# C# 프로그래밍

---

## 학습목표

- C#에 대해서 이해할 수 있다.
- C#을 이용해 프로그램을 제작할 수 있다.

## 개요

Unity 엔진의 스크립트 언어는 C#이다. 따라서 Unity 엔진을 가지고 여러 가지를 구현하려면 C#에 대해서 잘 이해하고 있어야 한다. 이번 강의에서는 Unity C#을 다루기 위한 기본적인 부분에 대해서 배우게 된다.

### 템플릿과의 차이점

C#의 제네릭은 C++의 템플릿과 동일한 수준의 유연성을 제공하고 있진 않다. 또한 특수화도 템플릿은 마치 매크로와 같이 전처리가 일어나며, 제네릭은 런타임에 해당 타입으로 대체된다. 아래에서 실제 코드 수준에서의 차이점을 알아보자

- 논타입 템플릿 매개 변수를 허용하지 않는다.

```
// in c++
template <size_t N>
class A
{

};
// in c#
class A<int N> // 컴파일 오류
{

}
```

- 명시적 특수화를 지원하지 않는다.

```
// in c++
template <typename T>
class A
{

};
```

```
// 가능
template <>
class A<int>
{

};
// in c#
class A<int> // 컴파일 오류
{

}
```

- 부분 특수화를 지원하지 않는다.

```
// in c++
template <typename T, typename U>
class A
{

};

template <typename T, int>
class A
{

};
// in c#
class A<T, U>
{

}

class A<T, int> // 컴파일 오류
{

}
```

- 타입 파라미터를 제네릭 타입에 대한 부모 클래스로 사용할 수 없다.

```
// in c++
template <typename T>
```

```
class A : public T
{

};
// in c#
class A<T> : T // 컴파일 오류
{

}
```

- 타입 파라미터에 기본 타입을 지정할 수 없다.

```
// in c++
template <typename T = int>
class A
{

};

A<> a; // A<int>로 만들어짐.

// in c#
class A<T = int> // 컴파일 오류
{

}
```

- 제네릭 타입 파라미터는 제네릭이 될 수 없다.

```
// in c++
template <typename T, template <typename> typename C>
class A
{
    C<T> a;
};
// in c#
class A<T, C>
{
    C<T> a;
}
```

- 제약 조건으로 특정 조건을 충족하는 타입만 타입 인자로 받을 수 있도록 할 수 있다.

```
// in c++
template <typename T>
class A
{
    public void foo()
    {
        a.boo(); // T 타입에 boo()가 없을 수 있음
    }

    T a;
};

// in c#
interface IBoo
{
    void Boo();
}
class A<T> where T : IBoo
{
    T a;

    public void Foo()
    {
        // 제약 조건으로 IBoo를 구현한 타입에 대해서만
        // 특수화 가능
        a.Boo();
    }
}
```

## 반복자 메소드

C++에서는 STL 컨테이너를 순회할 때 반복자를 사용한다. 마찬가지로 C#에서도 반복자가 있다. C#에서는 순회할 수 있는 타입인 IEnumerable과 IEnumerable 타입을 순회할 수 있는 IEnumerator가 있다.

C#에서는 이런 타입을 반환하는 특수한 메소드를 만들 수 있다. 이를 반복자 메소드라고 한다. 여기서는 yield문이라는 특수한 문을 사용할 수 있다.

반복자 메소드의 실행은 MoveNext()를 통해 이뤄지며, yield문을 만날 때까지 실행된다. 만약 첫 MoveNext() 실행 후 함수의 끝까지 다다르지 않았다면, 다음 MoveNext()에서는 이전에 끝낸 yield문의 다음 구문부터 실행된다. yield return문에서 반환된 값은 Current에서 접근할 수 있다.

