LU2 - C# & Unity Scripting

▼ Overview

- Revision of C# Coding
    - Mathematical statements;
    - Loops;
    - Decisions;
    - Arrays;
    - Methods;
    - Classes;
    - Inheritance;
    - Events;
    - Exceptions;
    - Files;
    - Unity and GitHub
- Implement Fundamental C# Principles in Game Development
    - Instantiate Scripts
    - Use Enumerations in Unity (enums)
    - Create Arrays
    - Use Update and Start Methods

▼ Theme 1 - C# Revision

▼ C# Basics

```
public static void Main(string[] args)
{
  // Variables
  string aWord = "A Word or Full Sentences";
  char aCharacter = 'a';
  char aNumberCharacter = '1';
  char aSpecialCharacter = '@';
  int aPositiveNumber = 1;
  int aNegativeNumber = -1;
  float aFloat = 0.123f; // single precision floating point data type (32bit)
  double aDouble = 0.123; // double precision floating point data type (64bit)
  bool aFlag = false;

  // Console Output
  Console.Write(); // No New Line after Writing to Console
  Console.WriteLine(); // Adds a New Line after Writing to the Console

  // Console Input
  // Reads all the Characters entered by the user
  // from the Standard Input Stream
  string input = Console.ReadLine();

  // Reads the Next Character from the Standard Input Stream
  Console.Read();

  // Gets the next character or function key pressed by the user
  ConsoleKeyInfo consoleKeyInfo = Console.ReadKey();
  Console.WriteLine(consoleKeyInfo.Key);

  // Mathematical Statements
  int a = 1;
  int b = 2;
  int c = 4;
  int d = 5;

  int sum = a + b;

  float division = a / b;

  int multiplication = a * b;

  double sqrt = Math.Sqrt(c);

  int absoluteValue = Math.Abs(-a);

  int someEquation = (a + b) * (c + d);

  // Loops

  // For Loop
  for(int i = 1; i <= 10; i++)
  {
    Console.WriteLine($"{i}");
  }

  // While Loop
  int i = 1;
  while(i <= 10)
  {
    Console.WriteLine($"{i}");
    i++;
  }
```

```
    // Do While Loop
    int j = 1;
    do
    {
      Console.WriteLine($"{j}");
      j++;
    }
    while(j <= 10);


    // If-Else Statements

    if(a < b)
    {
      Console.WriteLine("a is less than b");
    }
    else if(a > b)
    {
      Console.WriteLine("a is greater than b");
      a -= b;
      if(a < 0)
        Console.WriteLine("a is less than 0");
    }
    else
    {
      Console.WriteLine("a and b are maybe equal");
    }

    // Arrays and Lists

    int[] numbers = new int[10];
    string[] words = new string[]
    {
      "Hello, ", "World! ", "Awe, ", "Bru."
    };

    List<char> characters = new List<char>()
    {
      'a', 'e', 'i', 'o', 'u'
    };

    int[,] matrix = new int[,]
    {
        { 1, 2 },
        { 4, 5 }
    };

    List<List<int>> something_i_guess = new List<List<int>>()
    {
      new List<int>(),
      new List<int>()
    };
  }
```

- Methods

```
public void CountTillTen()
{
  for(int i = 1; i <= 10; i++)
  {
    Print($"{i}");
  }
}

public void Print(string message)
{
  Console.WriteLine(message);
}

public int Sum(int a, int b)
{
  return a + b;
}
```

▼ C# OOP

- Classes

```
public class Person
{
  #region VARIABLES | FIELDS

  public string Name;
  public string Surname;
  public int Age;

  #endregion

  #region CONSTRUCTORS

  public Person(){}

  public Person(string name, string surname, int age)
  {
    Name = name;
    Surname = surname;
    Age = age;
  }

  ~Person()
  {
    // This is a Deconstructor.
    // Here you can handle any logic for when this object gets destroyed
    // You cannot call the Deconstructor its handled by the Garbage Collector
  }

  #endregion

  #region METHODS
  // Validate Age
  // Validate Name and Surname
  // ToString

  #endregion
}

public static void Main(string[] args)
{
  Person unknown= new Person();
  myself.Name = "User";
  myself.Surname = "Unknown";
  myself.Age = -1;

  Person johnWick = new Person("John", "Wick", 21)

  Person randomPerson = new Person();
}
```

