互動程式設計III

Interactive Programming Design Integration

Introduction to Advanced Class Design

- In this lesson you'll learn:
  - Practical examples and implementation in C# OOP Class Design and How to model software systems using UML.
  - Benefits and usage of generic classes in C#.
  - Practical applications of reflection for dynamic type discovery and method invocation.
- Learning Objectives
  - Understand OOP Concepts: Grasp the foundational principles of Object-Oriented Programming and their implementation in C#.
  - Model with UML: Learn to create and interpret UML diagrams to effectively design and communicate software architecture.
  - Utilize Generics: Implement and leverage generics for type-safe and reusable code components.
  - Apply Reflection: Use reflection to dynamically inspect and manipulate code, enhancing flexibility and adaptability.
- Why This Chapter is Important
  - Foundation of Software Development:
    - OOP principles are fundamental to modern software design, enabling scalable, maintainable, and reusable code.
    - UML provides a standardized way to visualize system architecture, facilitating clear communication and documentation.
  - Enhanced Code Efficiency:
    - Generics allow for the creation of flexible and reusable code without sacrificing type safety, improving overall code quality and reducing redundancy.
  - Dynamic and Flexible Code:
    - Reflection offers powerful capabilities to interact with code at runtime, enabling dynamic type discovery, method invocation, and metadata inspection. This flexibility is crucial for building adaptable and robust applications.

互動程式設計III

Interactive Programming Design Integration

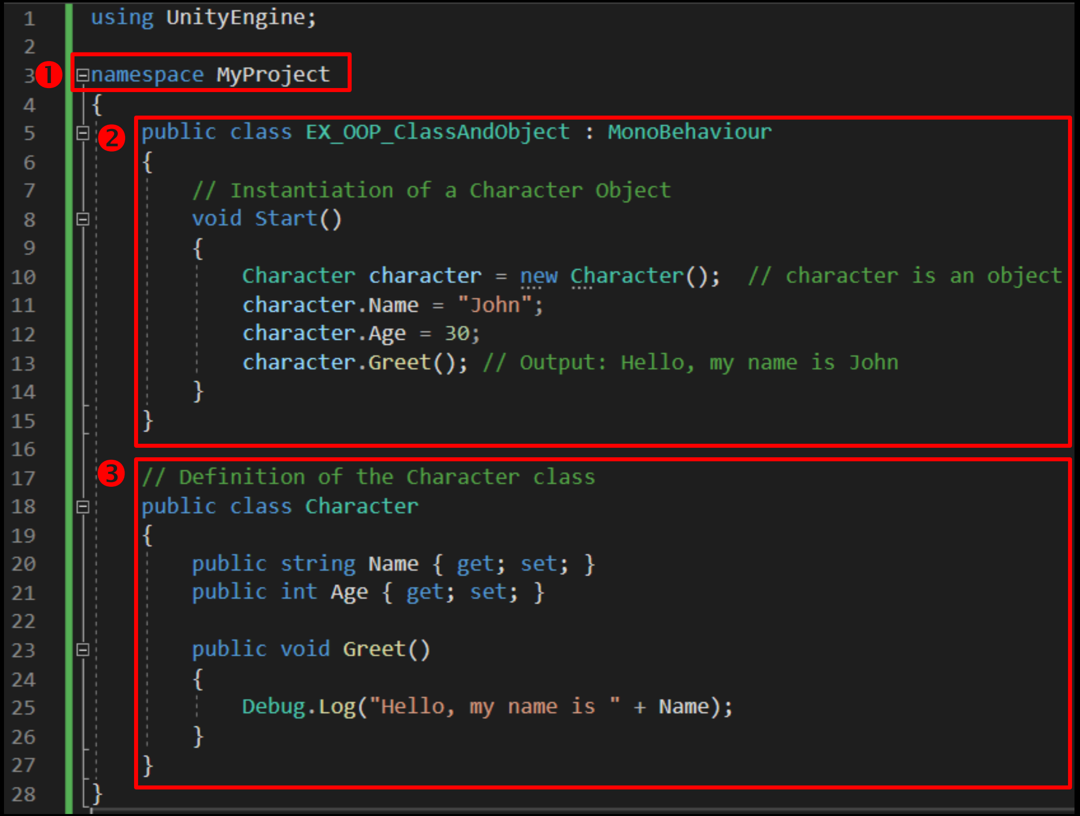
- Advanced Class Design
  - OOP
  - UML
  - Generics
  - Reflection
  - Attribute

OOP

- Introduction to Object-Oriented Programming (OOP)
  - Definition:
    - Explanation of OOP as a programming paradigm based on the concept of ”objects”.
  - Key Principles:
    - Encapsulation: Bundling the data (attributes) and methods (functions) that operate on the data into a single unit or class.
    - Abstraction: Hiding complex implementation details and showing only the necessary features of an object.
    - Inheritance: Mechanism by which one class can inherit the properties and methods of another class.
    - Polymorphism: Ability to present the same interface for different underlying data types.
  - Benefits:
    - Code reusability
    - Improved code maintainability
    - Easier debugging and testing

Class and Object

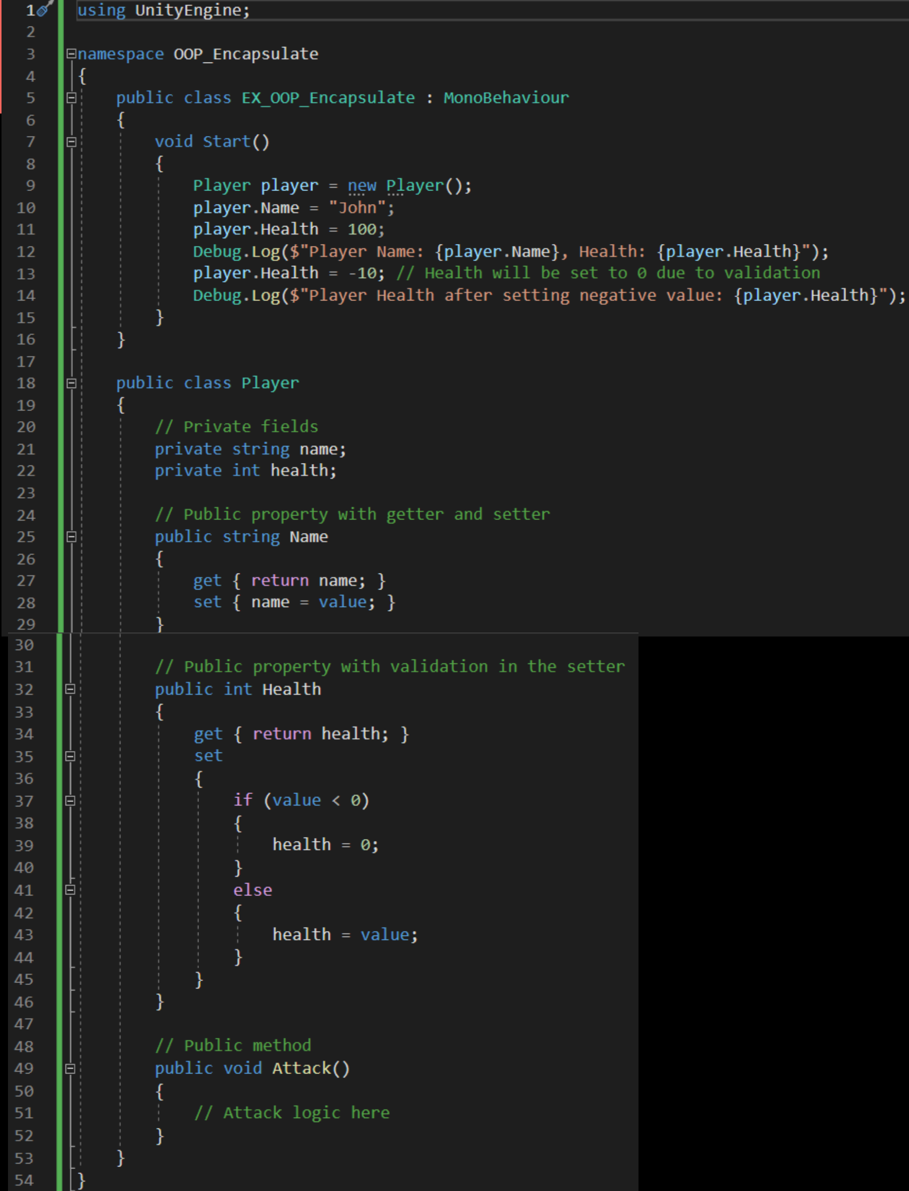
- Classes
  - A class is a blueprint for creating objects. It defines a datatype by bundling data and methods that work on the data into one single unit❷.
  - To avoid class name conflict, we can use namespace to separate the classes with the same name❶.
- Objects
  - An object is an instance of a class. It is created from a class and can use the class's methods and properties❸.
  - We can have more than one objects that are derived from the same class at the same time.



```
using UnityEngine;

namespace MyProject
{
    public class EX_OOP_ClassAndObject : MonoBehaviour
    {
        // Instantiation of a Character Object
        void Start()
        {
            Character character = new Character();  // character is an object
            character.Name = "John";
            character.Age = 30;
            character.Greet(); // Output: Hello, my name is John
        }
    }

    // Definition of the Character class
    public class Character
    {
        public string Name { get; set; }
        public int Age { get; set; }

        public void Greet()
        {
            Debug.Log("Hello, my name is " + Name);
        }
    }
}
```

OOP - Encapsulate

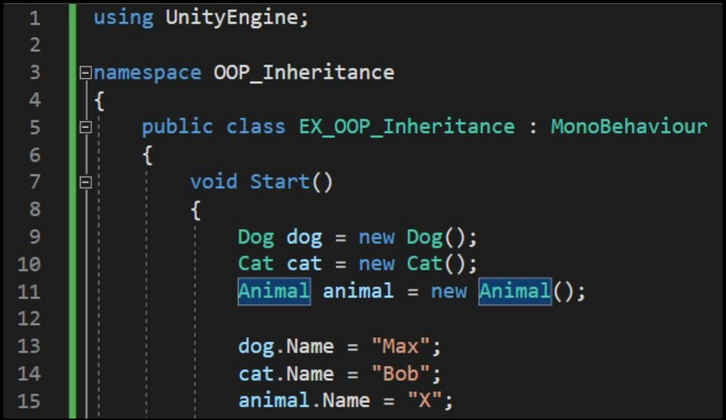
- Definition:
  - Encapsulation is the process of bundling the data (attributes) and methods (functions) that operate on the data into a single unit, typically a class.
- Key Points:
  - Access Modifiers: Control the visibility of class members.
    - Private: Members are accessible only within the same class.
    - Public: Members are accessible from any other class.
    - Protected: Members are accessible within the same class and by derived class instances.
    - Internal: Members are accessible within the same assembly.
  - Properties: Provide controlled access to class fields.
- Benefits:
  - Protects data from being accessed and modified directly.
  - Promotes modularity and maintainability.
  - Enables validation and logic enforcement through getters and setters.



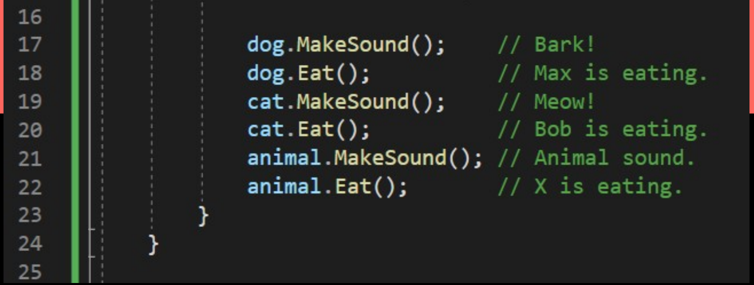
```csharp
using UnityEngine;

namespace OOP_Encapsulate
{
    public class EX_OOP_Encapsulate : MonoBehaviour
    {
        void Start()
        {
            Player player = new Player();
            player.Name = "John";
            player.Health = 100;
            Debug.Log($"Player Name: {player.Name}, Health: {player.Health}");
            player.Health = -10; // Health will be set to 0 due to validation
            Debug.Log($"Player Health after setting negative value: {player.Health}");
        }
    }

    public class Player
    {
        // Private fields
        private string name;
        private int health;

        // Public property with getter and setter
        public string Name
        {
            get { return name; }
            set { name = value; }
        }

        // Public property with validation in the setter
        public int Health
        {
            get { return health; }
            set
            {
                if (value < 0)
                {
                    health = 0;
                }
                else
                {
                    health = value;
                }
            }
        }

        // Public method
        public void Attack()
        {
            // Attack logic here
        }
    }
}
```

