Shadow Monsters – MonoGame Tutorial Series
Chapter 4
Shadow Monsters

This tutorial series is about creating a Pokemon style game with the MonoGame Framework called Shadow Monsters. The tutorials will make more sense if you read them in order as each tutorial builds on the previous tutorials. You can find the list of tutorials on my blog: Shadow Monsters. The source code for each tutorial will be available as well. I will be using Visual Studio 2019 Community for the series. The code should compile on the 2013, 2015 and 2017 versions as well.

I also want to mention that I assume you have a basic understanding of C# and MonoGame. If you don't I recommend that you learn basic C# and work with MonoGame a little. Enough to know the basics of fields, properties, methods, classes and the MonoGame framework.

In this tutorial I am going to cover adding shadow monsters for the player to interact with. I will be taking a bottom up approach. Shadow monsters have moves so I will start with moves. Open up your project from where we left of last time. Right click the ShadowMonsters project in the Solution Explorer, select Add and then New Folder. Name this new folder ShadowMonsters. Right click the ShadowMonsters folder, select Add and then New Item. From the list that comes up choose Interface. Name this new interface IMove. Here is the code for the interface.

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

namespace ShadowMonsters.ShadowMonsters
{
    public enum Target
    {
        Self, Enemy
    }

    public enum MoveType
    {
        Attack, Heal, Buff, Debuff, Status
    }

    public enum MoveElement
    {
        None, Dark, Earth, Fire, Light, Water, Wind
    }
```

```
    public interface IMove
    {
        string Name { get; }
        Target Target { get; }
        MoveType MoveType { get; }
        MoveElement MoveElement { get; }
        int UnlockedAt { get; set; }
        bool Unlocked { get; }
        int Duration { get; set; }
        int Attack { get; }
        int Defense { get; }
        int Speed { get; }
        int Health { get; }
        void Unlock();
        object Clone();
    }
}
```

Included in the interface are two enumerations. The first is the what type of target the move is against, self or enemy. The second is the type of move, Attack, Heal, Buff, Debuff and Status. Attack is an attack on the target. Heal will heal the target. Buff increases one, or more, of the shadow monster's attributes and debuff decreases the attributes. Status changes the status of the shadow monster such as poisoned or asleep. The next is the element associated with the move. Shadow monsters are one of six elements: Dark, Earth, Fire, Light, Wind and Water. Moves have either one of those elements or no element at all.

To implement the interface you must have a number of read only properties. The first is the name of the move. Next is the target followed by the type of move and the element of the move. Not all moves are known at once and are unlocked at a certain level. Next is if the move is unlocked or not. Duration is typically zero for attacks and a greater number for buffs, debuffs and status changes. Moves have a attack, defense, speed and health attributes associated with them. There are methods to unlock a move and clone a move. Yes, I'm a big fan of cloning objects.

Now let's create two basic moves: Tackle and Block. Tackle does physical damage and Block increases the shadow monster's defense slightly. Right click the ShadowMonsters folder in the Solution Explorer, select Add and then Class. Name this new class Tackle. Here is the code.

```
using ShadowMonsters.GameStates;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ShadowMonsters.ShadowMonsters
{
    public class Tackle : IMove
    {
        #region Field Region

        private readonly string name;
        private readonly Target target;
```

```
private readonly MoveType moveType;
private readonly MoveElement moveElement;
private bool unlocked;
private int unlockedAt;
private int duration;
private readonly int attack;
private readonly int defense;
private readonly int speed;
private readonly int health;

#endregion

#region Property Region

public string Name
{
    get { return name; }
}

public Target Target
{
    get { return target; }
}

public MoveType MoveType
{
    get { return moveType; }
}

public MoveElement MoveElement
{
    get { return moveElement; }
}

public int UnlockedAt
{
    get { return unlockedAt; }
    set { unlockedAt = value; }
}

public bool Unlocked
{
    get { return unlocked; }
}

public int Duration
{
    get { return duration; }
    set { duration = value; }
}

public int Attack
{
    get { return attack; }
}

public int Defense
{
    get { return defense; }
}
```

```
        public int Speed
        {
            get { return speed; }
        }

        public int Health
        {
            get { return health; }
        }

        #endregion

        #region Constructor region

        public Tackle()
        {
            name = "Tackle";
            target = Target.Enemy;
            moveType = MoveType.Attack;
            moveElement = MoveElement.None;
            duration = 1;
            unlocked = false;
            attack = MoveManager.Random.Next(0, 0);
            defense = MoveManager.Random.Next(0, 0);
            speed = MoveManager.Random.Next(0, 0);
            health = MoveManager.Random.Next(10, 15);
        }

