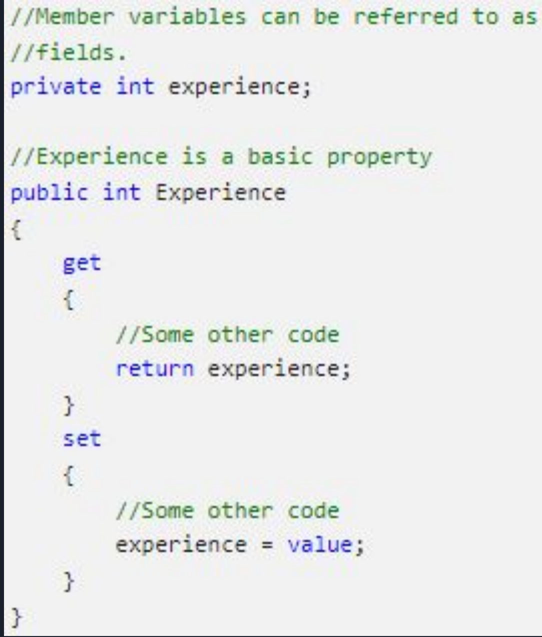
Intermediate Scripting

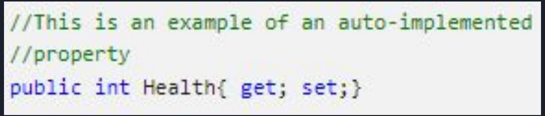
Concepts in C# and Unity

Creating Properties

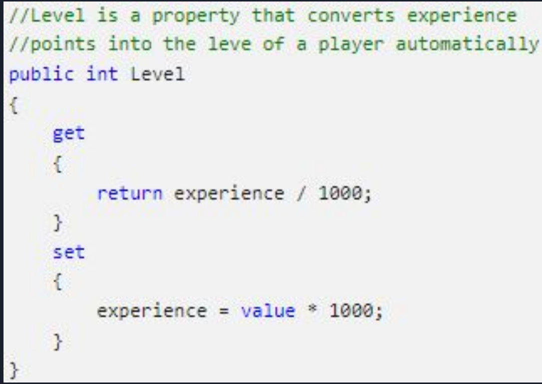
- We can utilize data hiding to use private/protected variables that are exposed in a limited and controlled manner
    - Having established naming conventions can go a long way
- Allow us to force read only/write only scenarios
- Also allows for input validation, updating other fields, triggering events, etc



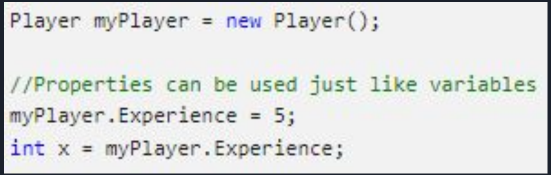
```
//Member variables can be referred to as
//fields.
private int experience;

//Experience is a basic property
public int Experience
{
    get
    {
        //Some other code
        return experience;
    }
    set
    {
        //Some other code
        experience = value;
    }
}
```



```
//This is an example of an auto-implemented
//property
public int Health{ get; set;}
```



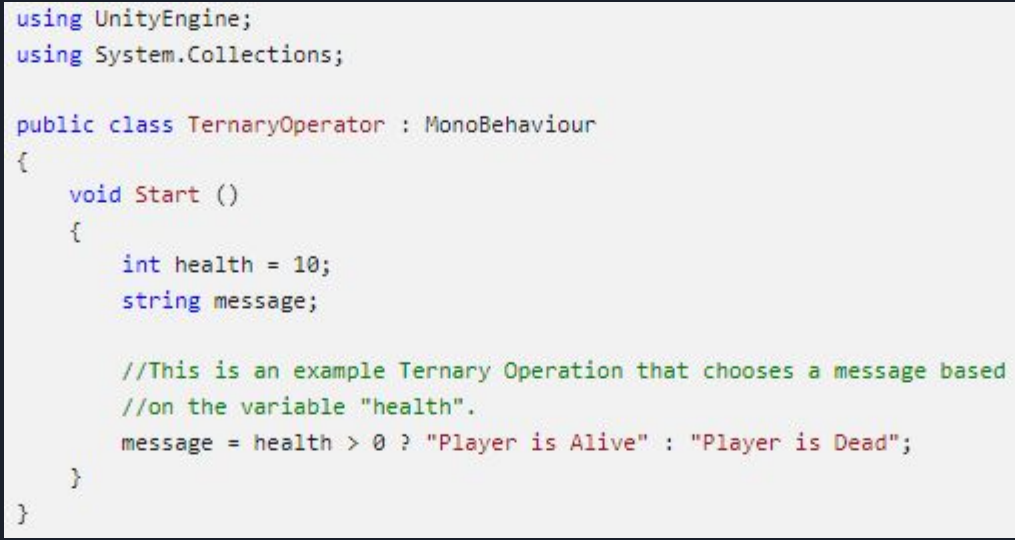
```
//Level is a property that converts experience
//points into the leve of a player automatically
public int Level
{
    get
    {
        return experience / 1000;
    }
    set
    {
        experience = value * 1000;
    }
}
```



```
Player myPlayer = new Player();

//Properties can be used just like variables
myPlayer.Experience = 5;
int x = myPlayer.Experience;
```

