

Bridge The Gap

2022/04/04 ~

박지원

<목차>

1. NetFramework vs .NetCore
2. C# Delegate
3. C# Event & EventHandler Class
4. Boxing & Unboxing
5. C# Multi-Thread (1) : Thread Class
6. C# Multi-Thread (2) : Task
7. C# Multi-Thread (3) : Async & Await With MSDN
8. Coroutine Vs Multi-Threading
9. C# List에서 하나의 데이터에 3개 이상의 항목을 저장해야 하는 경우

<.NetFramework vs .NetCore>

1. Dot NetFramework vs Dot NetCore

.Net Framework과 .Net Core 중 어떤 .Net을 사용해야 할까?

현재 .Net 진영에는 두 가지 .Net 프레임워크가 존재합니다.

- .Net Framework → .Net Framework 4.8...
- .Net Core → .Net 5,6,7...

MS에서는 현재 .Net Framework와 .Net Core간의 호환되게 개발할 계획이 없으며, 둘 중 하나를 선택해서 사용해야 합니다. .Net Framework와 .Net Core를 선택하는 기준은 간단합니다.

기존 앱이 .Net Core에서 사용할 수 없는 경우 .Net Framework를, 아닌 경우에는 .Net Core를 사용해야 합니다.

위 기준은 기존의 사용 중인 .Net이 .Net Framework를 계승해야 한다면 .Net Framework를 사용해야 한다는 의미입니다. .Net Core와 호환이 되지 않기 때문에 .Net Core로 새롭게 만들지 않는다면 .Net Framework를 사용하는 것이 좋습니다. 또한 .Net Core에서 지원하지 않는 타사 .Net 라이브러리 사용, NuGet 패키지 사용, .Net Framework만이 가진 기술 등을 사용할 때 .Net Framework를 사용하면 됩니다.

.Net Framework

.Net Framework는 CRL과 .Net Framework 클래스 라이브러리로 구성되어 있습니다. Windows에 종속된 관리형 실행 환경을 가집니다. 현재 MS에서는 4.8 이후 버전에 대한 개발은 없으며, 추후 버전을 올리고자 한다면 .Net Framework에서 벗어나 .NET5인 .Net Core를 사용하라고 권장하고 있습니다. 지속적인 업그레이드가 필요 없는 앱이라면 .Net Framework를 선택하는 것이 좋습니다.

.Net Framework가 가지고 있는 대표 라이브러리는 WPF, Windows Form, ASP.NET이 있습니다. 이를 활용한 프로젝트는 .Net Framework에서만 동작하기 때문에 .Net Framework를 사용하게 됩니다. Core가 나오기 전에 만들어진 윈도우 프로그램과 웹 프로그램이 여기에 속합니다.

.Net Core가 나오기 전에 개발된 앱들은 대부분 .Net Framework로 개발되었습니다. 오래된 프로젝트를 개선하거나 재개발하지 않는 이상 기존 .Net Framework를 사용해도 좋은 성능을 제공합니다.

.Net Core

.Net Core는 .Net Framework의 약점이었던 크로스 플랫폼을 지원합니다. 리눅스, OS X, 윈도우 모든 환경을 지원합니다. .Net Core는 클라우드 사용과 인터넷 연결 앱을 만들기 위한 플랫폼 간 고성능 오픈 소스 프레임워크라고 소개합니다. 오픈 소스라는 점이 특이하며, 클라우드, IoT 등 다양한 환경에서 동작하기 위해 MS에서 출시한 Next 프로젝트라고 보시면 됩니다.

.Net Core는 .Net Framework가 만들었던 대부분의 프로젝트를 다시 만들 수 있습니다. 다만, .Net Framework만 지원하는 라이브러리와 Nuget은 지원하지 않습니다. MS는 .Net Core를 개발자들이 사용할 수 있도록 다양한 이점을 소개했습니다.

- 웹 UI 및 웹 API를 동일한 과정으로 빌드
- 테스트 가능성을 고려
- Blazor, Razor Page 지원.
- gRPC, 클라우드 환경, 종속성 주입 기본 제공.
- 고성능 모듈식 HTTP 요청 파이프라인 지원(BanchMark)
- 다양한 환경에서 호스트 가능

- .netframework를 다운로드 했고, 거기에 존재하는 api 클래스를 모두 사용할 수 있게 됩니다.
- using을 통해 import하는 것이 모두 .netframework를 이용했습니다.
- .net core는 사용하지 않았습니다.

<C# Delegate>

1. 개요

- 개념은 알았지만 필요했던 경우가 적었기에 사용하지 않았습니다.
- 개념은 unity study에 있고, 이 녀석에 대한 심층 고민을 해봅니다.
- 델리게이트를 선언 할 때 최소 1개의 method는 추가되어야 하는 것 같습니다. 그래서 델리게이트 객체를 만들 때 인자가 void라면 선언 에러가 발생합니다.
- 여러 가지 method를 한 번에 호출할 수 있는 편리한 대리자이며, 다른 언어에서는 함수 포인터와 유사합니다.
- delegate는 함수(메서드)를 담을 수 있는 타입이다.
- 타입이기 때문에 메서드의 매개변수나 리턴타입 같은 곳에도 delegate타입을 선언 할 수 있다.
- 델리게이트에 메서드를 여러개 담을 수 있다.

: C#에서 메서드를 가리킬 수 있는 **타입**

(C++의 함수 포인터, 함수형프로그래밍에서 일급 함수와 유사하다.)

delegate로 선언한 타입이 메서드를 가리키기 때문에

그 메서드를 직접 호출하는 것 대신에 delegate로 그 메서드를 호출할 수 있다.

2. 형식

- 형식을 어렵지 않지만 익숙해지지 않으면 계속 헛갈릴 것 입니다.
- 하지만 개념을 확실히 이해했기에 사용하는데 무리가 없을 것 입니다.

3. 선언 위치

구문이 변수를 선언하는 것처럼 보일 수 있지만 실제로는 형식을 선언합니다. **클래스 내부**, 직접 네임스페이스 내부 또는 전역 네임스페이스에 대리자 형식을 정의할 수 있습니다.

① 참고

전역 네임스페이스에 직접 대리자 형식(또는 기타 형식)을 선언하는 것은 권장하지 않습니다.


4. Callback Method를 먼저 이해해야 합니다.

Callback function

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

Here is a quick example:

```
function greeting(name) {  
  alert('Hello ' + name);  
}  
  
function processUserInput(callback) {  
  var name = prompt('Please enter your name.');
```



```
  callback(name);  
}  
  
processUserInput(greeting);
```

5. 그래서 왜 쓸까?

<MSDN>

- 대리자는 C++ 함수 포인터와 유사하지만 C++ 함수 포인터와 달리 멤버 함수에 대해 완전히 개체 지향입니다. 대리자는 개체 인스턴스 및 메서드를 모두 캡슐화합니다.
- 대리자를 통해 메서드를 매개 변수로 전달할 수 있습니다.
- 대리자를 사용하여 콜백 메서드를 정의할 수 있습니다.
- 여러 대리자를 연결할 수 있습니다. 예를 들어 단일 이벤트에 대해 여러 메서드를 호출할 수 있습니다.
- 메서드는 대리자 형식과 정확히 일치하지 않아도 됩니다. 자세한 내용은 [대리자의 가변성 사용](#)을 참조하세요.

1) **method의 인자에 method 주소를 전달 할 수 있습니다.**

- method의 주소 값을 델리게이트에 저장하기 때문에 method를 인자로 넘겨주기에 용이합니다. 이는 call-back method 구현에 용이합니다.