        #endregion

        #region Method Region

        public void Unlock()
        {
            unlocked = true;
        }

        public object Clone()
        {
            Tackle tackle = new Tackle
            {
                unlocked = this.unlocked
            };

            return tackle;
        }

        #endregion
    }
}
```

The class implements the IMove interface. There are fields for the properties that are required by the interface and the properties themself. The constructor initializes the fields. I reference a class MoveManager that I will show you shortly for a random number generator. The move does not affect attack, defense or spee but does affect health. It deals 10 to 14 points of damage. This is because the upper bound of the overload of Next I'm using is exclusive. Unlock just sets the unlocked field to true. Clone creates a new Tackle object and sets the

unlocked field to be the value of the current instance and returns it.

Block is basically the same the difference is that it is a buff that only affects defense. Right click the ShadowMonsters folder, select Add and then Class. Name this new class Block. Here is the code for the Block class.

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

namespace ShadowMonsters.ShadowMonsters
{
    public class Block : IMove
    {
        #region Field Region

        private readonly string name;
        private readonly Target target;
        private readonly MoveType moveType;
        private readonly MoveElement moveElement;
        private bool unlocked;
        private int unlockedAt;
        private int duration;
        private readonly int attack;
        private readonly int defense;
        private readonly int speed;
        private readonly int health;

        #endregion

        #region Property Region

        public string Name
        {
            get { return name; }
        }

        public Target Target
        {
            get { return target; }
        }

        public MoveType MoveType
        {
            get { return moveType; }
        }

        public MoveElement MoveElement
        {
            get { return moveElement; }
        }

        public int UnlockedAt
        {
            get { return unlockedAt; }
            set { unlockedAt = value; }
        }
```

```
public bool Unlocked
{
    get { return unlocked; }
}

public int Duration
{
    get { return duration; }
    set { duration = value; }
}

public int Attack
{
    get { return attack; }
}

public int Defense
{
    get { return defense; }
}

public int Speed
{
    get { return speed; }
}

public int Health
{
    get { return health; }
}

#endregion

#region Constructor Region

public Block()
{
    name = "Block";
    target = Target.Self;
    moveType = MoveType.Buff;
    moveElement = MoveElement.None;
    unlocked = false;
    duration = 5;
    attack = MoveManager.Random.Next(0, 0);
    defense = MoveManager.Random.Next(2, 6);
    speed = MoveManager.Random.Next(0, 0);
    health = MoveManager.Random.Next(0, 0);
}

#endregion

#region Method Region

public void Unlock()
{
    unlocked = true;
}

public object Clone()
```

```
        {
            Block block = new Block
            {
                unlocked = this.unlocked
            };

            return block;
        }

        #endregion
    }
}
```

Now that we have a couple moves let's add the MoveManager class. Right click the ShadowMonsters folder in the Solution Explorer, select Add and then class. Name this new class MoveManager. Here is the code.

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

namespace ShadowMonsters.ShadowMonsters
{
    public static class MoveManager
    {
        #region Field Region

        private static readonly Dictionary<string, IMove> allMoves = new Dictionary<string,
IMove>();
        private static readonly Random random = new Random();

        #endregion

        #region Property Region

        public static Random Random => random;

        #endregion

        #region Constructor Region
        #endregion

        #region Method Region

        public static void FillMoves()
        {
            AddMove("Tackle", new Tackle());
            AddMove("Block", new Block());
        }

        public static IMove GetMove(string name)
        {
            if (allMoves.ContainsKey(name))
                return (IMove)allMoves[name].Clone();

            return null;
        }
```

```
        public static void AddMove(IMove move)
        {
            if (!allMoves.ContainsKey(move.Name))
                allMoves.Add(move.Name, move);
        }