Ternary Operator

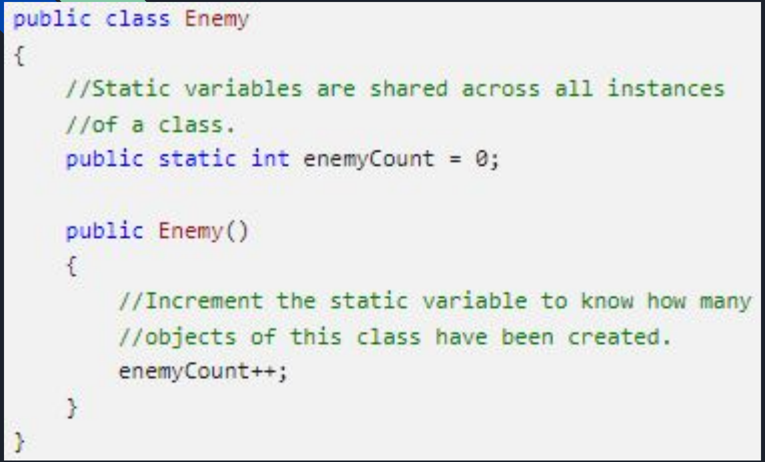
- -A more in-line version of an if-else statement
- Provides a value in all circumstances
- [Condition] ? [True Outcome] : [False Outcome]
- Bonus: If you are trying to access a reference that may be null (i.e. myList), avoid null reference errors for conditional access: myList?.Add(3);



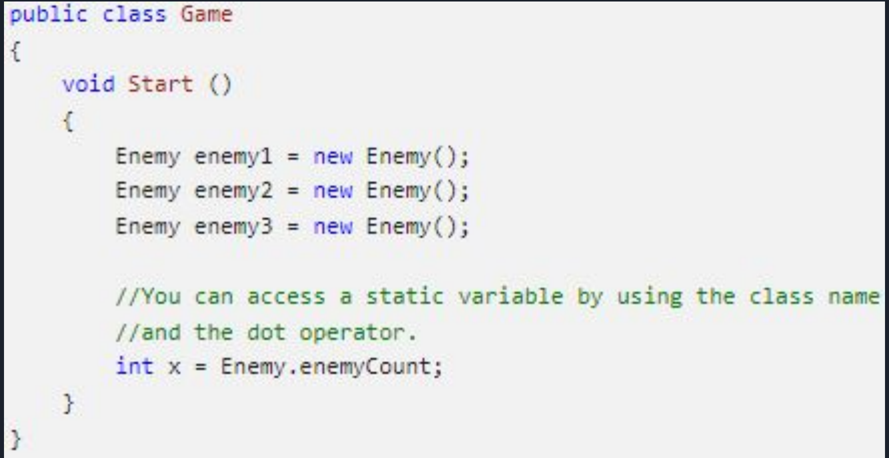
```
using UnityEngine;
using System.Collections;

public class TernaryOperator : MonoBehaviour
{
    void Start ()
    {
        int health = 10;
        string message;

        //This is an example Ternary Operation that chooses a message based
        //on the variable "health".
        message = health > 0 ? "Player is Alive" : "Player is Dead";
    }
}
```

Statics

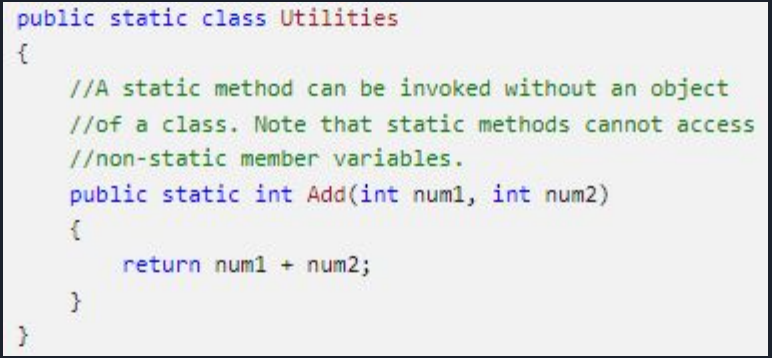
- While regular members of classes (like variables and methods) exist independently for every instance of a class, static members are shared by all instances of the class
- Static members can be accessed without any objects being declared at all!
- You can create a static class too; all fields and methods must be static, and you will not be able to declare an instance of the class



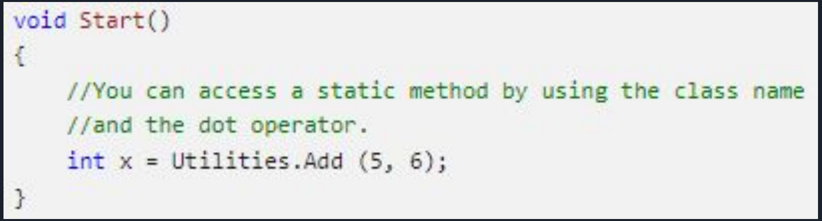
```
public class Enemy
{
    //Static variables are shared across all instances
    //of a class.
    public static int enemyCount = 0;

    public Enemy()
    {
        //Increment the static variable to know how many
        //objects of this class have been created.
        enemyCount++;
    }
}
```



```
public class Game
{
    void Start ()
    {
        Enemy enemy1 = new Enemy();
        Enemy enemy2 = new Enemy();
        Enemy enemy3 = new Enemy();

        //You can access a static variable by using the class name
        //and the dot operator.
        int x = Enemy.enemyCount;
    }
}
```



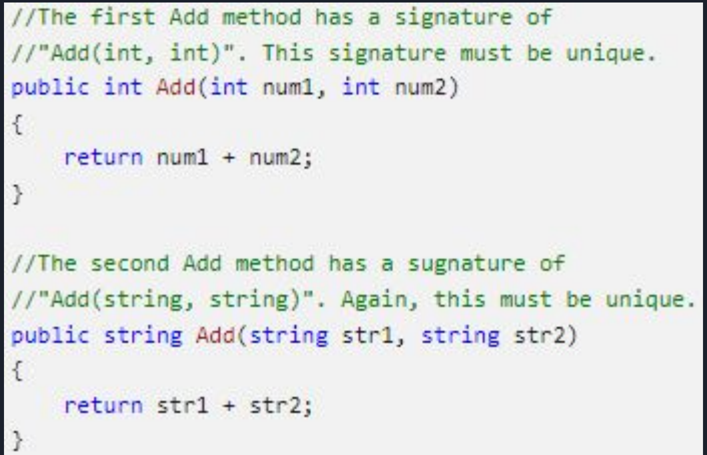
```
public static class Utilities
{
    //A static method can be invoked without an object
    //of a class. Note that static methods cannot access
    //non-static member variables.
    public static int Add(int num1, int num2)
    {
        return num1 + num2;
    }
}
```