OOP - Inheritance

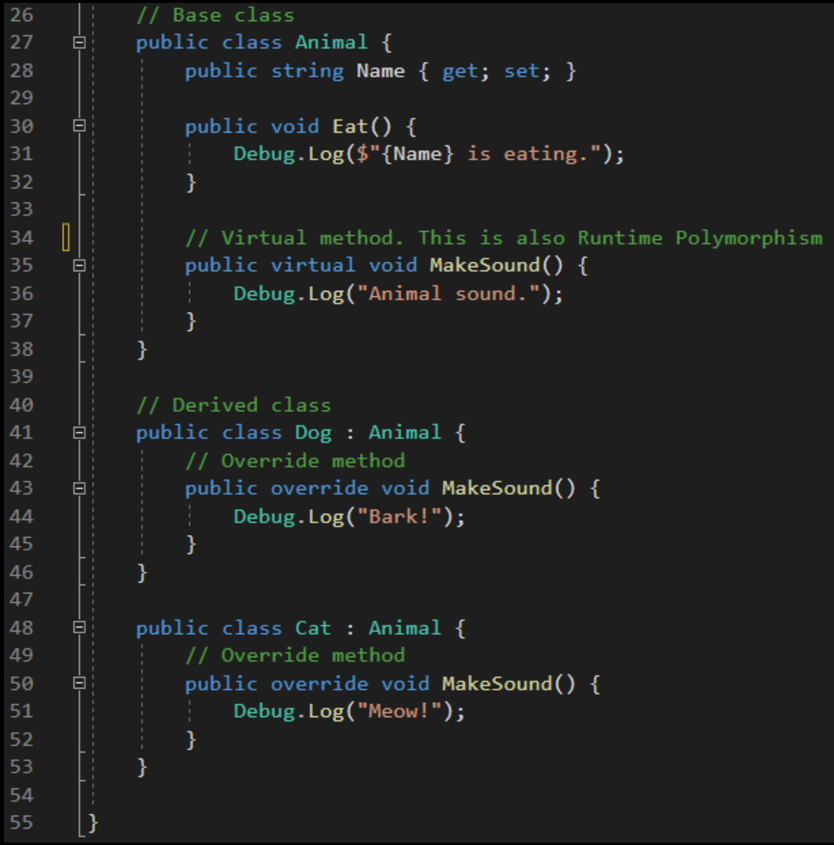
- Definition:
  - Inheritance is a fundamental concept in object-oriented programming where a new class (derived class) is created from an existing class (base class).
  - The derived class inherits fields, properties, and methods from the base class.
- Key Points:
  - Base Class: The class being inherited from.
  - Derived Class: The class that inherits from the base class.
  - Keywords: base, virtual, override



```csharp
using UnityEngine;

namespace OOP_Inheritance
{
    public class EX_OOP_Inheritance : MonoBehaviour
    {
        void Start()
        {
            Dog dog = new Dog();
            Cat cat = new Cat();
            Animal animal = new Animal();

            dog.Name = "Max";
            cat.Name = "Bob";
            animal.Name = "X";
```



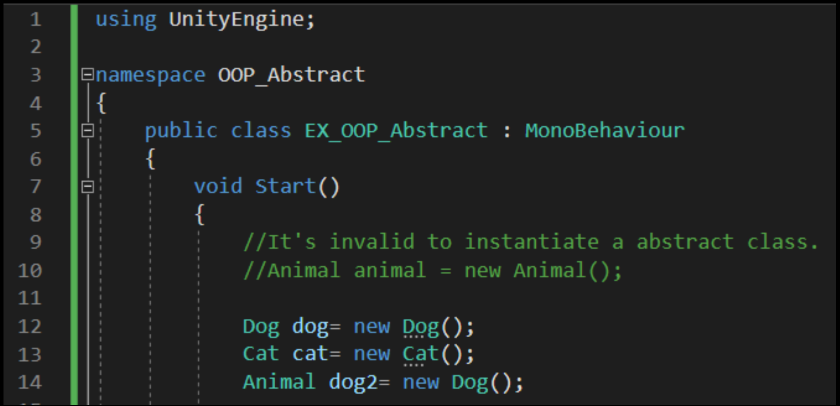
```csharp

            dog.MakeSound();    // Bark!
            dog.Eat();          // Max is eating.
            cat.MakeSound();    // Meow!
            cat.Eat();          // Bob is eating.
            animal.MakeSound(); // Animal sound.
            animal.Eat();       // X is eating.
        }
    }

```



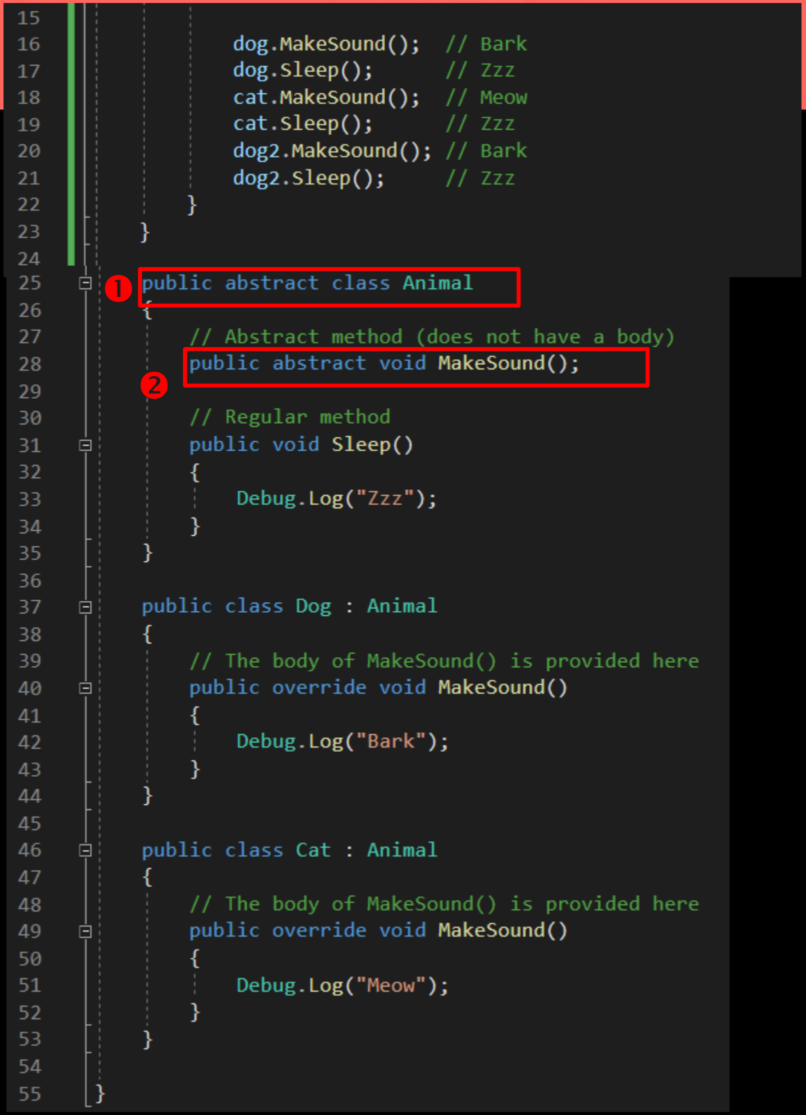
```csharp
    // Base class
    public class Animal {
        public string Name { get; set; }

        public void Eat() {
            Debug.Log($"{Name} is eating.");
        }

        // Virtual method. This is also Runtime Polymorphism
        public virtual void MakeSound() {
            Debug.Log("Animal sound.");
        }
    }

    // Derived class
    public class Dog : Animal {
        // Override method
        public override void MakeSound() {
            Debug.Log("Bark!");
        }
    }

    public class Cat : Animal {
        // Override method
        public override void MakeSound() {
            Debug.Log("Meow!");
        }
    }

}
```

OOP - Abstract

- Definition:
  - Abstract Class: A class that cannot be instantiated and is meant to be subclassed. It can contain both abstract methods (without implementation) and concrete methods (with implementation)❶.
  - Abstract Method: A method without implementation that must be overridden in a subclass❷.
- Key Points:
  - To provide a common base class with default behavior while enforcing a contract for subclasses to provide specific implementations.
- Usage:
  - Used when you have a base class that should not be instantiated on its own.
  - Helps in defining a template for a group of subclasses.



```
using UnityEngine;

namespace OOP_Abstract
{
    public class EX_OOP_Abstract : MonoBehaviour
    {
        void Start()
        {
            //It's invalid to instantiate a abstract class.
            //Animal animal = new Animal();

            Dog dog= new Dog();
            Cat cat= new Cat();
            Animal dog2= new Dog();
```



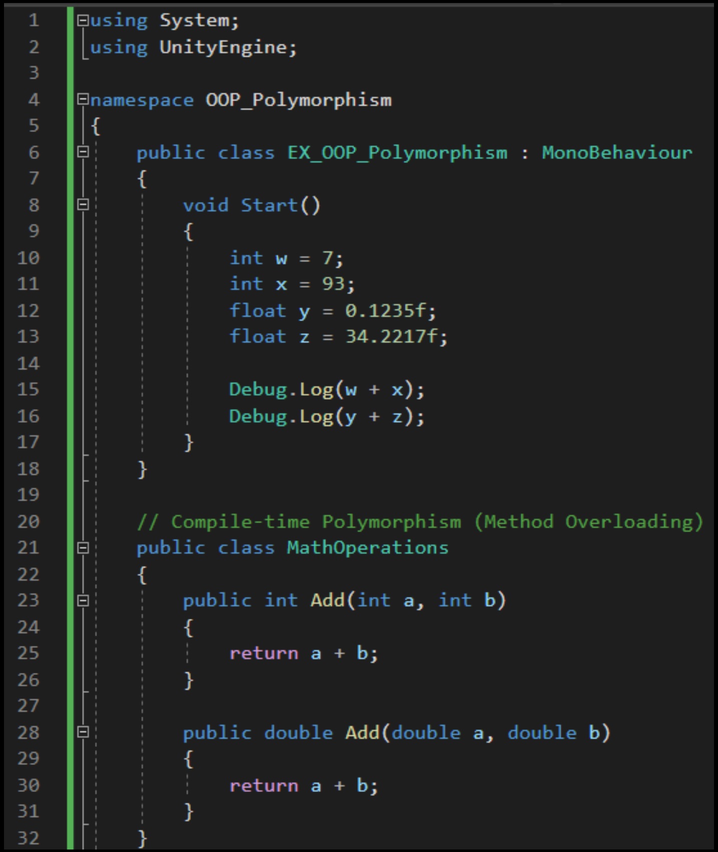
```

            dog.MakeSound();  // Bark
            dog.Sleep();      // Zzz
            cat.MakeSound();  // Meow
            cat.Sleep();      // Zzz
            dog2.MakeSound(); // Bark
            dog2.Sleep();     // Zzz
        }
    }

❶  public abstract class Animal
    {
        // Abstract method (does not have a body)
        public abstract void MakeSound();
❷
        // Regular method
        public void Sleep()
        {
            Debug.Log("Zzz");
        }
    }

    public class Dog : Animal
    {
        // The body of MakeSound() is provided here
        public override void MakeSound()
        {
            Debug.Log("Bark");
        }
    }

    public class Cat : Animal
    {
        // The body of MakeSound() is provided here
        public override void MakeSound()
        {
            Debug.Log("Meow");
        }
    }

}
```

Abstract Class VS. Interface

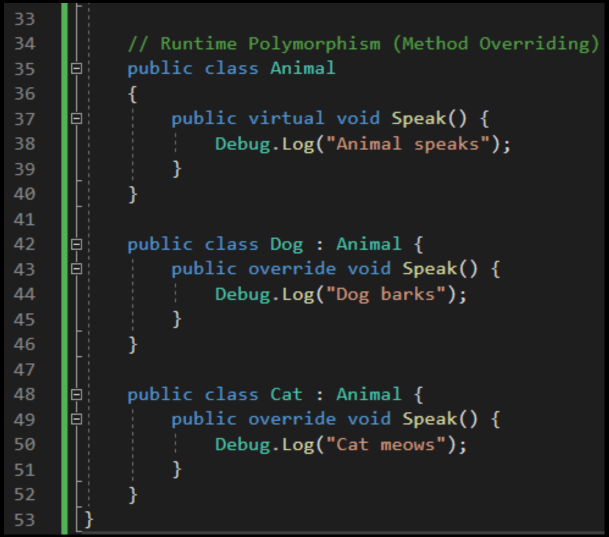
- Key Differences
  - Shown in Unity Editor:
    - Abstract Class: Can be exposed to Unity Editor
    - Interface: Can NOT be exposed to Unity Editor
  - Instantiation:
    - Abstract Class: Cannot be instantiated directly.
    - Interface: Cannot be instantiated directly.
  - Implementation:
    - Abstract Class: Can have both abstract methods (without implementation) and concrete methods (with implementation).
    - Interface: Can only have method and property declarations (no implementation).
  - Multiple Inheritance:
    - Abstract Class: A class can inherit only one abstract class (single inheritance).
    - Interface: A class can implement multiple interfaces (multiple inheritance).
  - Usage Scenarios:
    - Abstract Class: Use when you have a base class with shared code and need to provide a common foundation for derived classes.
    - Interface: Use when you need to define a contract that can be implemented by any class, regardless of where it sits in the class hierarchy.

