# Eyes of the Dragon Tutorials
# Part 15
# Skills, Spells, and Talents

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.

In this tutorial I'm going to get started with adding skills, talents, and spells to the game. I will be adding in place holders that will be filled in as the game progresses. First I want to explain what skills, spells, and talents are for.

Skills can be learned by any humanoid creature with intelligence. Spells are specific to characters that have an understanding of magic. Talents are available to characters that are not magic using, basically special moves like bashing an enemy with a shield, picking a lock, or disarming a trap.
Before I get to the game there is I thought I'd share with you that I use a lot. It is a pain to always be entering all of the **#region** and **#endregion** directives into your code but it is good for organizational purposes to use them. I made a snippet for adding them and added it to the snippet manager in Visual Studio. So, I can just right click in the code editor and insert the snippet. Right click your game, select **Add** and then **New Item**. Select the **XML File** entry and name it **regions.snippet**. This is the code.

```
<?xml version="1.0" encoding="utf-8" ?>
<CodeSnippets xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
	<CodeSnippet Format="1.0.0">
		<Header>
			<Title>
				Regions Snippet
			</Title>
		</Header>
		<Snippet>
			<Code Language="CSharp">
				<![CDATA[#region Field Region
#endregion

#region Property Region
#endregion

#region Constructor Region
#endregion

#region Method Region
#endregion

#region Virtual Method region
#endregion
]]>
			</Code>
		</Snippet>
	</CodeSnippet>
</CodeSnippets>
```

That is what a snippet file looks like for the snippet manager in Visual Studio. The **Title** tag is the title of the snippet in the snippet manager, I named it **Regions Snippet**. I set the **Language** attribute of the **Code** tag to **CSharp** as it is a C# snippet. In between the inner square brackets of **CDATA** is where you place to code to insert. Now, from the **File** menu select **Save As** to save it to a different directory. Navigate to the **\Visual Studio 2019\Code Snippets\Visual C#\My Code Snippets** folder and save the file there. Now, right click the **regions.snippet** file in the solution explorer and select **Remove**. When you want to use the snippet right click in the editor where you want to insert it, select **Insert Snippet**, **My Code Snippets**, then **Regions Snippet**. You can make many more snippets. I've just found this one incredibly useful when it comes to organizing code.

I want to add a static class to the **RpgLibrary** for handling game mechanics, like generating random numbers, resolving skill use, etc. Right click the **RpgLibrary**, select **Add** and then **Class**. Name this new class **Mechanics**. This is the code for that class.

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

namespace RpgLibrary
{
    public enum DieType { D4 = 4, D6 = 6, D8 = 8, D10 = 10, D12 = 12, D20 = 20, D100 = 100 }

    public static class Mechanics
    {
        #region Field Region

        static Random random = new Random();

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region

        public static int RollDie(DieType die)
        {
            return random.Next(0, (int)die) + 1;
        }

        #endregion

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

There is an enumeration, **DieType**, that has entries for different dice that are found in role playing games. I have it at the name space level so you don't have to use a class name as well as the enum name when referencing a **DieType** variable. I also give them values based on the type of die. For example, the **D8** member has a value of **8**. So you don't have to use if or switch statements to get the upper limit of the die. The value **8** is already associated with **D8**.

The class itself is static as the fields, properties, and methods are all meant to be static. You reference the fields, properties and methods using the class name rather than an object of the class type. It is common to group items that are similar together. A good example is the **MathHelper** class. There are a number of useful math functions in the **MathHelper** class. There are going to be many methods and properties that will be useful and I'm going to keep them all in the same place.

At the moment there is just one field and one method. The variable is of type **Random** and will be used for generating random numbers when they are needed. The static method, **RollDie**, will be called when you want to roll a die. You pass in the type of die that you want to roll. The overload of the **Next** method that I use takes two parameters. The first is the inclusive lower bound and the second is the exclusive upper bound. That means the first number is included in the range of numbers to create and the second is not. That is why I pass in zero for the first value and the die cast to an integer for the second and add one to the result. So, for the **D8** example the number generated will be between 0 and 7. Adding 1 to that gives the 1 to 8 range we are looking for.

