Session 03 – Inheritance and Polymorphism

Learning Outcomes

- Custom classes, Constructors & Deconstructors
- Class Inheritance
- Polymorphism
- Virtual methods & Overriding

Session Outline

- Our classes don't have to inherit from MonoBehaviour or AActor

```
public class CharStats
{
    public float strength;
    public float dexterity;
    public float energy;
    public float mana;
}
```

- Classes are instantiated via the '**new**' keyword

```
CharStats charStats = new CharStats()
```

- If we haven't specified a **constructor** then a 'default constructor' is used, i.e. no code is executed and all class member variables are initialised to their default value.

- However if we do want to run some code or set some initial values we can create a constructor

```
public class CharStats
{
    // snip ...

    public CharStats()      // note there is no return type (not even 'void')
    {                       // and the method name must match the class name
        strength = 100f;
        dexterity = 100f;
        energy = 100f;
        mana = strength / 4f + energy / 2f;
    }
}
```

- We can define multiple constructors in the same class, so long as they have a distinct method signature (i.e. number or type of parameters)

```
public class CharStats
{
	// snip ...

	public CharStats(float s, float d, float e, float m = -1)
	{
		strength = s;
		dexterity = d;
		energy = e;
		if (m == -1)
		{
			mana = strength / 4f + energy / 2f;
		} else {
			mana = m;
		}
	}
}
```

- This is called method '**overloading**' and it can be used with any method, not just constructors. Be sure that you understand the difference between 'overloaded' methods and 'overridden' methods (which we'll see later in this session).

- We can also specify a class **destructor**. This is called when the object is deleted by the Garbage collector or by the **delete** keyword . A destructor is signified by a tilde (~) in front of a method whose name matches the class name.

```
public class CharStats
{
	// snip ...

	~CharStats()
	{
		// execute any code required to 'clean up' the class,
		// e.g. free any large memory allocations
	}
}
```

- Class Inheritance ('is-a' relationship). Allows us to inherit all of the functionality of member variables and methods from a parent class without having to re-declare these in the child.

  - Note that members which are declared as 'private' in a parent class are <u>not</u> accessible from the child class. If you want to prevent your data from being public to any other class, but you do want to make it accessible to any child classes then you use the '**protected**' keyword.

  - Inheritance example. First we define a parent class (in this case, 'Target')

```
using UnityEngine;
```

```
using System.Collections;

public class Target : MonoBehaviour
{
	public int value;

	public void Hit()
	{
		Debug.Log("Target " + name + " has been hit!");
	}
}
```

- Then a child class when inherits from the parent (in this case, 'MovingTarget')

```
using UnityEngine;
using System.Collections;

public class MovingTarget : Target
{
	public void Move()
	{
		transform.Translate(Vector3.Up);
	}
}
```

- we don't need to re-implement **value** or **Hit()** in MovingTarget because they are inherited from Target

- Polymorphism – using a parent class's data type to store a reference to a child instance.
  - Now in some other script we can make a collection of Targets:

```
	public List<Target> targets = new List<Target>();
```

and then add both Target and MovingTarget objects to the collection:

```
	GameObject targetGO = GameObject.FindWithTag("Target");
	Target t = targetGO.GetComponent<Target>();

	GameObject movingTargetGO =
GameObject.FindWithTag("MovingTarget");
	MovingTarget mt = movingTargetGO.GetComponent<MovingTarget>();

	targets.Add(t);
	targets.Add(mt);	// both types can be added to the same collection
```

  - We can now call Hit() on each element of the collection:

```
	targets[0].Hit();
	targets[1].Hit();
```

However, we now cannot call targets[1].Move(); because the collection is of type **Target** and that class does not define a **Move()** method.

- Virtual Methods & Overrides – accessing child class functionality from when the referencing variable uses a parent data type.
    - Target defines a **virtual** method

```
public class Target  : MonoBehaviour
{
        // snip ...

        public virtual void GetType()
        {
                Debug.Log("Target");
        }
}
```

    - MovingTarget **overrides** the virtual method

```
public class MovingTarget : Target
{
        // snip ...

        public override void GetType()
        {
                Debug.Log("Moving target");
        }
}
```

    - Elsewhere in our code we can call GetType() on our collection of Target objects

```
foreach(Target x in targets)
{
        x.GetType();
{
```

      We find that the instances of MovingTarget run their overridden GetType() method.

    - Interfaces
        - Interfaces are a contract on functionality.
        - An interface defines what a class must do, but not how it will achieve it. This means an interface might contain one or more method signatures (return type, name, & parameters) but not the functionality. Another class implementing this interface will have to define the functionality for all such methods. These definitions might vary from one class implementing

this interface to the next.

- By convention, interfaces are always prefixed with a capitol 'I' and often end in the suffix '-able' because it defines what the implementing class is capable of. For example: IControllable
- Unlike inheritance, a class can implement multiple interfaces (C++ can have multiple inheritance but is frowned upon)

https://unity3d.com/learn/tutorials/modules/intermediate/scripting/interfaces

https://wiki.unrealengine.com/Interfaces_in_C%2B%2B