A delegate is a type-safe function pointer that can reference a method that has the same signature as that of the delegate. You can take advantage of delegates in C# to implement events and call-back methods. A multicast delegate is one that can point to one or more methods that have identical signatures.

2) 델리게이트에는 여러 개의 method 주소를 저장할 수 있습니다. (=Delegate Chain)

- 관리 하는 method의 형식(return type , parameter type)이 같아야 합니다.
- 일반화하기 위해 generic을 사용하기도 합니다.

3) 위의 장점 2가지를 설명하기 위한 예제

(1) Delegate 클래스를 만들고 delegate 객체를 선언합니다.

```
delegate void myDelegate(int number);  
myDelegate delegate_object;
```

(2) Delegate 객체에 저장되는 3가지 method

```
private void IncreaseHp(int number)  
{  
    jiwon.hp += number;  
}  
private void IncreaseMp(int number)  
{  
    jiwon.mp += number;  
}  
private void IncreaseExp(int number)  
{  
    jiwon.exp += number;  
}
```

(3) Delegate 객체에 최소 1개의 method를 추가하면서, 여러 가지 method를 추가합니다.

<기본 방식>

```
private void InitializeDelegateInstance()
{
    delegate_object = new myDelegate(IncreaseHp);
    delegate_object += IncreaseMp;
    delegate_object += IncreaseExp;
}
```

<권장하지 않는 방식>

```
private void InitializeDelegateInstance()
{
    delegate_object = null;
    //delegate_object = new myDelegate(IncreaseHp);
    delegate_object += IncreaseHp;
    delegate_object += IncreaseMp;
    delegate_object += IncreaseExp;
}
```

- 이게 정상적으로 동작은 하지만 객체를 null pointer로 초기화 하는 방식은 좋지 않을 것 입니다.

(4) 구현

```
private void Update()
{
    if(Input.GetKeyDown(KeyCode.Z) == true)
    {
        IncreaseThings(30, IncreaseHp);
        IncreaseThings(30, IncreaseMp);
        IncreaseAll(10);
        Show();
    }
}

//1. delegate for call-back
private void IncreaseThings(int number, myDelegate delegate_obj)
{
    delegate_obj(number);
    return;
}

//2. delegate chain
private void IncreaseAll(int number)
{
    delegate_object(number);
}
```

<delegate for call-back>

- IncreaseThings method의 매개변수로 delegate를 추가합니다.
- delegate를 전달하고, 해당 delegate에 저장된 함수 중 인자를 통해 호출이 가능합니다.

<delegate chain>

- IncreaseAll method는 delegate에 저장된 모든 함수들을 호출합니다.

7. Delegate vs 함수 포인터(주소) (최종 정리)

A delegate in C# is a type-safe function pointer with a built in iterator.

- Delegate는 함수 포인터 집합 입니다.
- Delegate 내부에는 1개의 함수 포인터가 존재할 수 있고, 여러 개의 함수 포인터가 존재할 수 있습니다.

<ThreadPool의 QueueUserWorkItem 메서드로 delegate 완벽히 이해하기>

1) 해당 메서드의 매개변수

```
public static class ThreadPool
{
    ...public static bool BindHandle(IntPtr osHandle);
    public static bool BindHandle(SafeHandle osHandle);
    public static void GetAvailableThreads(out int workerThreads, out int completionPortThreads);
    public static void GetMaxThreads(out int workerThreads, out int completionPortThreads);
    public static void GetMinThreads(out int workerThreads, out int completionPortThreads);
    public static bool QueueUserWorkItem(WaitCallback callBack);
    public static bool QueueUserWorkItem(WaitCallback callBack, object state);
}
```

- WaitCallback이 무엇인지 알아보아야 합니다.

2) WaitCallback

```
public delegate void WaitCallback(object state);
```

- WaitCallback은 delegate 입니다. 그러므로 메서드의 주소들을 저장하고 있습니다.
- 즉, 인자로 메서드의 주소를 전달하면 됩니다.
- 처음에는 delegate이므로 delegate를 전달해야 하는 줄 알았으나, delegate는 method의 주소와 동일하므로, method의 주소(method 이름)를 전달하면 됩니다.

3) 예제

```
delegate void MyDelegate();

private void Awake()
{
    DelegateParameter obj = new DelegateParameter();
    ThreadPool.QueueUserWorkItem(obj.Test);
    ThreadPool.QueueUserWorkItem(Test);
}

private void Test(object obj)
{
    Debug.Log("parameter is delegate or function");
}
```

- static method , object method와 상관없이 method의 주소를 전달시키면 실행합니다.

6. delegate 내부 구현

<https://tsyang.tistory.com/32>

7. 참고

- Unity Practice -> PracticeDelegate.cs

<C# Event & EventHandler Class>

1. Event vs Delegate

- Event는 특수한 형태의 Delegate
- Delegate에 두 가지 문제가 있어서 생겨났습니다. 아래의 글을 참고하십시오.

모든 이벤트(event)는 특수한 형태의 delegate이다. C#의 delegate 기능은 경우에 따라 잘못 사용될 소지가 있다. 예를 들어, 우리가 Button 컨트롤을 개발해 판매한다고 하자. 이 컨트롤은 delegate 필드를 가지고 있고, 버튼 클릭시 InvocationList에 있는 모든 메서드들을 차례로 실행하도록 하였다. 그런데, Button 컨트롤을 구입한 개발자가 한 컴포넌트에서 추가 연산(+=)을 사용 하지 않고 실수로 할당 연산자(=)를 사용하였다고 가정하자. 이 할당연산은 기존에 가입된 모든 메서드 리스트를 지워버리고 마지막에 할당한 메서드 한개만 InvocationList에 넣게 할 것이다. 즉, 누구든 할당 연산자를 한번 사용하면 기존에 가입받은 모든 메서드 호출요구를 삭제하는 문제가 발생한다.

이러한 문제점과 더불어 또다른 중요한 문제점은 delegate는 해당 클래스 내부에서 뿐만 아니라, 외부에서도 누구라도 메서드를 호출하듯 (접근 제한이 없다면) 해당 delegate를 호출할 수 있다는 점이다. 아래 예제는 할당연산자를 사용해서 기존 delegate를 덮어쓰는 예와 delegate를 외부에서 호출하는 예를 보여준다.

[보호 수준]

An **Event** declaration adds a layer of abstraction and protection on the **delegate** instance. This protection prevents clients of the delegate from resetting the delegate and its invocation list and only allows adding or removing targets from the invocation list.

An event is a delegate with safety constraint.

- You can only call an event from the class that holds it.
- You can only register/unregister to an event from outside the class (only +=/-=)
- You cannot pass an event as parameter
- You can only wipe clean an event from the class that holds it (eventName = null)

2. Event를 만들 때 Delegate가 항상 필요한가? 너무 귀찮다.

A *delegate* basically describes a function type. An *event* is a hook where registered functions matching a given function type are called when the event occurs.

Putting together these two simplified definitions, a delegate is also a descriptor for an event, therefore you cannot have an event without a related delegate.

Anyway, you are not forced to declare a new delegate for every event you declare. Instead, you can use any of the preexisting delegates such as `EventHandler`, `EventHandler<TEventArgs>` or even any of the `Action` overloads, choosing in the one that fits you best in each case.

