
互動程式設計III
Interactive Programming Design Integration

Introduction to Multitasking

- Introduction to Multitasking in Unity
  - Definition: Multitasking in Unity refers to the ability to perform multiple tasks concurrently, improving the efficiency and responsiveness of your game or application.
  - Key Techniques:
    - Coroutines: Execute tasks over multiple frames without blocking the main thread.
    - Async/Await: Perform asynchronous operations such as I/O tasks without freezing the game.
    - Threads: Use separate threads for CPU-intensive tasks to keep the main thread responsive.
- Learning Objectives
  - Understand the basics of multitasking in Unity.
  - Learn how to use Coroutines for frame-based multitasking.
  - Implement asynchronous operations using Async/Await.
  - Explore threading for handling CPU-bound tasks.
  - Identify scenarios where each multitasking technique is most appropriate.
- Why This Chapter is Important
  - Performance: Enhances game performance by ensuring tasks do not block the main game loop.
  - Responsiveness: Improves user experience by keeping the game responsive even during complex operations.
  - Efficiency: Allows for better utilization of system resources by distributing tasks appropriately.
  - Scalability: Prepares your game or application to handle more complex and resource-intensive features seamlessly.
  - Practical Skills: Equips you with essential techniques for modern game development, ensuring your projects meet industry standards.

互動程式設計III

Interactive Programming Design Integration

- Multitasking
  - Coroutine
  - Async/Await
  - Process/Thread/Task
  - File I/O
  - HTTP
  - Appendix

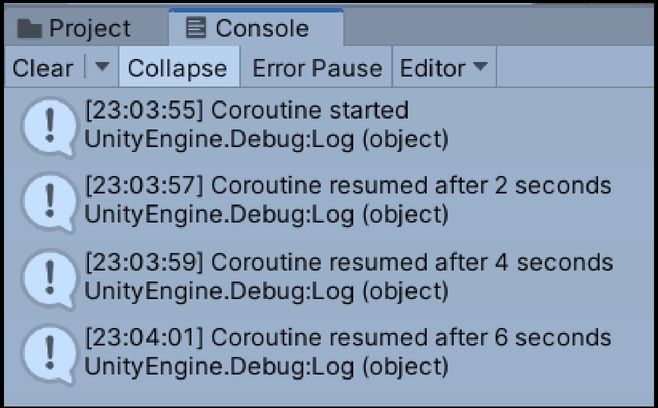
What is Coroutine

- Definition:
  - A coroutine is a lightweight form of virtual multitasking that simulates multithreading on a single thread. A coroutine can pause its execution and return control to the Unity game loop to perform a specified asynchronous task, then resume from the pause point when that task first yields or completes.
- Advantages:
  - Non-blocking: By using yield return, coroutines hand control back to the Unity game loop rather than freezing it.
  - Simple syntax: Compared to other multitasking approaches, coroutines are more straightforward to implement and read.
  - Seamless integration with Unity: They can be used for various Unity tasks—such as delayed events, animation sequences, and other asynchronous operations.
  - Minimal overhead: Unity coroutines are implemented as IEnumerator objects and are scheduled each frame by the engine. A single coroutine only occupies a small state-machine structure, so under normal use it is not a major performance bottleneck.
- Considerations:
  - Lifecycle management: Coroutines bound to a MonoBehaviour will stop unexpectedly if the associated GameObject is destroyed.
  - Error handling: Exceptions thrown inside a coroutine can be harder to debug, so you should explicitly catch and handle errors within coroutines.
  - Scope control: Coroutines still run on the main thread—they simply spread their execution across multiple frames, rather than running in parallel threads.

EX: Create a Coroutine

- To create a coroutine:
  - The return type must be IEnumerator.
  - At least one yield return should be placed in coroutine body.
  - Use StartCoroutine to start a coroutine.

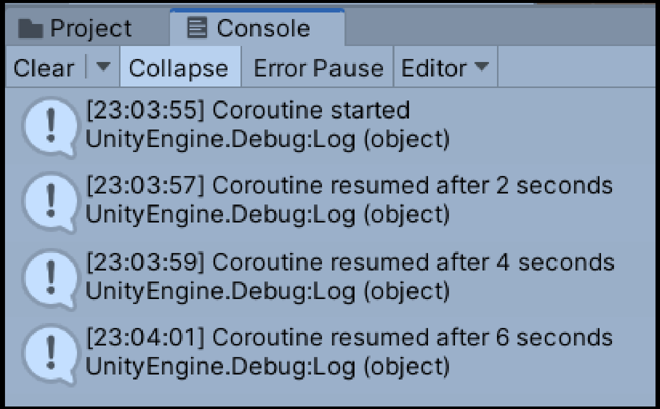
```csharp
using System.Collections;
using UnityEngine;

public class EX_Coroutine_01 : MonoBehaviour
{
    void Start()
    {
        StartCoroutine(MyCoroutine());
    }

    private IEnumerator MyCoroutine()
    {
        Debug.Log("Coroutine started");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 2 seconds");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 4 seconds");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 6 seconds");
        yield return null;
    }
}
```



EX: Create a Coroutine

- To create a coroutine:
  - The return type must be IEnumerator.
  - At least one yield return should be placed in coroutine body.
  - Use StartCoroutine to start a coroutine.

```csharp
using System.Collections;
using UnityEngine;

public class EX_Coroutine_01 : MonoBehaviour
{
    void Start()
    {
        StartCoroutine(MyCoroutine());
    }

    private IEnumerator MyCoroutine()
    {
        Debug.Log("Coroutine started");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 2 seconds");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 4 seconds");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 6 seconds");
        yield return null;
    }
}
```



EX: What Coroutine Does...

- The Unity Coroutine just creates a state machine for us.
  - It uses a simple switch to control state processes.
- The Coroutine class implements IEnumerator ❶ interfaces.

