Unity3D 프로그래머라면 알아야 할 최소한의 C#

최흥배
https://github.com/jacking75/choiHeungbae

Unityで覚えるC#

Learn C# in Unity

http://www.slideshare.net/lucifuges/unityc

C#에서의 클래스

- **클래스 정의**

```
using UnityEngine; // 사용할 라이브러리 선언

// MonoBehaviour를 상속하는 NewBehaviourScript 클래스
선언
public class NewBehaviourScript : MonoBehaviour
{
     // 멤버 필드 선언
     public Vector3 velocity;

     // 멤버 메소스 정의
     void Update()
     {
          transform.Translate(velocity * Time.deltaTime);
     }
}
```

멤버 선언에 public을 붙이면 클래스 밖에서도 접근할 수 있다. private를 붙이던가 아무것도 붙이지 않으면 클래스 밖에서 접근 할 수 없다. 내부에 무엇이 있고, 무엇을 할 수 있는지 숨기는 것이 좋다.

C#에서의 클래스

- **클래스가 멤버로 가지고 있는 것**

■**필드** 변수. 보통 바로 멤버 변수를 공개하지 않지만 Unity의 경우 공개 필드로 하지 않으면 인스펙트에 표시 되지 않는다.

■**메소드／생성자／소멸자** 함수. 생성자와 소멸자는 조금 특별한 함수.

■**프로퍼티**
접근자 간이 표기법

■**인덱서** 오브젝트를 배열과 같이 다룰 수 있도록 해주는 기능.

■**이벤트** 처리의 일부를 외부에 위임하거나 어떤 타이밍에서 통지하기 위한 기능.

C#에서의 클래스

- **생 성 자**

  오브젝트의 초기화 처리를 하는 메소드. 단 Unity 는 MonoBehaviour을 상속한 클래스가 생성자를 구현하는 것을 추천하지 않는다. Awake/Start를 사용하도록 하자.

```
public class Foo : Bar
{
      int i;
      // 생성자 정의. 반환값이 없고, 클래스 명과 같은 이름의 메소드로 한다.
      public Foo(int i)
      {
            this.i = i;
      }
      // 같은 클래스의 다른 생성자를 호출한다.
      public Foo() : this(0)
      {
      }
      // 부모의 생성자를 명시적으로 호출하는 방법. 생략하면 부모의 인자 없는 생성자가 호출된다.
      public Foo(float f) : base(f)
      {
      }
}

Foo f = new Foo(1.0f); // 3개의 생성자가 호출 된다.
```

C#에서의 클래스

- **소 멸 자**
  오브젝트의 파괴 처리를 하는 메소드. C#은 GC가 동작하므로 명시적으로 파괴하지 않아도 좋으므로 통상 의식할 필요가 없다.

```
public class Foo
{
    // 소멸자 정의. 반환값, 인수, 접근자 지정도 없다. 클래스 명에 ~을 붙인 이름의 메소드로 한다.
    ~Foo()
    {
    }
}
```

C#에서 파괴 타이밍을 관리하고 싶은 클래스는 IDisposable을 상속하고, Dispose() 메소드를 구현, using 문과 같이 사용한다. 단 Dispose() 구현은 익숙하지 않아서 조금 까다롭다…。

C#에서의 클래스

- **프로퍼티 : 접근자 간이 표기법**

```
public class NewBehaviourScript : MonoBehaviour
{
     // 멤버 프로퍼티 정의
     public Vector3 velocity
     {
          get { return rigidbody.velocity; }
          set { rigidbody.velocity = value; }
     }
     // 위는 아래의 접근자 정의를 간단하게 한 것이다.
     Vector3 GetVelocity()
     {
          return rigidbody.velocity;
     }
     void SetVelocity(Vector3 value)
     {
          rigidbody.velocity = value;
     }
     // 프로퍼티는 보통 멤버 필드와 같도록 접근 할 수 있다
     void Start()
     {
          velocity = Vector3.zero;
     }
}
```