- Event를 만들 때 Delegate는 항상 필요하고, 그 Delegate의 이름이 Event를 설명해주기 때문에 잘 지어야 합니다.
- 다행스럽게도, EventHandler 같이 미리 만들어진 Delegate 덕분에 매번 만들 필요가 없습니다.
- EventHandler = .Net 에서 제공하는 만들어진 Delegate

3. EventHandler를 사용하지 않는 경우와 사용하는 경우

1) 사용하지 않으면 매우 귀찮습니다.

```
public class MakeEventNotUsingEventHandler : MonoBehaviour
{
    delegate void MyHandler();
    event MyHandler my_handler;
    private void Start()
    {
        MakeDelegate();
        my_handler();
    }

    private void MakeDelegate()
    {
        my_handler += TestFunction;
    }
    private void TestFunction()
    {
        Debug.Log("Test");
        return;
    }
}
```

- 언제나 delegate를 만들어 주어야 합니다.

2) **EventHandler** 클래스를 사용하면 **Delegate**를 만들지 않아도 됩니다.

- 강제로 2개의 매개변수(object sender , EventArgs s) 두 개를 사용해야 하지만 delegate를 만들지 않습니다.

```
public class MakeEventHandler : MonoBehaviour
{
    EventHandler event_handler;

    private void Start()
    {
        event_handler += TestFunction;
        object tmp = null;
        EventArgs tmp2 = null;
        event_handler(tmp , tmp2);
    }

    private void TestFunction(object sender , EventArgs s)
    {
        Debug.Log("So Easy!");
        return;
    }
}
```

4. EventHandler를 적극적으로 사용합니다.

```
6 public class EventHandlerPractice : MonoBehaviour
7 {
8     EventHandler FirstBlood;
9     private Character nokturn;
10
11     private void Start()
12     {
13         nokturn = new Character();
14         FirstBlood += nokturn.GetGold; //퍼스트 블러드가 발생했을 때 호출할 메소드의 주소를 이벤트 처리기에 등록시킨다.
15         FirstBlood(this, EventArgs.Empty);
16     }
17 }
18
19 public class Character
20 {
21     public int gold { get; private set; } //객체의 상태는 숨긴다.
22
23     public Character()
24     {
25         gold = 0;
26     }
27     public void GetGold(object sender, EventArgs e) //객체의 행동은 나타낸다. 캐릭터가 자신의 보유 골드 상태를 증가시킨다.
28     {
29         gold += 400;
30         Debug.Log(gold);
31     }
32 }
33
```

- Delegate는 메서드의 집합이며, EventHandler는 delegate를 미리 만들어 놓은 것 입니다. 결국 어떤 사건이 발생했을 때 메서드가 호출되는 것은 동일합니다.
- 솔직히 명확하게 왜 사용해야 하는지는 모르겠습니다. 일단 가독성이 좋고, 메시지(=호출할 메서드)를 EventHandler에 모아두는 것이 장점이라고만 체감됩니다. object sender , EventArgs e도 명확한 쓰임새가 지금은 와 닿지 않습니다.
- 추후에 실무를 하면서 깨달은 것이 있다면 적어봅시다.

5. Delegate, Event, EventHandler를 클래스 밖에 선언하는 이유

"I'm curious on why this delegate needs to be declared outside the class, I also want to know why/what situations should I declare variables/types outside a class?"

The delegate does not have to be specified outside of the class. However, if you're creating an event, it typically makes sense to declare the delegate outside of the class since it will be used by other types, outside of the class. That being said, you can just as easily declare it inside the class - but other types will have to fully-qualify it with the class name to use it.

- 세 가지 개념 모두 클래스기 때문입니다.

6. 이벤트 주도적 프로그래밍

- HP가 50이하면 화면이 붉게 깜빡이는 상황을 구현해봅시다.
- Update에서 매 번 검사하는 방식은 좋지 않습니다. 결국 객체의 상태 중 HP에 method로 접근하는 순간이 존재할 것이고 그 때 call-back 함수로 붉게 깜빡이게 하면 됩니다.

<Boxing & Unboxing>

1. Boxing & Unboxing MSDN

다음 예제에서는 정수 변수 `i`를 *boxing*하고 개체 `o`에 할당합니다.

C#

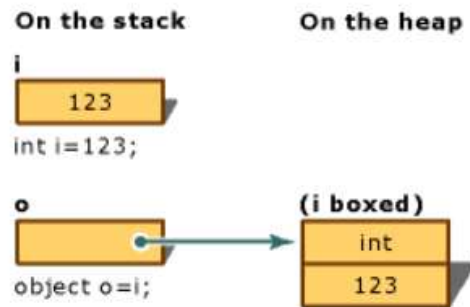
```
int i = 123;  
// The following line boxes i.  
object o = i;
```

그런 다음 `o` 개체를 unboxing하고 정수 변수 `i`에 할당할 수 있습니다.

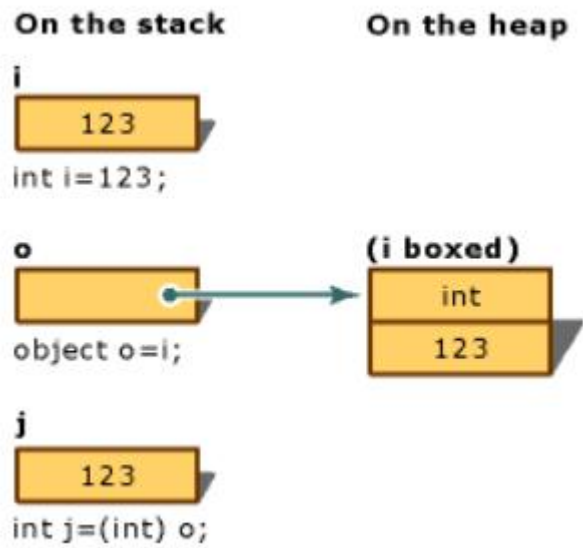
C#

```
o = 123;  
i = (int)o; // unboxing
```

<Boxing>



<UnBoxing>



2. 성능

성능

단순 할당에서는 boxing과 unboxing을 수행하는 데 **많은 계산 과정**이 필요합니다. 값 형식을 boxing할 때는 새로운 개체를 할당하고 생성해야 합니다. 정도는 약간 덜하지만 unboxing에 필요한 캐스트에도 상당한 계산 과정이 필요합니다. 자세한 내용은 성능을 참조하세요.

<C# Multi-Thread (1) : Thread Class>

1. 개요

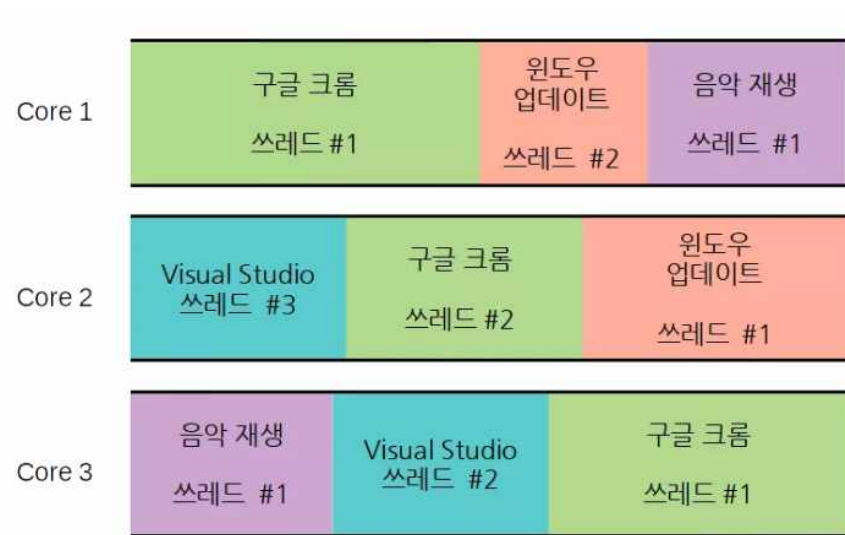
- 기본 개념과 과거에 사용했던 Thread class에 대해서 공부합니다.
- 회사에서 공부한 내용을 바탕으로 다시 완벽히 연마합니다.
- 회사에서 만든 프로그램은 맨 땅에 헤딩 하며 만든 multi thread program이었습니다. 이제는 제대로 알고 만들어야 합니다.