        #endregion
    }
}
```

The class is static because I only want one instance of the class. It could have implemented the singleton design pattern. For fields there is a Dictionary<string, IMove> that holds all of the moves in the game. There is also the Random field that moves use when generating random values. There is a property to expose the Random field.

The FillMoves method creates a new move for each type and calls the AddMove method. GetMove is used to retrieve a clone of the requested move if it exists or returns null. AddMove adds the move to the Dictionary of moves if the key does not already exist.

Now it is time to add the shadow monster. Right click the ShadowMonsters folder, select Add and then Class. Name this new class ShadowMonster. Here is the code for that class. Warning: This is a wall of code.

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ShadowMonsters.ShadowMonsters
{
    public enum AvatarElement
    {
        Dark, Earth, Fire, Light, Water, Wind
    }

    public class ShadowMonster
    {
        #region Field Region

        private static readonly Random random = new Random();

        private Texture2D texture;
        private string name;
        private string displayName;
        private AvatarElement element;
        private int level;
        private long experience;
        private int costToBuy;
        private int speed;
        private int attack;
        private int defense;
        private int health;
        private int currentHealth;
```

```
public Rectangle Source { get; private set; }

private readonly List<IMove> effects;
private Dictionary<string, IMove> knownMoves;
private bool isAsleep;
private bool isPoisoned;
private bool isConfused;
private bool isParalyzed;

#endregion

#region Property Region

public string Name
{
    get { return name; }
}

public string DisplayName
{
    get { return displayName; }
}

public int Level
{
    get { return level; }
    set { level = (int)MathHelper.Clamp(value, 1, 100); }
}

public long Experience
{
    get { return experience; }
}

public Texture2D Texture
{
    get { return texture; }
}

public Dictionary<string, IMove> KnownMoves
{
    get { return knownMoves; }
}

public AvatarElement Element
{
    get { return element; }
}

public List<IMove> Effects
{
    get { return effects; }
}

public static Random Random
{
    get { return random; }
}

public int BaseAttack
{
    get { return attack; }
}
```

```
public int BaseDefense
{
    get { return defense; }
}

public int BaseSpeed
{
    get { return speed; }
}

public int BaseHealth
{
    get { return health; }
}

public int CurrentHealth
{
    get { return currentHealth; }
}

public bool Alive
{
    get { return (currentHealth > 0); }
}

public bool IsAsleep { get => isAsleep; set => isAsleep = value; }
public bool IsPoisoned { get => isPoisoned; set => isPoisoned = value; }
public bool IsConfused { get => isConfused; set => isConfused = value; }
public bool IsParalyzed { get => isParalyzed; set => isParalyzed = value; }
public bool IsFainted { get; set; }
#endregion

#region Constructor Region

private ShadowMonster()
{
    level = 1;
    knownMoves = new Dictionary<string, IMove>();
    effects = new List<IMove>();
    IsAsleep = false;
    IsParalyzed = false;
    IsPoisoned = false;
}

#endregion

#region Method region

public void ResoleveMove(IMove move, ShadowMonster target)
{
    bool found;
    switch (move.Target)
    {
        case Target.Self:
            if (move.MoveType == MoveType.Buff)
            {
                found = false;
                for (int i = 0; i < effects.Count; i++)
                {
                    if (effects[i].Name == move.Name)
                    {
                        effects[i].Duration += move.Duration;
```

```
                            found = true;
                        }
                    }

                    if (!found)
                    {
                        effects.Add((IMove)move.Clone());
                    }
                }
                else if (move.MoveType == MoveType.Heal)
                {
                    currentHealth += move.Health;
                    if (currentHealth > health)
                    {
                        currentHealth = health;
                    }
                }
                else if (move.MoveType == MoveType.Status)
                {
                }

                break;
            case Target.Enemy:
                if (move.MoveType == MoveType.Debuff)
                {
                    found = false;
                    for (int i = 0; i < target.Effects.Count; i++)
                    {
                        if (target.Effects[i].Name == move.Name)
                        {
                            target.Effects[i].Duration += move.Duration;
                            found = true;
                        }
                    }

                    if (!found)
                    {
                        target.Effects.Add((IMove)move.Clone());
                    }
                }
                else if (move.MoveType == MoveType.Attack)
                {
                    float modifier = GetMoveModifier(move.MoveElement,
target.Element);

                    float tDamage = GetAttack() + move.Health * modifier -
target.GetDefense();

                    if (tDamage < 1f)
                    {
                        tDamage = 1f;
                    }

                    target.ApplyDamage((int)tDamage);
                }

                break;
        }
    }

    public static float GetMoveModifier(MoveElement moveElement, AvatarElement
avatarElement)
    {
```

```
float modifier = 1f;

switch (moveElement)
{
    case MoveElement.Dark:
        if (avatarElement == AvatarElement.Light)
        {
            modifier += .25f;
        }
        else if (avatarElement == AvatarElement.Wind)
        {
            modifier -= .25f;
        }

        break;
    case MoveElement.Earth:
        if (avatarElement == AvatarElement.Water)
        {
            modifier += .25f;
        }
        else if (avatarElement == AvatarElement.Wind)
        {
            modifier -= .25f;
        }

        break;
    case MoveElement.Fire:
        if (avatarElement == AvatarElement.Wind)
        {
            modifier += .25f;
        }
        else if (avatarElement == AvatarElement.Water)
        {
            modifier -= .25f;
        }

        break;
    case MoveElement.Light:
        if (avatarElement == AvatarElement.Dark)
        {
            modifier += .25f;
        }
        else if (avatarElement == AvatarElement.Earth)