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

public class EX_Coroutine_02 : MonoBehaviour
{
    private EX_Coroutine_02_Coroutine coroutine =
        new EX_Coroutine_02_Coroutine();

    void Update()
    {
        if (coroutine.MoveNext())
        {
            object nextObj = coroutine.Current;
            print(nextObj);
        }
    }
}

```

```

public class EX_Coroutine_02_Coroutine : IEnumerator<object>
{
    private int state;

    public object Current {
        get {
            switch (state)
            {
                case 1:
                    return 1;
                case 2:
                    return 2;
                case 3:
                    return 3;
                default:
                    throw new InvalidOperationException();
            }
        }
    }

    public void Dispose() { }

    public bool MoveNext() {
        state++;
        return state <= 3;
    }

    public void Reset() {
        state = 0;
    }
}
```

❷

EX: Yield Instructions

- In C#, the yield keyword is not exactly syntactic sugar, but it does simplify the process of creating iterators. Here’ s a detailed explanation of what yield does and how it compares to traditional iterator implementations:
  - Purpose: The yield keyword is used to return elements one at a time from an enumerator without the need to explicitly manage the state of the iteration.
  - Keywords:
    - yield return: Used to return each element one by one.
    - yield break: Used to end the iteration.
- When you use yield return in a method, the C# compiler generates a state machine behind the scenes. This state machine keeps track of the current position of the iteration and resumes execution from the point where yield return was last called.
  - This state machine is implemented as a class that implements the IEnumerable or IEnumerator interface.
- Key Differences
  - Complexity: Using yield greatly reduces the complexity of writing iterators. The manual implementation requires maintaining state and ensuring correct implementation of IEnumerable and IEnumerator interfaces.
  - Readability: Code using yield is more concise and easier to understand.
  - Maintainability: Less boilerplate code means fewer opportunities for errors and easier maintenance.
- Frequently used yield return instructions:
  - yield return null; (wait until next frame)
  - yield return new WaitForSeconds(seconds); (wait for a scaled duration)
  - yield return new WaitForEndOfFrame(); (wait until the end of the frame)
  - yield return new WaitForSecondsRealtime(seconds); (wait for a specified unscaled duration)

EX: Chaining Coroutines

-

```
using System.Collections;
using UnityEngine;

public class EX_Coroutine_03 : MonoBehaviour
{
    void Start()
    {
        StartCoroutine(FirstCoroutine());
    }

    private IEnumerator FirstCoroutine()
    {
        Debug.Log("First Coroutine");
        yield return new WaitForSeconds(1);
        yield return StartCoroutine(SecondCoroutine());
    }

    private IEnumerator SecondCoroutine()
    {
        Debug.Log("Second Coroutine");
        yield return null;
    }
}
```

EX: Stopping Coroutines

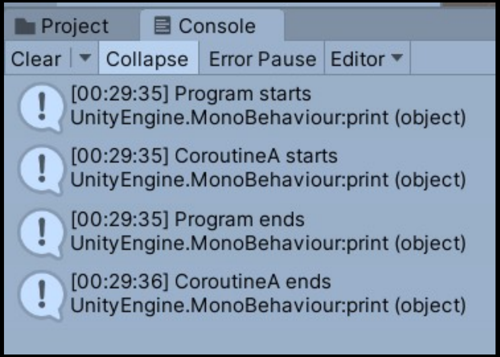
- Save the reference❶ of the coroutine enumerator in order to stop it.

```
using System.Collections;
using UnityEngine;

public class EX_Coroutine_04 : MonoBehaviour
{
    private Coroutine myCoroutine;

    private void Start()
    {
        myCoroutine = StartCoroutine(MyCoroutine());
    }

    private void Update()
    {
        if (Input.GetKeyDown("w"))
        {
            StopCoroutine(myCoroutine);
        }
    }

```

```
    private IEnumerator MyCoroutine()
    {
        Debug.Log("Coroutine started");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 2 seconds");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 4 seconds");
        yield return new WaitForSeconds(2);
        Debug.Log("Coroutine resumed after 6 seconds");
        yield return null;
    }
}
```

EX: Problem of Execution Order

- Can you explain the console logs?




```csharp
using System.Collections;
using UnityEngine;

public class EX_Coroutine_05 : MonoBehaviour
{
    void Start()
    {
        print("Program starts");

        StartCoroutine(CoroutineA());

        print("Program ends");
    }

    IEnumerator CoroutineA()
    {
        print("CoroutineA starts");

        yield return new WaitForSeconds(1);