- Inheritance

Base Car Class

```
public class Car
{
    #region VARIABLES

    public string Brand;
    public string Model;
    public int Year;

    #endregion

    #region CONSTRUCTORS

    public Car()
    {
    }

    public Car(string brand, string model, int year)
    {
        Brand = brand;
        Model = model;
        Year = year;
    }

    #endregion

    #region METHODS

    public override string ToString()
    {
        return $"{Brand} : {Model} : {Year}";
    }

    #endregion
}
```

Derived from the base Car Class

```
public class BMW : Car
{
    public BMW()
    {
        Brand = "BMW";
    }

    public BMW(string model, int year) : base("BMW", model, year)
    {
    }
}
```

Test

```
public class Program
{
  public static void Main(string[] args)
  {
      Car unknown = new Car("Unknown", "Unknown", -1);

      BMW m3_2021 = new BMW("M3", 2021);
      Console.WriteLine(m3_2021);
  }
}
```

▼ Abstract Classes

Abstract classes are classes that contain zero or more ***abstract*** methods and or zero or more **concrete** methods.

A ***concrete*** method is a method that has an ***implementation***.

An ***abstract*** method ***is*** a method that ***is*** declared, but contains ***no*** implementation. The ***implementing*** class provides it's implementation.

An ***abstract*** class cannot be instantiated.

▼ C# Events

Events

- Delegates
- Action
- Func
- Predicates
- EventHandlers

The class who defines and raises the events is called the **publisher** class.

Some other class that receives a notification is called the **subscriber**

Events use the **Publisher-Subscriber** model

Publishers raises an event when some action occurred. The Subscribers who are interested in getting notified when an action occurred, should register with an event and handle it.

▼ Delegate Example

```
public delegate void Notify();
public delegate bool Notify();
public delegate void Notify(string message);
public delegate bool Notify(string message);

// Publisher Class
public class ChocolateMaker
{
    // Event
    public event Notify ChocolateCompleted;

    public void StartMakingChocolate()
    {
        Console.Write("Start Making Chocolate!");
        OnChocolateCompleted();
    }

    // Raise Event Chocolate Completed
    public virtual void OnChocolateCompleted()
    {
        // Invoke the event and call all registered subscribers
        ChocolateCompleted?.Invoke();
    }
}

// Subscriber Class
public class Program
{
    public static void Main(string[] args)
    {
        ChocolateMaker chocolateMaker = new ChocolateMaker();
        // Subscribing to the ChocolateCompleted event
        chocolateMaker.ChocolateCompleted += LindtChocolateCompleted;
        chocolateMaker.StartMakingChocolate();
    }

    // Subscriber to ChocolateComplete Event
    public static void LindtChocolateCompleted()
    {
        Console.WriteLine("Lindt Chocolate is Done!");
    }
}
```

▼ Action and Func Examples

Action

```
public class ChocolateMaker
{
    // Event
    public event Action ChocolateCompleted;

    public void StartMakingChocolate()
    {
        Console.Write("Start Making Chocolate!");
        OnChocolateCompleted();
    }

    // Raise Event Chocolate Completed
    public virtual void OnChocolateCompleted()
    {
        // Invoke the delegate and call all registered subscribers
        ChocolateCompleted?.Invoke();
    }
}

// Subscriber Class
public class Program
{
    public static void Main(string[] args)
    {
        // Publisher Subscriber
        ChocolateMaker chocolateMaker = new ChocolateMaker();
        chocolateMaker.ChocolateCompleted += LindtChocolateCompleted;
        chocolateMaker.StartMakingChocolate();

        // Lamba Method
        Action<string> Print = message => Console.WriteLine(message);
    }

    // Subscriber to ChocolateComplete Event
    public static void LindtChocolateCompleted()
    {
        Console.WriteLine("Lindt Chocolate is Done!");
    }
}
```

Func

```
public class ChocolateMaker
{
    // Event
    public event Func<bool> ChocolateCompleted;

    public void StartMakingChocolate()
    {
        Console.Write("Start Making Chocolate!");
        bool result = OnChocolateCompleted();

        if(result) Console.WriteLine("Clean Chocolate Maker");
    }

    // Raise Event Chocolate Completed
    public virtual bool OnChocolateCompleted()
    {
        // Invoke the delegate and call all registered subscribers
        return ChocolateCompleted?.Invoke();
    }
}

// Subscriber Class
public class Program
{
    public static void Main(string[] args)
    {
        // Publisher Subscriber
        ChocolateMaker chocolateMaker = new ChocolateMaker();
        chocolateMaker.ChocolateCompleted += LindtChocolateCompleted;
        bool evaluate = chocolateMaker.StartMakingChocolate();

        // Func
        Func<int, int, int> sum = (a, b) => x + y;
        Console.WriteLine(sum(1, 2));
    }

    // Subscriber to ChocolateComplete Event
    public static bool LindtChocolateCompleted()
    {
        Console.WriteLine("Lindt Chocolate is Done!");
        return true;
    }
}
```