        {
            modifier -= .25f;
        }

        break;
    case MoveElement.Water:
        if (avatarElement == AvatarElement.Fire)
        {
            modifier += .25f;
        }
        else if (avatarElement == AvatarElement.Earth)
        {
            modifier -= .25f;
        }

        break;
    case MoveElement.Wind:
        if (avatarElement == AvatarElement.Light)
        {
            modifier += .25f;
```

```
            }
            else if (avatarElement == AvatarElement.Earth)
            {
                modifier -= .25f;
            }

            break;

    }

    return modifier;
}

public void ApplyDamage(int tDamage)
{
    currentHealth -= tDamage;

    if (currentHealth > health)
    {
        currentHealth = health;
    }
}

public void Heal(int amount)
{
    currentHealth += amount;

    if (currentHealth > BaseHealth)
    {
        currentHealth = BaseHealth;
    }
}

public void Update()
{
    for (int i = 0; i < effects.Count; i++)
    {
        effects[i].Duration--;

        if (effects[i].Duration < 1)
        {
            effects.RemoveAt(i);
            i--;
        }
    }
}

public int GetAttack()
{
    int attackMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
        {
            attackMod += move.Attack;
        }

        if (move.MoveType == MoveType.Debuff)
        {
            attackMod -= move.Attack;
        }
    }
```

```
    return attack + attackMod;
}

public int GetDefense()
{
    int defenseMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
        {
            defenseMod += move.Defense;
        }

        if (move.MoveType == MoveType.Debuff)
        {
            defenseMod -= move.Defense;
        }
    }

    return defense + defenseMod;
}

public int GetSpeed()
{
    int speedMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
        {
            speedMod += move.Speed;
        }

        if (move.MoveType == MoveType.Debuff)
        {
            speedMod -= move.Speed;
        }
    }

    return speed + speedMod;
}

public int GetHealth()
{
    int healthMod = 0;

    foreach (IMove move in effects)
    {
        if (move.MoveType == MoveType.Buff)
        {
            healthMod += move.Health;
        }

        if (move.MoveType == MoveType.Debuff)
        {
            healthMod += move.Health;
        }
    }

    return health + healthMod;
}
```

```
public void StartCombat()
{
    effects.Clear();
    //currentHealth = health;
}

public long WinBattle(ShadowMonster target)
{
    int levelDiff = target.Level - level;
    long expGained;
    if (levelDiff <= -10)
    {
        expGained = 10;
    }
    else if (levelDiff <= -5)
    {
        expGained = (long)(100f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 0)
    {
        expGained = (long)(50f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 5)
    {
        expGained = (long)(5f * (float)Math.Pow(2, levelDiff));
    }
    else if (levelDiff <= 10)
    {
        expGained = (long)(10f * (float)Math.Pow(2, levelDiff));
    }
    else
    {
        expGained = (long)(50f * (float)Math.Pow(2, target.Level));
    }

    experience += expGained;
    return expGained;
}

public long LoseBattle(ShadowMonster target)
{
    long expGained = (long)((float)WinBattle(target) * .5f);
    experience += expGained;

    return expGained;
}

public bool CheckLevelUp()
{
    bool leveled = false;

    if (experience >= 50 * (1 + (long)Math.Pow((level - 1), 2.5)))
    {
        leveled = true;
        level++;
    }

    return leveled;
}

public object Clone()
{
```

```
            ShadowMonster monster = new ShadowMonster
            {
                name = this.name,
                displayName = this.displayName,
                texture = this.texture,
                element = this.element,
                costToBuy = this.costToBuy,
                level = this.level,
                experience = this.experience,
                attack = this.attack,
                defense = this.defense,
                speed = this.speed,
                health = this.health,
                currentHealth = this.health,
                Source = this.Source
            };

            foreach (string s in this.knownMoves.Keys)
            {
                monster.knownMoves.Add(s, this.knownMoves[s]);
            }

            return monster;
        }

        #endregion

        public void AssignPoint(string s, int p)
        {
            switch (s)
            {
                case "Attack":
                    attack += p;
                    break;
                case "Defense":
                    defense += p;
                    break;
                case "Speed":
                    speed += p;
                    break;
                case "Health":
                    health += p;

                    if (currentHealth > 0)
                    {
                        currentHealth += p;
                    }

                    break;
            }
        }

        public static ShadowMonster FromString(string description, ContentManager
content)
        {
            ShadowMonster monster = new ShadowMonster();
            string[] parts = description.Split(',');

            monster.name = parts[0];
            monster.displayName = parts[1];
            monster.texture = content.Load<Texture2D>(@"ShadowMonsterImages\" +
parts[1]);
            monster.element = (AvatarElement)Enum.Parse(typeof(AvatarElement),
```

```
parts[2]);
            monster.costToBuy = int.Parse(parts[3]);
            monster.level = int.Parse(parts[4]);
            monster.attack = int.Parse(parts[5]);
            monster.defense = int.Parse(parts[6]);
            monster.speed = int.Parse(parts[7]);
            monster.health = int.Parse(parts[8]);
            monster.currentHealth = monster.health;
            monster.Source = new Rectangle(int.Parse(parts[9]), int.Parse(parts[10]),
64, 64);

            monster.knownMoves = new Dictionary<string, IMove>();

            for (int i = 11; i < parts.Length; i++)
            {
                string[] moveParts = parts[i].Split(':');

                if (moveParts[0] != "None")
                {
                    IMove move = MoveManager.GetMove(moveParts[0]);
                    move.UnlockedAt = int.Parse(moveParts[1]);

                    if (move.UnlockedAt <= monster.Level)
                    {
                        move.Unlock();
                    }

                    monster.knownMoves.Add(move.Name, move);
                }
            }
            return monster;
        }
    }
}
```