## 대리자 및 이벤트

대리자

대리자(delegate)는 함수 포인터와 비슷한 참조 타입으로 기능은 System.Delegate와 System.MulticastDelegate로부터 받게 된다. 아래는 사용 예시이다.

```
class TimeManager
{
    public static TimeManager Instance { get; }

    // delegate 객체
    // 만드는 방법은 여기를 참고하라
    public SecondElapsedDelegate OnSecondElapsed = logTime;

    // delegate 만들기
    // (Access Specifier) delegate [Return Type] [Identifier]
[Parameter List];
    private delegate void SecondElapsedDelegate();

    private float _totalTime = 0f;
    private float _lastTime = 0f;

    void Update()
    {
        _totalTime += Time.time;

        if (_totalTime - _lastTime >= 1f)
        {
            _lastTime = _totalTime;

            // delegate에 연결된 메소드 전부 호출
            // 혹은 OnSecondElapsed.Invoke();
            OnSecondElapsed();
        }
    }

    private void logTime()
    {
        Console.WriteLine($"1 sec elapsed.");
    }


}
```

```
class Plant
{
    public Plant()
    {
        // -= 및 += 연산자를 이용해 delegate에 메소드 삭제 및 추가 가능
        TimeManager.Instance.OnSecondElapsed -= grow;
        TimeManager.Instance.OnSecondElapsed += grow;
    }
    private void grow()
    {
        Console.WriteLine("자람");
    }
}
```

.NET에는 미리 만들어진 대리자가 있는데 Action과 Func가 그것이다. 둘은 반환형의 유무 차이 밖에 없다.

**C++** 함수 포인터와의 차이점
함수 포인터와 비슷한 컨셉을 갖고 있지만 몇 가지 측면에서 다른 점이 있다.

- 함수 포인터는 어떤 메서드의 주소를 가르키지만, 대리자는 내부적으로 객체의 주소와 메서드 주소를 함께 갖고 있어, 정적 메서드나 인스턴스 메서드 등을 가리지 않고 다룰 수 있다.
- 함수 포인터는 멤버 함수를 가리킬 시, 한 타입에 대해서만 사용할 수 있지만 대리자는 여러 타입의 메소드를 참조할 수 있다.
- 함수 포인터는 단일의 메소드만 참조할 수 있지만 대리자는 여러 개의 메소드를 참조할 수 있다.
- 함수 포인터는 타입 안전성이 없지만, 대리자는 타입 안전성을 갖고 있다.

이벤트
이벤트는 옵저버 패턴을 좀 더 사용하기 쉽게 구현된 기능이다. 클래스 내부의 변경 사항을 구독자에게 알리는 데 사용한다. 아래는 사용 예제다.

```
class TimeManager
{
    public static TimeManager Instance { get; }

    // 이벤트 인수는 해당 이벤트가 발생될 때, 구독자에게 바뀐 상태를 전달하기 위한 데이터다.
    // 보통 EventArgs에서 파생한다.
    public TimeElapsedEventArgs : EventArgs
    {
        public float LastTime { get; set; }
```

```
        public float TotalTime { get; set; }
        public float ElapsedTime { get { return TotalTime - LastTime;
} }

        public TimeElapsedEventArgs(float lastTime, float totalTime)
        {
            LastTime = lastTime;
            TotalTime = totalTime;
        }
    }

    // Event 객체
    // [Access Specifier] event EventHandler (Identifier)
    public event EventHandler<TimeElapsedEventArgs> OnSecondElapsed;

    private float _totalTime = 0f;
    private float _lastTime = 0f;

    void Update()
    {
        _totalTime += Time.time;

        if (_totalTime - _lastTime >= 1f)
        {
            _lastTime = _totalTime;

            // 구독자에게 이벤트 발생했음을 알림
            // 구독자가 없으면 null일 수 있기에 null 체크를 한다.
            OnSecondElapsed?.Invoke(this, new
TimeElapsedEventArgs(_lastTime, _totalTime));
        }
    }
}

class Plant
{
    public Plant()
    {
        // -= 및 += 연산자를 이용해 이벤트에 구독 혹은 해제 가능
        // 무명 메소드를 추가할 수도 있지만, 취소가 필요한 경우라면 안 쓰는
것이 좋다.
        TimeManager.Instance.OnSecondElapsed -= grow;
```

```
        TimeManager.Instance.OnSecondElapsed += grow;
    }

    private void grow(object sender, TimeManager.TimeElapsedEventArgs
args)
    {
        Console.WriteLine($"이전 시간은 {args._lastTime}");
        Console.WriteLine("자람");
    }
}
```

## 참고자료

- C# 문서 - 시작, 자습서, 참조.
- C# 스터디
- The Memory Management Reference