OOP - Polymorphism

- Definition:
  - Method Polymorphism allows methods to do different things based on the object it is acting upon, even if they share the same name.
- Types of Method Polymorphism:
  - Compile-time (Static) Polymorphism:
    - Achieved through method overloading.
    - Methods have the same name but different signatures (different parameters).
  - Runtime (Dynamic) Polymorphism:
    - Achieved through method virtual/override.
    - Methods have the same name and signature but are defined in a base class and derived class.
- Benefits:
  - Simplifies code readability and maintenance.
  - Enables flexibility and integration of new functionality without altering existing code.



```csharp
using System;
using UnityEngine;

namespace OOP_Polymorphism
{
    public class EX_OOP_Polymorphism : MonoBehaviour
    {
        void Start()
        {
            int w = 7;
            int x = 93;
            float y = 0.1235f;
            float z = 34.2217f;

            Debug.Log(w + x);
            Debug.Log(y + z);
        }
    }

    // Compile-time Polymorphism (Method Overloading)
    public class MathOperations
    {
        public int Add(int a, int b)
        {
            return a + b;
        }

        public double Add(double a, double b)
        {
            return a + b;
        }
    }
```



```csharp

    // Runtime Polymorphism (Method Overriding)
    public class Animal
    {
        public virtual void Speak() {
            Debug.Log("Animal speaks");
        }
    }

    public class Dog : Animal {
        public override void Speak() {
            Debug.Log("Dog barks");
        }
    }

    public class Cat : Animal {
        public override void Speak() {
            Debug.Log("Cat meows");
        }
    }
}
```

- Advanced Class Design
  - OOP
  - UML
  - Generics
  - Reflection
  - Attribute

互動程式設計III

Interactive Programming Design Integration

Introduction to UML(1)

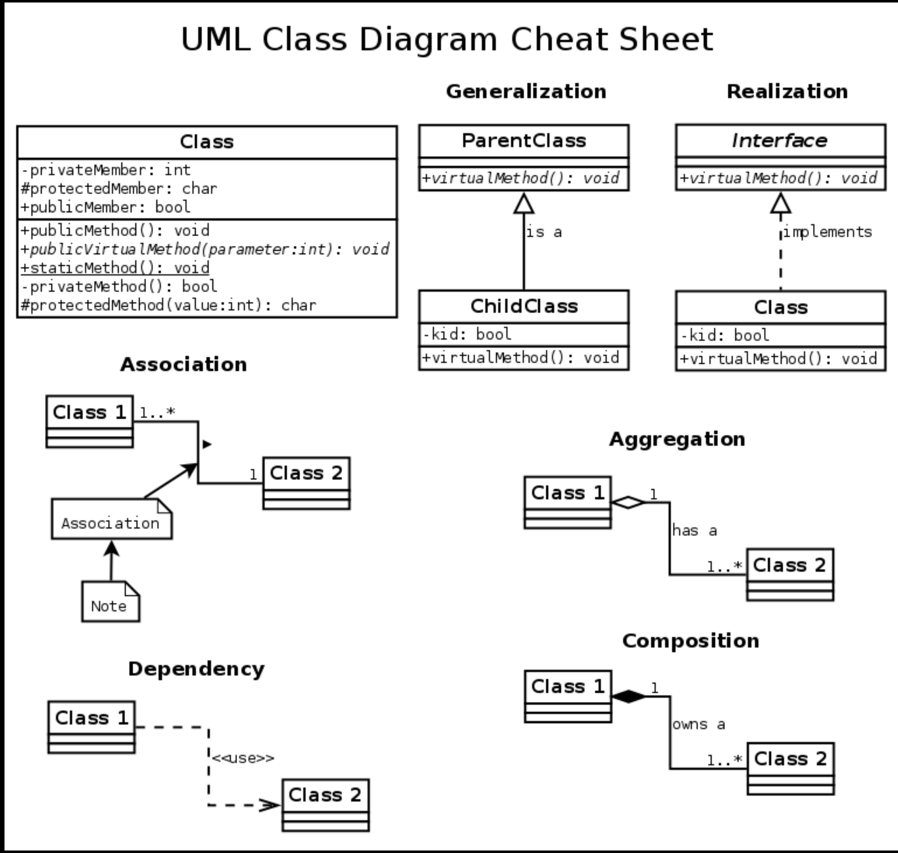
- What is UML?
  - Definition: Unified Modeling Language (UML) is a standardized visual language for creating models of object-oriented software systems.
  - Purpose: Helps in specifying, visualizing, constructing, and documenting the artifacts of a software system.
- Key Components of UML
  - Structural Diagrams:
    - Class Diagram: Shows the static structure of the system, including classes, their attributes, operations, and relationships.
    - Object Diagram: Represents the structure of a system at a particular point in time.
    - Component Diagram: Illustrates the organization and dependencies among software components.
    - Deployment Diagram: Depicts the physical deployment of artifacts on nodes.
  - Behavioral Diagrams:
    - Use Case Diagram: Describes the functional requirements of the system, representing interactions between actors and use cases.
    - Sequence Diagram: Shows how objects interact in a particular sequence of time.
    - Activity Diagram: Represents the flow of activities in a system, similar to a flowchart.
    - State Machine Diagram: Describes the states of an object and transitions between those states.

Introduction to UML(2)

- Benefits of Using UML
  - Generative AI: Improve the accuracy of code generation.
  - Standardization: Provides a standard way to visualize system design.
  - Communication: Enhances communication among stakeholders, including developers, designers, and business analysts.
  - Documentation: Offers clear documentation of the system architecture and design.
  - Analysis and Design: Facilitates analysis and design by providing different perspectives of the system.
- Free UML textbook:
  - https://link.springer.com/book/10.1007/978-3-319-12742-2

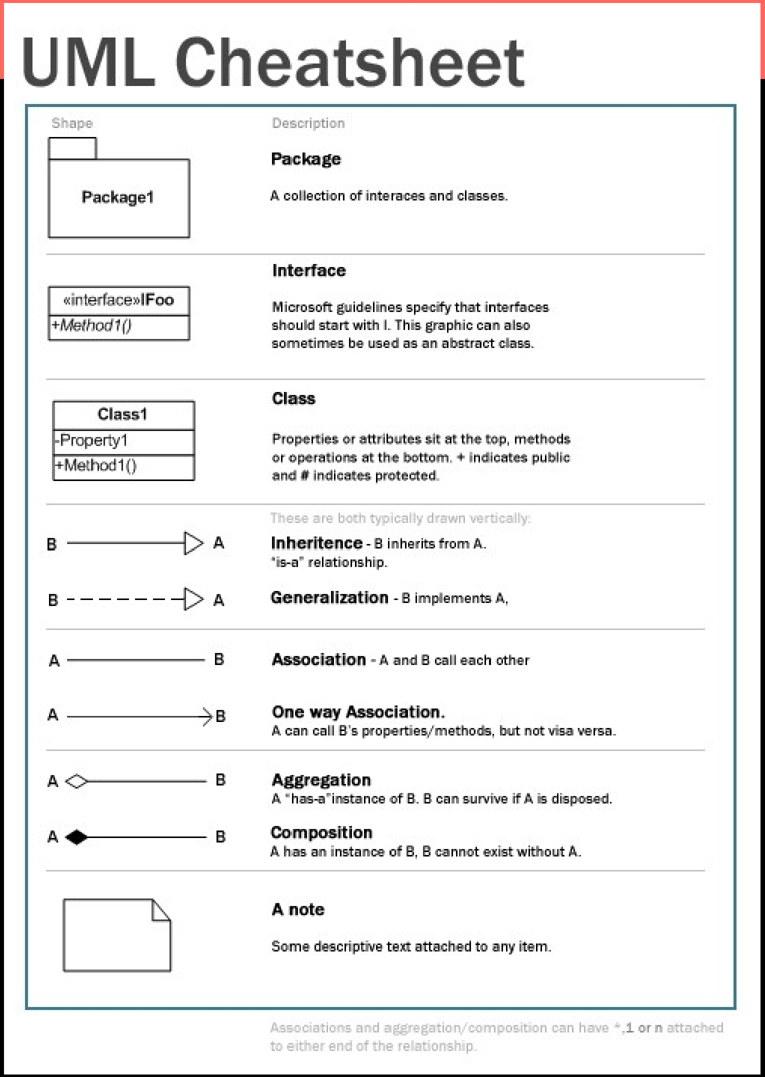
Introduction to UML Class Diagram

- Definition:
  - UML (Unified Modeling Language) Class Diagram is used to visually represent the relationships and structure of a system by showing its classes, attributes, methods, and the relationships among objects[*].
- Key Components:
  - Classes: Represented by rectangles divided into three sections: class name, attributes, and methods.
  - Relationships:
  - Inheritance: Shows "is-a"relationship, represented by a solid line with a hollow arrowhead.
  - Implement: Show "implement" relationship, represented by a dashed line with hallow arrorhead.
  - Association: General relationship between classes, represented by a solid line.
  - Aggregation: Special form of association representing a whole-part relationship, represented by a solid line with a hollow diamond. <Has-a>
  - Composition: Stronger form of aggregation with a lifecycle dependency, represented by a solid line with a filled diamond. <Owns a>



Introduction to UML Class Diagram

- https://yetanotherchris.dev/cheatsheet/uml-cheat-sheet/



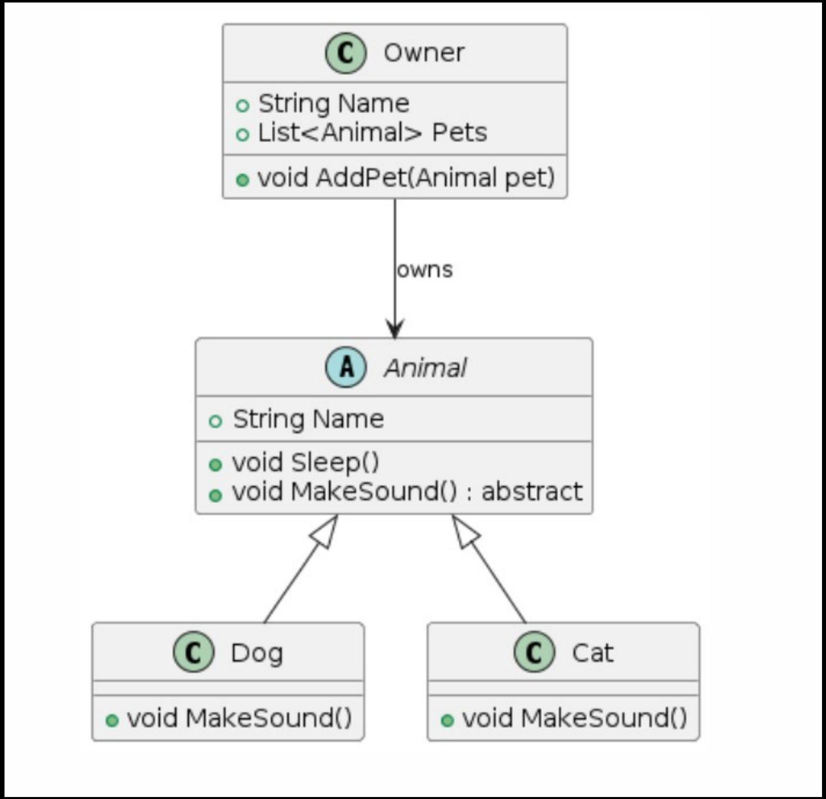
UML Cheatsheet

Shape	Description
Package1	**Package** A collection of interaces and classes.
«interface»IFoo *+Method1()*	**Interface** Microsoft guidelines specify that interfaces should start with I. This graphic can also sometimes be used as an abstract class.
Class1 -Property1 +Method1()	**Class** Properties or attributes sit at the top, methods or operations at the bottom. + indicates public and # indicates protected.
B ——▷ A B - - - -▷ A	These are both typically drawn vertically: **Inheritence** - B inherits from A. "is-a" relationship. **Generalization** - B implements A,
A —— B A ——> B	**Association** - A and B call each other **One way Association.** A can call B's properties/methods, but not visa versa.
A ◇—— B A ◆—— B	**Aggregation** A "has-a"instance of B. B can survive if A is disposed. **Composition** A has an instance of B, B cannot exist without A.
	A note Some descriptive text attached to any item.

Associations and aggregation/composition can have *,1 or n attached to either end of the relationship.

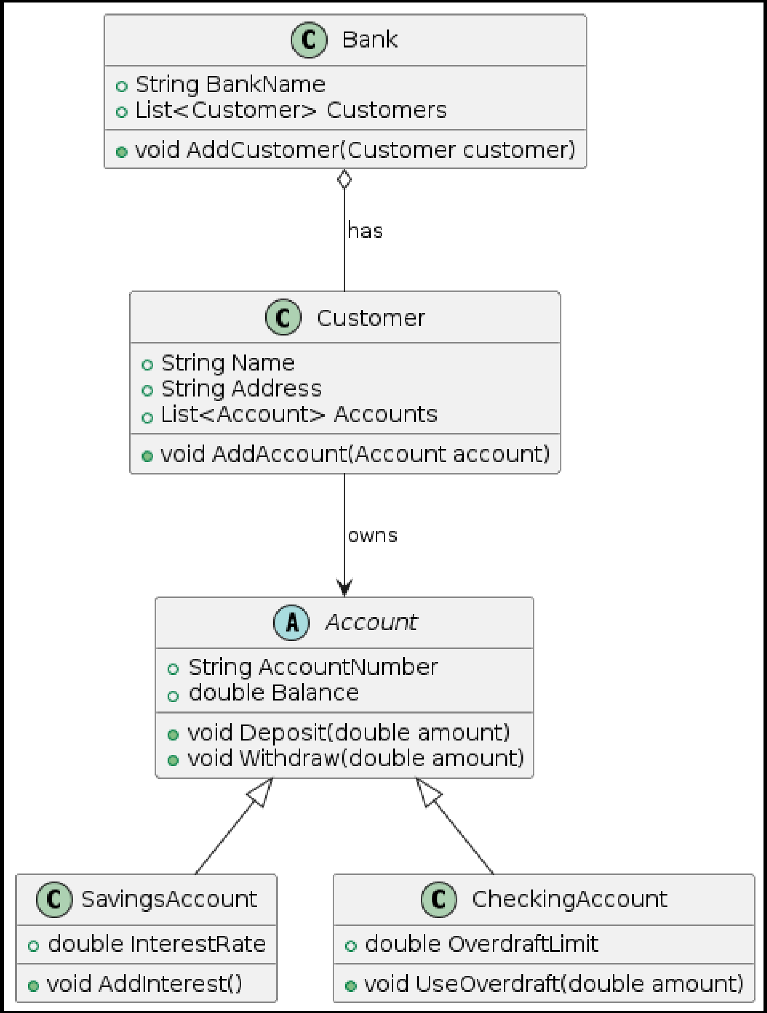
EX: Pet

- Explanation:
  - Animal Class: Abstract class with an abstract method MakeSound() and a concrete method Sleep().
  - Dog and Cat Classes: Inherit from Animal and provide implementations for MakeSound().
  - Owner Class: Represents an owner with a name and a collection of pets (Animals). The relationship between Owner and Animal is shown as an association.