        print("CoroutineA ends");
    }
}
```

What we expected

Program starts
CoroutineA starts
<Wait for 2 seconds>
CoroutineA ends
Program ends

What we get

Program starts
CoroutineA starts
Program ends
<Wait for 2 seconds>
CoroutineA ends

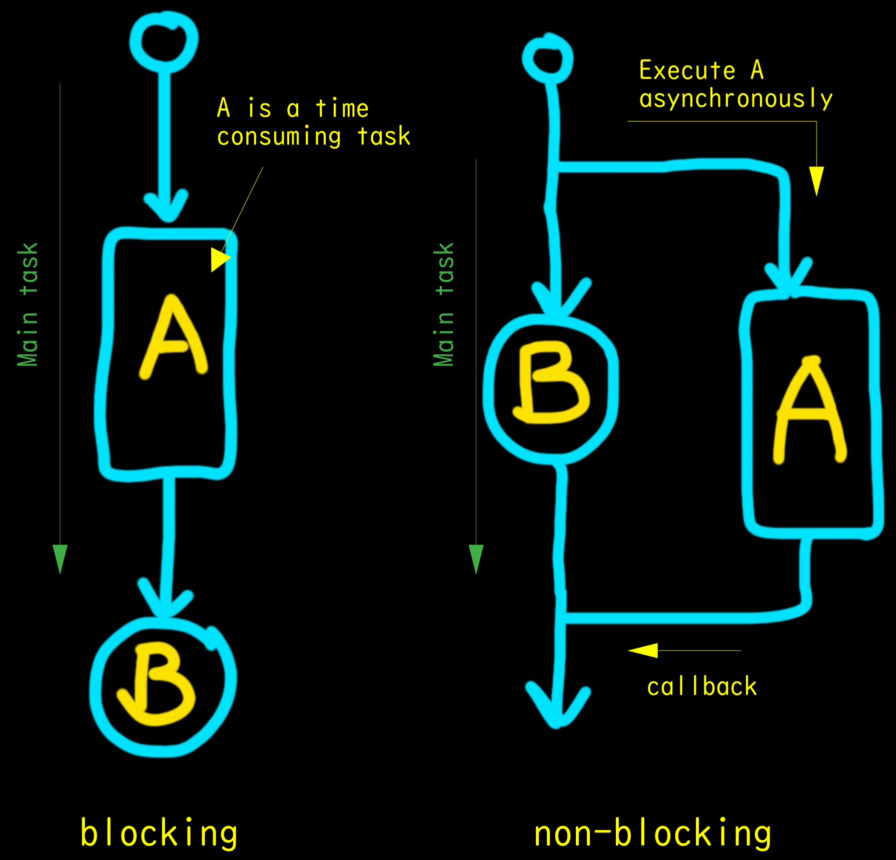
互動程式設計III

Interactive Programming Design Integration

- Multitasking
  - Coroutine
  - Async/Await
  - Process/Thread/Task
  - File I/O
  - HTTP
  - Appendix

What is Async/Await

- \- Introduction to asynchronous programming
  - Definition: Asynchronous programming allows for non-blocking operations, enabling tasks to run in the background while keeping the main thread responsive.
  - Benefits: Improved performance, Enhanced responsiveness, Better resource utilization
- \- Task-based asynchronous patterns
  - Task: Represents an asynchronous operation. Will be introduced later.
  - Async/Await Keywords:
    - async: Marks a method as asynchronous and turn the method into a Task.
    - await: Pauses the execution of current Task and wait for the new Task to be completed.
- \- You can use async with or without await.



Task-based Asynchronous Pattern (TAP)

- C# primarily uses the Task-based Asynchronous Pattern (TAP) to implement asynchronous operations. The Task and Task<T> types represent asynchronous operations. When you use the async keyword to define a method, and the await keyword to await a task, you're leveraging TAP.
- How async and await Work
  - Async Method:
    - The async keyword transforms the method into a state machine that can handle the asynchronous Task operation.
    - When you mark a method with the async keyword, it allows the method to use the await keyword within its body.
  - Await Keyword:
    - The await keyword is used to pause the execution of the Task until the awaited Task completes.
    - When await is encountered, the control is returned to the calling method, and the rest of the method is scheduled to continue after the awaited task completes.
  - Task Handling:
    - The Task type represents an asynchronous operation. The task can be completed, running, waiting to run, or canceled.
    - The Task is generally run on a thread pool, which is managed by the .NET runtime.

How await Works

- Suspending the Current Task:
  - When a Task encounters await, it suspends its execution and waits for the awaited Task to complete.
  - After suspension, control returns to the calling method, allowing other operations to continue.
- Scheduling Asynchronous Operations:
  - Asynchronous operations (like I/O operations) are scheduled and run, usually without consuming CPU time.
  - Code after await continues execution only after the awaited Task completes.
- Non-Blocking:
  - The await keyword allows the method to wait for a Task to complete without blocking the thread. This lets the thread handle other work.
- Summary
  - When using async/await, Tasks can call and wait for each other.
  - When a Task encounters await, it suspends and returns control to the calling method, allowing the calling method to continue other work.
  - A single thread does not execute multiple Tasks simultaneously but schedules and executes them in sequence.

Asynchronous Programming in C# (Under the Hood)

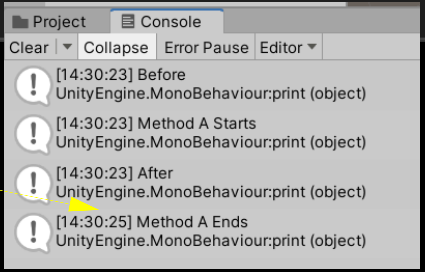
- State Machine:
  - The C# compiler translates async methods into state machines. This state machine keeps track of the method's state and knows how to resume the method once the awaited task completes.
  - The state machine allows the method to be paused and resumed without blocking the main thread.
- Thread Pool:
  - The .NET runtime uses a thread pool to manage and execute tasks. This pool is a collection of worker threads that can be reused to perform multiple tasks over their lifetime.
  - When an async method awaits a task, the task is often scheduled to run on a thread from the thread pool.
- Continuation:
  - When the awaited task completes, the state machine is resumed, and the continuation of the async method is executed.
  - The continuation may run on the same thread that completed the task or be scheduled to run on a different thread, depending on the synchronization context.

EX: Use Async/Await to replace Coroutine

- Use Async/Await to replace Coroutine
  - By marking MethodA as async, we indicates that MethodA contains asynchronous operations❷.
    - But we call MethodA like an normal method call (without await)❶. This means we are not going to "wait" for the whole methodA to be finished. But since MethodA is a async method, it will run asynchronously.
    - When an await method is called inside methodA❸, the program will fork to another new Task to execute the await call.
    - When await is encountered❸, the control returns to the calling method (Start in this case), and the remaining code in Start continues executing❹.
    - After the await is complete, the execution of MethodA resumes from where it was paused❺.

```
using System.Threading.Tasks;
using UnityEngine;

public class EX_AsyncAwait_01 : MonoBehaviour
{
    void Start()
    {
        print("Before");

     ❶  MethodA();

     ❹  print("After");
    }

 ❷  async Task MethodA()
    {
        print("Method A Starts");

     ❸  await Task.Delay(2000); // Wait for 2 seconds.

     ❺  print("Method A Ends");
    }
}
```

<Wait for 2 seconds here>



EX: Use Async/Await to replace Coroutine

- This time let's we call MethodA with await.
  - In order to use await in Start(), we have to set Start() as async❶.
  - We call MethodA with await❷, this will pause the Start() and wait for await to be completed.
- Inside MethodA,
  - The await Task.Delay forks the

```csharp
using System.Threading.Tasks;
using UnityEngine;

public class EX_AsyncAwait_02 : MonoBehaviour
{
    async void Start()
    {
        print("Before");

        await MethodA();

        print("After");
    }

    async Task MethodA()
    {
        print("Method A Starts");

        await Task.Delay(2000); // Wait for 2 seconds.