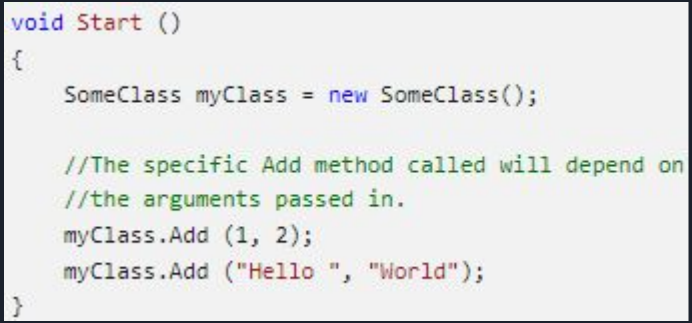
```
void Start()
{
    //You can access a static method by using the class name
    //and the dot operator.
    int x = Utilities.Add (5, 6);
}
```

Method Overloading

- Methods (or Functions) are unique based on their name and argument list
- This is often used to create multiple accepted lists of arguments for the same “method”, which we refer to as method overloading
- Default arguments can not be used to differentiate between overloaded methods
- In practice, many overloaded methods will make subsequent calls to other overloaded methods, to make code more efficient and maintainable (better than separately implementing the same method dozens of times!)



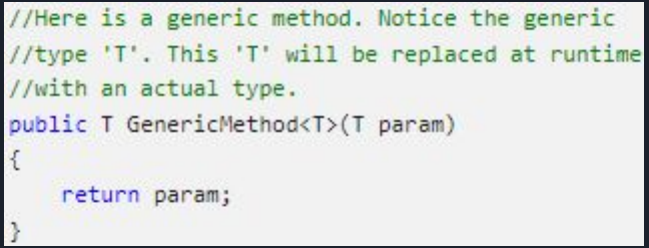
```
//The first Add method has a signature of
//"Add(int, int)". This signature must be unique.
public int Add(int num1, int num2)
{
    return num1 + num2;
}

//The second Add method has a sugnature of
//"Add(string, string)". Again, this must be unique.
public string Add(string str1, string str2)
{
    return str1 + str2;
}
```



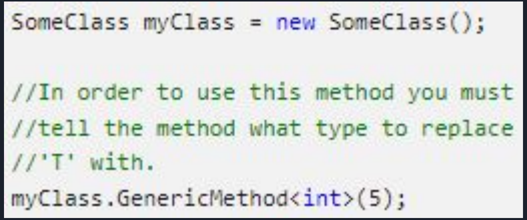
```
void Start ()
{
    SomeClass myClass = new SomeClass();

    //The specific Add method called will depend on
    //the arguments passed in.
    myClass.Add (1, 2);
    myClass.Add ("Hello ", "World");
}
```

Generics (aka Templates)

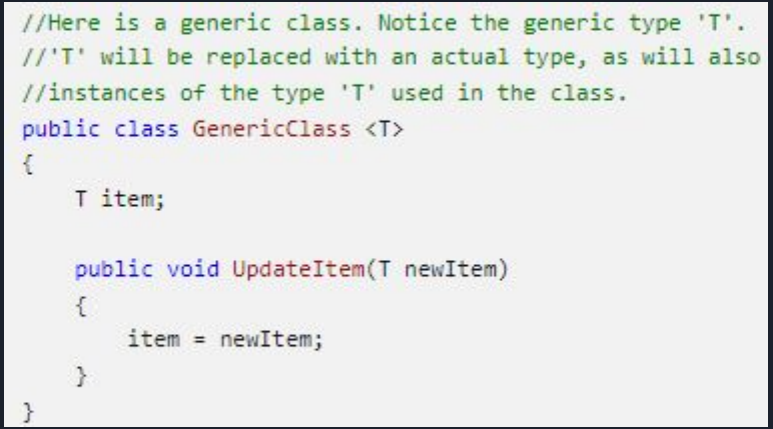
- Functions or Classes that are applied to an unknown type
- Examples include: List, Dictionary, GetComponent, FindObjectOfType



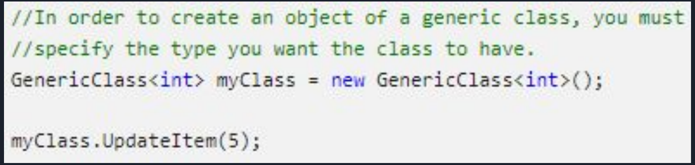
```
//Here is a generic method. Notice the generic
//type 'T'. This 'T' will be replaced at runtime
//with an actual type.
public T GenericMethod<T>(T param)
{
    return param;
}
```



```
SomeClass myClass = new SomeClass();

//In order to use this method you must
//tell the method what type to replace
//'T' with.
myClass.GenericMethod<int>(5);
```