2. CPU Thread vs Process Thread

- 완전히 다른 개념입니다.
- 제 컴퓨터에는 8개의 cpu thread가 존재하고, 컴퓨터마다 cpu thread 개수가 다릅니다. 그러나 프로그램 단계에서 우리는 계속 thread를 생성할 수 있게 됩니다.
- 현대에는 cpu thread가 cpu의 개수로 판단해도 무방하다고 생각합니다. 그리고 프로그램(프로세스)단계에서의 thread는 해당 프로그램을 실행 흐름으로 구분한 단위기 때문에 개수가 무제한도 이론상 가능하겠지요.

3. 하나의 cpu thread는 하나의 thread만 처리하는지? 잠정적으로 Yes

- 1개의 코어로 이루어진 cpu는 동시에 1개의 작업만 가능합니다.
- 2개의 코어로 이루어진 cpu는 동시에 2개의 작업만 가능합니다.
- 4개의 코어로 이루어진 cpu고, 1개의 코어가 2개의 cpu thread로 이루어졌다면 동시에 8개의 작업만 가능합니다.
- 제가 내린 결론 : 보통 1core에 2 cpu thread기 때문에 8core면 총 16개의 cpu thread고, 일단은 1개의 cpu thread가 1개의 core와 동등하게 1개의 업무를 처리할 수 있다고 가정합니다. 그 결과, 16 cpu thread라면 각각 1개의 process thread를 처리할 수 있기에 16개의 process thread를 동시에 처리할 수 있게 됩니다. 그 중에서 먼저 끝난 cpu thread에서 다음 process thread를 작업하게 됩니다.



여러 코어들에서 쓰레드들이 실행되는 모습

4. process thread는 무한히 생성 가능할까? X

There is **no limit** that I know of, but there are two practical limits:

1. The virtual space for the stacks. For example in 32-bits the virtual space of the process is 4GB, but only about 2G are available for general use. By default each thread will reserve 1MB of stack space, so **the top value are 2000 threads**. Naturally you can change the size of the stack and make it lower so more threads will fit in (parameter `dwStackSize` in `CreateThread` or option `/STACK` in the linker command). If you use a 64-bits system this limit practically disappears.
2. **The scheduler overhead.** Once you reach the thousands of threads, just scheduling them will eat nearly 100% of your CPU time, so they are mostly useless anyway. This is not a hard limit, just your program will be slower and slower the more threads you create.

② Multi-Thread

- = 하나의 프로세스 내에서 여러 개의 thread를 지원하는 기능입니다.
- PCB처럼 thread를 추상화한 구조체인 TCB(Thread Control Block)가 존재합니다.
- 하나의 프로세스를 쪼개서 독립적인 미니 프로세스인 Thread로 스케줄링을 운용합니다.
- **스케줄링의 단위가 Thread입니다.**
- 이론적으로는 제한이 없으나 스택의 할당량과 context switching overhead가 발생합니다.

5. ThreadPool

1) static class

```
namespace System.Threading
{
    ...public static class ThreadPool
    {
        ...public static bool BindHandle(SafeHandle osHandle);
        ...public static bool BindHandle(IntPtr osHandle);
        ...public static void GetAvailableThreads(out int workerThreads, out int completionPortThreads);
        ...public static void GetMaxThreads(out int workerThreads, out int completionPortThreads);
        ...public static void GetMinThreads(out int workerThreads, out int completionPortThreads);
        ...public static bool QueueUserWorkItem(WaitCallback callBack);
        ...public static bool QueueUserWorkItem(WaitCallback callBack, object state);
        ...public static RegisteredWaitHandle RegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimerCallba
        ...public static RegisteredWaitHandle RegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimerCallba
        ...public static RegisteredWaitHandle RegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimerCallba
        ...public static RegisteredWaitHandle RegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimerCallba
        ...public static bool SetMaxThreads(int workerThreads, int completionPortThreads);
        ...public static bool SetMinThreads(int workerThreads, int completionPortThreads);
        ...public static bool UnsafeQueueNativeOverlapped(NativeOverlapped* overlapped);
        ...public static bool UnsafeQueueUserWorkItem(WaitCallback callBack, object state);
        ...public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimer
        ...public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimer
        ...public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimer
        ...public static RegisteredWaitHandle UnsafeRegisterWaitForSingleObject(WaitHandle waitObject, WaitOrTimer
    }
}
```

- 죄다 static method 입니다.

Rules for Static Class

1. Static classes cannot be instantiated.
2. All the members of a static class must be static; otherwise the compiler will give an error.
3. A static class can contain static variables, static methods, static properties, static operators, static events, and static constructors.
4. A static class cannot contain instance members and constructors.
5. Indexers and destructors cannot be static
6. `var` cannot be used to define static members. You must specify a type of member explicitly after the `static` keyword.
7. Static classes are sealed class and therefore, cannot be inherited.
8. A static class cannot inherit from other classes.
9. Static class members can be accessed using `ClassName.MemberName`.
10. A static class remains in memory for the lifetime of the application domain in which your program resides.

In C#, the static class contains two types of static members as follows:

- **Static Data Members:** As static class always contains static data members, so static data members are declared using static keyword and they are directly accessed by using the class name. The memory of static data members is allocating individually without any relation with the object.

Syntax:

```
static class Class_name
{
    public static nameofdatamember;
}
```

- **Static Methods:** As static class always contains static methods, so static methods are declared using static keyword. These methods only access static data members, they can not access non-static data members.

Syntax:

- ThreadPool은 객체를 생성할 수 없는 static class 입니다.
- Static class의 구성 멤버는 모두 static!

2) ThreadPool 이용

- ThreadPool에서 수행을 하기 전에 queue에 저장합니다.
- queue가 가득 차면 threadpool에서 thread를 추가적으로 생성합니다.

프로세스당 하나의 스레드 풀이 있습니다. .NET Framework 4부터 프로세스에 대한 스레드 풀의 기본 크기는 가상 주소 공간의 크기와 같은 여러 요인에 따라 달라집니다.

프로세스에서 스레드를 새로 만들었다 삭제하는 것을 반복하면 스레드 생성에 대한 오버헤드가 발생한다. 그래서 일반적으로 프로세스에 스레드 풀을 생성하고 스레드 작업이 필요한 경우 풀에서 스레드를 꺼내와 사용하고 작업이 끝나면 해제하는 것이 아니라 스레드 풀에 되돌려 주는 방식으로 성능의 향상을 꾀한다.

C#에서는 이런 스레드풀을 언어 레벨에서 지원한다. 스레드 수행 할 작업을 스레드 풀의 큐에 넣으면 내부적으로 스레드를 생성하거나 기존 생성되어 있던 스레드를 가져와 작업을 수행 후 스레드를 삭제하는 것이 아니라 대기 상태로 유지 시킨다. CLR[?]은 이 스레드풀에 최적의 스레드 개수를 유지하도록 관리해준다.