▼ Keywords

- delegate - The delegate is a reference type data type that defines the method signature ; the delegate must refer to a method with the same signature
- Action - Action is a delegate has 0 or more input parameters and has a void signature ; it can be used with a method that has a void return type ; Actions can have 16 input parameters
- Func - Func is a delegate that has 1 output parameter and 0 or more input parameters ; The last parameter is considered an out parameter ; Func can have 16 input parameters and only 1 output
- event - declare an event in a publisher class ; can only be invoked in the publisher class

▼ C# Exceptions

▼ Try Catch

```
public void division(int num1, int num2) {

  double result = 0;
  try
  {
    // statements causing exception
    result = num1 / num2;
  }
  catch (DivideByZeroException e)
  {
    // error handling code
    Console.WriteLine("Exception caught: {0}", e);
  }
  finally
  {
    // statements to be executed
    Console.WriteLine("Result: {0}", result);
  }
}
```

▼ User Defined Exceptions

```
// Derive from Exception Class
public class TempIsZeroException: Exception {

  // Constructor implements the base constructor from Exception
  public TempIsZeroException(string message): base(message)
  {
  }

}
```

▼ Throw Exception

In c#, the **throw** is a keyword, and it is useful to throw an exception manually during the execution of the program, and we can handle those thrown exceptions using try-catch
blocks based on our requirements.

For more reading on throw please go here.

```
public class Temperature
{
  int temperature = 0;

  public void showTemp()
  {
    if(temperature == 0)
    {
      // Throw a object derived from the System.Exception class
      throw new TempIsZeroException("Zero Temperature found");
    }
    else
    {
      Console.WriteLine("Temperature: {0}", temperature);
    }
  }
}


public void CheckValidTemp()
{
  try
  {
    showTemp();
  }
  catch (Exception ex)
  {
    // error handling code
    Console.WriteLine(ex.Message);
  }
  finally
  {
    // statements to be executed
    Console.WriteLine("Valid temperature.");
  }
}
```

