2: Introduction to Object Orientated Programming

GAM140: Individual Creative Computing Project

Reminder – Week 3

- **Proposal Review**
  - **Describe** the game design that will form the basis for your interface
  - **Illustrate** basic research into electronic component and physical form factors for controllers
  - **Analyse** the design of the controller in detail;
  - **List** the key electronic components of your controller
  - and **List** the key user stories.

Learning outcomes

- **Understand** core principles of OOP
- **Understand** the basic constructs in OOP
- **Develop** some basic classes

Introduction

- Object Orientated Programming (OOP) is the dominant paradigm in programming
- Most modern Programming Languages are OOP
  – C++ - Mixed Paradigm, but supports OOP
  – C#-OOP
  – Java – OOP
  – Python - Mixed Paradigm, but supports OOP
  – Javascript - Mixed Paradigm, but supports OOP
- It has became dominant because of how we as humans describe the world
  – E.g. Designing games we talk about ’enemies’ or ’weapons’ with their properties and abilities these have

OOP PRINCIPLES

Introduction

- OOP has certain core features which make it ideal for complex reusable software such as games
  - **Encapsulation** – Data hiding
  - **Composition –** Objects can be composed of other objects
  - **Inheritance –** Objects can be related to other objects
  - **Polymorphism –** Objects can be used via references to their parent class (more about this later!)

SOLID

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Research SOLID

- **Take 10 minutes to research these areas**
  - **S**ingle Responsibility Principle
  - **O**pen Closed Principle

OOP CONCEPTS IN C#

Classes

- In C# a **Class** allows us to create a custom data type
- These are a collection of **variable** and **functions**
- The function usually operates on the variable some manner

Classes

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Player : MonoBehaviour
{
    [SerializeField]
    private int Health;

    void TakeDamage(int damage)
    {
        Health -= damage;
        if (Health<0)
        {
            Death();
        }
    }

    void Death()
    {
        Destroy(gameObject);
    }
}
```

FALMOUTH UNIVERSITY

Classes – Using Statements

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Player : MonoBehaviour
{
   [SerializeField]
   private int Health;

   public void TakeDamage(int damage)
   {
      Health -= damage;
      if (Health<0)
      {
         Death();
      }
   }

   void Death()
   {
      Destroy(gameObject);
   }
}
```

Using Statements allow you to Import and use libraries, these can be built in such as ‘System.*’

Or can be from a Third Party such as Unity (see Unity Engine).

FALMOUTH UNIVERSITY

Classes – Names and Inheritance

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Player : MonoBehaviour
{
   [SerializeField]
   private int Health;

   public void TakeDamage(int damage)
   {
      Health -= damage;
      if (Health<0)
      {
         Death();
      }
   }

   void Death()
   {
      Destroy(gameObject);
   }
}
```

All classes start with the **class** keyword, anything after the **:** is what the class inherits from, and essence became a parent class.

C# has a simple *single inheritance,* this means any class can only inherit from one class.

In this case the **Player** class is a child class of MonoBehaviour, which means that it gains all the functions, and variables of the class

Classes – Access Modifiers

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Player : MonoBehaviour
{
   [SerializeField]
   private int Health;

   public void TakeDamage(int damage)
   {
      Health -= damage;
      if (Health<0)
      {
         Death();
      }
   }

   void Death()
   {
      Destroy(gameObject);
   }
}
```

Access modifies, these specify the access level of class, variable or function.

private: This is the default access for classes only the owning class has access.

protected: Same as private, except child child classes can access.

public: No restrictions on access. In Unity a variable will appear in the Inspector

Classes - Attributes

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Player : MonoBehaviour
{
    [SerializeField]
    private int Health;

    public void TakeDamage(int damage)
    {
        Health -= damage;
        if (Health<0)
        {
            Death();
        }
    }

    void Death()
    {
        Destroy(gameObject);
    }
}
```

Attributes are used to mark Classes, Functions and Variables. This provides some meta-data of how the variable is treated by Unity or another API

SerializeField means that this variable will appear in the Inspector regardless of access modifier.

https://docs.unity3d.com/Manual/Attributes.html

Classes - Variables

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Player : MonoBehaviour
{
    [SerializeField]
    private int Health;

    public void TakeDamage(int damage)
    {
        Health -= damage;
        if (Health<0)
        {
            Death();
        }
    }

    void Death()
    {
        Destroy(gameObject);
    }
}
```