.NET의 Thread 클래스를 이용하여 스레드를 하나씩 만들어 사용하는 것이 아니라, 이미 존재하는 스레드풀에서 사용가능한 작업 스레드를 할당 받아 사용하는 방식이 있는데, 이는 다수의 스레드를 계속 만들어 사용하는 것보다 효율적이다.

.NET의 쓰레드풀은 기본적으로 (by default) CPU 코어당 최소 1개의 쓰레드에서 최대 N 개의 작업쓰레드를 생성하여 운영하게 된다. 여기서 최대 N 개는 .NET의 버전에 따라 다른데, .NET 2.0까지는 CPU 코어 당 25개, .NET 3.5(CLR 2.0 SP1+)에서는 CPU 코어 당 250개, .NET 4.0의 32bit에서 1023개, .NET 4.0의 64bit 환경에서는 32768개 등과 같이 다양한 값을 가질 수 있다. 예를 들어, 만약 해당 컴퓨터가 4개의 CPU 코어를 가지고 있고 .NET 3.5를 사용하고 있다면, ThreadPool은 최소 4개의 쓰레드와 최대 1000개의 작업 쓰레드를 가질 수 있게된다.

3) 연습

```
private void Awake()
{
    int thread_count = -1;
    int thread_count_sub = -1;
    ThreadPool.GetAvailableThreads(out thread_count, out thread_count_sub);
    Debug.Log("thread count " + " " + thread_count + " " + thread_count_sub);
    ThreadPool.QueueUserWorkItem(Repeat, 'a');
    ThreadPool.QueueUserWorkItem(Repeat, 'b');
    ThreadPool.QueueUserWorkItem(Repeat, 'c');
}

private void Repeat(object obj)
{
    for(int i=0; i<100; i++)
    {
        Debug.Log(obj + " " + i);
    }
}
```

- 결과는 3개의 Repeat 함수가 비동기적으로 수행됩니다.
- 3개의 스레드를 올바르게 사용함을 알 수 있습니다.

6. Main Thread(=UI Thread) vs Worker Thread

.NET에서 UI Application을 만들기 위해 Windows Forms(윈폼)이나 WPF (Windows Presentation Foundation)을 사용한다. 이들 WinForm이나 WPF는 그 UI 컨트롤을 생성한 스레드만(UI 스레드)이 해당 UI 객체를 액세스할 수 있다는 스레드 선호도(Thread Affinity) 규칙을 지키도록 설계되었다. 만약 개발자 이러한 기본 규칙을 따르지 않는다면, 에러가 발생하거나 예기치 못한 오동작을 할 수 있다. **UI 컨트롤을 생성하고 이 컨트롤의 윈도우 핸들을 소유한 스레드를 UI Thread라 부르고**, 이러한 UI 컨트롤들을 갖지 않는 스레드를 작업스레드(Worker Thread)라 부른다. 일반적으로 UI 프로그램은 하나의 UI Thread (주로 메인스레드)를 가지며, 여러 개의 작업 스레드를 갖는다. 하지만 필요한 경우 여러 개의 UI 스레드를 갖게 하는 것도 가능하다.

<C# Multi-Thread (2) : Task>

1. Action Class

1) Task를 이해하기 위해 Action을 알아야 하는 이유

```
public class Task : IAsyncResult, IDisposable
{
    public Task(Action action);
    public Task(Action action, CancellationToken cancellationToken);
    public Task(Action action, TaskCreationOptions creationOptions);
    public Task(Action<object> action, object state);
    public Task(Action action, CancellationToken cancellationToken, TaskCreationOptions creationOptions);
    public Task(Action<object> action, object state, CancellationToken cancellationToken);
    public Task(Action<object> action, object state, TaskCreationOptions creationOptions);
    public Task(Action<object> action, object state, CancellationToken cancellationToken, TaskCreationOptions creationOptions);
}
```

- 생성자의 매개변수가 Action 입니다.

2) Action을 사용하는 이유

Action delegate is an in-built generic type delegate. This delegate saves you from defining a custom delegate as shown in the below examples and make your program more readable and optimized. It is defined under *System* namespace. It can contain minimum 1 and maximum of 16 input parameters and does not contain any output parameter. The Action delegate is generally used for those methods which do not contain any return value, or in other words, Action delegate is used with those methods whose return type is void. It can also contain parameters of the same type or of different types.

- Action은 Delegate에 포함되는 개념입니다.
- Delegate를 만드는 것이 귀찮기 때문에 만들어진 Delegate를 사용하는 것입니다.
- Return이 Void 이고, Parameter는 최대 1~16개까지 담을 수 있는 Method들의 Delegate = Action
- Delegate이므로 Action에도 형식만 일치하면 다수의 method를 추가시킬 수 있습니다.

3) 예제 (parameter가 있는 Action , Parameter가 없는 Action)

```
void Start()
{
    Action non_params = new Action(DoSomething2);
    non_params();

    Action<string> parameters = new Action<string>(DoSomething);
    parameters("do1");
}

private static void DoSomething(string str)
{
    Debug.Log(str);
}

private static void DoSomething2()
{
    Debug.Log("do2");
}
```

2. 왜 Thread Class와 ThreadPool 대신에 Task가 출현했는가?

1) Thread는 상당히 무거운 객체입니다. 보통 1mb를 할당합니다.

2) Thread의 작업 완료 시점을 알 수 없습니다. 메인스레드는 작업스레드가 뭘 하든 신경 쓰지 않습니다.

3) Thread의 취소 / 예외 처리를 할 수 없습니다.

3. Thread vs Task

Task

- C#에 `Task` 클래스가 있다.
- `ThreadPool`을 더 편리하게 사용하도록 만든 라이브러리이다.

Thread

- 스레드가 생성되어 장시간 동작해야 되는 경우 사용한다.

Task

- 단발적이고 짧은 동작들을 수행하는 경우 사용한다.
- 장시간 동작해야 하는 경우, `ThreadPool`의 `Thread`를 하나 점유하지 않도록 다음과 같이 생성한다.

Task work asynchronously manages the the unit of work. In easy words Task doesn't create new threads. Instead it efficiently manages the threads of a threadpool. Tasks are executed by `TaskScheduler`, which queues tasks onto threads.

- Task는 결국 `ThreadPool`의 thread 1개를 차지하기 때문에 장기간의 작업이라면 Thread를 따로 하나 생성해서 독립적으로 이용하는 것이 훨씬 효율적입니다.

4. Task는 Static method를 이용할까 , 객체를 생성해서 일반 method를 사용할까?

1) Task 클래스는 static method 와 일반 method가 존재합니다.

```
public void Start();  
public void Start(TaskScheduler scheduler);
```

```
public static Task Run(Func<Task> function, CancellationToken cancellationToken);  
public static Task Run(Action action, CancellationToken cancellationToken);  
public static Task Run(Action action);
```

- 어떤 걸 사용할지 아래를 읽어봅시다.

2) **Task.Run()**을 사용합니다. **Task객체.Start()**은 사용하지 않습니다.

Don't ever create a `Task` and call `Start()` unless you find an extremely good reason to do so. It should only be used if you have some part that needs to create tasks but **not schedule them** and another part that schedules without creating. That's almost never an appropriate solution and could be dangerous. More in ["Task.Factory.StartNew" vs "new Task\(...\).Start"](#)

In conclusion, mostly use `Task.Run` , use `Task.Factory.StartNew` if you must and never use `Start` .