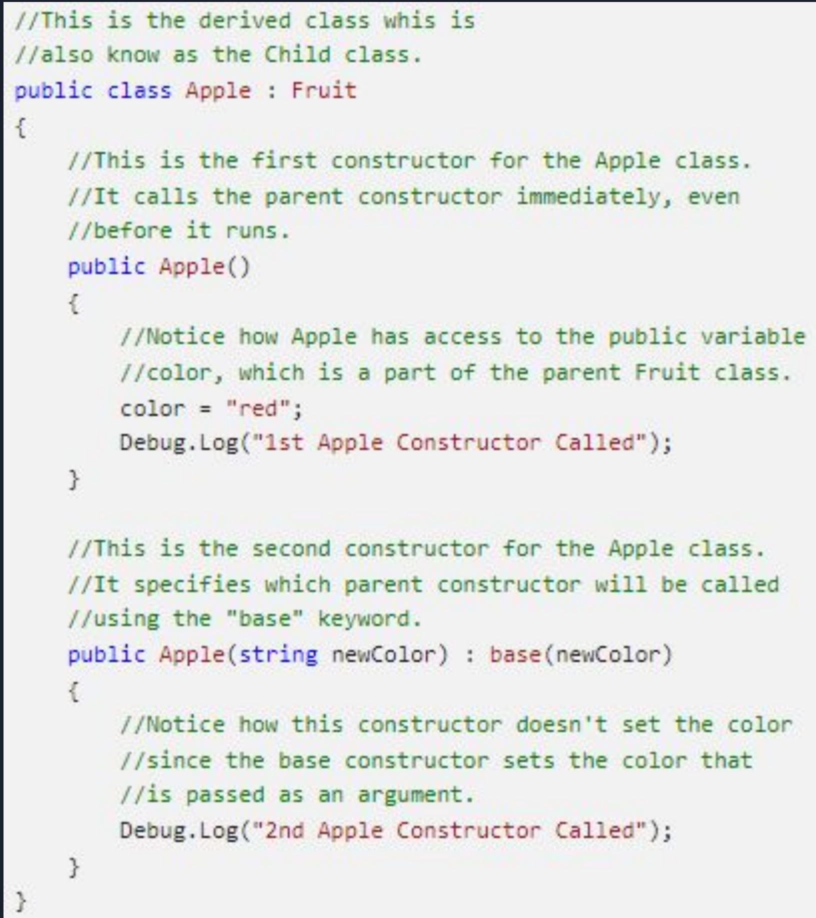
```
//Here is a generic class. Notice the generic type 'T'.
//'T' will be replaced with an actual type, as will also
//instances of the type 'T' used in the class.
public class GenericClass <T>
{
    T item;

    public void UpdateItem(T newItem)
    {
        item = newItem;
    }
}
```



```
//In order to create an object of a generic class, you must
//specify the type you want the class to have.
GenericClass<int> myClass = new GenericClass<int>();

myClass.UpdateItem(5);
```

Inheritance

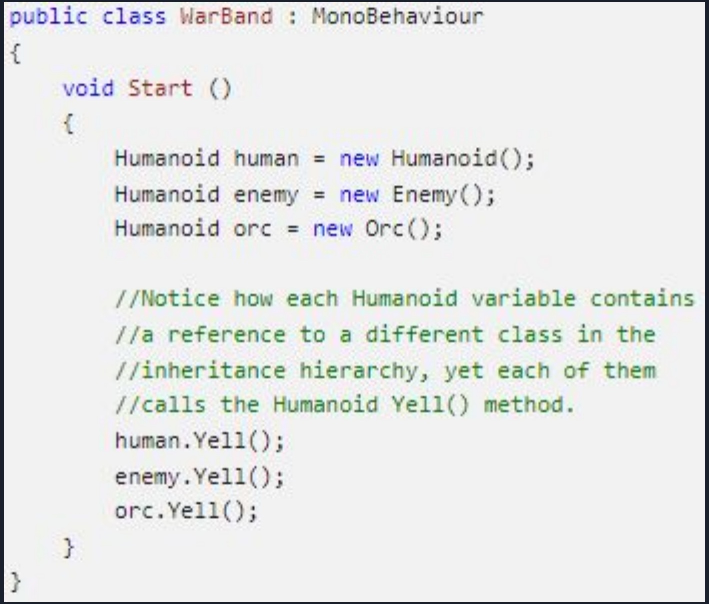
- Inheritance allows us to create a class that is closely based off of another
- Example: All components inherit from the monobehaviour class
- The class that inherits from another is called the derived class, the class being inherited from is called the base class
- Derived classes will maintain access to all public (and protected) members of the base class
    - This is the purpose of introducing the protected visibility option, it behaves just like private variables with the exception of being available to derived classes



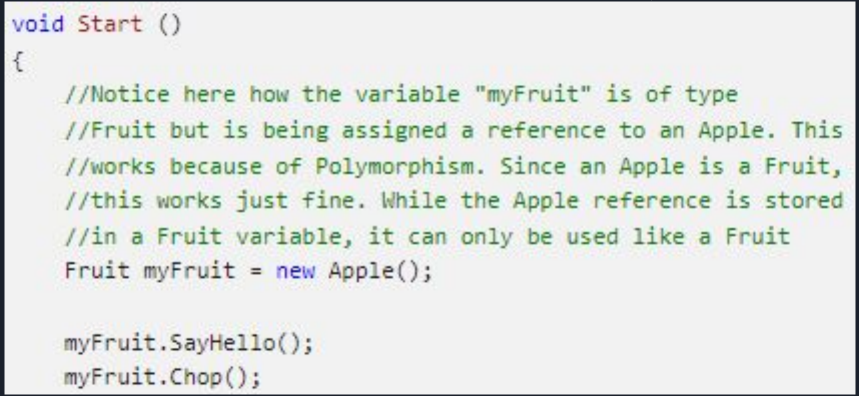
```
//This is the derived class whis is
//also know as the Child class.
public class Apple : Fruit
{
    //This is the first constructor for the Apple class.
    //It calls the parent constructor immediately, even
    //before it runs.
    public Apple()
    {
        //Notice how Apple has access to the public variable
        //color, which is a part of the parent Fruit class.
        color = "red";
        Debug.Log("1st Apple Constructor Called");
    }

    //This is the second constructor for the Apple class.
    //It specifies which parent constructor will be called
    //using the "base" keyword.
    public Apple(string newColor) : base(newColor)
    {
        //Notice how this constructor doesn't set the color
        //since the base constructor sets the color that
        //is passed as an argument.
        Debug.Log("2nd Apple Constructor Called");
    }
}
```

