Eyes of the Dragon Tutorials
Part 10
Character Classes

I'm writing these tutorials for the MonoGame 3.8 framework using Visual Studio 2019. The tutorials will make more sense if they are read in order. You can find the list of tutorials on the Eyes of the Dragon page of my web blog. I will be making each version of the project available on GitHub here. It will be included on the page that links to the tutorials.

This tutorial is about adding in classes related to characters in the game. When I say characters I mean the player character, non-player characters, and monsters. Monster is a rather generic term to mean any enemy that the player will face. They can be anything for a bandit to a wolf to a dragon. Player characters and non-player characters will be a little different from monsters. A monster will be a more generic version than player characters and non-player characters. There are a number of things that all three will share in common though.

I will be adding items related to characters to the **RpgLibrary** project. Right click the **RpgLibrary** project in the solution explorer, select **Add** and then **New Folder** and call it **CharacterClasses**. I'm going to add in a class to represent an attribute that has a current and maximum value like health and mana. Right click the **CharacterClasses** folder, select **Add** and then **Class**. Call this new class **AttributePair**. The code for that class follows.

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

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

        int currentValue;
        int maximumValue;

        #endregion

        #region Property Region

        public int CurrentValue
        {
            get { return currentValue; }
        }

        public int MaximumValue
        {
            get { return maximumValue; }
        }

        public static AttributePair Zero
        {
            get { return new AttributePair(); }
        }
```

```
        #endregion

        #region Constructor Region

        private AttributePair()
        {
            currentValue = 0;
            maximumValue = 0;
        }

        public AttributePair(int maxValue)
        {
            currentValue = maxValue;
            maximumValue = maxValue;
        }

        #endregion

        #region Method Region

        public void Heal(ushort value)
        {
            currentValue += value;

            if (currentValue > maximumValue)
                currentValue = maximumValue;
        }

        public void Damage(ushort value)
        {
            currentValue -= value;

            if (currentValue < 0)
                currentValue = 0;
        }

        public void SetCurrent(int value)
        {
            currentValue = value;

            if (currentValue > maximumValue)
                currentValue = maximumValue;
        }

        public void SetMaximum(int value)
        {
            maximumValue = value;

            if (currentValue > maximumValue)
                currentValue = maximumValue;
        }

        #endregion
    }
}
```

I had considered making the **AttributePair** a structure rather than a class. In the end I decided to go with a class as one of the main reasons for using a structure rather than a class is you will be

frequently creating and destroying objects of that type. That is not something that I will be doing. There are two fields in this class. The first is **currentValue** which represents the current value of the pair. The other is **maximumValue** and represents the maximum value the pair can have. I want to force the use of methods that allow for validation to modify the fields so there are read only properties to expose the fields. **CurrentValue** exposes the **currentValue** field and **MaximumValue** exposes the **maximumValue** field. There is a static property, **Zero**, that just returns an attribute pair with the default values of zero.

Their are two constructors in the class. The first is a private constructor that sets the fields to 0. It was called from the static property **Zero** that returns an attribute pair with the values set to 0. The other constructor takes an integer parameter for the maximum value of the attribute pair. It sets the current value and maximum values to that parameter.

I used role playing game terms for two of the methods, **Heal** and **Damage**. The **Heal** method is used to increase the **currentValue** field up to the maximum value. The **Damage** method is used to decrease the **currentValue** field. They both take a **value** parameter that is an unsigned short the value to increase or decrease. I decided to use unsigned short as passing in negative numbers would reverse the effect. The **Heal** method adds the **value** parameter to the **currentValue** field. It then checks to make sure that it doesn't exceed the **maximumValue** field. If it does then **currentValue** is set to **maximumValue**. The **Damage** method decreases the **currentValue** field. If **currentValue** is less than zero it is set to zero.

The **SetCurrent** method is used to set the **currentValue** field to a specific value. It first sets the **currentValue** field to the value passed in. It checks to make sure that **currentValue** is not greater than **maximumValue**. If it is it The **SetMaximum** method is use to set the **maximumValue** field. It checks to make sure that the **currentValue** field is not greater than the **maximumValue** field. If it is it then sets **currentValue** to be **maximumValue**.

I'm now going to add in a class that can be used to read in information about character classes. This again includes the player character, non-player character, and monsters in the game. I'm giving the three a basic name of **Entity**. Right click the **CharacterClasses** folder, select **Add** and then **Class**. Name this new class **EntityData**. This the code for that class.

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

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

        public string ClassName;
        public int Strength;
        public int Dexterity;
        public int Cunning;
        public int Willpower;
        public int Magic;
```

```
        public int Constitution;
        public string HealthFormula;
        public string StaminaFormula;
        public string MagicFormula;

        #endregion

        #region Constructor Region

        private EntityData()
        {
        }

        #endregion

        #region Static Method Region

        public static void ToFile(string filename)
        {
        }

        public static EntityData FromFile(string filename)
        {
            EntityData entity = new EntityData();

            return entity;
        }
        #endregion
    }
}
```

