



# C# -CHAPTER3-

SOUL SEEK

# 목차

1. Delegate & Event
2. 람다식
3. 가비지 컬렉션(GC)

The background is a dark blue gradient with a large, faint, light blue circle in the center. In the four corners, there are white, stylized circuit-like lines with small circles at the ends, resembling a network or data flow diagram.

# DELEGATE & EVENT

# 1. DELEGATE & EVENT

## Delegate

- **Callback**을 만들기 위한 형식 - 메소드에 의한 참조
- 메소드를 대신해서 호출하는 역할을 한다. 특정 메소드를 처리할 때 그 메소드를 직접 호출해서 실행시켜야 했지만 **Delegate**를 사용하면 그 메소드를 대신하여 호출할 수 있다.

```
delegate int myDelegate(int a, int b);  
class Delegate  
{  
    //사용할 함수들  
    public static int plus(int a, int b)  
    {  
        return a + b;  
    }  
  
    public static int minus(int a, int b)  
    {  
        return a - b;  
    }  
}
```

단지 선언하고 호출할 때의 모양만 보았다  
이렇게 쓸려고만 했으면 쓸 필요가 없다.  
**delegate**를 쓰는 진짜 목표는 콜백 메서드의  
역할인 것이다

```
static void Main(string[] args)
```

```
{
```

```
    //기본 형태
```

```
    //delegate 변수 선언
```

```
    myDelegate caculate;
```

```
    //함수를 delegate에 선언
```

```
    caculate = new myDelegate(plus);
```

```
    int sum = caculate(11, 22);
```

```
    Console.WriteLine("11 + 22 = {0}", sum);
```

```
    //함수를 delegate에 선언
```

```
    caculate = new myDelegate(minus);
```

```
    Console.WriteLine("22 - 11 = {0}",
```

```
    caculate(22, 11));
```

```
}
```

```
}
```

# 1. DELEGATE & EVENT

//Callback 메서드

```
myDelegate Plus = new myDelegate(plus);  
myDelegate Minus = new myDelegate(minus);  
myDelegate Multiply = new myDelegate(multiply);
```

```
Calculator(11, 22, Plus);  
Calculator(33, 22, Minus);  
Calculator(11, 22, Multiply);
```

- 콜백함수를 만들어 쓰게 되는 형식의 **delegate**는 **Unity**에서 스크립트 형식의 참조를 하기 때문에 스크립트끼리의 통신 예를 들면 매니저 함수에서 버튼 역할을 하는 **UI**의 특정 값처리 결과를 받고 싶을때 해당 객체를 구하는 형식으로 해야한다.그럴경우 **delegate**를 활용해서 콜백으로 처리 할 수 있게 하면 될것이다.
- **<T>**형식의 **delegate**도 만들 수 있다.

```
delegate T myDelegate<T>(T a, T b);
```

```
myDelegate<int> Plus_int = new myDelegate<int>(plus);  
myDelegate<float> Plus_float = new myDelegate<float>(plus);  
myDelegate<double> Plus_double = new myDelegate<double>(plus);
```

```
Calculator(11, 22, Plus_int);  
Calculator(3.3f, 4.4f, Plus_float);  
Calculator(5.5, 6.6, Plus_double);
```

# 1. DELEGATE & EVENT

- 체인 이라고 하나의 메소드만 사용할 수 있는 것이 아니라 메소드를 여러개 추가해서 함께 사용할수 있는 기능이 있다. 예를 보자 +=추가되고 -=제거되는데, 추가한 순서대로 차례로 호출된다.

```
myDelegate dele;  
dele = new myDelegate(func0);  
dele += func1;  
dele += func2;
```

```
dele();
```

```
Console.WriteLine();
```

```
dele -= func0;  
dele -= func2;
```

```
dele();
```

```
void func0()  
{  
    Console.Write("첫 번째");  
}
```

```
void func1()  
{  
    Console.Write("두 번째");  
}
```

```
void func2()  
{  
    Console.Write("세 번째");  
}
```

# 1. DELEGATE & EVENT

## Event

- **delegate** 타입을 선언해준 뒤 그대로 **delegate** 변수로 선언할 수 있지만 **event** 변수로도 선언이 가능하지만 **delegate**와 차이점이 있다.
- **delegate**는 **public** 이나 **internal**을 써서 클래스 외부참조가 가능하지만 **event**는 **public**으로 선언하여도 클래스 외부에서 참조가 불가능하다.
- **delegate**는 콜백 전용으로 쓰고 **event**는 특정클래스 안에서 상태변화에 따른 속성변화 같은 용도로 활용하자.

```
delegate void myDelegate(int a);
```

```
//이벤트매니저 클래스
```

```
class EventManager
```

```
{
```

```
    //이벤트 타입으로 선언
```

```
    public event myDelegate eventCall;
```

```
    public void NumberCheck(int num)
```

```
    {
```

```
        //콜백 호출
```

```
        if (num % 2 == 0)
```

```
            eventCall(num);
```

```
    }
```

```
}
```

```
class Event
```

```
{
```

```
    //이벤트에 등록이 될 함수.
```

```
    static void EventNumber(int num)
```

```
    {
```

```
        Console.WriteLine("{0}는 짝수", num);
```

```
    }
```

```
    static void Main(string[] args)
```

```
    {
```

```
        //매니저 생성
```

```
        EventManger eventManger = new EventManager();
```

```
        //이벤트에 delegate를 체인한다.
```

```
        eventManger.eventCall += new
```

```
        myDelegate(EventNumber);
```

```
        for (int i = 1; i < 10; i++)
```

```
            eventManger.NumberCheck(i);
```

```
        Console.ReadKey();
```

```
        Console.ReadLine();
```

```
    }
```


```
}
```