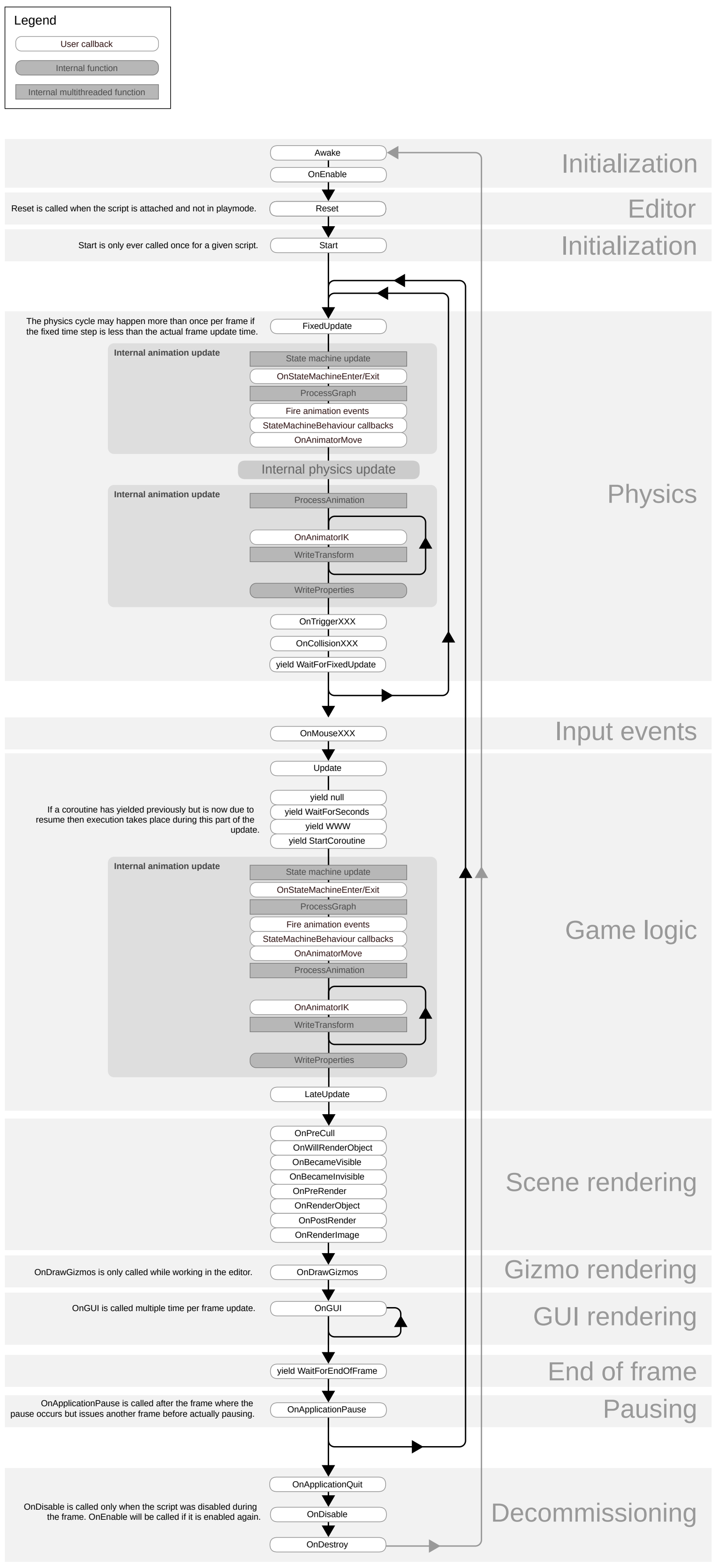
▼ C# File Handling

```
// Read File
public string[] ReadEntireFile(string path)
{
    string[] fileContents = null;
    // List<string> fileContents = null;
    try
    {
        // fileContents = File.ReadLines(path).ToList();
        fileContents = File.ReadAllLines(path);
    }
    catch (FileNotFoundException fileNotFoundException)
    {
        Console.WriteLine(fileNotFoundException.Message);
    }
    catch (DirectoryNotFoundException directoryNotFoundException)
    {
        Console.WriteLine(directoryNotFoundException.Message);
    }

    return fileContents;
}


// Write Text To File
public void WriteToFile(string path, string text)
{
    try
    {
        File.WriteAllText(path, text);
    }
    catch (FileNotFoundException fileNotFoundException)
    {
        Console.WriteLine(fileNotFoundException.Message);
    }
    catch (DirectoryNotFoundException directoryNotFoundException)
    {
        Console.WriteLine(directoryNotFoundException.Message);
    }
}
```

▼ Source Control - GitHub

- Install Source Tree | Fork

▼ Theme 2 - Unity & C# Integration


Legend
User callback
Internal function
Internal multithreaded function
Initialization
Awake
OnEnable
Editor
Reset is called when the script is attached and not in playmode.
Reset
Initialization
Start is only ever called once for a given script.
Start
Physics
The physics cycle may happen more than once per frame if the fixed time step is less than the actual frame update time.
FixedUpdate
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
Internal physics update
Internal animation update
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
OnTriggerXXX
OnCollisionXXX
yield WaitForFixedUpdate
Input events
OnMouseXXX
Game logic
Update
If a coroutine has yielded previously but is now due to resume then execution takes place during this part of the update.
yield null
yield WaitForSeconds
yield WWW
yield StartCoroutine
Internal animation update
State machine update
OnStateMachineEnter/Exit
ProcessGraph
Fire animation events
StateMachineBehaviour callbacks
OnAnimatorMove
ProcessAnimation
OnAnimatorIK
WriteTransform
WriteProperties
LateUpdate
Scene rendering
OnPreCull
OnWillRenderObject
OnBecameVisible
OnBecameInvisible
OnPreRender
OnRenderObject
OnPostRender
OnRenderImage
Gizmo rendering
OnDrawGizmos is only called while working in the editor.
OnDrawGizmos
GUI rendering
OnGUI is called multiple time per frame update.
OnGUI
End of frame
yield WaitForEndOfFrame
Pausing
OnApplicationPause is called after the frame where the pause occurs but issues another frame before actually pausing.
OnApplicationPause
Decommissioning
OnApplicationQuit
OnDisable is called only when the script was disabled during the frame. OnEnable will be called if it is enabled again.
OnDisable
OnDestroy