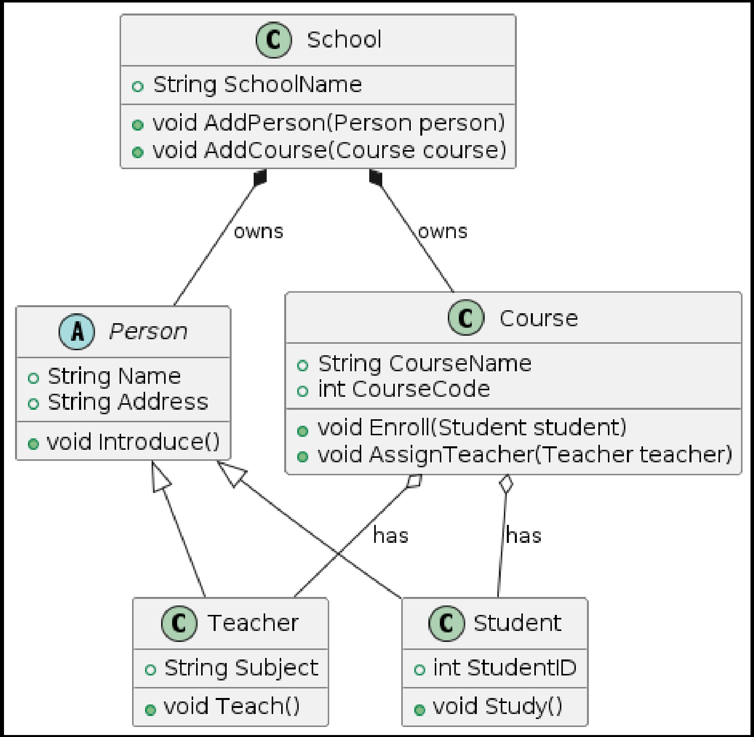
EX: Bank Customer

- Explanation:
  - Account Class: An abstract base class with common properties and methods for all types of accounts.
  - SavingsAccount and CheckingAccount Classes: Inherit from Account and add specific attributes and methods.
  - Customer Class: Represents a customer with multiple accounts, demonstrating association.
  - Bank Class: Represents a bank with multiple customers, demonstrating aggregation.
- Key Points:
  - Inheritance: SavingsAccount and CheckingAccount inherit from the Account class.
  - Association: Customer owns multiple Accounts.
  - Aggregation: Bank has multiple Customers, representing a whole-part relationship without a strong lifecycle dependency.



EX: Bank Customer

- Explanation:
  - Person Class: An abstract base class with common properties and methods for all types of persons.
  - Teacher and Student Classes: Inherit from Person and add specific attributes and methods.
  - Course Class: Represents a course, demonstrating aggregation with teachers and students.
  - School Class: Represents a school, demonstrating composition with persons and courses.
- Key Points:
  - Inheritance: Teacher and Student inherit from the Person class.
  - Composition: School employs/enrolls Persons and offers Courses, indicating a strong relationship.
  - Aggregation: Course includes Teachers and Students as part of its structure, indicating a whole-part relationship.



互動程式設計III

Interactive Programming Design Integration

- Advanced Class Design
  - OOP
  - UML
  - Generics
  - Reflection
  - Attribute

Introduction to Generics

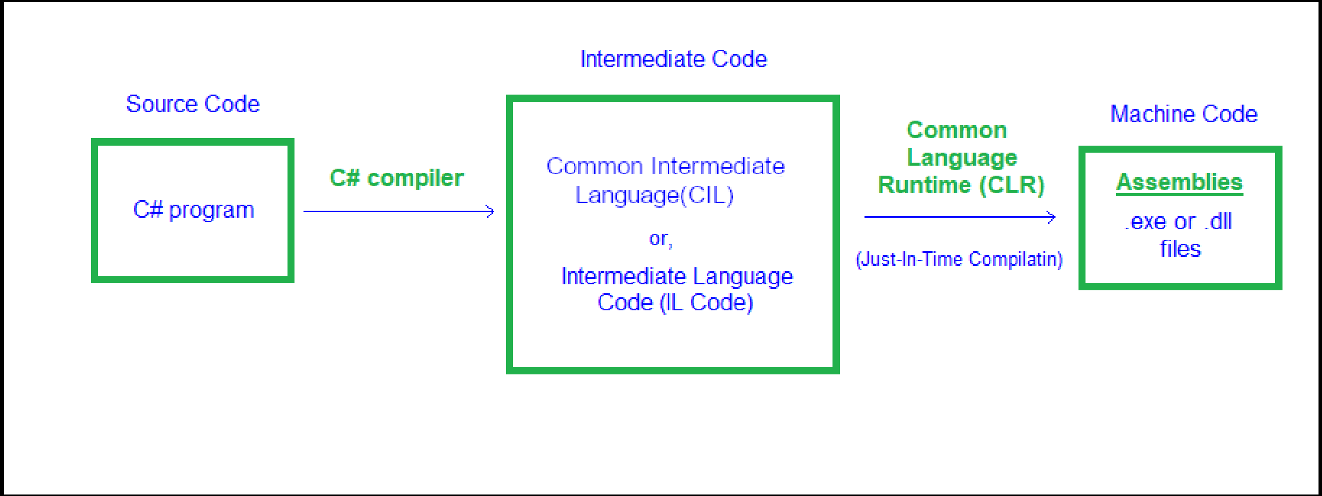
- Definition:
  - Generics allow you to define classes, methods, delegates, and interfaces with a placeholder for the type of data they store or use.
  - Syntax: GenericClass<T> or GenericMethod<T>()
- Benefits:
  - Type Safety: Ensures that the data type is consistent.
  - Code Reusability: Write a method or class once and reuse it for different data types.
  - Performance: Reduces the need for boxing/unboxing, improving performance.
  - Maintainability: Simplifies code maintenance and reduces redundancy.

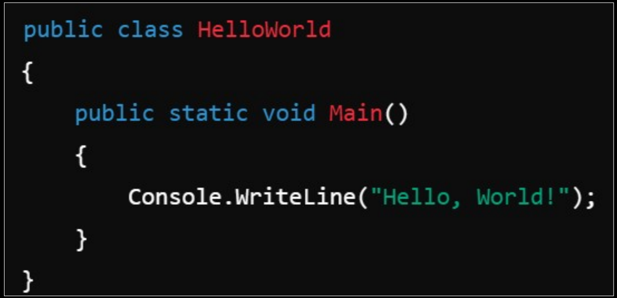
How C# Compiles a Generic Class/Method

- What Happens to Generic Classes at Compile Time
- Compile time:
  - IL Code Generation with Placeholders: During compile time, the C# compiler generates IL (Intermediate Language) code for generic classes and methods. In this IL code, the generic types are represented by placeholder symbols (often referred to as type parameters). These placeholders do not specify a concrete type; instead, they indicate that a type will be provided later during runtime.
- Runtime:
  - Runtime Instantiation: When a generic class or method is instantiated at runtime with specific types (e.g., GenericClass<int> or List<string>), the CLR (Common Language Runtime) replaces these placeholder symbols with the actual types. This process is known as "type instantiation."
  - Code Sharing and JIT Compilation: The CLR utilizes a technique called "code sharing" to optimize the memory usage and performance of generic types. For reference types, a single implementation of the generic type is shared. For value types, the CLR generates specific IL and native code for each value type used with the generic type. This is done by the Just-In-Time (JIT) compiler, which compiles the IL code into native machine code specific to the runtime environment.

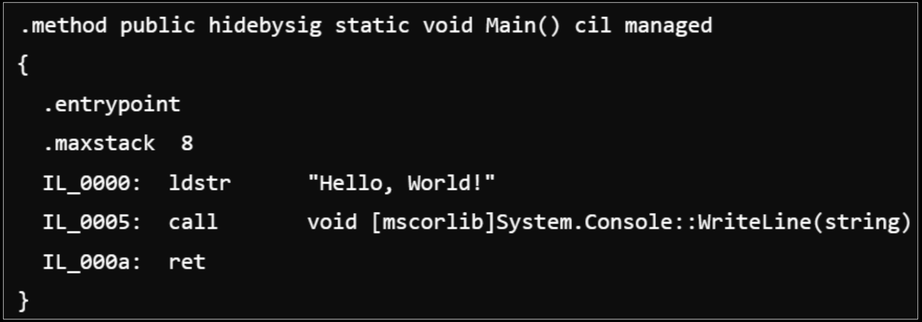
C# Execution Process – JIT Compiler

- How C# Code Gets Compiled and Executed?[*]
- Compile twice: C# compiler, and JIT compiler
- Try to generate some IL codes by yourself: https://sharplab.io/

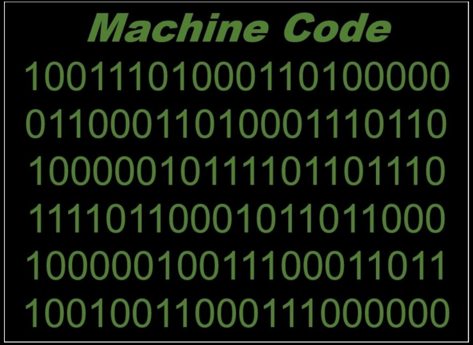

Source Code
C# program
C# compiler
Intermediate Code
Common Intermediate Language(CIL)
or,
Intermediate Language Code (IL Code)
Common Language Runtime (CLR)
(Just-In-Time Compilatin)
Machine Code
Assemblies
.exe or .dll files



```
public class HelloWorld
{
    public static void Main()
    {
        Console.WriteLine("Hello, World!");
    }
}
```



```
.method public hidebysig static void Main() cil managed
{
  .entrypoint
  .maxstack  8
  IL_0000:  ldstr      "Hello, World!"
  IL_0005:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000a:  ret
}
```



Machine Code

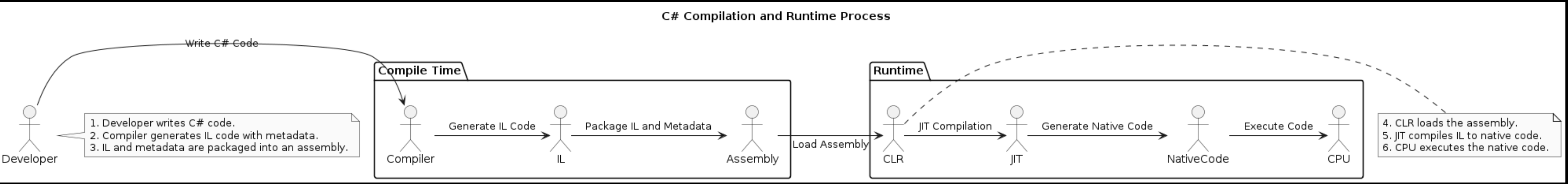
```
1001110100011010000
0110001101000111011
1000001011110110111
1111011000101101100
1000001001110001101
1001001100011100000
```

JIT vs IL2CPP in Unity

- JIT (Just-In-Time) Compilation: JIT compilation is a runtime process where the Intermediate Language (IL) code is compiled into native machine code just before execution.
  - Runtime Compilation: Converts IL code to machine code at runtime.
  - Flexibility: Allows for dynamic code generation and execution.
  - Debugging: Easier debugging since the code is compiled and executed line-by-line.
- IL2CPP (Intermediate Language to C++): IL2CPP is a Unity-specific AOT (Ahead-Of-Time) compilation technology that converts IL code to C++ code, which is then compiled to native machine code.
  - Ahead-Of-Time Compilation: Converts IL code to native code before execution, not at runtime.
  - Performance: Improved runtime performance due to precompiled native code.
  - Platform Support: Required for certain platforms (e.g., iOS, WebGL) where JIT compilation is not allowed.

C# Execution Process

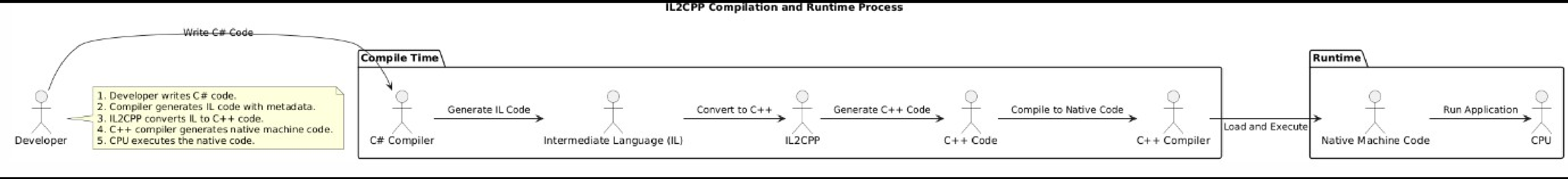
•



1. Developer writes C# code.
2. Compiler generates IL code with metadata.
3. IL and metadata are packaged into an assembly.