        print("Method A Ends");
    }
}
```

```
[16:08:32] Before
UnityEngine.MonoBehaviour:print (object)
[16:08:32] Method A Starts
UnityEngine.MonoBehaviour:print (object)
[16:08:34] Method A Ends
UnityEngine.MonoBehaviour:print (object)
[16:08:34] After
UnityEngine.MonoBehaviour:print (object)
```

<Wait for 2 seconds here>

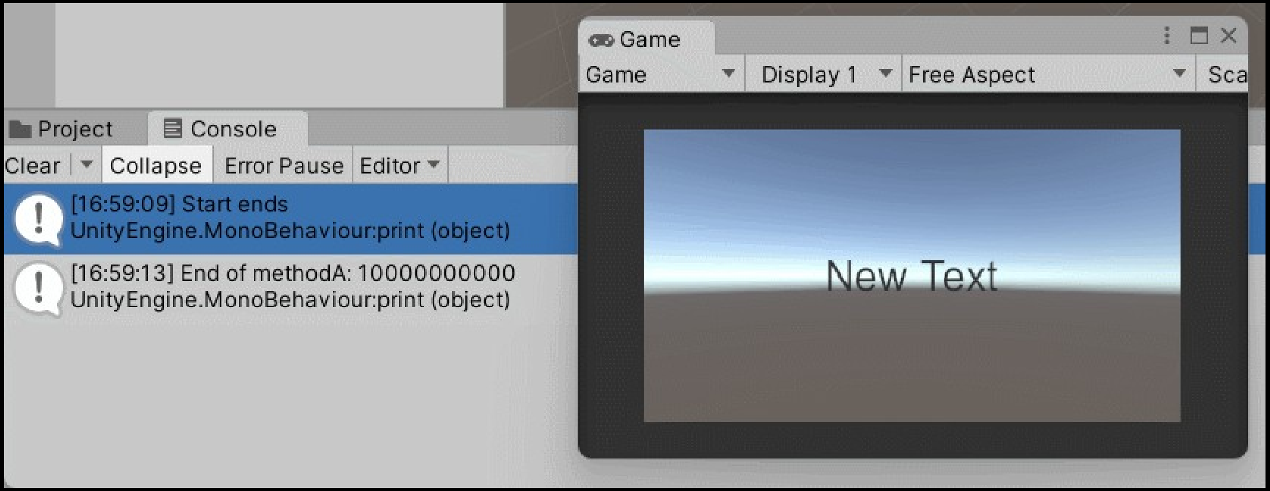
EX: Overloading in Start(1)

- Now let's intentionally overload the Start().
  - Put a super large for loop in MethodA❶, and invoke methodA in Start().
  - The "start ends" will be delayed.
  - The Update is blocked since the only Unity main thread is overloaded by Start().
- When this Unity procedure is halt, we are still able to open a Chrome or edit a Word file. This is because of the Multi-process behaviour.


```
using UnityEngine;
using UnityEngine.UI;

public class EX_AsyncAwait_04 : MonoBehaviour
{
    public Text text;

    void Start() {
        MethodA();
        print("Start ends");
    }

    void Update() {
        if (Input.GetKey("w")) {
            text.text = "W is pressed.";
        }
        else {
            text.text = "";
        }
    }

    void MethodA() {
        long a = 0;
❶       for (long i = 0; i < 30000000000; i++) {
            a++;
        }
        print("end of MethodA: " + a);
    }
}
```

EX: Overloading in Start(2)

- We can create a new thread for methodA in order not to block the Unity main thread.
  - Use Task() constructor to wrap the method into a task❶.
  - Use Task.Start() to create a new thread and run the task❶.
- Run the project and you'll see that the MethodA will not block the execution of Start() and Update() anymore.




```
using System.Threading.Tasks;
using UnityEngine;
using UnityEngine.UI;

public class EX_AsyncAwait_04 : MonoBehaviour
{
    public Text text;

    void Start()
    {
        Task task = new Task(MethodA);
        task.Start();
        print("Start ends");
    }

    void Update()
    {
        if (Input.GetKey("w"))
        {
            text.text = "W is pressed.";
        }
        else
        {
            text.text = "";
        }
    }

    void MethodA()
    {
        long a = 0;
        for (long i = 0; i < 5000000000; i++)
        {
            a++;
        }
        print("End of methodA: " + a);
    }
}
```

❶

EX: Async to Yield from Overloading

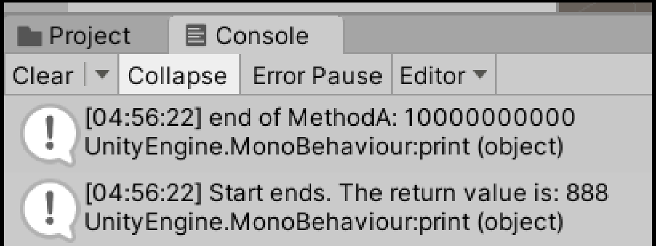
- Let's put all the heavy loading codes in a lambda method, and send it to a Task.Run(…)❶. This enables the codes to be invoke as an await task.
- Since we use await in MethodA, so it should be modified as an async method with a Task type return❷.
- Task.Run() will get a new thread from C# CLR thread pool and offload the for loop to that thread.


```csharp
using System.Threading.Tasks;
using UnityEngine;
using UnityEngine.UI;

public class EX_AsyncAwait_05 : MonoBehaviour
{
    public Text text;

    void Start() {
        MethodA();
        print("Start ends");
    }

    void Update() {
        if (Input.GetKey("w")) {
            text.text = "W is pressed.";
        }
        else {
            text.text = "";
        }
    }
                                                    ❷
    async Task MethodA() {
        await Task.Run(() => {
            long a = 0;                             ❶
            for (long i = 0; i < 10000000000; i++) {
                a++;
            }
            print("end of MethodA: " + a);
        });
    }
}
```

Question: The "Start ends" message is supposed to be shown after the for loop is finished. How to fix this problem?

EX: Return from an Async Task(2)

- In order to return some value from async Task, we have to make some changes:
  - Set the return type as a generic type T in Task<T>❶.
  - Set the return value in MethodA❷.
  - Get the return value in Start() by making the invocation of MethodA as an await Task❸.
  - Since we placed an await in Start(), we have to make it an async Task with Task type return❹.