Polymorphism + Member Hiding

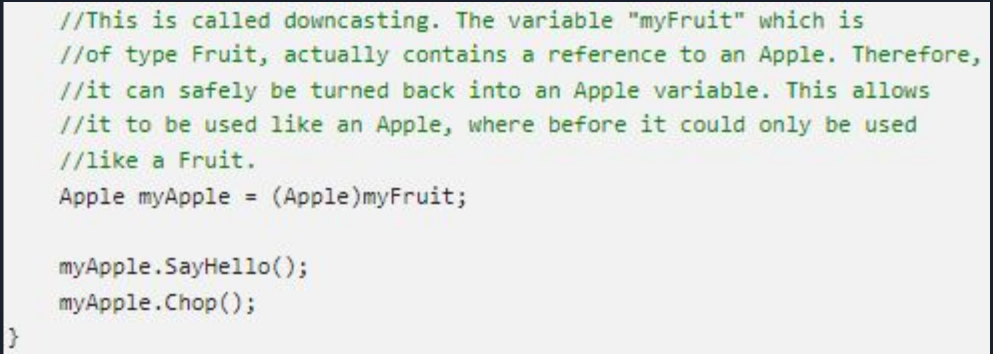
- Derived classes can be cast as their base class
  - The derived class exists fully, but when it is held in a variable of the base type, it will only be capable of base operations
  - A derived class being treated as a base class is called “upcasting”
  - Casting a base class as a derived class is called “downcasting”
- It is possible to provide new definitions for existing functions
  - This does require the new keyword to prevent errors
  - Used when accessed as a derived class, not a base class



```
public class WarBand : MonoBehaviour
{
    void Start ()
    {
        Humanoid human = new Humanoid();
        Humanoid enemy = new Enemy();
        Humanoid orc = new Orc();

        //Notice how each Humanoid variable contains
        //a reference to a different class in the
        //inheritance hierarchy, yet each of them
        //calls the Humanoid Yell() method.
        human.Yell();
        enemy.Yell();
        orc.Yell();
    }
}
```



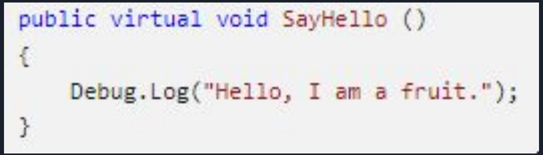
```
void Start ()
{
    //Notice here how the variable "myFruit" is of type
    //Fruit but is being assigned a reference to an Apple. This
    //works because of Polymorphism. Since an Apple is a Fruit,
    //this works just fine. While the Apple reference is stored
    //in a Fruit variable, it can only be used like a Fruit
    Fruit myFruit = new Apple();

    myFruit.SayHello();
    myFruit.Chop();
```



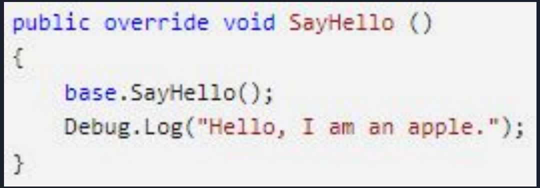
```
    //This is called downcasting. The variable "myFruit" which is
    //of type Fruit, actually contains a reference to an Apple. Therefore,
    //it can safely be turned back into an Apple variable. This allows
    //it to be used like an Apple, where before it could only be used
    //like a Fruit.
    Apple myApple = (Apple)myFruit;

    myApple.SayHello();
    myApple.Chop();
}
```

Overriding

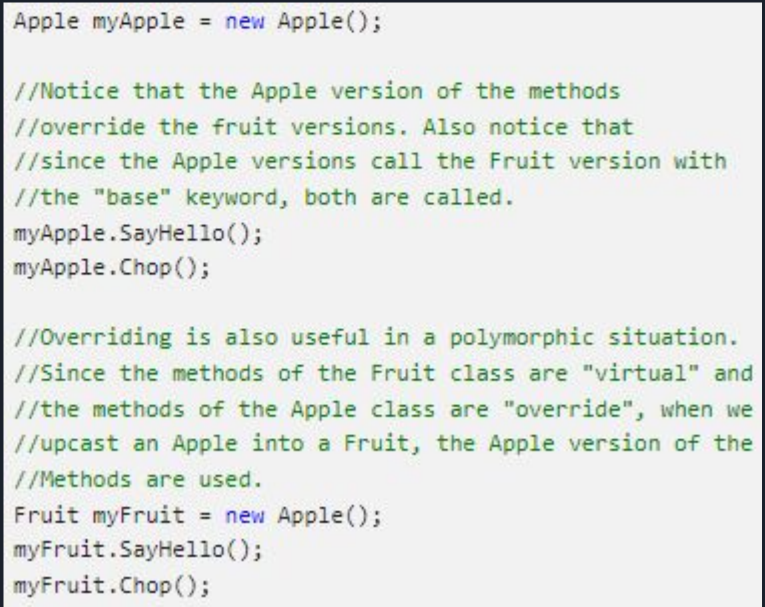
- Sometimes we want to provide a function that will have its code modified in derived classes
- In the base class, include the *virtual* keyword before the return type
- In the derived class, redefine that function, replacing *virtual* with *override*
- Now the function override will be used for any derived classes, even if they are accessed after being upcasted



```
public virtual void SayHello ()
{
    Debug.Log("Hello, I am a fruit.");
}
```



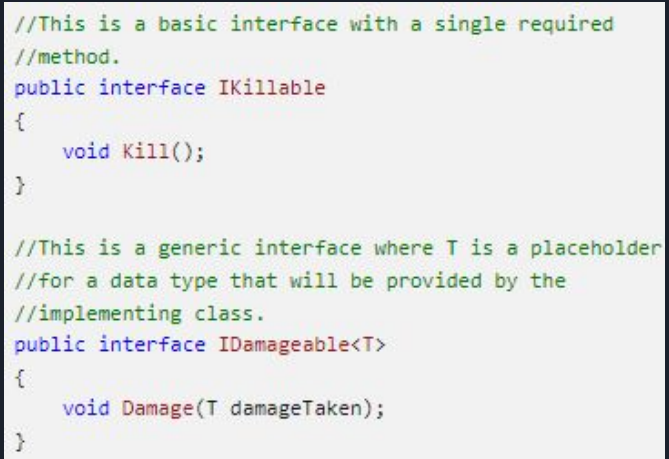
```
public override void SayHello ()
{
    base.SayHello();
    Debug.Log("Hello, I am an apple.");
}
```



