonus)
            : base(shieldName, price, weight, size)
        {
            DefenseValue = defenseValue;
            DefenseBonus = defenseBonus;
        }
    }
}
```

The tutorial was shorter than I thought it would be so I decided to add more to this tutorial. I will start with having items in the chests. To do this, I changed the **Chest** class. I will give you the new code for the **Chest** class with the changes in bold so you can see what has changed. This is the new code for the **Chest** class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using New2DRPG.SpriteClasses;

namespace New2DRPG.ItemClasses
{
    class Chest
    {
        static Random random = new Random();
        ItemSprite sprite;
        static float collisionRadius = 64;
        int goldMinimum;
        int goldMaximum;
        int gold;
        List<BaseItem> items;

         public Chest(Game game,
                Texture2D texture,
                Vector2 position,
                List<BaseItem> items)
        {
            sprite = new ItemSprite(game, texture, position);
            gold = 0;
            this.items = items;
        }

        public Chest(Game game,
                Texture2D texture,
                Vector2 position,
                int goldMinimum,
                int goldMaximum,
                List<BaseItem> items)
        {
            sprite = new ItemSprite(game, texture, position);
            this.goldMinimum = goldMinimum;
            this.goldMaximum = goldMaximum;
            if (goldMinimum == goldMaximum)
                gold = goldMinimum;
            else
                gold = Chest.random.Next(goldMinimum, goldMaximum + 1);
            this.items = items;
        }

        public static float CollisionRadius
        {
            get { return collisionRadius; }
        }

        public Vector2 Position
```

```
        {
            get { return sprite.Position; }
        }

        public Vector2 Origin
        {
            get { return sprite.Origin; }
        }

        public int Gold
        {
            get { return gold; }
        }

        public List<BaseItem> Items
        {
            get { return items; }
        }

        public void Update(GameTime gameTime)
        {
            sprite.Update(gameTime);
        }

        public void Draw(GameTime gameTime)
        {
            sprite.Draw(gameTime);
        }
    }
}
```

The first thing you will see is that I added a new field to the class **items**, which is a **List<BaseItem>**. Using a **List<BaseItem>** means that the chests can hold a number of items inside of them with no maximum. The next change was in the constructors. What I did was add another parameter, **items**, of type **List<BaseItem>**. This allowed me to pass items to the chest. The first constructor can be used to make a chest that has only items in and the second can be used to create chests with both gold and items. If you pass **null** into the constructors that will mean there are no items in the chest at all. The last change was I added a public get only property to retrieve the list of items from the chest.

fjhf

Making those changes will break the **CreateChests** method in the **Game1** class because there is no longer a constructor that will take the parameters that are being passed in. Replace the old **CreateChests** method with the following.

```
private void CreateChests()
{
    Chest tempChest;

    for (int i = 0; i < 2; i++)
    {
        tempChest = new Chest(
            this,
            Content.Load<Texture2D>(@"Items\chest"),
            new Vector2(random.Next(3, 3 + 5), random.Next(3, 3 + 5)),
            random.Next(100),
```

```
            random.Next(100, 200),
            null);
        chests.Add(tempChest);
    }

    List<BaseItem> items = new List<BaseItem>();

    Weapon sword = new Weapon(
        "Short Sword",
        100,
        5,
        ItemSize.Small,
        Hands.One,
        5,
        0);

    Armor leather = new Armor(
        "Leather Armor",
        10,
        10,
        ItemSize.Medium,
        5,
        0);

    items.Add(sword);
    items.Add(leather);

    tempChest = new Chest(
        this,
        Content.Load<Texture2D>(@"Items\chest"),
        new Vector2(random.Next(5, 11), random.Next(5, 11)),
        random.Next(100),
        random.Next(100, 200),
        items);

    chests.Add(tempChest);

    List<BaseItem> items2 = new List<BaseItem>();
    items2.Add(sword);

    tempChest = new Chest(
        this,
        Content.Load<Texture2D>(@"Items\chest"),
        new Vector2(random.Next(5, 11), random.Next(5, 11)),
        items2);

    chests.Add(tempChest);
}
```

What I did was first declare a local variable **tempChest** that will hold the chest to add to the game. In a for loop I created two chests like before but added an extra argument for the last parameter set to **null** so the chest will not have items in it. I then added the chests to the list of chests in the game.

I then created a **List<BaseItem>** that will hold some items to put in the chest. I then created an object of type **Weapon** called **sword** to represent a short sword. I also created an object of type **Armor** called **leather** to represent leather armor. I then add both of these to the list of items. I created a chest using the constructor that takes the game object, texture, position, minimum gold, maximum gold, and

a list of items. I then created another **List<BaseItem>** because the variable is a reference variable. That means changing the original **List<BaseItem>** would change the items in the other chest. I added the **sword** variable to the new **List<BaseItem>** and then created a new chest using the constructor that requires the game object, texture, position, and a list of items. I then added the chest to the list of chests in the game.

The last thing I will cover in this tutorial is showing the contents of the chest when the player opens a chest. I will handle adding the items to the player's inventory in another tutorial. This will be done in the **Draw** method of the **TreasureScreen** class.

```
public override void Draw(GameTime gameTime)
{
    base.Draw(gameTime);

    string outstring;
    Vector2 position = new Vector2(
        15 + imagePosition.X,
        130 + imagePosition.Y);

    if (chest != null)
    {
        if (chest.Gold != 0)
        {
            outstring = "You found " + chest.Gold.ToString() + " gold.";
            spriteBatch.DrawString(spriteFont,
                outstring,
                position,
                Color.Yellow);
            position.Y += spriteFont.LineSpacing * 2;
        }
        if (chest.Items != null)
        {
            if (chest.Gold != 0)
                outstring = "As well as:";
            else
                outstring = "You found:";

            spriteBatch.DrawString(spriteFont,
                outstring,
                position,
                Color.Yellow);
            foreach (BaseItem item in chest.Items)
            {
                position.Y += spriteFont.LineSpacing;
                outstring = " " + item.Name;
                spriteBatch.DrawString(spriteFont,
                    outstring,
                    position,
                    Color.Yellow);
            }
        }
    }
}
```

What did, is after the call to **base.Draw** is create two variables: **outstring** and **position**. **outstring** will hold a string to be drawn and **position** will hold where to draw the string. The **position**

variable's value is set the same way as in the previous tutorials. There is then an if statement that checks to make sure that there is a chest to work with.

Inside the if statement there is another if statement that checks to see if there is gold in the chest. If there is gold I create a string that tells how much gold the player has found. I draw the string and then increment the **Y** property of **position** by the **LineSpacing** property of the font times 2. This will allow any other text drawn to move to a different line.

There is another if statement that checks to see if there are items inside of the chest. Inside that if there is an if statement that checks to see if there was gold in the chest. If there was gold I set the **outstring** to say **You also found:**, otherwise it is set to **You found:**. I then draw **outstring**.

Next there is a foreach loop that will loop through all of the items in a chest. I first increase the **Y** property of **position** to draw on a different line. I then create a string that will hold what the item is and finally draw the string.

Well, that is it for this tutorial. I will be adding in more and more functionality to the game that you would expect to find in a role playing game. In the next tutorial I will finish picking up items. I encourage you to keep either visiting my site http://xnagpa.net or my blog, XNA Game Programming Adventures Blog for the latest news on my tutorials.