I first want to add classes for skills, spells, and talents. Right click the **RpgLibrary**, select **Add** and then **New Folder**. Name this new folder **SkillClasses**. Repeat the process twice and name the folders **SpellClasses** and **TalentClasses**.

Now to each of those folders I want to add in a class. Right click each of the folders, select **Add** and then **Class**. To the **SkillClasses** folder you want to add a class **Skill**, to the **SpellClasses** folder **Spell**, and to the **TalentClasses** folder **Talent**. Right click each of the folders again, select **Add** and then **Class**. To **SkillClasses** add a new class **SkillData**, to **SpellClasses** add a new class **SpellData**, and to **TalentClasses** add a new class **TalentData**. One last time, right click each of the folders, select **Add** and then **Class**. To **SkillClasses** you want to add **SkillDataManager**, to **SpellClasses** you want to add **SpellDataManager**, and to **TalentClasses** you want to add **TalentDataManager**.

To the class and class data classes I added in regions and made the classes public. This is the code that I added to the **Skill** and **SkillData** classes. I included an enum called **DifficultyLevel** in the **Skill** class. This is how hard it is to use the skill. It can be very easy to perform a skill or it can be next to impossible. The harder a skill is to perform the lower its value.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.SkillClasses
{
    public enum DifficultyLevel
    {
        Master = -50,
        Expert = -25,
        Improved = -10,
        Normal = 0,
        Easy = 25,
    }
    public class Skill
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.SkillClasses
{
    public class SkillData
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

I did that using the snippet I showed you at the start of the tutorial. Alternatively after adding the regions for the **Skill** class you could copy and paste the region code. The code for the **Spell**, **SpellData**, **Talent** and **TalentData** classes follows next.

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

namespace RpgLibrary.SpellClasses
{
    public class Spell
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

        #region Virtual Method Region
        #endregion
    }
}

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

namespace RpgLibrary.SpellClasses
{
    public class SpellData
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}

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

namespace RpgLibrary.TalentClasses
{
    public class Talent
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion
        #region Method Region
        #endregion

        #region Virtual Method Region
        #endregion
    }
}

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

namespace RpgLibrary.TalentClasses
{
    public class TalentData
    {
        #region Field Region
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

To the manager classes I added a bit more code. I added in private readonly fields, public properties to expose those fields, and in the constructor I create the fields. The code for my **SkillDataManager**, **SpellDataManager**, and **TalentDataManager** classes follows next.

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

namespace RpgLibrary.SkillClasses
{
    public class SkillDataManager
    {
        #region Field Region

        readonly Dictionary<string, SkillData> skillData;

        #endregion

        #region Property Region

        public Dictionary<string, SkillData> SkillData
        {
            get { return skillData; }
        }

        #endregion

        #region Constructor Region

        public SkillDataManager()
        {
            skillData = new Dictionary<string, SkillData>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace RpgLibrary.SpellClasses
{
    public class SpellDataManager
    {
        #region Field Region

        readonly Dictionary<string, SpellData> spellData;

        #endregion

        #region Property Region

        public Dictionary<string, SpellData> SpellData
        {
            get { return spellData; }
        }

        #endregion

        #region Constructor Region

        public SpellDataManager()
        {
            spellData = new Dictionary<string, SpellData>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

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

namespace RpgLibrary.TalentClasses
{
    public class TalentDataManager
    {
        #region Field Region

        readonly Dictionary<string, TalentData> talentData;

        #endregion

        #region Property Region

        public Dictionary<string, TalentData> TalentData
        {
            get { return talentData; }
        }

        #endregion

        #region Constructor Region

        public TalentDataManager()
        {
            talentData = new Dictionary<string, TalentData>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

There will be modifiers to skills, spells, and talents. I say modifiers rather than bonuses because I feel that bonuses are beneficial and there could be negative modifiers. I'm going to add a class to represent a modifier. Right click the **RpgLibrary** project, select **Add** and then **Class**. Name the class **Modifier**. This is the code.