C#에서의 클래스

- **프로퍼티를 좀 더 편리하게 사용**

프로퍼티는 구현을 생략해도OK

```
public class NewBehaviourScript : MonoBehaviour
{
     public Vector3 velocity { get; set; }

     // 위는 아래와 같은 암묵적인 멤버 필드를 가지고 있다고 생각하면 OK
     Vector3 m_velocity;
     public Vector3 velocity
          get { return m_velocity; }
          set { m_velocity = value; }
     }
}
```

프로퍼티는 get/set 마다 접근자 지정을 붙일 수 있다

```
public class NewBehaviourScript : MonoBehaviour
{
     public Vector3 velocity
     {
          get; private set;
     }
}
```

C#에서의 클래스

- **프로퍼티를 좀 더 편리하게 사용**
  읽기만 허용하는 프로퍼티는 set을 생략할 수 있다.

```
public class NewBehaviourScript : MonoBehaviour
{
     public Vector3 velocity
     {
          get { return rigidbody.velocity; }
     }
}
```

C#에서의 클래스

- 인 덱 서

```
public class NewBehaviourScript : MonoBehaviour
{
     int[] data;
     // 인덱서를 정의
     public int this[int index]
     {
          get { return data[index]; }
          set { data[index] = value; }
     }
     // 인덱서를 멤버로 가지는 오브젝트는 배열과 비슷하게 접근 할 수 있다
     void Start()
     {
          NewBehaviourScript nbs = new NewBehaviourScript();
          print(nbs[0]);
     }
}
```

C#에서의 클래스

- **이 벤 트**

  중요한 기능이지만 이해가 어려우므로 뒤에 다시 설명

```
public class Foo
{
     // 델리게이트 정의
     public delegate void OnDamageDelegate(GameObject victim, float amount);
     // 정의한 형의 이벤트를 선언
     public event OnDamageDelegate onDamage;
     // 데미지 처리
     float hp { get; private set; }
     public void DoDamage(float amount) {
          hp -= amount;
          onDamage(gameObject, amount); // onDamage에 등록되어 있는 메소드를 호출
     }
}
// 예를들면 데미지를 먹었던 타이밍에서 이 메소드를 호출 하고 싶은 경우
void PrintDamage(GameObject victim, float amount)
{
     print(victim.name + " got damage " + amount);
}
// 이벤트에 데릴게이트를 등록
Foo f = new Foo();
f.onDamage = PrintDamage;
// 데미지 처리를 호출하면 이벤트 기능으로 PrintDamage 메소드가 호출된다
f.DoDamage(10);
```

C#에서의 클래스

- 클래스 멤버와 인스턴스 멤버

```
public class Foo : MonoBehaviour
{
      // 클래스 필드 정의
      public static int classField = 0;
      // 인스턴스 필드 정의
      public int instanceField = 0;
      // 클래스 메소드 정의
      public static void Hoge() { print("Hoge! " + classField); }
      // 인스턴스 메소드 정의
      public void Mage() { print("Mage! " + classField); }
}

// 클래스 멤버에는 인스턴스가 없어도 접근할 수 있다
Foo.classField = 10;
Foo.Hoge(); // "Hoge! 10"
Foo foo = new Foo();
print(foo.instanceField); // 0
foo.Mage(); // "Mage! 10"
```

C#에서의 클래스

- **클래스 멤버만 가진 클래스**
  모든 멤버가 static한 클래스는 클래스 자체도 static으로 하여 인스턴스를 만드는 것을 명시적으로 막는다.

```
public static class Foo
{
    // 클래스 필드 정의
    public static int classField = 0;
    // 클래스 메소드 정의
    public static void Hoge() { print("Hoge! " + classField); }
    // static한 클래스는 정적 생성자(클래스 생성자)로 초기화 한다
    static Foo()
    {
        classField = 10;
    }
}
```

C#에서의 클래스

- **클래스 수식자**

  sealed

  sealed 수식자를 붙인 클래스는 상속할 수 없다.