```
using System.Threading.Tasks;
using UnityEngine;
using UnityEngine.UI;

public class EX_AsyncAwait_05 : MonoBehaviour
{
    public Text text;

    async Task Start() {
        int x = await MethodA();
        print("Start ends. The return value is: " + x);
    }

    void Update() {
        if (Input.GetKey("w")) {
            text.text = "W is pressed.";
        }
        else {
            text.text = "";
        }
    }

    async Task<int> MethodA() {
        await Task.Run(() => {
            long a = 0;
            for (long i = 0; i < 10000000000; i++) {
                a++;
            }
            print("end of MethodA: " + a);
        });
        return 888;
    }
}
```


- Multitasking
  - Coroutine
  - Async/Await
  - Process/Thread/Task
  - File I/O
  - HTTP
  - Appendix
互動程式設計III
Interactive Programming Design Integration

What is Thread

- Process
  - A process is an instance of a program running in a computer. It has its own memory space and system resources.
    - Isolation: Each process runs independently and is isolated from others.
    - Resource Management: Managed by the operating system; each process has its own memory and system resources.
    - Heavyweight: Creating and managing processes is resource-intensive.
    - Need some IPC mechanism to communicated with other process.
- Thread
  - A Thread is the smallest unit of a process that can be scheduled by the operating system.
  - It represents an independent path of execution.
    - Shared Memory: Threads within the same process share memory and resources.
    - Lightweight: Less overhead compared to processes; faster to create and manage.
    - Concurrency: Enables parallelism within a single process, improving performance for CPU-bound tasks.
    - Can communicate easily with other threads within the same process.
- Task
  - A Task represents an asynchronous operation and provides a higher-level abstraction over threads.
  - It is part of the Task Parallel Library (TPL) and is used for concurrent programming.
    - Asynchronous Programming: Simplifies writing concurrent and asynchronous code.
    - Task-based Asynchronous Pattern (TAP): Utilizes async and await keywords for asynchronous operations.
    - Flexible: Can represent various types of work, such as computations or I/O operations.
    - Built on top of the thread pool

Comparison

Key Points

- Processes are isolated and resource-intensive, suitable for running separate applications.
- Threads share memory within a process, enabling concurrency with lower overhead than processes.
- Tasks provide a higher-level abstraction for asynchronous programming, leveraging the TPL for efficient task management.

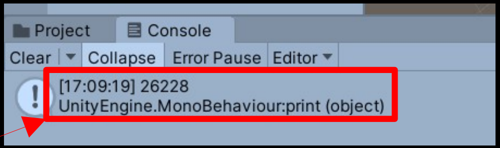
Feature	Process	Thread	Task
Isolation	Yes	No	No
Creation Overhead	High	Low	Low
Communication	Inter-process communication (IPC)	Shared memory	Easier with async/await syntax
Use Case	Running separate applications	Concurrent execution within app	Asynchronous programming, TAP

Comparison

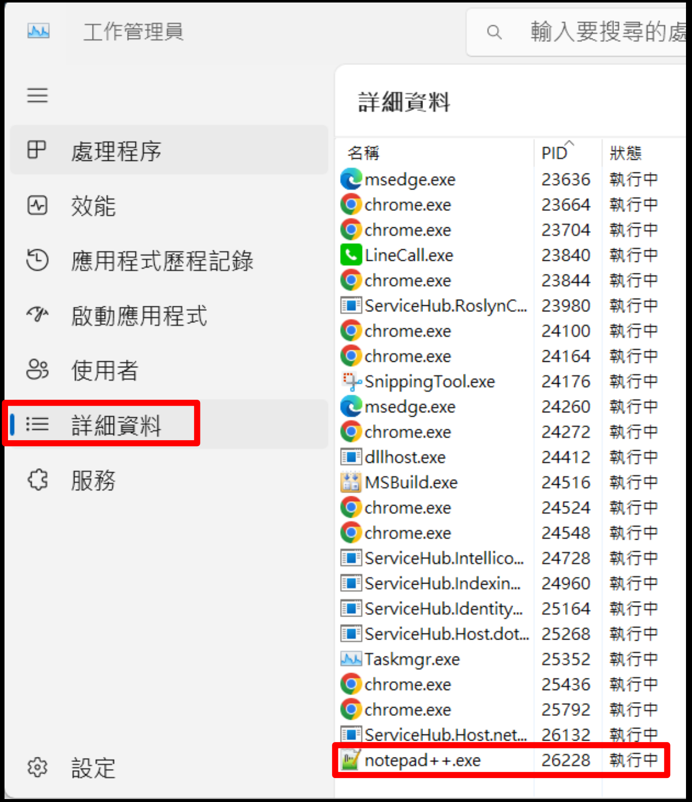
- Process:
  - Benefits:
    - Each process runs independently, preventing issues in one process from affecting others.
    - Crashes in one process do not affect others.
  - Considerations:
    - High overhead for creation and management.
    - Needs IPC for communication. More complex than thread communication.
- Threads:
  - Benefits:
    - Direct control over thread lifecycle.
    - Suitable for long-running background tasks.
  - Considerations:
    - More complex synchronization and resource management.
    - Higher resource consumption due to direct management.
- Tasks:
  - Benefits:
    - Simplified syntax for asynchronous operations.
    - Integrated with the async/await keywords for easier concurrency management.
    - More efficient resource management.
  - Considerations:
    - Overhead of task creation and context switching.
    - Not suitable for very fine-grained parallelism.

EX: Create a Process in Unity3D

- Use Process.Start() to initiate a new process.
- Use Process.CloseMainWindow() or Process.Kill() to close the process.
- You can find the process id with Task manager(Ctrl + Shift + Esc).