람다식



## 2. 람다식

- **C# 3.0**부터 사용 - **Delegate**보다 간결, 문 형식
- 람다식은 메소드를 단순한 **계산식**으로 표현한 것이다.  
*//싱글라인*

```
int add(int i, int j)  myDelegate1 add = (a, b) => a + b;  
{  
    return i + j;  
}  
myDelegate2 lamda = () => Console.WriteLine("람다식");  
Console.WriteLine("11 + 22 = {0}", add(11, 22));  
lamda();
```

*//다중라인.(delegate가 매개변수를 대입 받는 형식으로만, 값리턴 형식으로 X)*

```
myDelegate Compare = (a, b) =>  
{  
    if (a > b)  
        Console.Write("{0}보다 {1}가 크다", b, a);  
    else if (a < b)  
        Console.Write("{0}보다 {1}가 크다", a, b);  
    else  
        Console.Write("{0}, {1}는 같다", a, b);  
};  
  
Compare(11, 22);
```

# 가비지 컬렉션(GC)

### 3. 가비지 컬렉션(GC)

- 힙 메모리를 차지하는 객체를 생성하게 되면 스택에는 힙 메모리상의 위치를 가리키는 값이 들어가게 됩니다. 스택에서 사라지는 순간 힙 메모리의 루트를 잃어버리는 것이 되기 때문에 가비지 컬렉터는 이것을 체크하고 루트가 없다고 판단하여 쓰레기로 간주해서 버리는 행위를 하게 되는데 이를 가비지 컬렉션이라고 한다.

#### Level

- 힙에는 여러 가지의 객체들이 상주하고 있고 **GC**는 쓰레기인걸 처음부터 알고 있는 것이 아니어서 일단 모든 객체들의 공간을 루트가 없다고 인식하고 시작한다.
- 열심히 객체들의 루트를 찾는다 그러다 찾지 못한 애들은 쓰레기로 판명하고 찾은 애들과 못 찾는 애들을 따로 모아서 정리 해준 뒤 쓰레기들을 정리해서 차곡차곡 순서대로 다시 쌓아준다.
- 쌓여가는 순서대로 일정량 이상이 되면 우선순위의 레벨을 매겨준다.

#### LevelCheck

1. **GC**는 힙 메모리 내에서 **3**단계로 나누어 두고 막 생성된 객체들은 **0**단계에 **0**단계가 가득 차면 쓰레기를 수거하고 그래도 넘어가거나 쓰레기가 없다면 **1**단계로 넘겨주고 다시 **0**단계에 차곡차곡 쌓기 시작한다.
2. **0**단계도 더 이상 쌓을 수 없고 **1**단계도 쌓을 수 없다면 다시 정리를 시작한다 그렇게 정리하고 넘치는 것은 **1**단계를 **2**단계로 **0**단계를 **1**단계로 옮겨둡니다.
3. 단계별 관리가 문제가 되는 이유가 **0**단계에서 **GC**가 발생하면 **0**단계만 정리를 하고 **1**단계에서 **GC**가 발생하면 **0**단계와 **1**단계를 모두 정리하고 **2**단계는 **0**단계와 **1**단계를 모두 일어나게 한다. **2**단계까지 올라가서 **2**단계에서의 **GC**가 많이 발생할수록 전체 **GC**를 가동하게 되는 셈이다. 그렇게 되면 심하게는 **app**의 구동을 멈춘 채 **GC**를 가동하는데 재원을 모두 활용하게 되는 참사가 발생합니다 그렇기 때문에 **GC**를 일어나게 하는 방법을 최대한 방어 하는게 좋다.

### 3. 가비지 컬렉션(GC)

#### 효율적인 코드 작성

1. 객체수가 많아질수록 메모리도 많이 차지하므로 **0**세대가 빨리 포화되면서 뒤로 밀릴 수가 있다.
2. 한번에 너무 큰 객체를 되도록이면 피해라 당연히 클래스 같은걸 내부에 너무 많은 객체들을 포함하지 말자. 부득이하게 쓸 수는 있지만 되도록이면 피하자.
3. 너무 복잡하거나 너무 많은 루트를 만들지 말자. - 참조 관계가 복잡하고 루트가 많은 만큼 **GC**는 파악하는데 시간이 걸리기 때문입니다.
4. 유니티에서 프로파일이라는 걸 보면서 성능상에서 **CG** 메모리라는걸 보면서 이렇게 쌓여가는 메모리를 확인할 수 있다. 이걸 최대한 낮추는게 앞으로의 과제라고 보면된다.