UdonSharp 1.x Feature Overview

How to take advantage of some of the latest features introduced in UdonSharp 1.x

Abstract Classes

Just like C#, which UdonSharp is based on, now supports abstract classes!

FOLLOWING PAGE CONTAINS MATERIAL THAT IS WORK IN PROGRESS!

What are abstract classes?

The correct definition of "abstract" is "*missing or incomplete implementation*".

In simple terms, abstract is useful for creating a base class that defines what properties and/or methods a derived class should have.

Example 1 - Basics

Here is a basic example of abstract and derived classes

```
public abstract class BaseClass : UdonSharpBehaviour
{
    // Abstract classes can include variables, properties and methods
    // Using the abstract keyword on any of the previously mentioned will require the implementation in the derived class
    public abstract bool RequiredProperty { get; set; }

    public abstract void RequiredAbstractMethod(); // Abstract methods can't have body, their behaviour must be implemented in the derived class

    // The "virtual" keyword allows us to define the "default" behaviour of a method, which can be overriden in the derived class
    public virtual void VirtualMethod()
    {
        RequiredAbstractMethod();
        RequiredProperty = true;
    }
}
```

```
public class DerivedClass : BaseClass
{
    // Properties can be implemented as we wish, getters and setters can point to to anything
    // In this example the property just points to a private boolean in the derived class
    public override bool RequiredProperty { get => privateBoolean; set => privateBoolean = value; }

    private bool privateBoolean;

    // Abstract methods must be implemented in derived classes
    public override void RequiredAbstractMethod()
    {
        // Here we add any code we want to execute that we can call with either
    }
}
```

Example 2 - Vehicle

In this example we want to create an abstract Vehicle that has shared characteristics across all transportation methods via land, air and sea.

```
[RequireComponent(typeof(Rigidbody))] // RequireComponent attributes will be inherited by derived class
[RequireComponent(typeof(Animator))]
[RequireComponent(typeof(VRCObjectSync))]
[UdonBehaviourSyncMode(BehaviourSyncMode.Continuous)] // Sync mode will be inherited by derived classes, so you may define it in any class you want
public abstract class Vehicle : UdonSharpBehaviour
{
    // Abstract classes can have private variables that won't be accessible from derived classes
    private Rigidbody rb;
    private VRCObjectSync objectSync;
    private Animator animator;

    // In this example, you obviously wouldn't want any script tampering with the private variables of the abstract class, so we will add our own properties with only getter exposed
```

```
    public Rigidbody RB { get => rb; }
    public VRCObjectSync ObjectSync { get => objectSync; }
    public Animator Animator { get => animator; }

    // By adding the "protected" access modifier, we can make private variables,
properties or methods accessible to the derived class
    private protected void InitializeAbstractVehicle()
    {
    // Any operations shared by all derived classes can be included in the abstract class
    rb = GetComponent<Rigidbody>();
        objectSync = GetComponent<VRCObjectSync>();
        animator = GetComponent<Animator>();
    }

    // By adding a "virtual" keyword we can define the "default" behaviour of a method,
which can overriden in derived classes
    // In this example we want to be able to call Vehicle._RespawnVehicle() or
LandVehicle._RespawnVehicle() and always call the respawn method on the object sync
    public virtual void _RespawnVehicle()
    {
    objectSync.Respawn();
    }
}

public class LandVehicle : Vehicle
{

}

public class AirVehicle : Vehicle
{

}

public class SeaVehicle : Vehicle
{

}
```

Check out the official documentation for using abstract here:Abstract (C# Reference)