```csharp
using System.Diagnostics;
using UnityEngine;

public class EX_Thread_01 : MonoBehaviour
{
    Process process;

    void Start()
    {
        process = Process.Start("C:\\Program Files\\Notepad++\\notepad++.exe");
        print(process.Id);
    }

    void Update()
    {
        if (Input.GetKeyDown("w"))
        {
            process.CloseMainWindow();
            //process.Kill();       // Enforce OS to terminate the process.
        }
    }
}
```



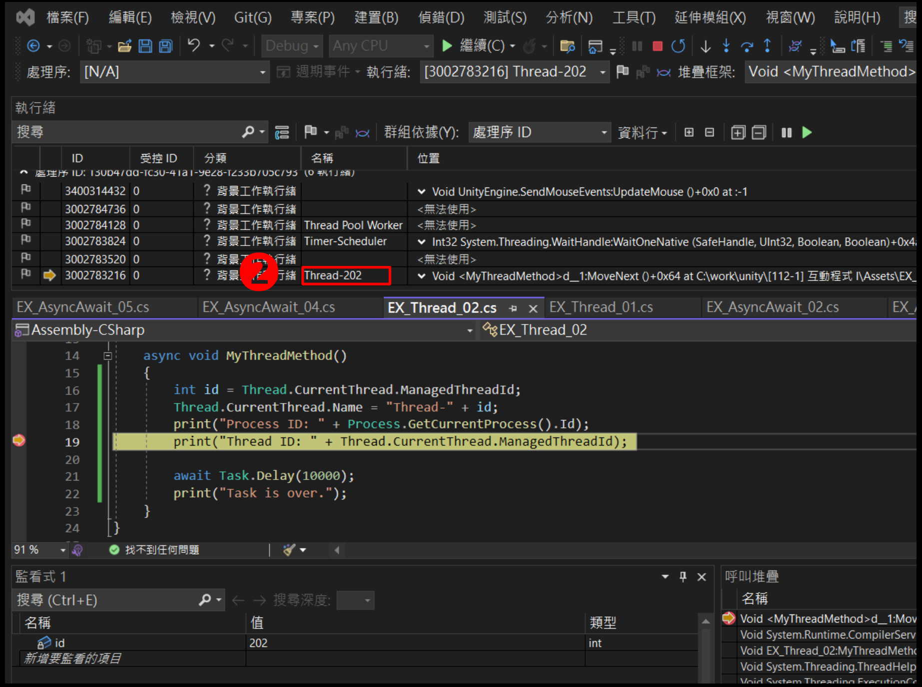
EX: Create a Thread in Unity3D

- The Thread constructor needs an delegate to specify the signature for the Thread method. There are two acceptable delegates for a Thread constructor[*]:
  - If the method has no arguments, you pass a ThreadStart delegate to the constructor.
  - If the method has an argument, you pass a ParameterizedThreadStart delegate to the constructor. The type of the only argument should be object.
- We can place a breakpoint❶ in VS studio to check the Thread id. But there are some bugs in the VS thread browser to display a managed thread id[*], so we can put the thread id into its name and check for the name in thread browser❷.

Assembly-CSharp — EX_Thread_02

```
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;
using UnityEngine;

public class EX_Thread_02 : MonoBehaviour
{
    void Start()
    {
        Thread thread = new Thread(new ThreadStart(MyThreadMethod));
        thread.Start();
    }

    async void MyThreadMethod()
    {
        int id = Thread.CurrentThread.ManagedThreadId;
        Thread.CurrentThread.Name = "Thread-" + id;
        print("Process ID: " + Process.GetCurrentProcess().Id);
        print("Thread ID: " + Thread.CurrentThread.ManagedThreadId);

        await Task.Delay(10000);
        print("Task is over.");
    }
}
```

❶



EX: Create a Task in Unity3D

- This had been introduced in last section.
  Here we have some examples.

```csharp
using System.Threading.Tasks;
using UnityEngine;

public class EX_Thread_03 : MonoBehaviour
{
    void Start()
    {
        Task task = new Task(MyTaskMethod);
        task.Start();
    }

    void MyTaskMethod()
    {
        for (int i = 0; i < 10000; i++)
        {
            print(i);
        }
    }
}
```

A simple blocking task.
Just turn a method into a task and start it.

```csharp
using System.Threading.Tasks;
using UnityEngine;

public class EX_Thread_04 : MonoBehaviour
{
    void Start()
    {
        MyTaskMethod();
        print("start finished.");
    }

    async Task MyTaskMethod()
    {
        await Task.Run(() => {
            for (int i = 0; i < 10000; i++)
            {
                //print(i);
            }
        });
        print("loop finished.");
    }
}
```

An async/await non-blocking task.

```csharp
using System.Threading.Tasks;
using UnityEngine;

public class EX_Thread_04 : MonoBehaviour
{
    async void Start()
    {
        await MyTaskMethod();
        print("start finished.");
    }

    async Task MyTaskMethod()
    {
        await Task.Run(() => {
            for (int i = 0; i < 10000; i++)
            {
                //print(i);
            }
        });
        print("loop finished.");
    }
}
```

An async/await non-blocking task with await invocation.

- Multitasking
  - Coroutine
  - Async/Await
  - Process/Thread/Task
  - File I/O
  - HTTP
  - Appendix

互動程式設計III
Interactive Programming Design Integration

What is Async File I/O

- Definition
  - Asynchronous file I/O operations allow reading from and writing to files without blocking the main thread. This ensures the application remains responsive, especially during long-running file operations.
- Benefits of Async File I/O
  - Non-Blocking: Prevents the main thread from being blocked, improving responsiveness in UI applications.
  - Scalability: Handles multiple file operations concurrently, making it suitable for applications dealing with a large number of files.
  - Performance: Efficiently utilizes system resources by leveraging asynchronous I/O operations.
- Considerations:
  - Error Handling: Always use try-catch blocks to handle potential exceptions during file I/O operations.
  - File Locks: Be aware of file locks that may occur when multiple operations access the same file concurrently.
  - Large Files: For very large files, consider using Stream methods like ReadAsync and WriteAsync for more granular control over the I/O process.

EX: Read a Text File

- Put your text file in <Project folder>/StreamingAssets
- Write some text in the file

```csharp
using System;
using System.IO;
using System.Threading.Tasks;
using UnityEngine;

public class EX_FileIO_01 : MonoBehaviour
{
    async private void Start()
    {
        string path = Path.Combine(Application.streamingAssetsPath, "test.txt");
        string content = await ReadFileAsync(path);
        print(content);
    }

    public async Task<string> ReadFileAsync(string filePath)
    {
        string content = "";
        try
        {
            content = await File.ReadAllTextAsync(filePath);
        }
        catch (Exception ex)
        {
            print("Error reading file: " + ex.Message);
        }
        return content;
    }
}
```

EX: Write a Text File

- Put your text file in <Project folder>/StreamingAssets
- Write some sentences to the content variable.