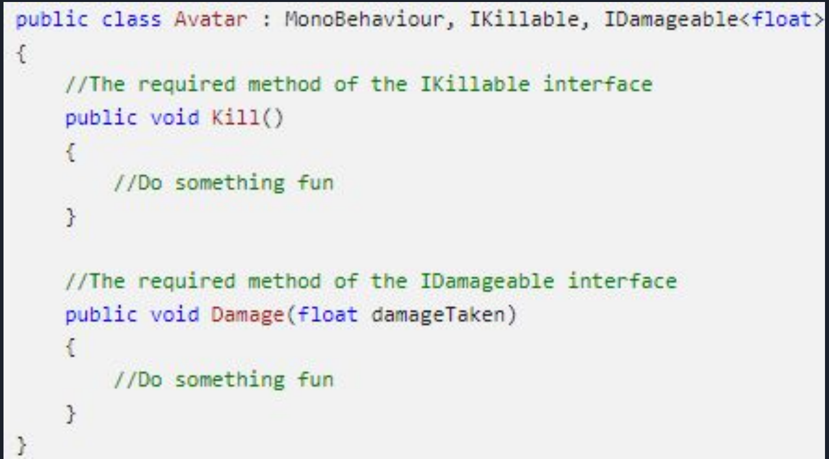
```
Apple myApple = new Apple();

//Notice that the Apple version of the methods
//override the fruit versions. Also notice that
//since the Apple versions call the Fruit version with
//the "base" keyword, both are called.
myApple.SayHello();
myApple.Chop();

//Overriding is also useful in a polymorphic situation.
//Since the methods of the Fruit class are "virtual" and
//the methods of the Apple class are "override", when we
//upcast an Apple into a Fruit, the Apple version of the
//Methods are used.
Fruit myFruit = new Apple();
myFruit.SayHello();
myFruit.Chop();
```

Interfaces

- Interfaces are assurances that certain specified members are implemented in all classes that implement them
- If a class implements an interface, it will get a compiler error if it does not implement all methods and properties
- “Implementing” an interface behaves syntactically similar to inheriting from a base class, but interfaces can not be created, they describe behaviors that exist in other classes
- However, we can treat collections of various objects that implement an interface as interface objects, using a common form of interaction for all of them
  - These various objects may have very different implementations, the only assurance is that functions and properties that are specified exist!



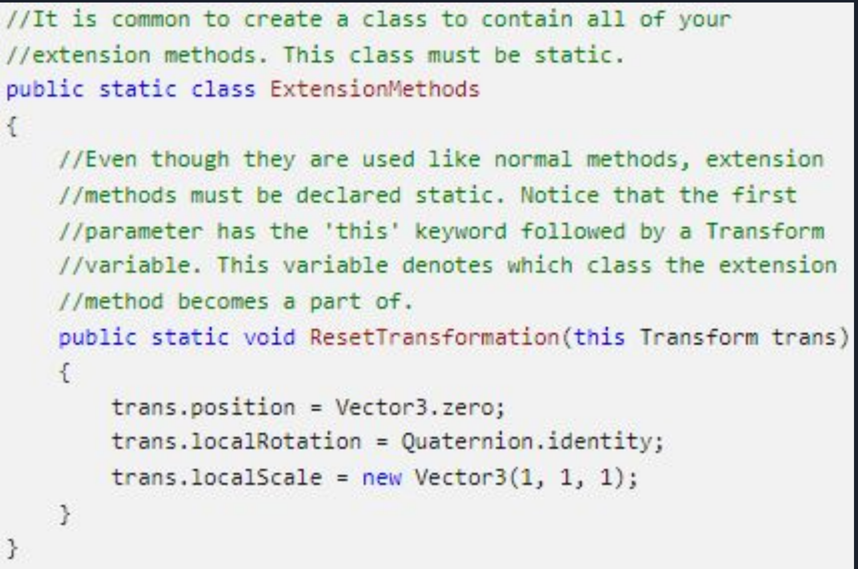
```
//This is a basic interface with a single required
//method.
public interface IKillable
{
    void Kill();
}

//This is a generic interface where T is a placeholder
//for a data type that will be provided by the
//implementing class.
public interface IDamageable<T>
{
    void Damage(T damageTaken);
}
```



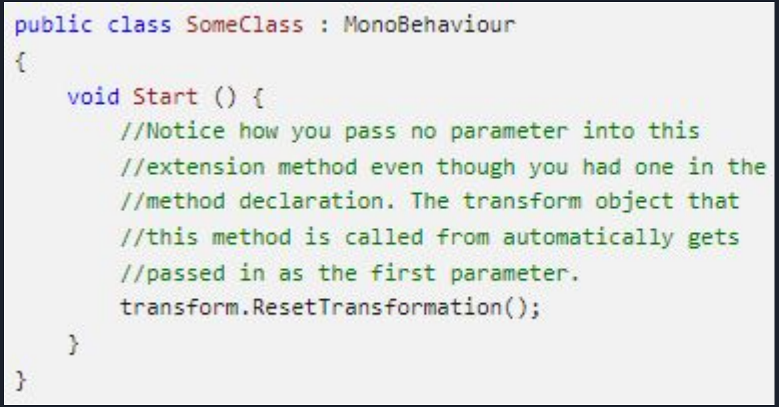
```
public class Avatar : MonoBehaviour, IKillable, IDamageable<float>
{
    //The required method of the IKillable interface
    public void Kill()
    {
        //Do something fun
    }

    //The required method of the IDamageable interface
    public void Damage(float damageTaken)
    {
        //Do something fun
    }
}
```