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

namespace RpgLibrary
{
    public struct Modifier
    {
        #region Field Region

        public int Amount;
        public int Duration;
        public TimeSpan TimeLeft;

        #endregion

        #region Constructor Region
        public Modifier(int amount)
        {
            Amount = amount;
            Duration = -1;
            TimeLeft = TimeSpan.Zero;
        }

        public Modifier(int amount, int duration)
        {
            Amount = amount;
            Duration = duration;
            TimeLeft = TimeSpan.FromSeconds(duration);
        }

        #endregion

        #region Method Region

        public void Update(TimeSpan elapsedTime)
        {
            if (Duration == -1)
                return;

            TimeLeft -= elapsedTime;

            if (TimeLeft.TotalMilliseconds < 0)
            {
                TimeLeft = TimeSpan.Zero;
                Amount = 0;
            }
        }

        #endregion

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

The first thing you will notice is that I made this a structure rather than a class. The reason I did this is that you will be adding and removing them frequently. That is one reason to use a structure rather than a class.

There are three fields in the structure and they are all public. The first, **Amount**, is the amount of the modifier. **Duration** is how long the modifier lasts, measured in seconds, and the **TimeLeft** tracks how much time is left before the modifier expires. If you set the **Duration** to -1 the modifier lasts until it is dispelled some how.

There are two constructors in the structure. The first takes the amount of the modifier as a parameter. The second takes the amount and the duration. The first constructor sets the **Amount** field with the value passed in, **Duration** to -1, and **TimeLeft** to **TimeSpan.Zero**. The second sets the **Amount** and **Duration** fields to the values passed in. It then creates the **TimeLeft** field using the **FromSeconds** method of the **TimeSpan** class.

The **Update** method takes as a parameter a **TimeSpan** object. I can't pass in a **GameTime** object because this library has nothing to do with MonoGame. The **GameTime** class's **ElapsedGameTime** property is a **TimeSpan** object so that can be passed to the **Update** method. I first check to see if **Duration** is -1 and if it is I exit the method as this modifier lasts until it is dispelled in some way. I then reduce the **TimeLeft** field by the **elapsedTime** parameter. If the **TotalMilliseconds** is less than zero I set **TimeLeft** to be **TimeSpan.Zero** and **Amount** to 0. If there is a modifier that has an amount of 0 it will be removed from the list of modifiers.

I will be adding skills that have crafting formulas or recipes. I decided to go with the term recipe for them in the end. Since they are related to skills I put them into the **SkillClasses** folder. Right click the **SkillClasses** folder, select **Add** and then **Class**. Name this new class **Recipe**. I added in a structure to this class as well. Add the following code to the **Recipe** class.

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

namespace RpgLibrary.SkillClasses
{
    public struct Reagents
    {
        #region Field Region

        public string ReagentName;
        public ushort AmountRequired;

        #endregion

        #region Property Region
        #endregion

        #region Constructor Region
        public Reagents(string reagent, ushort number)
        {
            ReagentName = reagent;
            AmountRequired = number;
        }

        #endregion

        #region Method Region
        #endregion
    }

    public class Recipe
    {
        #region Field Region

        public string Name;
        public Reagents[] Reagents;
        #endregion

        #region Property Region
        #endregion

        #region Constructor Region

        private Recipe()
        {
        }

        public Recipe(string name, params Reagents[] reagents)
        {
            Name = name;
            Reagents = new Reagents[reagents.Length];

            for (int i = 0; i < reagents.Length; i++)
                Reagents[i] = reagents[i];
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

The structure **Reagents** is for the different reagents in a recipe. I use reagent to mean herbs and other items that can be used to make potions, poisons, traps, etc. Since it is a structure I made the two fields public. The first field, **ReagentName**, is the name of the reagent from the item classes. I will add that class shortly. The **AmountRequired** field is the amount of that reagent required to craft the item and is an unsigned short so its value can never be negative. The constructor takes two parameters. A string for the name of the reagent and the amount of the reagent that is required.

I decided against creating a data class for recipes. That just seemed unnecessary as there isn't anything difficult about recipes. The **Recipe** class has two public fields. The first is a string for the name of the recipe, **Name**, and the second is an array of **Reagents** that holds the reagents required for the recipe.