Sorry for the wall of code but it is easier than adding method by method. Shadow monsters have one of six elements so there is an enumeration that describes those elements. In case there is a need for random numbers there is a static Random field. I know that there are a lot of Random fields and I said it was bad but in this case it makes sense to split responsibility between classes. Shadow monsters have images so there is a field for that, the name and display name. Name is something like Fire1 where as display name is like Charmander. This is because there might be a level 1 Charmander, Fire1, and a level 70 Charmander, Fire70. There are also fields for element, level, experience, costToBuy, speed, attack, defense, health, current health, source rectangle if the image is a sprite sheet, a List<IMove> which holds the current buffs and debuffs, Dictionary<string, IMove> which holds the moves that the shadow monster can know. There are also some status fields, isAsleep, isPoisoned, isConfused and isParalyzed.

There are properties to expose all of the fields. The status ones are all read and write where the others are read only. There is just a private constructor. It just initializes fields to default values.

The first method is ResolveMove which is used to apply a move to a shadow monster. There is a local variable found that is used when searching for buffs and debuffs. That is because adding a buff or debuff does not make it stronger, it makes it last longer. There is a switch on the move target. If the target is Self I check if the move is buff. If it is I set found to false and

loop through the active effects to see if the buff is already there. If it is I set found to true and increase the duration. If it is not found I add it to the list. If the move is a Heal I increment the currentHealth field then if it is greater than the maximum health I set it to the maximum health.

If the target is Enemy I check to see if the move is a debuff. If it is I send found to false and loop through the active effects to see if there is already that debuff. If there is I increase the duration. If not I add it to the list of effects.

If the move is an attack I call a method GetMoveModifier that I will get to shortly. What it does is allow for elemental strengths and weaknesses. For example, fire moves are weak against water type shadow monsters. I then calculate the damage done using the attack attribute of the shadow monster, the Health property of the move, the modifier and the defense. I call methods GetAttack and GetDefense that calculate the current values based on their base value and any effects. I will get to those methods shortly. A move will always do at least one damage so I check if it is less than one and set it to one if it is. Then I apply the damage.