4. CLR loads the assembly.
5. JIT compiles IL to native code.
6. CPU executes the native code.

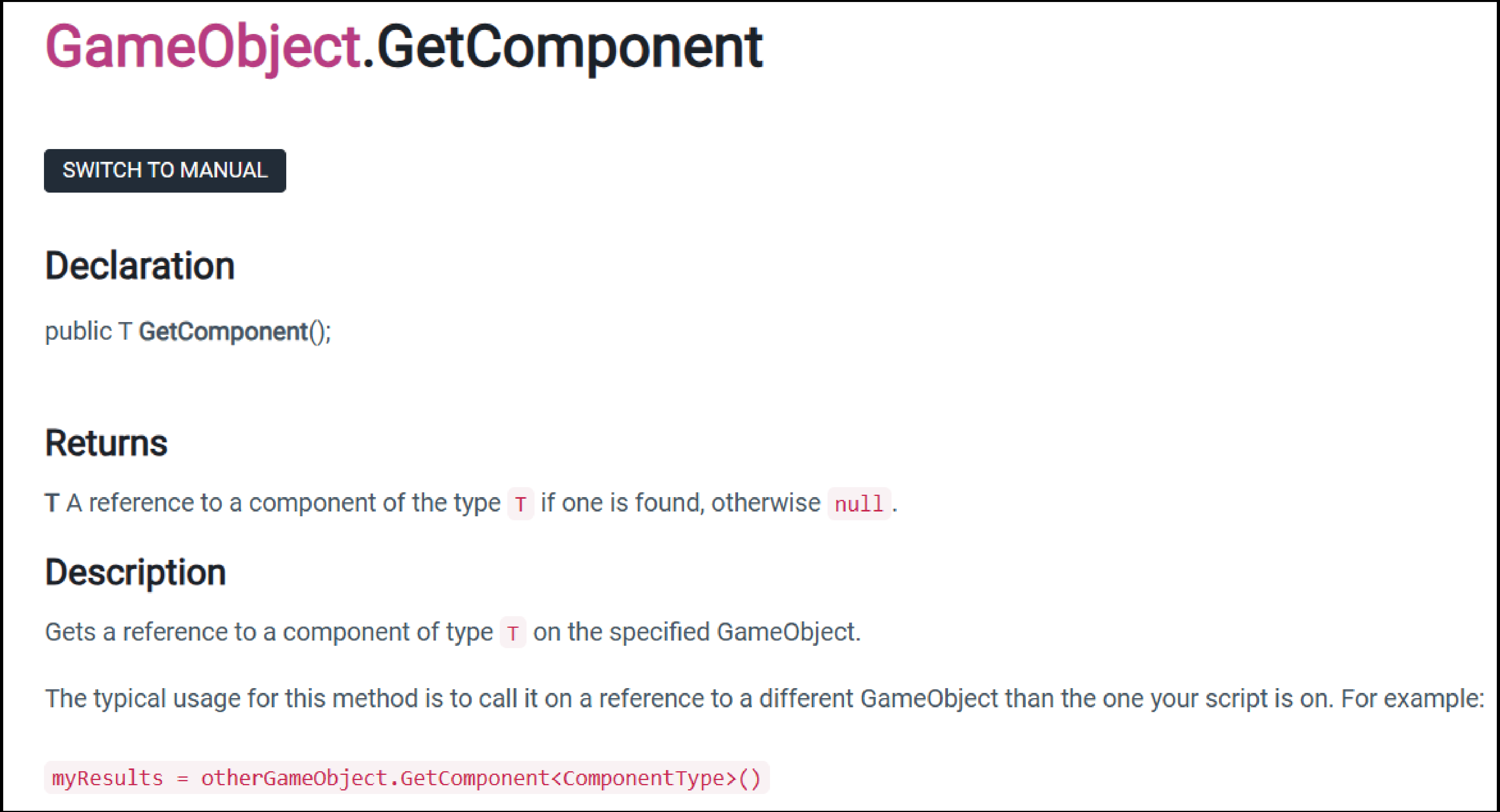
JIT (Just-In-Time) Compilation



1. Developer writes C# code.
2. Compiler generates IL code with metadata.
3. IL2CPP converts IL to C++ code.
4. C++ compiler generates native machine code.
5. CPU executes the native code.

IL2CPP (Intermediate Language to C++)

EX: Unity GetComponent<T>(1)

- 

GameObject.GetComponent

SWITCH TO MANUAL

Declaration

public T **GetComponent**();

Returns

T A reference to a component of the type `T` if one is found, otherwise `null`.

Description

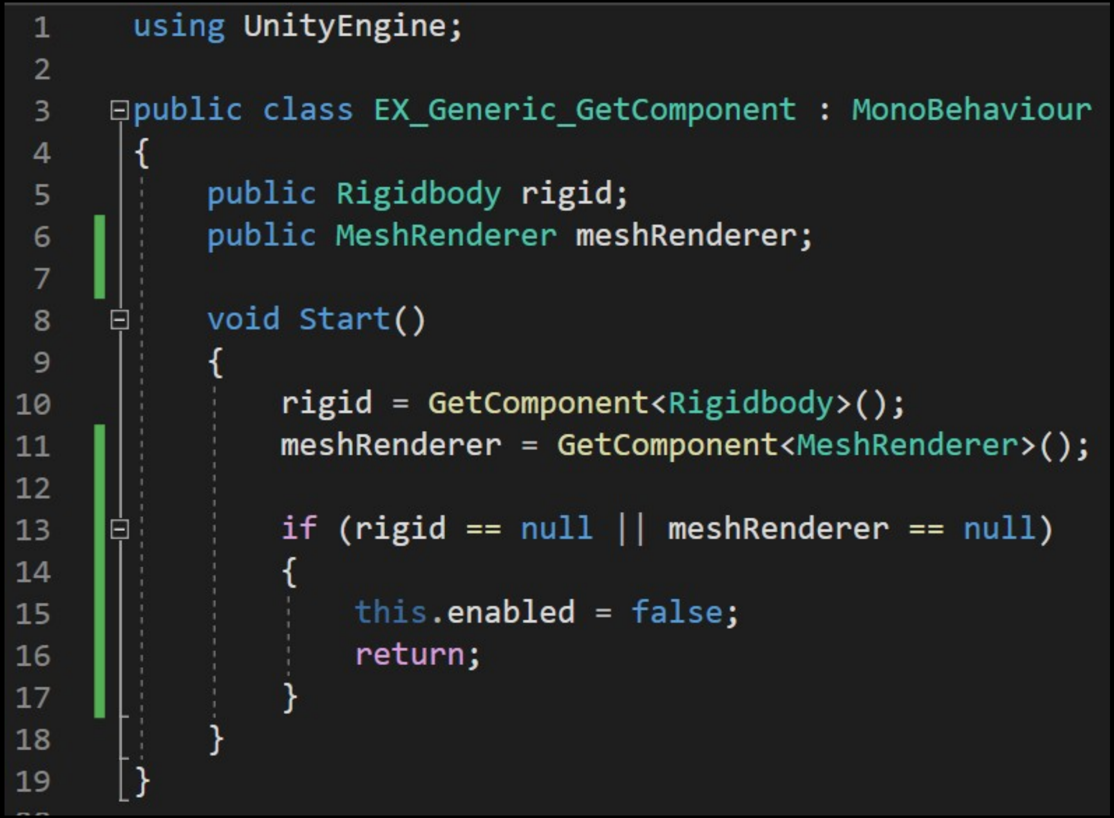
Gets a reference to a component of type `T` on the specified GameObject.

The typical usage for this method is to call it on a reference to a different GameObject than the one your script is on. For example:

```
myResults = otherGameObject.GetComponent<ComponentType>()
```

EX: Unity GetComponent<T>(2)

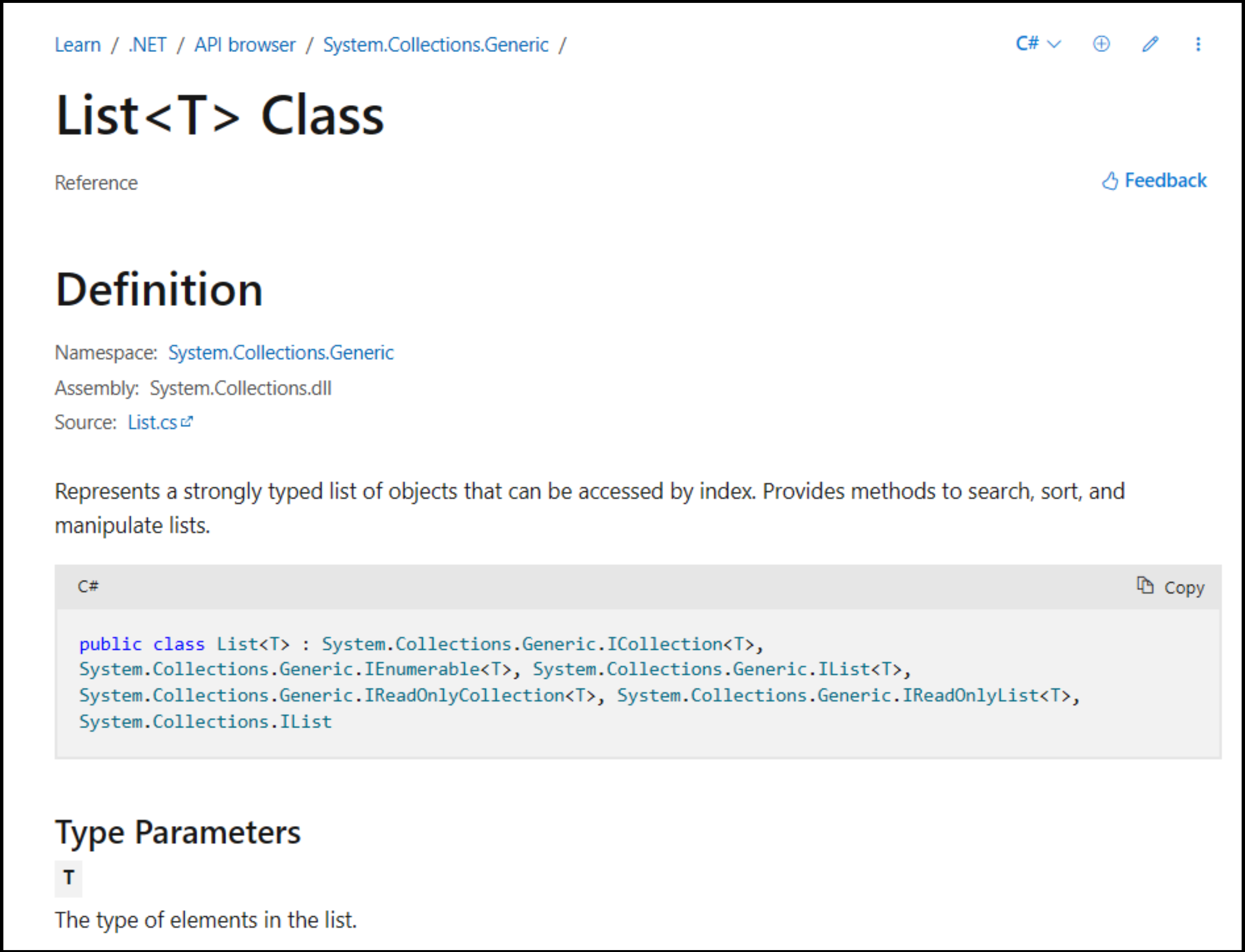
- The return type of GetComponent<T>() is T.
- The generic design enhances type safety and code readability.



```
using UnityEngine;

public class EX_Generic_GetComponent : MonoBehaviour
{
    public Rigidbody rigid;
    public MeshRenderer meshRenderer;

    void Start()
    {
        rigid = GetComponent<Rigidbody>();
        meshRenderer = GetComponent<MeshRenderer>();

        if (rigid == null || meshRenderer == null)
        {
            this.enabled = false;
            return;
        }
    }
}
```

EX: C# List(1)

•



List<T> Class

Reference

Definition

Namespace: System.Collections.Generic

Assembly: System.Collections.dll

Source: List.cs

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

```csharp
public class List<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>,
System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.Generic.IReadOnlyList<T>,
System.Collections.IList
```

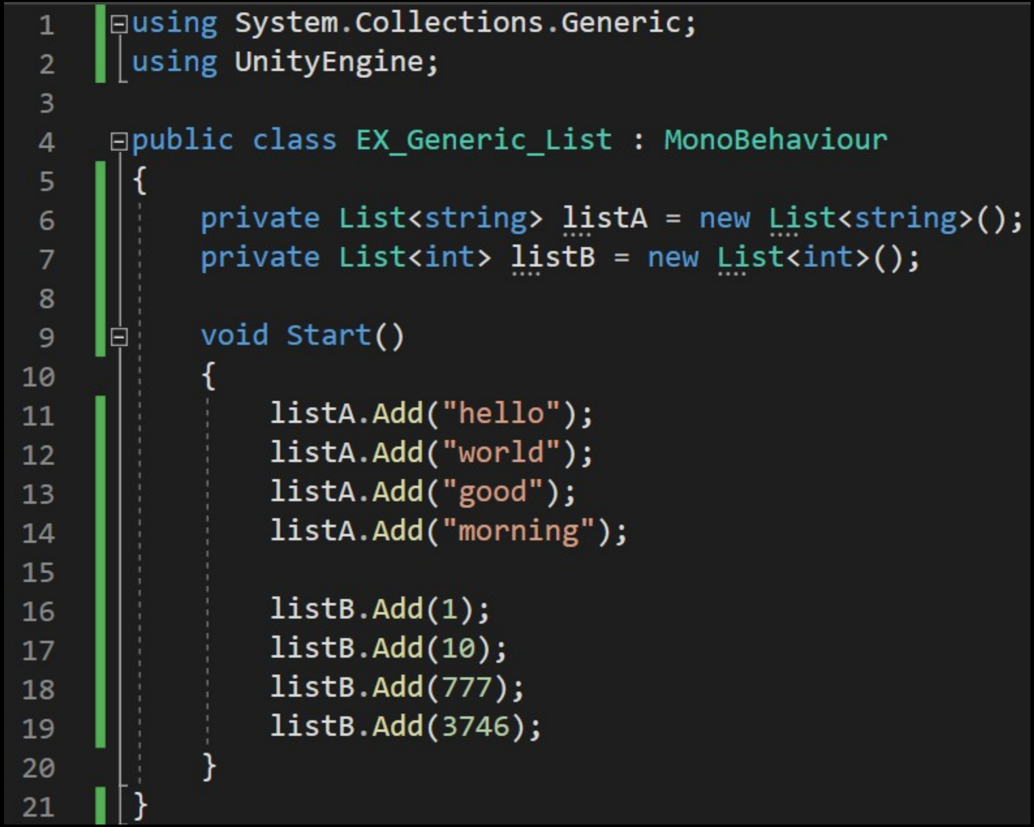
Type Parameters

T

The type of elements in the list.