Extension Methods

- Way to add functionality to a class that we can't go in and modify ourselves (ie Transform)
  - Deriving a new class doesn't make sense if we're just trying to add functionality to all instances of the class
- Must be static functions placed in a static, non-generic class
  - Common to create a class dedicated to extension methods
- First argument is "this", which will automatically be filled by the object calling the method



```
//It is common to create a class to contain all of your
//extension methods. This class must be static.
public static class ExtensionMethods
{
    //Even though they are used like normal methods, extension
    //methods must be declared static. Notice that the first
    //parameter has the 'this' keyword followed by a Transform
    //variable. This variable denotes which class the extension
    //method becomes a part of.
    public static void ResetTransformation(this Transform trans)
    {
        trans.position = Vector3.zero;
        trans.localRotation = Quaternion.identity;
        trans.localScale = new Vector3(1, 1, 1);
    }
}
```



```
public class SomeClass : MonoBehaviour
{
    void Start () {
        //Notice how you pass no parameter into this
        //extension method even though you had one in the
        //method declaration. The transform object that
        //this method is called from automatically gets
        //passed in as the first parameter.
        transform.ResetTransformation();
    }
}
```

Namespaces

- Separate spaces that are able to hold groups of classes while preventing naming conflicts
- Can also be used to import these classes for easy use and access
- If your class is being created within a namespace, you have immediate access to all other classes within that namespace
- For classes that are outside of your namespace, they must be imported with the “using” keyword, or access the class through [Namespace].[Classname]
- Namespaces may be nested, enclose one namespace declaration within another
    - I.e. System.Collections
- Specific information about namespaces/classes available in https://docs.unity3d.com/ScriptReference/
- Consider establishing and using your own namespace to pre-emptively avoid naming conflicts (ie size, object, cur, count). This can be used as a common space for reusable assets

Lists and Dictionaries

- Lists (reference) implement a Vector within C#
  - One dimensional, indexed and ordered array of values
  - Generic, capable of holding any type
  - Adding/Removing to end is more optimal than the center or the start!
  - Has access to a host of Quality of Life features, including Sort()
    - Sort requires the type being held to implement the IComparable Interface
- Dictionaries (reference) implement a hash table with two generic types, one for the key/"index", and one for the value
  - Suitable for creating a relationship table for any two arbitrary classes (ie "Pikachu" and the corresponding PokemonData object)
  - Not serializable! If you wish to save the contents of a library, look into alternative serializing implementations, or find another way to save the data (ie in two lists)

Coroutines

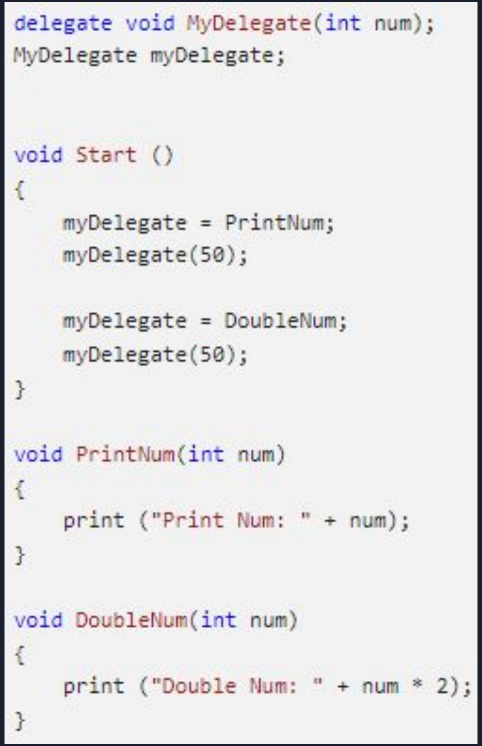
- Coroutines are a good way to spread out a task over multiple frames
- Hold return type of IEnumerator
- Called with the StartCoroutine function call
- When calling yield return null; execution will cease until the next frame
- Can also yield return new WaitForSeconds(<float>); to delay execution for <float> seconds
- This creates an alternative to using the Update loop
- Check previous and linked examples!

Quaternions

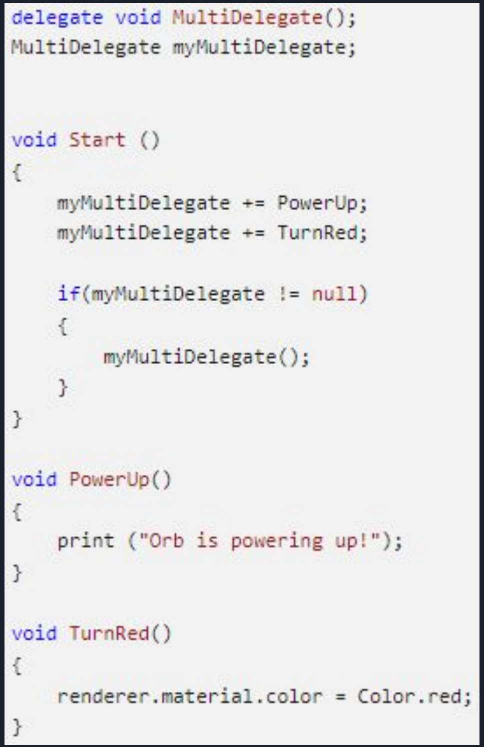
- When we look at a transform within the inspector, we see its angle represented as a *Euler Angle*, which is intuitive but subject to some technical limitations
    - EulerAngles function, rotatetowards
- Unity actually stores the rotations of transforms as Quaternions, which are less intuitive and have more variables
    - But, importantly, they don’t suffer from the technical limitations of euler angles!
- Manually altering quaternions is a bad idea (except for math majors)
- Many functions exist to make logically dealing with quaternions possible
    - Quaternion.LookRotation(Vector3 v3) will give us the Quaternion that will look in the specified direction
        - Accepts another vector3 indicating “up” direction, if desired (for rotated perspectives)
    - Quaternion.Slerp(Quaternion current, Quaternion target, float t) can be used to smoothly transition from one angle to another
        - When t is close to 0, the function returns an angle close to current
        - When t is close to 1, the function returns an angle close to target
- Alternatives often exist, like transform.Rotate