```csharp
using System;
using System.IO;
using System.Threading.Tasks;
using UnityEngine;

public class EX_FileIO_02 : MonoBehaviour
{
    async private void Start()
    {
        string path = Path.Combine(Application.streamingAssetsPath, "test.txt");
        string content = await WriteFileAsync(path);
        print(content);
    }

    public async Task<string> WriteFileAsync(string filePath)
    {
        string content = "您好：有什麼需要我為您服務的嗎?";
        try
        {
            await File.WriteAllTextAsync(filePath, content);
        }
        catch (Exception ex)
        {
            print("Error reading file: " + ex.Message);
        }
        return content;
    }
}
```

Difference Between File and Stream in C#

- File:
  - The File class in C# provides static methods for creating, copying, deleting, moving, and opening files. It also helps in reading from and writing to files in one go.
    - Static Methods: The File class consists of static methods, meaning you don't need to create an instance of the File class to use them.
    - Convenience: Provides high-level methods for common file operations like ReadAllText, WriteAllText, ReadAllBytes, WriteAllBytes, etc.
    - Use Cases: Ideal for simple file operations where the entire file is read or written at once.
- Stream:
  - A Stream is an abstract base class for working with sequences of bytes, such as file streams, memory streams, network streams, etc. Streams provide a way to read and write data in chunks.
    - Instance Methods: You need to create an instance of a stream class to use it (e.g., FileStream, MemoryStream).
    - Flexibility: Streams provide more control over how data is read and written, allowing for operations on parts of a file or handling continuous data streams.
    - Use Cases: Ideal for scenarios where you need to process large files, read or write data incrementally, or work with network data.
- Summary
  - File: Best for simple, high-level file operations where convenience and ease of use are important.
  - Stream: Best for scenarios requiring detailed control over data processing, large files, and continuous data streams.

EX: Read a Text File Stream

- Put your text file in <Project folder>/StreamingAssets
- Use FileStream to open file. Use StreamReader to read file.

```csharp
using System.IO;
using System.Threading.Tasks;
using UnityEngine;

public class EX_StreamIO_01 : MonoBehaviour
{
    async void Start()
    {
        string path = Path.Combine(Application.streamingAssetsPath, "test.txt");
        await ReadFileAsync(path);
    }

    public async Task ReadFileAsync(string filePath)
    {
        using (FileStream fs = new FileStream(filePath, FileMode.Open, FileAccess.Read, FileShare.Read, 4096, true))
        using (StreamReader reader = new StreamReader(fs))
        {
            string line;
            while ((line = await reader.ReadLineAsync()) != null)
            {
                // Process one line of data.
                print(line);
            }
        }
    }
}
```

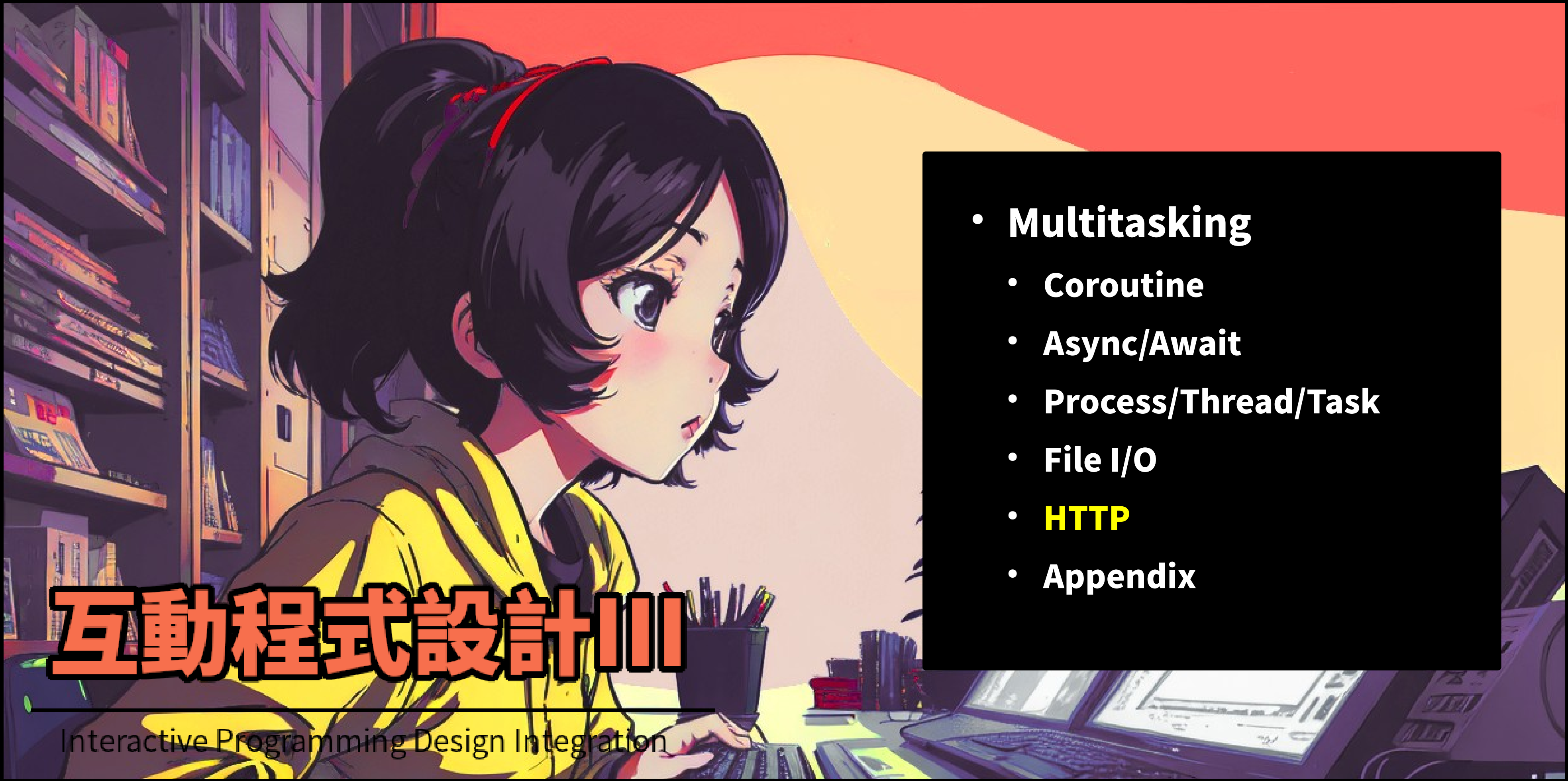
EX: Write a Text File Stream

- Put your text file in <Project folder>/StreamingAssets
- Use FileStream to open file. Use StreamWriter to write file.