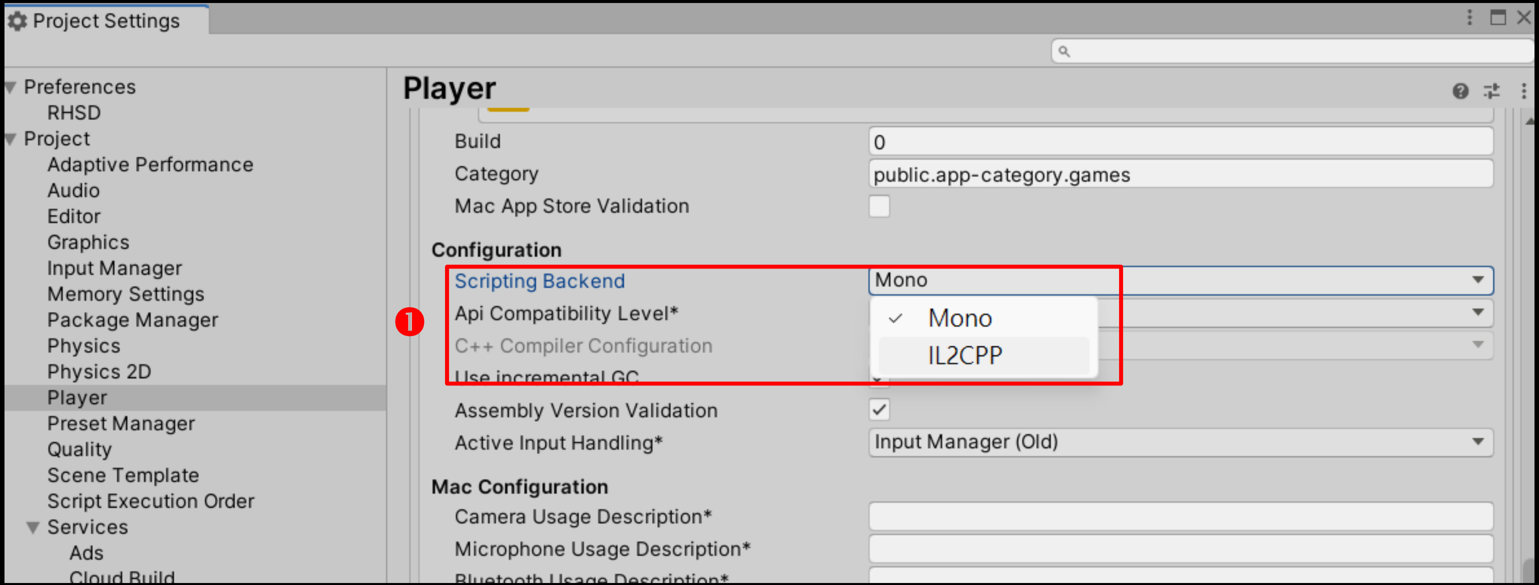
EX: C# List(2)

- The element type of List<T>() is T.
- The generic design clearly specifies the element data type of the list under operation.



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

public class EX_Generic_List : MonoBehaviour
{
    private List<string> listA = new List<string>();
    private List<int> listB = new List<int>();

    void Start()
    {
        listA.Add("hello");
        listA.Add("world");
        listA.Add("good");
        listA.Add("morning");

        listB.Add(1);
        listB.Add(10);
        listB.Add(777);
        listB.Add(3746);
    }
}
```

EX: Enabling IL2CPP in Unity



Project Settings

- Preferences
  - RHSD
- Project
  - Adaptive Performance
  - Audio
  - Editor
  - Graphics
  - Input Manager
  - Memory Settings
  - Package Manager
  - Physics
  - Physics 2D
  - Player
  - Preset Manager
  - Quality
  - Scene Template
  - Script Execution Order
  - Services
    - Ads
    - Cloud Build

Player

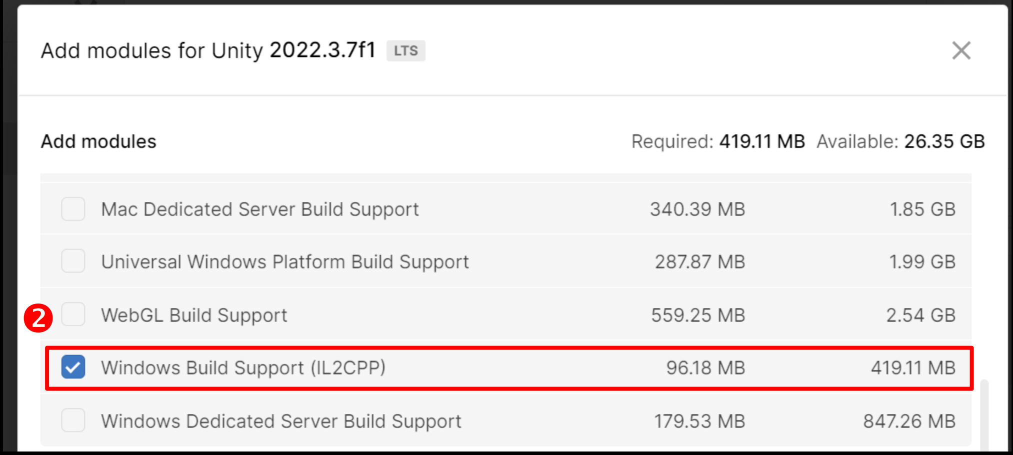
Build	0
Category	public.app-category.games
Mac App Store Validation	☐

Configuration

Scripting Backend	Mono
Api Compatibility Level*	
C++ Compiler Configuration	
Use incremental GC	
Assembly Version Validation	✓
Active Input Handling*	Input Manager (Old)

Mac Configuration

Camera Usage Description*	
Microphone Usage Description*	
Bluetooth Usage Description*	



Add modules for Unity 2022.3.7f1 LTS

Add modules — Required: 419.11 MB Available: 26.35 GB

☐	Mac Dedicated Server Build Support	340.39 MB	1.85 GB
☐	Universal Windows Platform Build Support	287.87 MB	1.99 GB
☐	WebGL Build Support	559.25 MB	2.54 GB
✓	Windows Build Support (IL2CPP)	96.18 MB	419.11 MB
☐	Windows Dedicated Server Build Support	179.53 MB	847.26 MB

- In Project ->Player Setting, set scripting backend to IL2CPP❶.
- In Unity Hub, make sure your Unity has Windows Build Support (IL2CPP) module❷.

- Advanced Class Design
  - OOP
  - UML
  - Generics
  - Reflection
  - Attribute

互動程式設計III

Interactive Programming Design Integration

Introduction to Reflection

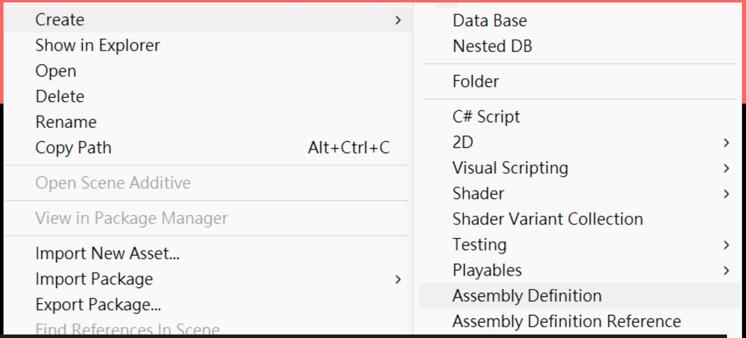
- Definition
  - A feature in C# that allows the inspection and manipulation of object types and their members (methods, properties, fields) at runtime.
  - Provides metadata about assemblies, modules, and types.
- Using reflection for dynamic programming
  - Dynamic Type Inspection:
    - Determine the type of an object at runtime.
    - Access type information without knowing it at compile time.
  - Creating Instances Dynamically:
    - Instantiate objects of a type dynamically.
  - Method Invocation:
    - Invoke methods dynamically based on the method name or signature.
  - Accessing Fields and Properties:
  - Read or modify fields and properties dynamically.
- Reflection plays the key role for all the game editors!

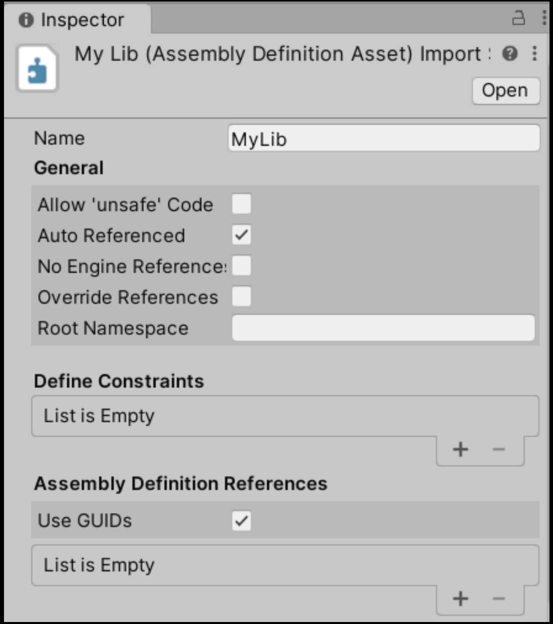
Performance considerations

- Overhead:
  - Reflection operations are slower than direct code execution.
  - Reflection involves runtime type checking and metadata inspection, adding overhead.
- Best Practices:
  - Minimize the use of reflection in performance-critical paths.
  - Cache reflection results when repeatedly accessing the same metadata.
  - Use reflection only when necessary, such as for dynamic plugins or configuration-driven code.

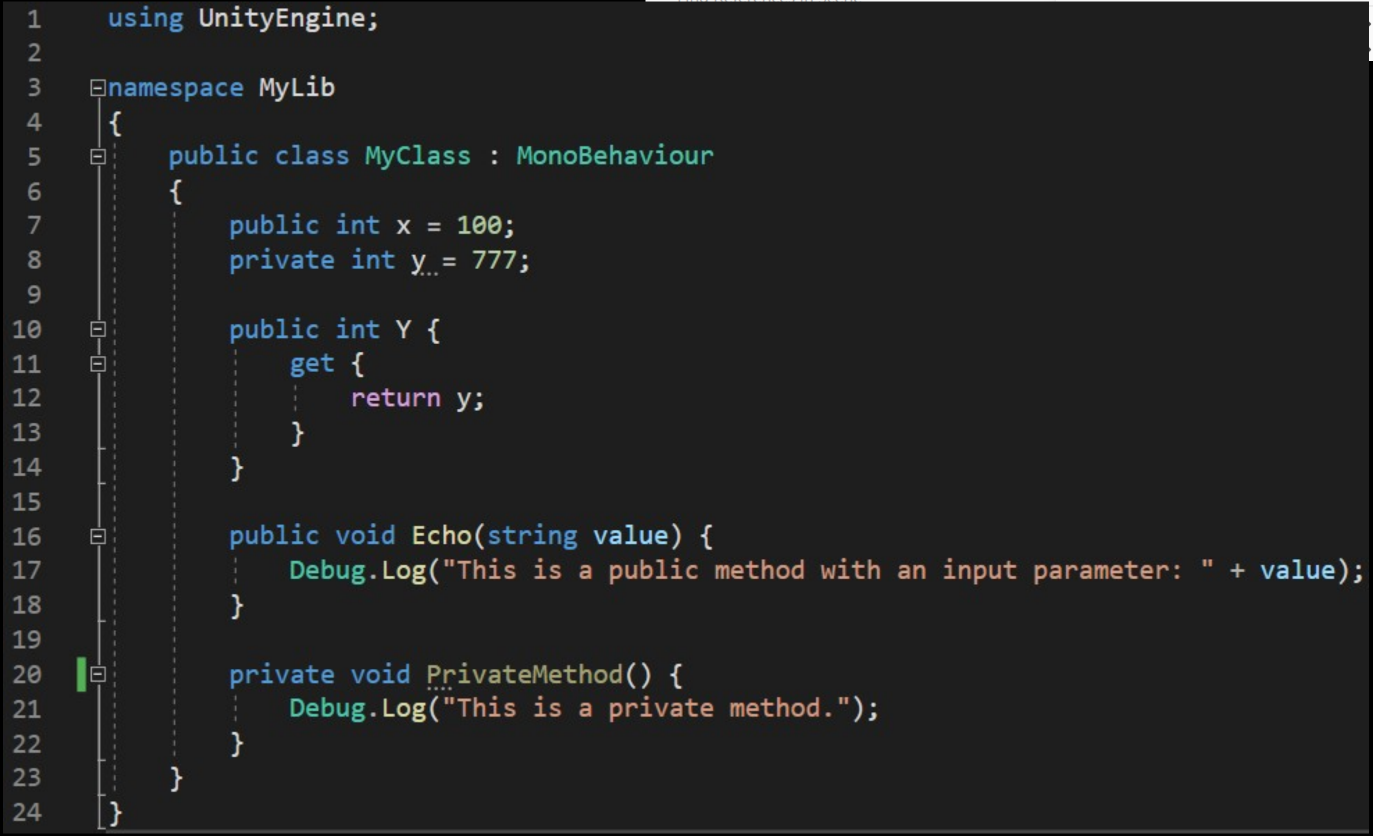
EX: Create a DLL in Unity(1)

- Create a dedicated folder and put a new class called MyClass.cs in it.
- Create a assembly definition file called MyLib.asmdef in the same folder.
  - Set the name as MyLib
  - Check auto-referenced





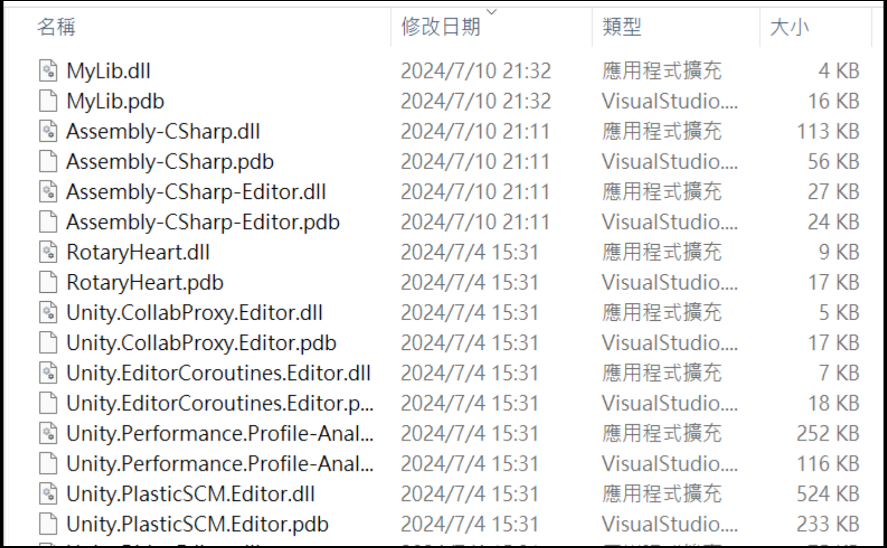
Name	MyLib
General	
Allow 'unsafe' Code	☐
Auto Referenced	✓
No Engine Reference:	☐
Override References	☐
Root Namespace	
Define Constraints	
List is Empty	
Assembly Definition References	
Use GUIDs	✓
List is Empty	



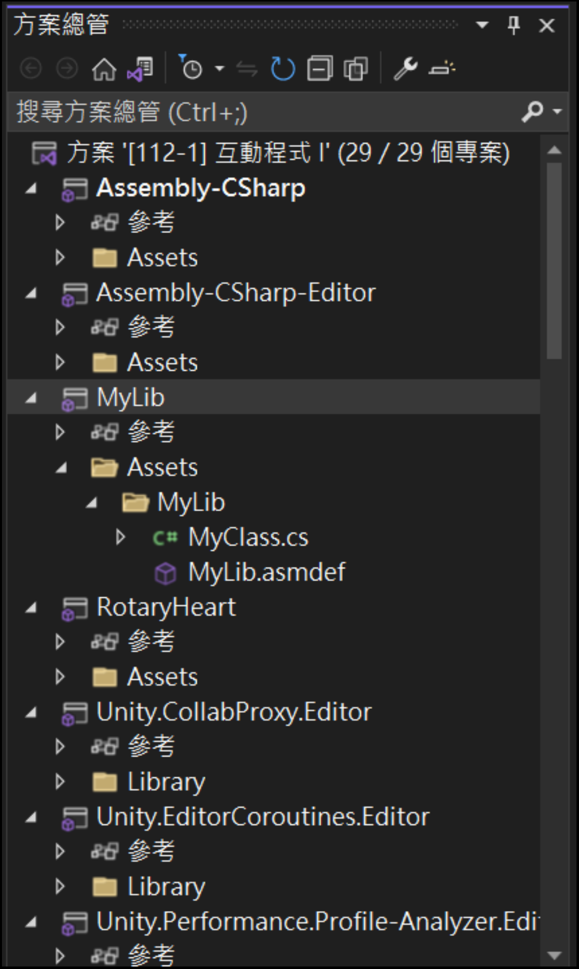
```
using UnityEngine;