This is a pretty basic class with a few fields, a private constructor and a static method. There is field for name of the entity, **EntityName**. This will be the name of the entity like **Fighter** or **Wolf**. All entities in the game will share some attributes. **Strength** measures how strong an entity is. **Dexterity** is a measure of the entity's agility. **Cunning** is a measure of the entity's mental reasoning and perception. **Willpower** determines an entity's stamina and mana and measures how quickly the entity tires in combat. **Magic** is used to determine a magic using entities spell power and how effective healing items and spells are. **Constitution** measures how healthy an entity is and is used to determine the entity's health.

The next three fields will be used to determine the entity's health, mana, and stamina. Different entities will have different health, mana, and stamina. A dwarf could have higher health than other entities for example. I will explain how the formulas will work later. I included them as they will be needed.

There is a private constructor in the class that is there to be expanded later as well. I also included two static methods **ToFile** and **FromFile**. Both of them take a string parameter that is the file name. These will be used to write an entity to a file or read in an entity from a file. The **FromFile** method creates a new instance of type **EntityData** and returns the entity. I will be adding the code to read and write entities later.

With that I'm going to create an abstract base class for all entities in the game. Right click the **CharacterClasses** folder, select **Add** and then **Class**. Name this new class **Entity**. The code for the

Entity class follows next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.CharacterClasses
{
    public enum EntityGender { Male, Female, NonBinary, Unknown }
    public abstract class Entity
    {
        #region Vital Field and Property Region

        protected string entityName;
        protected string entityType;
        protected EntityGender gender;

        public string EntityName
        {
            get { return entityName; }
        }

        public string EntityType
        {
            get { return entityType; }
        }

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

        #endregion

        #region Basic Attribute and Property Region

        protected int strength;
        protected int dexterity;
        protected int cunning;
        protected int willpower;
        protected int magic;
        protected int constitution;
        protected int strengthModifier;
        protected int dexterityModifier;
        protected int cunningModifier;
        protected int willpowerModifier;
        protected int magicModifier;
        protected int constitutionModifier;

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

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

```
}

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

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

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

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

#endregion

#region Calculated Attribute Field and Property Region

protected AttributePair health;
protected AttributePair stamina;
protected AttributePair mana;

public AttributePair Health
{
    get { return health; }
}

public AttributePair Stamina
{
    get { return stamina; }
}

public AttributePair Mana
{
    get { return mana; }
}

protected int attack;
protected int damage;
protected int defense;

#endregion

#region Level Field and Property Region

protected int level;
protected long experience;
```

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

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

        #endregion

        #region Constructor Region

        private Entity()
        {
            Strength = 0;
            Dexterity = 0;
            Cunning = 0;
            Willpower = 0;
            Magic = 0;
            Constitution = 0;
            health = new AttributePair(0);
            stamina = new AttributePair(0);
            mana = new AttributePair(0);
        }

        public Entity(EntityData entityData)
        {
            entityType = entityData.ClassName;
            Strength = entityData.Strength;
            Dexterity = entityData.Dexterity;
            Cunning = entityData.Cunning;
            Willpower = entityData.Willpower;
            Magic = entityData.Magic;
            Constitution = entityData.Constitution;
            health = new AttributePair(0);
            stamina = new AttributePair(0);
            mana = new AttributePair(0);
        }
        #endregion
    }
}
```

There is an enumeration at the name space level so it will be available with out having to use the class name to reference it. It is for the gender of the entity. You won't always need to know the gender of an entity, like a green slime from D&D, so there is an **Unknown** value as well as **Male, Female** and **NonBinary**. By default an entity will have an **Unknown** gender that you can override in any class that inherits from **Entity**.

There are a number of regions in this class to split it up logically. The first region is the **Vital Field and Property** region. This region has fields and properties for the entity's name, entity's type and the entity's gender.

The next region is the **Basic Attribute Field and Property** region. This region has fields for the six

basic attributes: **Strength**, **Dexterity**, **Cunning**, **Willpower**, **Magic**, and **Constitution**. There are also modifier fields for each of the attributes. There are properties for each of the six basic attributes. The get part is public and returns the attribute plus the modifier. The set part is protected and just sets the field of the property name starting with a lower case letter.

Then comes the **Calculated Attribute Field and Property** region. There are fields and properties the three paired attributes: **health**, **mana**, and **stamina**. The properties that expose them are public and read only. This allows use of the methods to modify their values but not modify them directly. I also included fields for the **attack**, **damage**, and **defense** attributes of the entity. I will add properties for them down the road when they are needed.

There is also a **Level Field and Property** region that has fields and properties for the level of the entity and the experience of the entity. Like the basic attributes the properties that expose the fields and public get and protected set parts to them.

The last region in this class is the **Constructor** region. It has two constructors. There is a private constructor and a public constructor. The private constructor just sets values to 0. It will be useful later on. The public constructor takes an **EntityData** parameter. It sets the **entityType** field and the basic attribute fields. It then creates the calculated fields and sets them to 0 as well. Later you will use the formula fields to calculate their values.

The next step in the tutorial is to add in the base character classes. Namely classes for fighters, rogues, wizards and priests. The code is pretty much the same. This is why I had been thinking about the class system to be dynamic rather than static. I will write tutorials on how to use a dynamic rather than static system for those who are interested in it.

The first class will be the **Fighter** class. Right click the **CharacterClasses** folder, select **Add** and then **Class**. Name this new class **Figher**. This is the code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.CharacterClasses
{
    public class Fighter : Entity
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public Fighter(EntityData entityData)
        : base(entityData)
        {
        }

        #endregion
```

```
        #region Method Region
        #endregion
    }
}
```