Variables are the data that is owned by the class. This is usually used to represent things like health, states, score, etc

Variables can be of the following data types

https://www.w3schools.com/cs/cs_data_types.asp

Including your own data types created with classes, or Unity's types

https://learn.unity.com/tutorial/data-types

Classes - Variables

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Player : MonoBehaviour
{
    [SerializeField]
    private int Health;

    public void TakeDamage(int damage)
    {
        Health -= damage;
        if (Health<0)
        {
            Death();
        }
    }

    void Death()
    {
        Destroy(gameObject);
    }
}
```

Functions are bundles of reusable code which can be called depending on access modifier. Functions can receive multiple variables and return one variable. If you are not returning a value the return type is **void.**

Usually functions operate on the variables in the class or perform some sort of action on the class. You should always consider access any variables through a function, rather than through a public variable.

FALMOUTH UNIVERSITY

Object Instances

```
Player player = new Player();
```

Objects are instances of classes, in traditional C# you initialise classes with the **new** keyword.

In Unity, instances are usually initialised by the engine through the creation of GameObjects.

You can still use **new** for any objects that don't inherit from **Monobehaviour**.

Classes - Inheritance

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Enemy : MonoBehaviour
{
   [SerializeField]
   protected int AttackDamage;

   public void Attack(Player p)
   {
     p.TakeDamage(AttackDamage);
   }

   public virtual void SpecialAttack(Player p)
   {
   }
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Boss : Enemy
{
   [SerializeField]
   protected int SpecialDamage;

   public override void SpecialAttack(Player p)
   {
          p.TakeDamage(AttackDamage*
                                 SpecialDamage);
   }
}
```

FALMOUTH UNIVERSITY

Classes - Inheritance

The Boss class conforms to the **Open Closed Principle** by extending the functionality of the enemy class by implementing the SpecialAttack function and adding a new variable which attacks like a multiplier.

This means the Boss is-a type of enemy, we have a built a relationship here. The enemy class is **Parent** class of the **Boss** class.

Does the Boss also inherit from the Monobehavior class?

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Boss : Enemy
{
   [SerializeField]
   protected int SpecialDamage;

   public override void SpecialAttack(Player p)
   {
          p.TakeDamage(AttackDamage*
                               SpecialDamage);
   }
}
```

FALMOUTH UNIVERSITY

Classes - Inheritance

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Enemy : MonoBehaviour
{
   [SerializeField]
   protected int AttackDamage;

   public void Attack(Player p)
   {
     p.TakeDamage(AttackDamage);
   }

   public virtual void SpecialAttack(Player p)
   {
   }
}
```

The **virtual** keyword indicates that you intend for the function to be overridden by a child class.

You can not use this keyword on private functions

Classes - Inheritance

The **override** keyword means that **Boss** class overrides the functionality provided by the parent class.

You can call the parent class version of this function in the child class using the following

base.SpecialAttack(p);

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Boss : Enemy
{
   [SerializeField]
   protected int SpecialDamage;

   public override void SpecialAttack(Player p)
   {
           p.TakeDamage(AttackDamage*
                                   SpecialDamage);
   }
}
```

Interfaces

```
interface ICelebrator
{
    void StartCelebration();
    void StopCelebration();
}
```

An **Interface** defines a contract that must be implemented by a class.

The interface doesn't provide any Functionality, just a signature that **must** be **implement** by a **class** which implements the interface.

Interfaces

```
interface ICelebrator
{
     void StartCelebration();
     void StopCelebration();
}
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Boss : Enemy,  ICelebrator
{
     //This is the boss example above, with some
    //code omitted for brevity
    public Animator animator;

    public void StartCelebration()
    {
        animator.SetTrigger(”DoDance”);
    }

    public void StopCelebration()
    {
       animator.SetTrigger(”Idle”);
    }
}
```

Polymorphism

https://www.w3schools.com/cs/cs_polymorphism.asp

EXAMPLES

Conclusion

- Today we learned about the basics features of OOP
- In addition we learned about some principles of OOP
- Finally we saw some examples of OOP in C#