There are two constructors for the class. The first a private constructor with no parameters to be used to deserialize a recipe. The second takes a string parameter for the name and a **params** of **Reagents**. You could just as easily use an array of **Reagents** or a **List<Reagents>** for it. It might be a good idea as you could enforce that there be at least one **Reagents** for a recipe. The second constructor sets the **Name** field to the **name** parameter. I then creates a new array of **Reagents** the same length as the reagents passed in. I then copy the values from the **reagents** parameter to the **Reagents** field. Structures are value types so you don't need to worry about unexpected side effects from changing values like you do in a class that is a reference type.

You will also want a class to manage all of the recipes in the game. Right click the **SkillClasses** folder, select **Add** and then **Class**. Name this new class **RecipeManager**. The code for that class follows next.

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

namespace RpgLibrary.SkillClasses
{
    public class RecipeManager
    {
        #region Field Region

        readonly Dictionary<string, Recipe> recipies;

        #endregion
        #region Property Region

        public Dictionary<string, Recipe> Recipies
        {
            get { return recipies; }
        }

        #endregion

        #region Constructor Region

        public RecipeManager()
        {
            recipies = new Dictionary<string, Recipe>();
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region
        #endregion
    }
}
```

Nothing really new here. It is like all of the other manager classes I've created so far. There is a private readonly field for the recipes. It is a **Dictionary<string, Recipe>**. There is a read only field to expose the field. The constructor just creates a new instance of the dictionary.

The last thing I'm going to do is add in a classes to the **ItemClasses** folders for reagents. Right click the **ItemClasses** folder, select **Add** and then **Class**. Name this new class **ReagentData**. This is the code for that class.

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

namespace RpgLibrary.ItemClasses
```

```
{
    public class ReagentData
    {
        public string Name;
        public string Type;
        public int Price;
        public float Weight;

        public ReagentData()
        {
        }

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

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

            return toString;
        }
    }
}
```

There are only four fields of interest to us from the **BaseItem** class. You want the name of the reagent, the type of the reagent, the price of the reagent, and the weight of the reagent. You could probably even get away with out the weight as for the inventory I won't be keeping track of weights. Weight will be of importance to what a character is actually carrying. I will show you how I'm going to deal with the fact that **BaseItem** requires zero or more strings for classes that are allowed to use the item in a moment.

The **ToString** method returns the a string with the name, type, price, and weight of the reagent. Right click the **ItemClasses** folder again, select **Add** and then **Class**. Name this new class **Reagent**. The code for that class follows next.

There are no new fields or properties in this class and as you can see it doesn't have the **params** parameter at the end. This is where the properties **params** allowing zero or more parameters comes into play. I can just pass null in to the call to the base constructor. So, what is the purpose of inheriting from **BaseItem**? Well, doing that you can still use polymorphism, the ability of a base class to act as an inherited class at run time. When you get to inventory you can have a backpack of **BaseItem** instead of all of the different item types for the items the player/party is carrying. In the **Clone** method you see that I don't need to even pass in null as a parameter to the constructor. Just the values that I'm interested in.

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

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

        #region Property Region
        #endregion

        #region Constructor Region

        public Reagent(
            string reagentName,
            string reagentType,
            int price,
            float weight)
            : base(reagentName, reagentType, price, weight, null)
        {
        }

        #endregion

        #region Method Region
        #endregion

        #region Virtual Method region

        public override object Clone()
        {
            Reagent reagent = new Reagent(Name, Type, Price, Weight);

            return reagent;
        }

        #endregion
    }
}
```

The last thing I'm going to do in this tutorial is update the **ItemDataManager** class. What I'm going to do is add in a field for the **ReagentData** objects and a property to expose it. Change that class 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>();

        #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; }
        }

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region
        #endregion
    }
}
```

I think that this is enough for this tutorial. It covers the original tutorial with little modification. So, I encourage you to visit my blog at, https://cynthiamcmahon.ca/blog/, for the latest news on my tutorials and other goodness.

Good luck in your game programming adventures!
Cynthia