This class inherits from **Entity**. There are regions in the class but the only region with code is the **Constructor** region. There is a constructor because the public constructor for **Entity** requires an **EntityData** parameter. The other classes have the same code the only difference is they are called **Rogue**, **Priest**, and **Wizard**. Add classes to the **CharacterClasses** folder like you did for the **Fighter** class but call them **Rogue**, **Priest**, and **Wizard**. The code for each class follows next in the same order.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.CharacterClasses
{
    public class Rogue : Entity
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public Rogue(EntityData entityData)
        : base(entityData)
        {
        }

        #endregion

        #region Method Region
        #endregion
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.CharacterClasses
{
    public class Priest : Entity
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public Priest(EntityData entityData)
        : base(entityData)
```

```
        {
        }

        #endregion

        #region Method Region
        #endregion
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.CharacterClasses
{
    public class Wizard : Entity
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        public Wizard(EntityData entityData)
        : base(entityData)
        {
        }

        #endregion

        #region Method Region
        #endregion
    }
}
```

What I'm going to do next is add a class to the game. This class will represent the world the game takes place in. This class will have some MonoGame elements in it. It will also require some of the elements from the **RpgLibrary**. The solution I chose to go with is add the world class into the **MGRpgLibrary** and add a reference for the **RpgLibrary** to the **MGRpgLibrary**.

To start, right click the **MGRpgLibrary** project and select the **Add** and then **Project Reference** item. In the dialog box that pops up select the **Projects** tab. From that tab select the **RpgLibrary** entry. The **RpgLibrary** is now referenced in the **MGRpgLibrary** project so you can use the classes from that project there. Right click the **MGRpgLibrary** project, select **Add** and then **New Folder**. Name this new folder **WorldClasses**. This folder will hold classes related to the world the player will explore. Right click the **WorldClasses** folder, select **Add** and then **Class**. Name this class **World**. This is the code for the **World** class.

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

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using RpgLibrary.CharacterClasses;
using RpgLibrary.ItemClasses;
using MGRpgLibrary.TileEngine;
using MGRpgLibrary.SpriteClasses;

namespace MGRpgLibrary.WorldClasses
{
    public class World
    {
        #region Graphic Field and Property Region

        Rectangle screenRect;

        public Rectangle ScreenRectangle
        {
            get { return screenRect; }
        }

        #endregion

        #region Item Field and Property Region

        ItemManager itemManager = new ItemManager();

        #endregion

        #region Level Field and Property Region
        #endregion

        #region Constructor Region

        public World(Rectangle screenRectangle)
        {
            screenRect = screenRectangle;
        }

        #endregion

        #region Method Region

        public void Update(GameTime gameTime)
        {
        }

        public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
        {
        }

        #endregion
    }
}
```

It needs to be fleshed out a bit but it is a good beginning. There are using statements for XNA Framework name space that will be used. There are also using statements for the two name spaces in the **RpgLibrary** project. There are also using statements for the tile engine sprite classes of the **MGRpgLibrary**.

There are a few field and property regions in this class. The **Graphic Field and Property** region is for fields and properties related to graphics like the **screenRect** field that represents the screen as a rectangle and the property that exposes it. The **Item Field and Property** region will hold fields an properties related to the items in the game. Again, this is items that can be in your game. It is not for inventory, that will be implemented later on. The **Level Field and Property** region will be for fields and properties related to levels in the game.

The constructor takes the area for the screen represented by a rectangle and sets the field. This class will need to update itself and draw itself so there are method stubs for updating and drawing. Both methods take **GameTime** parameters and the **Draw** method takes an additional **SpriteBatch** parameter.

Things are starting to come together but there is still a lot of work to be done. I think this is 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, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials.

Good luck in your game programming adventures!
Cynthia