Delegates

- Delegates allow for classes to hold references to functions as variables, which can allow for extreme amounts of customization and modularity
- Just like a function, delegates have a return type, a name, and a parameter list
- Can be changed at runtime - changing the current delegate, adding more delegates, and removing one (or all) delegates
    - To remove all delegates assigned, set the delegate to null



```
delegate void MyDelegate(int num);
MyDelegate myDelegate;


void Start ()
{
    myDelegate = PrintNum;
    myDelegate(50);

    myDelegate = DoubleNum;
    myDelegate(50);
}

void PrintNum(int num)
{
    print ("Print Num: " + num);
}

void DoubleNum(int num)
{
    print ("Double Num: " + num * 2);
}
```



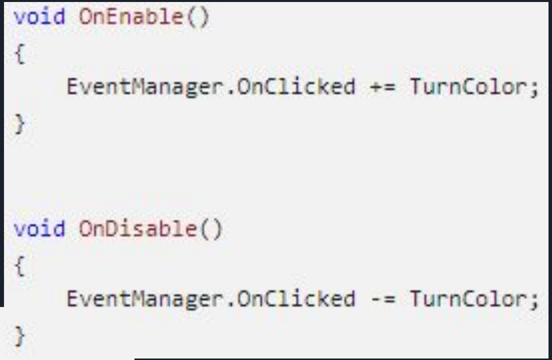
```
delegate void MultiDelegate();
MultiDelegate myMultiDelegate;


void Start ()
{
    myMultiDelegate += PowerUp;
    myMultiDelegate += TurnRed;

    if(myMultiDelegate != null)
    {
        myMultiDelegate();
    }
}

void PowerUp()
{
    print ("Orb is powering up!");
}

void TurnRed()
{
    renderer.material.color = Color.red;
}
```

Attributes

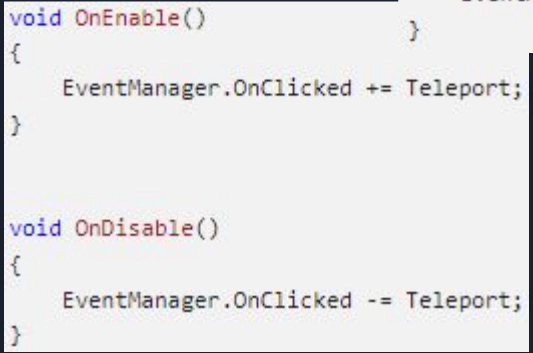
- Attributes are a way to attach information to a class, function, or variable as you declare it
- They vary wildly in how they are used, both within C# at large and Unity specifically
- Examples include:
  - Range(minimum, maximum) - Constricts numeric values
  - ExecuteInEditMode - Object's code will run even while the project is open in Edit mode
    - Changes made in edit mode will be permanent, not temporary like changes in play mode!
  - RPC(SendTo.Server) (along with alternative targets) - Used by a network connected user or server to signal the execution of code on (an) other target(s)

Events

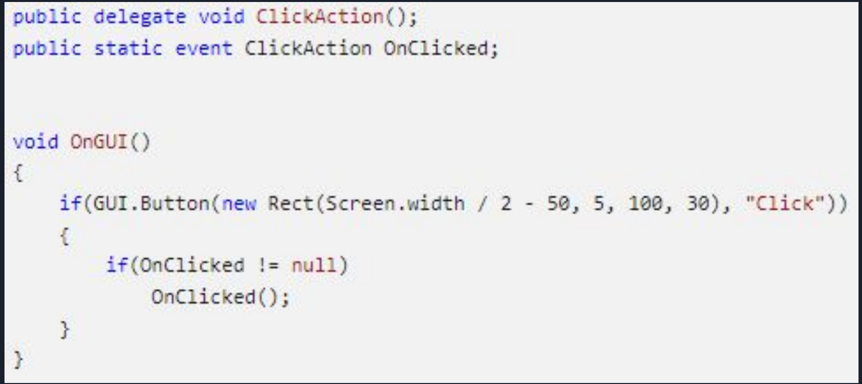
- Specialized delegates that allow for other classes to subscribe to alerts
  - In practice, work essentially like public multicast delegate
  - Don't forget to unsubscribe if an object will be destroyed, or suffer a null reference exception!
  - More secure implementation than making a delegate public, as any class could interfere with the delegate for other classes!
  - UnityEvent also serializes , preventing the need for you to subscribe and unsubscribe to the event in your code, while otherwise behaving similarly



```
void OnEnable()
{
    EventManager.OnClicked += TurnColor;
}


void OnDisable()
{
    EventManager.OnClicked -= TurnColor;
}
```



```
void OnEnable()
{
    EventManager.OnClicked += Teleport;
}


void OnDisable()
{
    EventManager.OnClicked -= Teleport;
}
```



```
public delegate void ClickAction();
public static event ClickAction OnClicked;


void OnGUI()
{
    if(GUI.Button(new Rect(Screen.width / 2 - 50, 5, 100, 30), "Click"))
    {
        if(OnClicked != null)
            OnClicked();
    }
}
```

Readability and Usability

- Use of #region <name> and #endregion make navigation a breeze, and helps you organize
- Use consistent naming conventions to reduce headaches later
  - This involves consistent casing as well as consistent perspective/tone in names
- If you or somebody else will be working with this component a lot in the editor, have you done what is possible to make using it a pleasure that enables the most productive interactions?
  - Labels, attributes like range
- /// for function and parameter explanations, if writing a piece of code that coders will be interacting with
- Refactor feature in VS Code