Task.Run 메서드

참조



정의

네임스페이스: [System.Threading.Tasks](#)

어셈블리: [System.Runtime.dll](#)

지정한 작업을 ThreadPool에서 실행하도록 큐에 대기시키고 해당 작업에 대한 작업 또는 [Task<TResult>](#) 핸들을 반환합니다.

3) Task.Run vs Task.Factory.Startnew

- 둘은 완벽하게 동일한 개념입니다.

So, in the [.NET Framework 4.5 Developer Preview](#), we've introduced the new Task.Run method. This in no way obsoletes Task.Factory.StartNew, but rather should simply be thought of as a quick way to use Task.Factory.StartNew without needing to specify a bunch of parameters. It's a shortcut. In fact, Task.Run is actually implemented in terms of the same logic used for Task.Factory.StartNew, just passing in some default parameters. When you pass an Action to Task.Run:

```
Task.Run(someAction);
```

that's exactly equivalent to:

```
Task.Factory.StartNew(someAction,  
    CancellationToken.None, TaskCreationOptions.DenyChildAttach,  
    TaskScheduler.Default);
```

4) Task.Run()을 가리키는 변수는 중요합니다.

- static method인 Task.Run(action)을 수행하면 Threadpool에 존재하는 thread에서 action을 수행합니다. Task.Run(action)의 return 값이 threadpool에서 해당 task가 실행되도록 준 task(=thread)입니다. 그러므로 개발자는 어떤 Task 변수에 해당 return 값을 캐싱 해서 유용하게 사용합니다.

<Run method의 return 값이 Task인 것을 알 수 있습니다.>

```
public static Task Run(Func<Task> function);  
public static Task<TResult> Run<TResult>(Func<Task<TResult>> function, CancellationToken cancellationToken);  
public static Task<TResult> Run<TResult>(Func<Task<TResult>> function);  
public static Task<TResult> Run<TResult>(Func<TResult> function, CancellationToken cancellationToken);  
public static Task<TResult> Run<TResult>(Func<TResult> function);  
public static Task Run(Func<Task> function, CancellationToken cancellationToken);  
public static Task Run(Action action, CancellationToken cancellationToken);  
public static Task Run(Action action);
```

<Task 변수로 참조하기>

```
Task my_task = Task.Run(action);
```

5. Task 클래스를 통해 운영체제의 스케줄링을 비슷하게 구현해보자.

1) Task.Run()

① MSDN 예제

```
public class Example
{
    public static void Main()
    {
        ShowThreadInfo("Application");

        var t = Task.Run(() => ShowThreadInfo("Task"));
        t.Wait();
    }

    static void ShowThreadInfo(String s)
    {
        Console.WriteLine("{0} thread ID: {1}",
                           s, Thread.CurrentThread.ManagedThreadId);
    }
}
```

② 나의 예제

```
//action1은 매개변수 1개의 메소드로 이루어져 있고 2개의 메소드가 저장되어 있다.
Action<object> action1 = new Action<object>(Func);
action1 += Func2;

//action2는 매개변수 2개의 메소드로 이루어져 있고 1개의 메소드가 저장되어 있다.
Action<object, object> action2 = new Action<object, object>(Func3);

Task.Run(() => action1("a"));
Task.Run(() => action2("a", "b"));
```

- action1과 action2는 다른 스레드에서 비동기적으로 수행되기 때문에 동시에 각각의 action을 수행합니다.
- action1은 Func부터 수행한 후, Func2를 수행합니다.

<C# Multi-Thread (3) : Async & Await With MSDN>

[Reference] <https://docs.microsoft.com/ko-kr/dotnet/csharp/programming-guide/concepts/async/>

[비동기로 아침 요리를 구현해봅시다.]

1. 아침 요리 구현 메서드

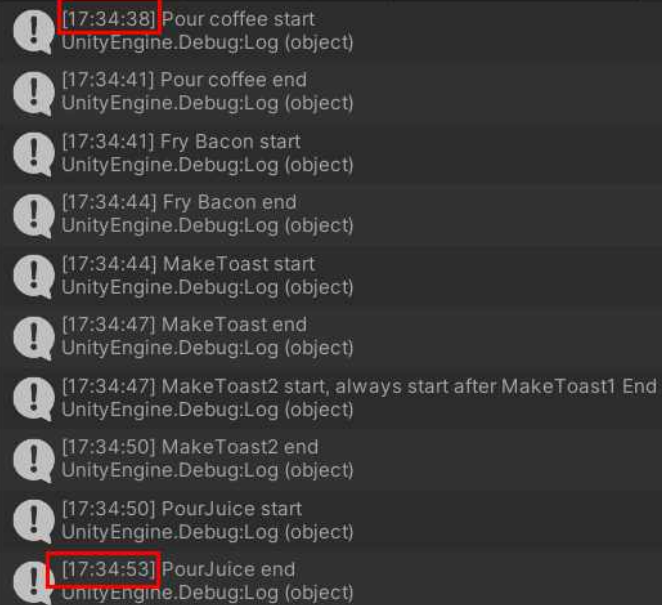
```
public static void PourCoffee()
{
    Debug.Log("Pour coffee start");
    Task.Delay(3000).Wait();
    Debug.Log("Pour coffee end");
}

public static void FryBacon()
{
    Debug.Log("Fry Bacon start");
    Task.Delay(3000).Wait();
    Debug.Log("Fry Bacon end");
}
```

- 모든 아침 요리 메서드는 위의 형식과 동일합니다.
- 3초 지연
- 아래의 '2. ~ 4.'를 통해 세 단계의 아침 요리 방식을 구현합니다.

2. 첫 번째 방식 : 아침 요리 전체를 동기적으로 구현하면 생기는 문제점

```
public static void DoExampleOne()
{
    PourCoffee();
    FryBacon();
    MakeToast();
    MakeToast2();
    PourJuice();
}
```



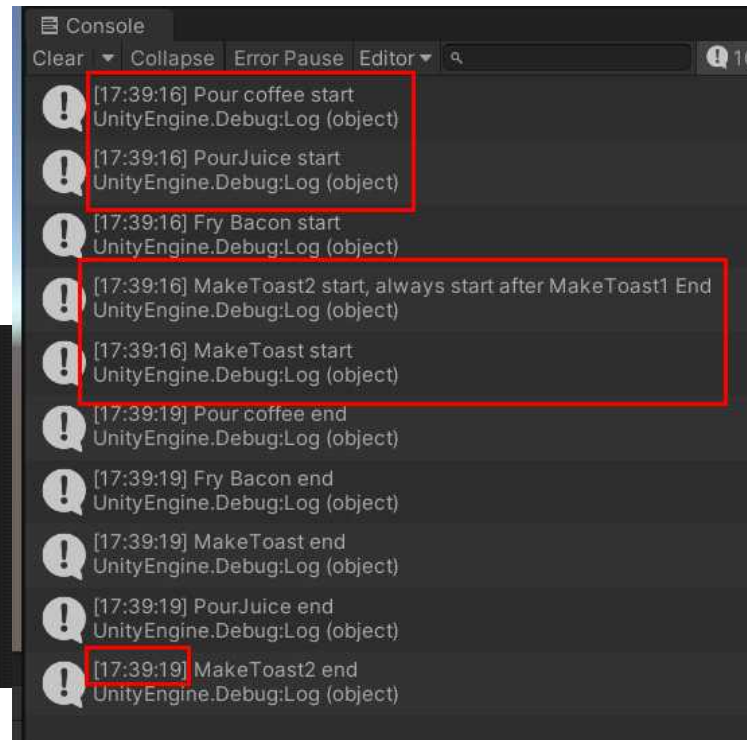
The screenshot shows a list of Unity console logs. Each log entry starts with a timestamp in square brackets, followed by a task name and 'UnityEngine.Debug:Log (object)'. The tasks are executed sequentially, with each subsequent task starting after the previous one has finished. The timestamps are: [17:34:38] Pour coffee start, [17:34:41] Pour coffee end, [17:34:41] Fry Bacon start, [17:34:44] Fry Bacon end, [17:34:44] MakeToast start, [17:34:47] MakeToast end, [17:34:47] MakeToast2 start, always start after MakeToast1 End, [17:34:50] MakeToast2 end, [17:34:50] PourJuice start, and [17:34:53] PourJuice end. The first and last log entries are highlighted with red boxes.