The next method I want to tackle is the GetMoveModifier method. There is a base 100% effectiveness for moves. If a move is effective it does an additional 25% damage. If it is not effective it does 25% less damage. There is a switch on the element of the move. The cases work the same way. If the avatarElement is one value I increment the modifier by .25, or 25%, if it is the another element I decrement the modifier by .25. The choices were pretty random and I'd recommend fine tuning them to make sense in your own game.

The next method in the class is the ApplyDamage method which attacks a shadow monster's health. Since it is an attack I subtract the amount from the currentHealth field. There is a check to see if currentHealth is greater than the maximum health and if it is it sets current health to maximum health. This is not typically necessary but I included it in the off chance a negative value is passed into the method. There is a Heal method that works in reverse to the Attack method.

There is a method Update that removes expired buffs and debuffs. It loops over the items in the effects field. For each effect I decrease the duration by one, If the duration is less than one I remove it.

Next there are four methods GetAttack, GetDefense, GetSpeed and GetHealth that all work the same way but on the attack, defense, speed and health fields respectively. They loop over all of the moves. If it is a buff they increase a modifier and if it is a debuff it decrements the modifier. Finally it returns the attribute plus the modifier.

There is a method StartCombat that will be called when combat starts. All it does is clear the active effects. At one point I did have a shadow monster start at full health so there is a commented line that will do that. It is up to you if you want to have that feature.

Next there are two methods: WinBattle and LoseBattle, that are called when a shadow monster wins or loses a battle. It first calculates the difference in levels between the two shadow monsters. Then in a series of if-else-if I check what the difference is. Based on the difference I return different experience values. LoseBattle returns half of what WinBattle does so even if the shadow monster lost the battle they gain some experience.

After the WinBattle and LoseBattle methods comes the CheckLevelUp method that checks to see if the shadow monster has levelled up. I use a simple, but expensive, formula to check if the shadow monster has levelled up. You would want to use a better formula in your own game. If returns if the shadow monster has levelled up or not.

The next method is Clone, yes, again. It is used to make a copy of the shadow monster. The only piece that isn't setting fields is looping over the known moves. Known moves are all the moves the shadow monster can know. Some just won;t be unlocked yet.

When a shadow monster levels up they gain points in their attributes. The AssignPoints method assigns points to these attributes.

That leaves the FromString method that is used to create shadow mosters, It takes as parameters the string that defines the shadow monster and a ContentManager for loading the image. The string is a comma delimited string. The format of the string is name, display name, texture name, element, cost to buy, level, attack, defense, speed, health, current health, source rectangle X coordinate, source rectangle Y coordinate, and known moves. The method splits the string into parts on the comma. It then assigns the fields using the parts. In my game the shadow monsters were 64 by 64 so I use that when creating source rectangles. You will either want to include the height and width as parts or hard code them like I did in your game. For moves they are colon separated. I split each move on a colon. I get the move using the left side and the level unlocked on the right. If the shadow monster's level is greater than or equal to the unlock level I unlock it.

Next I want to add in a class to manage shadow monsters. Right click the ShadowMonsters folder in the Solution Explorer, select Add then Class. Name this class ShadowMonsterManager. Here is the code.

```
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ShadowMonsters.ShadowMonsters
{
    public class ShadowMonsterManager
    {
        #region Field Region

        private static readonly Dictionary<string, ShadowMonster> monsterList = new
Dictionary<string, ShadowMonster>();

        #endregion

        #region Property Region

        public static Dictionary<string, ShadowMonster> ShadowMonsterList
        {
            get { return monsterList; }
        }
```

```
        #endregion

        #region Constructor Region

        #endregion

        #region Method Region

        public static void AddShadowMonster(string name, ShadowMonster monster)
        {
            if (!monsterList.ContainsKey(name))
            {
                monsterList.Add(name, monster);
            }
        }

        public static ShadowMonster GetShadowMonster(string name)
        {
            return monsterList.ContainsKey(name) ? (ShadowMonster)monsterList[name].Clone() :
null;
        }

        public static void FromFile(string fileName, ContentManager content)
        {
            using (Stream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read))
            {
                try
                {
                    using (TextReader reader = new StreamReader(stream))
                    {
                        try
                        {
                            string lineIn = "";

                            do
                            {
                                lineIn = reader.ReadLine();
                                if (lineIn != null)
                                {
                                    ShadowMonster monster = ShadowMonster.FromString(lineIn,
content);
                                    if (!
monsterList.ContainsKey(monster.Name.ToLowerInvariant()))
                                        monsterList.Add(monster.Name.ToLowerInvariant(),
monster);
                                }
                            } while (lineIn != null);
                        }
                        catch (Exception exc)
                        {
                            exc.GetType();
                        }
                        finally
                        {
                            if (reader != null)
                                reader.Close();
                        }
                    }
                }
```

```
                catch (Exception exc)
                {
                    exc.GetType();
                }
                finally
                {
                    if (stream != null)
                        stream.Close();
                }
            }
        }

        #endregion
    }
}
```

There is one static field in the class. It is a Dictionary<string, ShadowMonster> and holds all of the defined shadow monsters. There is a property to expose the field.