namespace MyLib
{
    public class MyClass : MonoBehaviour
    {
        public int x = 100;
        private int y = 777;

        public int Y {
            get {
                return y;
            }
        }

        public void Echo(string value) {
            Debug.Log("This is a public method with an input parameter: " + value);
        }

        private void PrivateMethod() {
            Debug.Log("This is a private method.");
        }
    }
}
```

EX: Create a DLL in Unity(2)

- You can find the new assembly configuration in Visual Studio
  - Right click on MyLib and select build to generate DLL
  - You can find your DLL in:
    <Project>/Library/ScriptAssemblies
  - Copy the MyLib.dll to
    <Project>/Assets/StreamingAssets/DLLs/
  - Then delete the whole /MyLib folder
  - Now you made a DLL for further experiments.



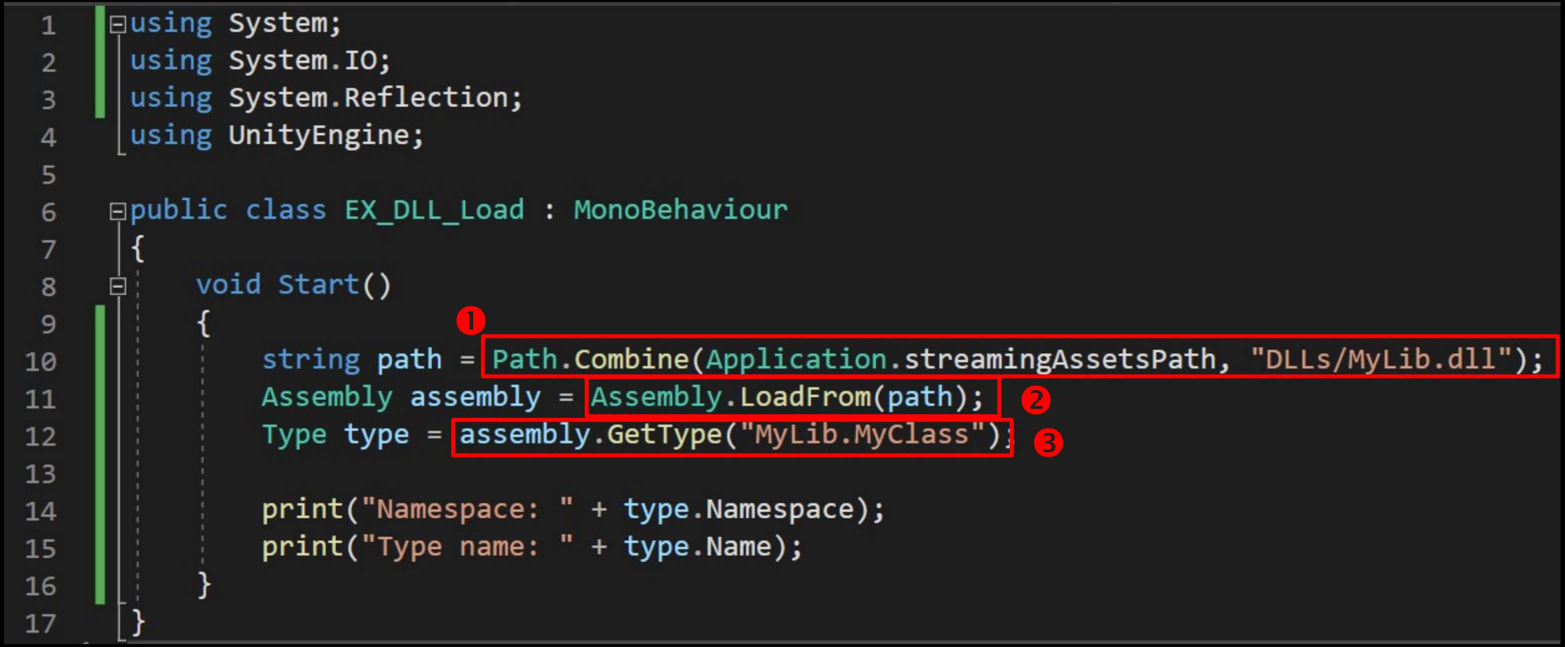
名稱	修改日期	類型	大小
MyLib.dll	2024/7/10 21:32	應用程式擴充	4 KB
MyLib.pdb	2024/7/10 21:32	VisualStudio....	16 KB
Assembly-CSharp.dll	2024/7/10 21:11	應用程式擴充	113 KB
Assembly-CSharp.pdb	2024/7/10 21:11	VisualStudio....	56 KB
Assembly-CSharp-Editor.dll	2024/7/10 21:11	應用程式擴充	27 KB
Assembly-CSharp-Editor.pdb	2024/7/10 21:11	VisualStudio....	24 KB
RotaryHeart.dll	2024/7/4 15:31	應用程式擴充	9 KB
RotaryHeart.pdb	2024/7/4 15:31	VisualStudio....	17 KB
Unity.CollabProxy.Editor.dll	2024/7/4 15:31	應用程式擴充	5 KB
Unity.CollabProxy.Editor.pdb	2024/7/4 15:31	VisualStudio....	17 KB
Unity.EditorCoroutines.Editor.dll	2024/7/4 15:31	應用程式擴充	7 KB
Unity.EditorCoroutines.Editor.p...	2024/7/4 15:31	VisualStudio....	18 KB
Unity.Performance.Profile-Anal...	2024/7/4 15:31	應用程式擴充	252 KB
Unity.Performance.Profile-Anal...	2024/7/4 15:31	VisualStudio....	116 KB
Unity.PlasticSCM.Editor.dll	2024/7/4 15:31	應用程式擴充	524 KB
Unity.PlasticSCM.Editor.pdb	2024/7/4 15:31	VisualStudio....	233 KB



```
方案 '[112-1] 互動程式 I' (29 / 29 個專案)
  Assembly-CSharp
    參考
    Assets
  Assembly-CSharp-Editor
    參考
    Assets
  MyLib
    參考
    Assets
      MyLib
        MyClass.cs
        MyLib.asmdef
  RotaryHeart
    參考
    Assets
  Unity.CollabProxy.Editor
    參考
    Library
  Unity.EditorCoroutines.Editor
    參考
    Library
  Unity.Performance.Profile-Analyzer.Edi
    參考
```

EX: Load a DLL in Unity

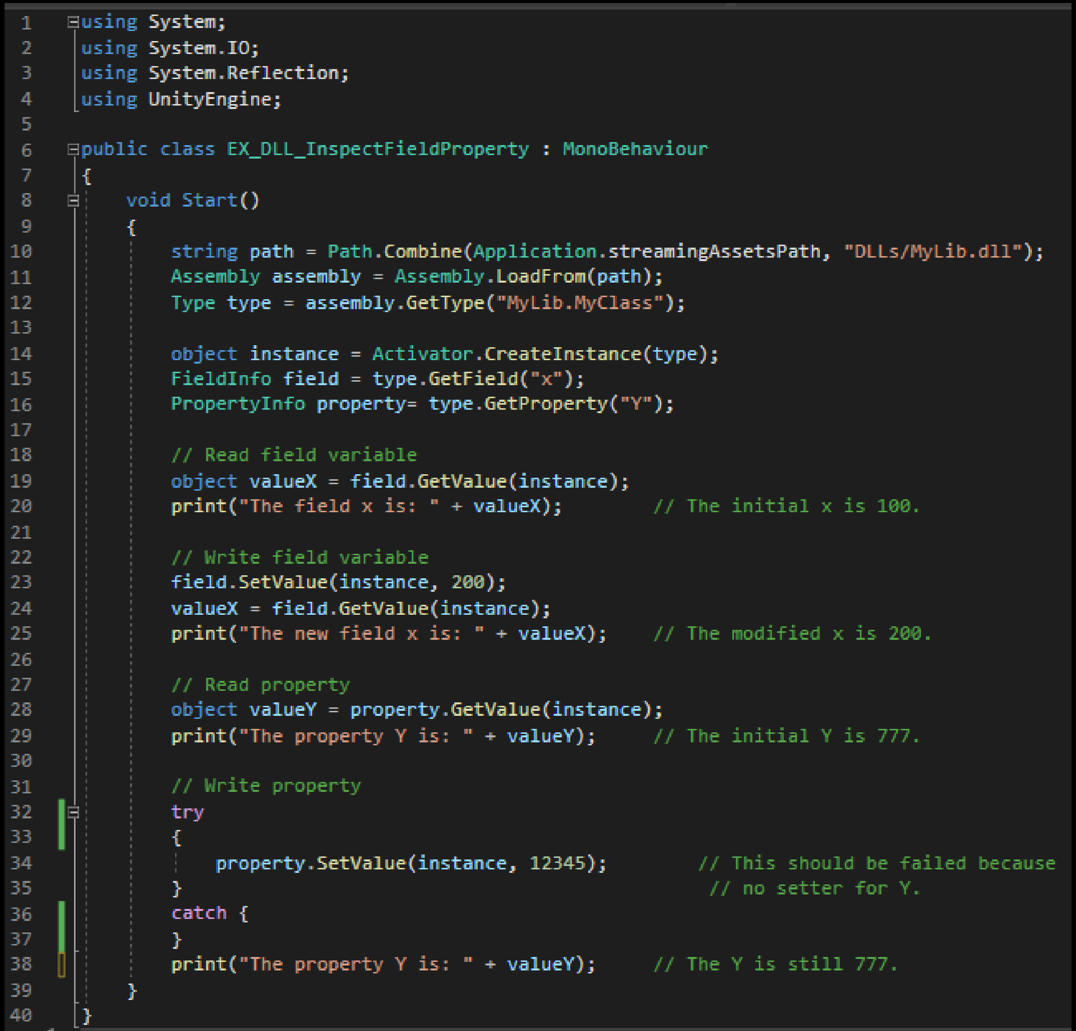
- Use Path.Combine to concatenate folder path and DLL filename❶.
- Use Assembly.LoadFrom() to load the DLL❷
- Use Assembly.GetType to get the type of MyClass❸.