[17:34:38] Pour coffee start
UnityEngine.Debug:Log (object)
[17:34:41] Pour coffee end
UnityEngine.Debug:Log (object)
[17:34:41] Fry Bacon start
UnityEngine.Debug:Log (object)
[17:34:44] Fry Bacon end
UnityEngine.Debug:Log (object)
[17:34:44] MakeToast start
UnityEngine.Debug:Log (object)
[17:34:47] MakeToast end
UnityEngine.Debug:Log (object)
[17:34:47] MakeToast2 start, always start after MakeToast1 End
UnityEngine.Debug:Log (object)
[17:34:50] MakeToast2 end
UnityEngine.Debug:Log (object)
[17:34:50] PourJuice start
UnityEngine.Debug:Log (object)
[17:34:53] PourJuice end
UnityEngine.Debug:Log (object)

- 커피 -> 베이컨 -> 토스트 -> 토스트2 -> 주스를 순서대로(동기적으로) 작업합니다.
- 동기적으로 수행할 경우 커피를 만드는 작업이 완료되기 전까지 베이컨 튀기기를 시작하지 않습니다. 그러므로 시간이 오래 걸립니다.
- 15초 수행
- 메인스레드가 Delay 함수로 멈추기 때문에 유니티도 15초 동안 멈춥니다.

3. 두 번째 방식 : 아침 요리 전체를 비동기적으로 구현하면 생기는 문제점

```
public static void DoExampleTwo()
{
    Task task1 = Task.Run(PourCoffee);
    Task task2 = Task.Run(FryBacon);
    Task task3 = Task.Run(MakeToast);
    Task task4 = Task.Run(MakeToast2);
    Task task5 = Task.Run(PourJuice);
}
```



- 커피 -> 베이컨 -> 토스트 -> 토스트2 -> 주스를 동시에 시작합니다.
- Task를 이용하기 때문에 요리사가 총 5명입니다.
- 동시에 시작하기 때문에 총 작업 시간은 3초밖에 걸리지 않습니다. 하지만 토스트2는 토스트1의 제작이 완료되고 진행되어야 하는데 동시에 하는 오류가 발생합니다.
- 확실하지는 않으나 현재 PourJuice가 FryBacon 보다 같은 시간이지만 출력이 먼저 되는 것을 보아 미세하게 먼저 시작했다고 가정할 수 있습니다. 이러한 미세한 시간도 결국 실행 순서의 보장이 불가능하다는 것을 자명하게 합니다.

4. 세 번째 방식 : 아침 요리의 일부를 비동기적으로 구현하기

① 중요

동기 작업이 뒤따르는 비동기 작업으로 구성된 작업은 비동기 작업입니다. 즉 작업의 일부가 비동기이면 전체 작업이 비동기입니다.

```
public static async void DoExampleThree()
{
    Task task1 = Task.Run(PourCoffee);
    await task1;

    Task task2 = Task.Run(FryBacon);
    Task task3 = Task.Run(MakeToast);
    await task3;
    Task task4 = Task.Run(MakeToast2);
    Task task5 = Task.Run(PourJuice);
}
```

[17:44:31] Pour coffee start
UnityEngine.Debug:Log (object)

[17:44:34] Pour coffee end
UnityEngine.Debug:Log (object)

[17:44:34] MakeToast start
UnityEngine.Debug:Log (object)

[17:44:34] Fry Bacon start
UnityEngine.Debug:Log (object)

[17:44:37] Fry Bacon end
UnityEngine.Debug:Log (object)

[17:44:37] MakeToast end
UnityEngine.Debug:Log (object)

[17:44:37] PourJuice start
UnityEngine.Debug:Log (object)

[17:44:37] MakeToast2 start, always start after MakeToast1 End
UnityEngine.Debug:Log (object)

[17:44:40] MakeToast2 end
UnityEngine.Debug:Log (object)

[17:44:40] PourJuice end
UnityEngine.Debug:Log (object)

- 커피 작업이 완료되면 이후의 일을 진행하도록 구현했고, 토스트1의 작업이 완료되면 토스트2를 시작하도록 구현했습니다.
- 결과에서 나타나듯, 올바르게 실행 순서를 지키고 있습니다.
- Await을 통해 해당 키워드가 붙은 부분을 동기적으로 구현할 수 있습니다. (실행흐름 관리)
- 전체 시간도 9초로 감소했습니다.

5. async / await

C# async는 컴파일러에게 해당 메서드가 await를 가지고 있음을 알려주는 역할을 한다. async라고 표시된 메서드는 await를 1개 이상 가질 수 있는데, 하나도 없는 경우라도 컴파일은 가능하지만 Warning 메시지가 표시된다. async를 표시한다고 해서 자동으로 비동기 방식으로 프로그램을 수행하는 것은 아니고, 일종의 보조 역할을 하는 컴파일러 지시어로 볼 수 있다. async 메서드의 리턴 타입은 대부분의 경우 Task<TResult> (리턴값이 있는 경우) 혹은 Task (리턴값이 없는 경우)인데, 예를 들어 리턴값이 string일 경우 async Task<string> method()와 같이 정의하고 return "문자열"과 같이 문자열만 리턴한다. C# 컴파일러는 return 문의 문자열을 자동으로 Task<string>로 변환해 준다. 또 다른 async 메서드의 리턴 타입으로 void 타입이 있는데, 특히 이벤트핸들러를 위해 void 리턴을 허용하고 있다.

- 코드에 await을 붙이면 비동기 코드를 동기적으로 만들 수 있습니다.
- 내 생각 : 비동기를 통한 멀티스레딩은 작업 속도와 퍼포먼스의 향상은 존재하겠지만 작업의 순서를 보장할 수 없습니다. 작업의 순서를 보장하기 위해 일부분은 await을 통해 순서를 부여하여 동기적으로 변경해 줍니다.

<Coroutine Vs Multi-Threading>

1. coroutine vs multi-threading

유니티에서의 대부분 로직은 동기 방식으로 작성되며, 흐름(Flow, Stream)을 관리하거나 시간을 조절해 시행해야하는 작업은 코루틴 Coroutine이라는 기능을 활용하여 구현됩니다. 특히 유니티는 기본적으로 메인스레드가 아닌 곳에서의 네이티브 API 호출을 허용하고 있지 않습니다. 비동기 방식으로 로직을 작성하더라도 유니티 API를 사용해야 하는 끝단에서는 메인스레드에서 함수 Call 작업을 마무리 해 주어야 합니다. 이 때 일반적으로 사용되는 방식이 액션을 큐에 추가해 유니티 Update 루프에서 처리해주는 디스패처입니다. 이 방법론이 필요한 이유와 한계, 장단점을 알고 계시다면 아래 내용은 스킵해도 됩니다.

왜 게임 개발에서는 비동기 사용이 적을까요?