There are methods to add a shadow monster and get a shadow monster. The get method returns a clone of the shadow monster.

The FromFile method reads a text file that holds the shadow monster definitions. First it creates a FileStream that points to the file name passed in. It then creates a TexrReader to read the file inside a try-catch block. I then read the file line by line. If a line is not null I call the FromString method of the ShadowMonster class passing in the line and the ContentManager that was passed into the method. If a shadow monster by that name does not exist in the dictionary I call the AddShadowMonster method. There are catches to catch any exceptions. For now they do nothing useful but in the future we will do something meaningful with the exceptions, In the finally clauses I close the reader and the stream.

The last thing that I'm going to tackle is adding in some shadow monsters. Right click the Content folder, select Add and then New Item. On the left select the General tab and on the right Text File. Name this text file ShadowMonsters.txt. Select the ShadowMonsters.txt file in the Solution Explorer. In the Properties window set Copy to Output Directory to Copy always. Add the following text to that file.

```
Dark1,Dark,Dark,100,1,9,12,10,50,0,0,Tackle:1,Block:1
Earth1,Earth,Earth,100,1,10,10,9,60,0,0,Tackle:1,Block:1
Fire1,Fire,Fire,100,1,12,8,10,50,0,0,Tackle:1,Block:1
Light1,Light,Light,100,1,12,9,10,50,0,0,Tackle:1,Block:1
Water1,Water,Water,100,1,9,12,10,50,0,0,Tackle:1,Block:1
Wind1,Wind,Wind,100,1,10,10,12,50,0,0,Tackle:1,Block:1
```

Add the following field to the GamePlayState.

```
        private ShadowMonsterManager monsterManager = new ShadowMonsterManager();
```

Finally update the LoadContent to the following.

```
        protected override void LoadContent()
        {
            ShadowMonsterManager.FromFile(@".\Content\ShadowMonsters.txt", content);
            TileSet set = new TileSet();
```

```
            set.TextureNames.Add("tileset1");
            set.Textures.Add(content.Load<Texture2D>(@"Tiles\tileset16-outdoors"));

            TileLayer groundLayer = new TileLayer(100, 100, 0, 1);
            TileLayer edgeLayer = new TileLayer(100, 100);
            TileLayer buildingLayer = new TileLayer(100, 100);
            TileLayer decorationLayer = new TileLayer(100, 100);

            for (int i = 0; i < 1000; i++)
            {
                decorationLayer.SetTile(random.Next(0, 100), random.Next(0, 100), 0,
random.Next(2, 4));
            }

            map = new TileMap(set, groundLayer, edgeLayer, buildingLayer, decorationLayer,
"level1");

            engine.SetMap(map);

            Animation animation = new Animation(3, 32, 36, 0, 0);
            animations.Add(AnimationKey.WalkUp, animation);

            animation = new Animation(3, 32, 36, 0, 36);
            animations.Add(AnimationKey.WalkRight, animation);

            animation = new Animation(3, 32, 36, 0, 72);
            animations.Add(AnimationKey.WalkDown, animation);

            animation = new Animation(3, 32, 36, 0, 108);
            animations.Add(AnimationKey.WalkLeft, animation);

            sprite = new AnimatedSprite(content.Load<Texture2D>(@"Sprites\mage_f"), animations)
            {
                CurrentAnimation = AnimationKey.WalkDown
            base.LoadContent();
            };
        }
```

There is a problem but I'm going to address it in the next tutorial. The problem is there are no images to load so loading the shadow monsters will fail. There is also a security concern because everything is in plain text. That will be addressed in a future tutorial.

I'm going to wrap this tutorial up here. I will be starting work on the next tutorial shortly. Keep checking back on the blog for news on that tutorial. I hope to have it up in the next week or so.

I wish you the best in your MonoGame Programming Adventures!