```
using System;
using System.IO;
using System.Reflection;
using UnityEngine;

public class EX_DLL_Load : MonoBehaviour
{
    void Start()
    {                                   ❶
        string path = Path.Combine(Application.streamingAssetsPath, "DLLs/MyLib.dll");
        Assembly assembly = Assembly.LoadFrom(path);  ❷
        Type type = assembly.GetType("MyLib.MyClass");  ❸

        print("Namespace: " + type.Namespace);
        print("Type name: " + type.Name);
    }
}
```

EX: Type Inspection - Field/Property

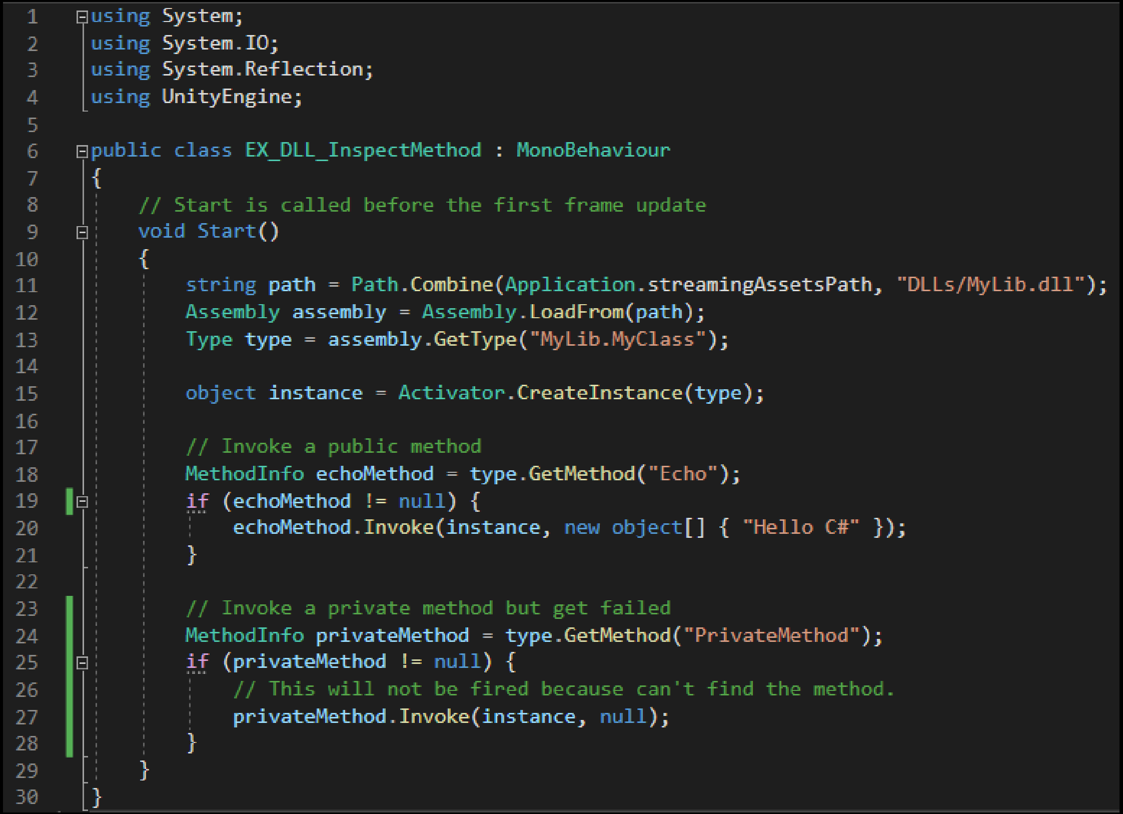
- Get the field variable info(metadata) first, then we can read/write the public field variable by calling FieldInfo.GetValue/SetValue.
- How C# reflection read/write a field?
  - Field Offsets: The CLR calculates the offsets of each field within the object based on the type's layout. These offsets are used to locate the fields relative to the start of the object's memory block.
  - Direct Access for Instance Fields: For instance fields, the runtime computes the address of the field by adding the field's offset to the address of the object. This allows direct access to the field's memory location.
  - Direct Access for Static Fields: Static fields are not stored within the instance but rather in a separate memory area reserved for the type. The CLR maintains a table of static fields, and each field's address can be directly computed from this table.
- The property works in similar pattern but acting upon get/set accessor methods.



```csharp
using System;
using System.IO;
using System.Reflection;
using UnityEngine;

public class EX_DLL_InspectFieldProperty : MonoBehaviour
{
    void Start()
    {
        string path = Path.Combine(Application.streamingAssetsPath, "DLLs/MyLib.dll");
        Assembly assembly = Assembly.LoadFrom(path);
        Type type = assembly.GetType("MyLib.MyClass");

        object instance = Activator.CreateInstance(type);
        FieldInfo field = type.GetField("x");
        PropertyInfo property= type.GetProperty("Y");

        // Read field variable
        object valueX = field.GetValue(instance);
        print("The field x is: " + valueX);         // The initial x is 100.

        // Write field variable
        field.SetValue(instance, 200);
        valueX = field.GetValue(instance);
        print("The new field x is: " + valueX);     // The modified x is 200.

        // Read property
        object valueY = property.GetValue(instance);
        print("The property Y is: " + valueY);      // The initial Y is 777.

        // Write property
        try
        {
            property.SetValue(instance, 12345);         // This should be failed because
        }                                                // no setter for Y.
        catch {
        }
        print("The property Y is: " + valueY);      // The Y is still 777.
    }
}
```

EX: Type Inspection - Method

- Get the method info(metadata) first, then we can invoke the public method by calling MethodInfo.Invoke()
  - The parameters should be in object array format



```csharp
using System;
using System.IO;
using System.Reflection;
using UnityEngine;

public class EX_DLL_InspectMethod : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        string path = Path.Combine(Application.streamingAssetsPath, "DLLs/MyLib.dll");
        Assembly assembly = Assembly.LoadFrom(path);
        Type type = assembly.GetType("MyLib.MyClass");

        object instance = Activator.CreateInstance(type);

        // Invoke a public method
        MethodInfo echoMethod = type.GetMethod("Echo");
        if (echoMethod != null) {
            echoMethod.Invoke(instance, new object[] { "Hello C#" });
        }

        // Invoke a private method but get failed
        MethodInfo privateMethod = type.GetMethod("PrivateMethod");
        if (privateMethod != null) {
            // This will not be fired because can't find the method.
            privateMethod.Invoke(instance, null);
        }
    }
}
```

EX: Is It Really Impossible to Read/Write Private Members?

- Actually you can read/write private field/property and also invoke private method. You are encouraged to find the solution by yourself.

- Advanced Class Design
  - OOP
  - UML
  - Generics
  - Reflection
  - Attribute

互動程式設計III

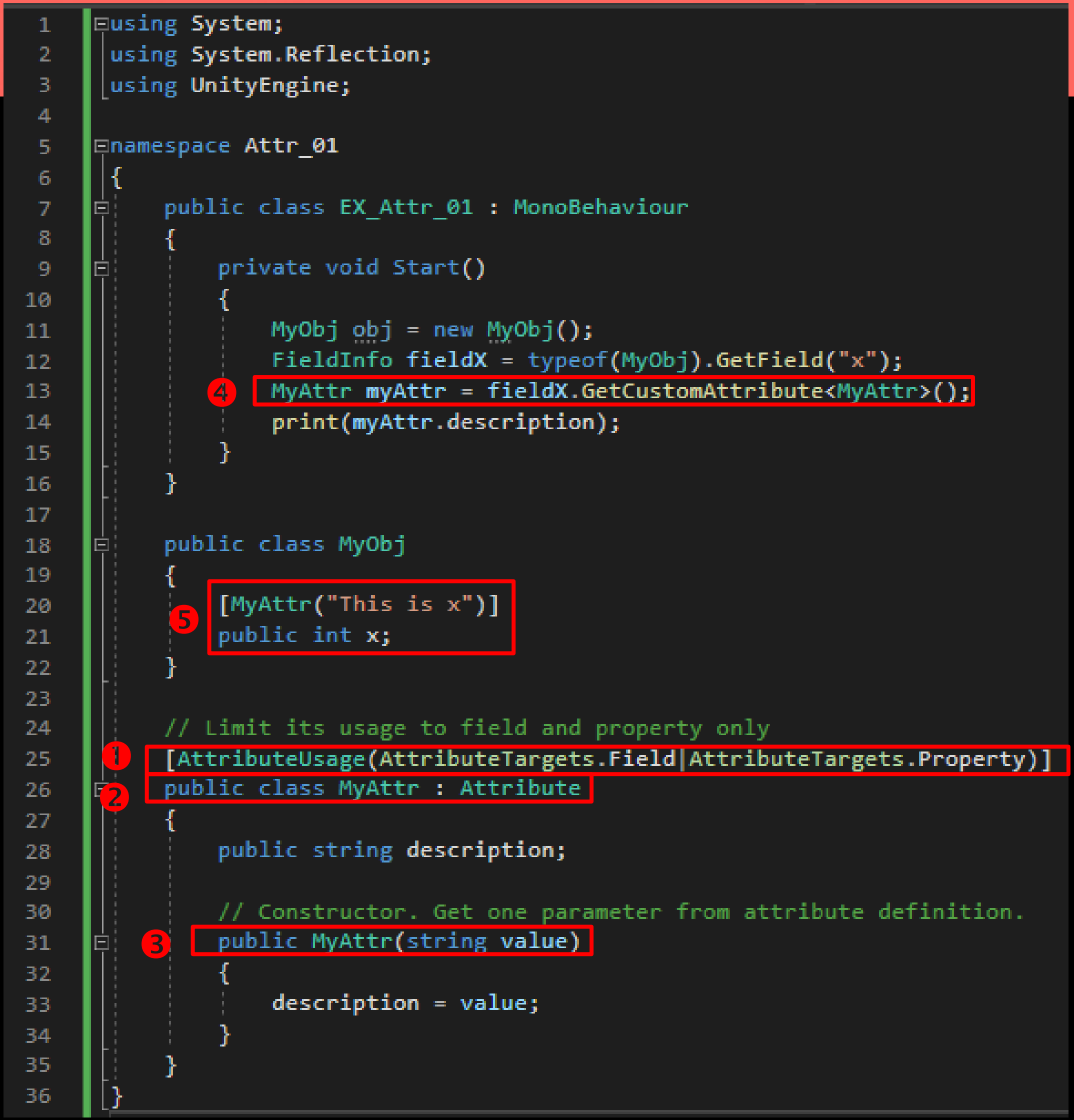
Interactive Programming Design Integration

Introduction to Attributes in C#

- Definition:
  - Attributes in C# provide a powerful method to add metadata to code elements (classes, methods, properties, etc.). They are used to store declarative information and can be accessed at runtime using reflection.
- Usage:
  - Commonly used for configuration, validation, documentation, and more.

EX: Defining an Attribute

- The custom attribute class must inherit Attribute class❶.
- Use AttributeUsage to limit the usage of custom attribute❷.
- You can pass some parameters to the constructor but it only accepts simple types❸.
- Use GetCustomAttribute<T>() to get the attached attribute❹.
- This is what you frequently seen in Unity C# codes❺.



```csharp
using System;
using System.Reflection;
using UnityEngine;

namespace Attr_01
{
    public class EX_Attr_01 : MonoBehaviour
    {
        private void Start()
        {
            MyObj obj = new MyObj();
            FieldInfo fieldX = typeof(MyObj).GetField("x");
       ❹    MyAttr myAttr = fieldX.GetCustomAttribute<MyAttr>();
            print(myAttr.description);
        }
    }

    public class MyObj
    {
        [MyAttr("This is x")]
   ❺   public int x;
    }

    // Limit its usage to field and property only
 ❶ [AttributeUsage(AttributeTargets.Field|AttributeTargets.Property)]
 ❷ public class MyAttr : Attribute
    {
        public string description;

        // Constructor. Get one parameter from attribute definition.
    ❸  public MyAttr(string value)
        {
            description = value;
        }
    }
}
```

互動程式設計 *III*

Interactive Programming Design Integration

國立臺北科技大學 互動設計系 傅子恒老師

互動程式設計III

Interactive Programming Design Integration

Q&A