```
using System.IO;
using System.Threading.Tasks;
using UnityEngine;

public class EX_StreamIO_02 : MonoBehaviour
{
    async void Start()
    {
        string path = Path.Combine(Application.streamingAssetsPath, "test.txt");
        string content = "您好：有什麼需要我為您服務的？";
        await WriteFileAsync(path, content);
    }

    public async Task WriteFileAsync(string filePath, string content)
    {
        using (FileStream fs = new FileStream(filePath, FileMode.Create, FileAccess.Write, FileShare.None, 4096, true))
        using (StreamWriter writer = new StreamWriter(fs))
        {
            await writer.WriteAsync(content);
        }
    }
}
```

Using using(...) Statement with Streams in C#

- Using()
  - The using statement in C# provides a convenient syntax for ensuring that IDisposable objects, such as streams, are properly disposed of when they are no longer needed. Ensures that resources are released in a timely manner, preventing resource leaks and improving application reliability.
  - Benefits of Using using Statement
    - Automatic Disposal: Automatically calls the Dispose method on the object when the block is exited, even if an exception occurs.
    - Resource Management: Ensures that resources such as file handles, network connections, and memory buffers are properly cleaned up.
    - Code Clarity: Simplifies the code by eliminating the need for explicit try-finally blocks for resource cleanup.


• Multitasking
• Coroutine
• Async/Await
• Process/Thread/Task
• File I/O
• HTTP
• Appendix
互動程式設計III
Interactive Programming Design Integration

Asynchronous HTTP Content Retrieval

- Definition:
  - Asynchronous HTTP content retrieval involves making non-blocking web requests to fetch data from web servers. This keeps the Unity game or application responsive while waiting for the server response.
- Common Use Cases:
  - Fetching data from REST APIs
  - Downloading assets or resources
  - Sending data to web servers
- Benefits of Asynchronous HTTP Requests
  - Non-Blocking: Prevents the main thread from being blocked, ensuring the game or application remains responsive.
  - Efficiency: Allows other operations to continue while waiting for the HTTP response.
  - Scalability: Handles multiple web requests concurrently, improving performance in network-intensive applications.

Asynchronous HTTP Content Retrieval

- Using UnityWebRequest API

```
using System.Collections;
using UnityEngine;
using UnityEngine.Networking;

public class EX_HTTP_01 : MonoBehaviour
{
    void Start()
    {
        // A correct website page.
        StartCoroutine(GetRequest("https://www.example.com"));

        // A non-existing page.
        //StartCoroutine(GetRequest("https://error.html"));
    }

    IEnumerator GetRequest(string uri)
    {
        using (UnityWebRequest webRequest = UnityWebRequest.Get(uri))
        {
            // Request and wait for the desired page.
            yield return webRequest.SendWebRequest();

            switch (webRequest.result)
            {
                case UnityWebRequest.Result.ConnectionError:
                case UnityWebRequest.Result.DataProcessingError:
                    Debug.LogError(uri + ": Error: " + webRequest.error);
                    break;
                case UnityWebRequest.Result.ProtocolError:
                    Debug.LogError(uri + ": HTTP Error: " + webRequest.error);
                    break;
                case UnityWebRequest.Result.Success:
                    Debug.Log(uri + ":\nReceived: " + webRequest.downloadHandler.text);
                    break;
            }
        }
    }
}
```

Asynchronous HTTP Content Retrieval

- Using System.Net.Http API

```csharp
using System;
using System.Net.Http;
using System.IO;
using System.Threading.Tasks;
using UnityEngine;

public class EX_HTTP_02 : MonoBehaviour
{
    private static readonly HttpClient client = new HttpClient();

    async void Start()
    {
        string url = "https://example.com";
        await FetchAndProcessDataAsync(url);
    }

    private async Task FetchAndProcessDataAsync(string url)
    {
        try
        {
            using (HttpResponseMessage response =
                await client.GetAsync(url, HttpCompletionOption.ResponseHeadersRead))
            {
                // 下列代碼會檢查是否response.IsSuccessStatusCode==true，成立即拋出錯誤
                response.EnsureSuccessStatusCode();
                using (Stream stream = await response.Content.ReadAsStreamAsync())
                using (StreamReader reader = new StreamReader(stream))
                {
                    {
                        string content = await reader.ReadToEndAsync();
                        Debug.Log("Received content: " + content);
                    }
                }
            }
        }
        catch (Exception ex)
        {
            Debug.LogError("Error fetching data: " + ex.Message);
        }
    }
}
```

- Multitasking
  - Coroutine
  - Async/Await
  - Process/Thread/Task
  - File I/O
  - HTTP
  - Appendix

互動程式設計III

Interactive Programming Design Integration

Self Learning: Iterator Design Pattern

- Resources:
  - English:
    - Refactoring Guru:
      - https://refactoring.guru/design-patterns/iterator
    - Derek Banas
      - https://www.youtube.com/watch?v=VKIzUuMdmag
    - Gang of four: Design Pattern, 5 Behavioral Pattern - Iterator
      - https://github.com/media-lib/prog_lib/blob/master/general/Gang%20of%20Four%20-%20Design%20Patterns%20-%20Elements%20of%20Reusable%20Object-Oriented%20Software.pdf
  - Chinese:
    - 結城浩(數學少女作者): DESIGN PATTERNS 於 JAVA 語言上的實習應用 Ch.01
      - https://www.drmaster.com.tw/bookinfo.asp?BookID=PG20214
    - 劉韜： 秒懂設計模式 Ch.15
      - https://www.tenlong.com.tw/products/9786263240261

互動程式設計III

Interactive Programming Design Integration

Q&A