상용 엔진에서는 async/await를 사용해야만 하는 경우가 극히 드뭅니다. 일반적으로 그 외적인 방법으로도 컨트롤 할 수 있는 영역 내에서 스크립팅이 진행되고, Scalable한 비동기 API를 제공한다고 해도 그 효과를 적극적으로 누릴 수 있는 분야는 다소 제한적이기 때문입니다. 렌더링을 비롯한 게임 로직을 비동기로 동시에 사용할 수 있게 제공하더라도 그 효율성보다 그 준비단계에서의 부담이 더 크므로 그 필요성을 느끼지 못하는 것입니다.

(사용자의 입장에서 async/await를 사용하고 싶을 수도 있지만, 엔진 개발 및 제공자의 입장에서 매력적인 옵션은 아닙니다.)

While Coroutines seem to work like threads at first glance, they actually aren't using any multithreading. They are executed sequentially until they `yield`. The engine will check all yielded coroutines as part of its own main loop (at what point exactly depends on the type of `yield`, [check this diagram for more information](#)), continue them one after another until their next `yield`, and then proceed with the main loop.

This technique has the advantage that you can use coroutines without the headaches caused by problems with real multithreading. You won't get any deadlocks, race conditions or performance problems caused by context switches, you will be able to debug properly and you don't need to use thread-safe data containers. This is because when a coroutine is being executed, the Unity engine is in a controlled state. It is safe to use most Unity functionality.

With threads, on the other hand, you have absolutely no knowledge about what state the Unity main loop is in at the moment (it might in fact no longer be running at all). So your thread might cause quite a lot of havoc by doing something at a time it isn't supposed to do that thing. **Do not touch any native Unity functionality from a sub-thread.** If you need to communicate between a sub-thread and your main thread, have the thread write to some thread-safe(!) container-object and have a MonoBehaviour read that information during the usual Unity event functions.

The disadvantage of not doing "real" multithreading is that you can not use coroutines to parallelize CPU-intense calculations over multiple CPU cores. You can use them, though, to split a calculation over multiple updates. So instead of freezing your game for one second, you just get a lower average framerate over multiple seconds. But in that case you are responsible to `yield` your coroutine whenever you want to allow Unity to run an update.

Conclusion:

- If you want to use asynchronous execution to express game logic, use coroutines.
- If you want to use asynchronous execution to utilize multiple CPU cores, use threads.

2. 코루틴에 대한 글

- 코루틴은 동기적 이라고 생각해왔습니다. 그러나 공부하고 정리하던 도중 아래의 예시를 떠올렸습니다.

ex) 카페에 손님이 10명 왔습니다. 완벽한 동기 프로그래밍은 1번 ~ 10번 까지 손님의 음료를 순차적으로 처리합니다. 코루틴은 1번 ~ 10번 까지 손님의 음료를 1명의 직원이 1번 음료 만들다가 3번 음료 만들다가 5번 음료 만들다가 다시 1번 음료를 만드는 느낌입니다. 완벽한 비동기 프로그래밍은 10명의 직원이 각각 일대일 대응으로 자신이 맡은 음료를 동시에 만듭니다. 가장 빠르겠지만 그 과정에서 재료를 공유하기 때문에 원하지 않는 결과가 나올 수 있습니다. 코루틴은 결국 동시에 하지 않으므로 비동기는 아니지만 순차적으로 하지도 않습니다.

<C# List에서 하나의 데이터에 3개 이상의 항목을 저장해야 하는 경우>

[기존에는 struct를 습관적으로 이용하거나 분할해서 자료를 저장해왔습니다. 하지만 struct보다 class를 사용하는 것이 좋아 보입니다.]

1. Struct는 예러가 발생합니다.

```
C# 복사  
  
List<myStruct> list = {...};  
list[0].Name = "MyStruct42"; //CS1612
```

구조체를 수정하려면 먼저 지역 변수에 할당하고 변수를 수정한 다음 컬렉션의 항목에 변수를 다시 할당합니다.

```
C# 복사  
  
List<myStruct> list = {...};  
MyStruct ms = list[0];  
ms.Name = "MyStruct42";  
list[0] = ms;
```

이 오류는 할당 시 값 형식이 복사되기 때문에 발생합니다. 속성 또는 인덱서에서 값 형식을 검색하는 경우 개체 자체에 대한 참조가 아니라 개체 복사본을 얻게 됩니다. 반환되는 복사본은 스토리지 위치 (변수)가 아니라 실제로 메서드이기 때문에 속성 또는 인덱서에서 저장되지 않습니다. 선언하는 변수에 복사본을 저장해야 수정할 수 있습니다.

이 경우 속성 또는 인덱서가 스토리지 위치인 기존 개체에 대한 참조를 반환하기 때문에 참조 형식에서는 오류가 발생하지 않습니다.

2. struct vs class

Value Type vs Reference Type

C#은 Value Type과 Reference Type을 지원한다. C#에서는 `struct`를 사용하면 Value Type을 만들고, `class`를 사용하면 Reference Type을 만든다.

The general rule to follow is that structs should be small, simple (one-level) collections of related properties, that are immutable once created; for anything else, use a class.

C# is nice in that structs and classes have no explicit differences in declaration other than the defining keyword; so, if you feel you need to "upgrade" a struct to a class, or conversely "downgrade" a class to a struct, it's mostly a simple matter of changing the keyword (there are a few other gotchas; structs can't derive from any other class or struct type, and they can't explicitly define a default parameterless constructor).

I say "mostly", because the more important thing to know about structs is that, because they are value types, treating them like classes (reference types) can end up a pain and a half. Particularly, making a structure's properties mutable can cause unexpected behavior.

3) 예시

```
public class Machine
{
    public class bottle_info
    {
        //이런 순수 자료 클래스는 굳이 private set을 할 필요가 있을까
        //public string name { get; private set; }
        //public int count { get; private set; }
        //public int price { get; private set; }
        public string name;
        public int count;
        public int price;

        public bottle_info(string a, int b, int c)
        {
            name = a;
            count = b;
            price = c;
        }
    };

    private List<bottle_info> bottles;

    //객체지향의 세상에서는 자판기가 자신의 수치를 관리한다.
    Machine()
    {
        bottles.Add(new bottle_info("coke", 10, 500));
        bottles.Add(new bottle_info("orange_juice", 12, 500));
    }
}
```

<자유로운 내용>

1. visual studio 에서 f12를 통해 reference를 확인할 수 있다.

2. unity에서 멤버 변수 종류 초기화 하는 법칙

Monobehaviours have special methods that are called by the unity engine. Unity documentation says:

- **Awake:** This function is always called before any Start functions and also just after a prefab is instantiated. (If a GameObject is inactive during start up Awake is not called until it is made active.)
- **Start:** Start is called before the first frame update only if the script instance is enabled.
- **Update:** Update is called once per frame. It is the main workhorse function for frame updates.

Awake and Start methods are used to initialize components before they become active in the scene.

Experienced C# developers know that classes have constructors and the constructor is called once for every instance created. We put initialization code (e.g., setting default values) in the constructor and we should avoid adding application logic to constructors.

- 변수 종류에 대한 선언은 메서드 문치 위에 독립적으로 하고, awake에서 초기화 합니다.

3. 생성자의 디폴트는 public

By default, constructors are defined in public section of class.

- 보통 다른 클래스에서 객체를 생성할 텐데, private이면 보호수준으로 인해 멤버 변수를 초기화하지 못합니다.
- 생성자를 이용한 초기화라면 멤버 변수를 초기화하지 않는 불상사를 최소화 할 수 있어 보입니다.