- **멤버 필드 수식자**

  const

  실행 중에 값이 변하지 않는디 (변할 수 없다) 필드에 붙여서 값을 바꾸는 코드를 쓰면 컴파일 시에 에러가 발생한다.

  readonly

  의미적으로는 const와 같다. const는 값 타입에만 사용할 수 있지만 readonly는 참조 타입에도 붙일 수 있다

C#에서의 클래스

- **멤버 메소드 수식자**

virtual

overdeir

exernt
외부에서 구현한 메소드를 선언 할 때 DlmportII 속성과 함께 사용. Unity는 네이티브 플러그인 메소드를 호출 하고 싶을 때 사용

C#에서의 클래스

- **속성 (어트리뷰트)**

  C#은 어트리뷰트라는 기능을 사용하여 클래스 자체나 그 멤버에 메타 데이터를 붙일 수 있다. Unity에서는 자신이 만든 클래스에 Serializable 속성을 붙여서 인스펙트에 표시 할 수 있도록 하거나, 에디터에 확장에서 자주 사용한다

```
[ExecuteInEditMode()] // 이 속성을 붙이면 에디터 상에서 비 실행 시에도 Start나 Update를 호출 가능
public class NewBehaviourScript : MonoBehaviour
{
     [System.Serializable()] // 이 속성을 붙이지 않으면 Foo 형 필드가 인스펙터에 표시 되지 않는다.
     public class Foo
     {
          public int hoge;
     }
     public Foo f;
}
```

C#에서의 구조체

- **구조체 정의 사양**

```
public struct Point
{
     // 멤버 필드 선언
     public int x, y;

     // 멤버 메소드 정의
     public Point(int x, int y)
     {
          this.x = x;
          this.y = y;
     }
}
```

기본적으로는 클래스 정의와 같지만 구조체는 클래스와 같은 구현 상속은 할 수 없다. 인터페이스를 상속하여 자신이 구현하는 것은 가능하다. 또 클래스는 참조형이지만 구조체는 값형이라는 것이 다르다.

C#에서의 인터페이스

- **인터페이스 정의 사양**

```
interface ICollidable
{
     // 멤버 메소드 선언(정의는 할 수 없다)
     bool IsCollide(GameObject other);
}

public class NewBehaviourScript : MonoBehaviour, ICollidable
{
     // 인터페이스 구현
     public bool IsCollide(GameObject other)
     {
          if (gameObject.collider == null || other.collider == null)
               return false;
          return IsCollide(gameObject.collider, other.collider);
     }
}
```

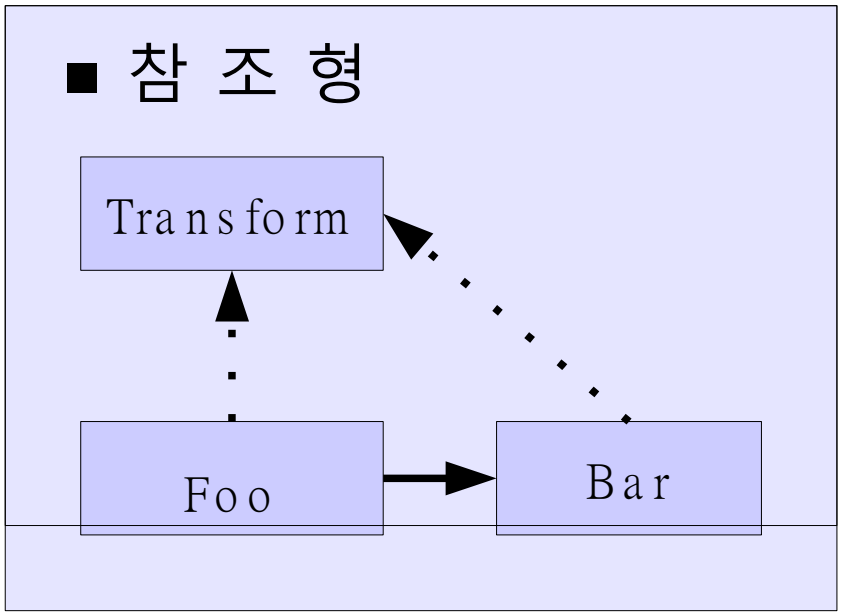
인터페이스는 그것을 상속한 오브젝트가 어떤 멤버 메소드, 프로퍼티, 인덱서, 이벤트를 가질지를 명시 하기 위한 것이다. 멤버 필드는 선언 할 수 없고 구현도 할 수 없기 때문에 단독으로는 사용할 수 없다.

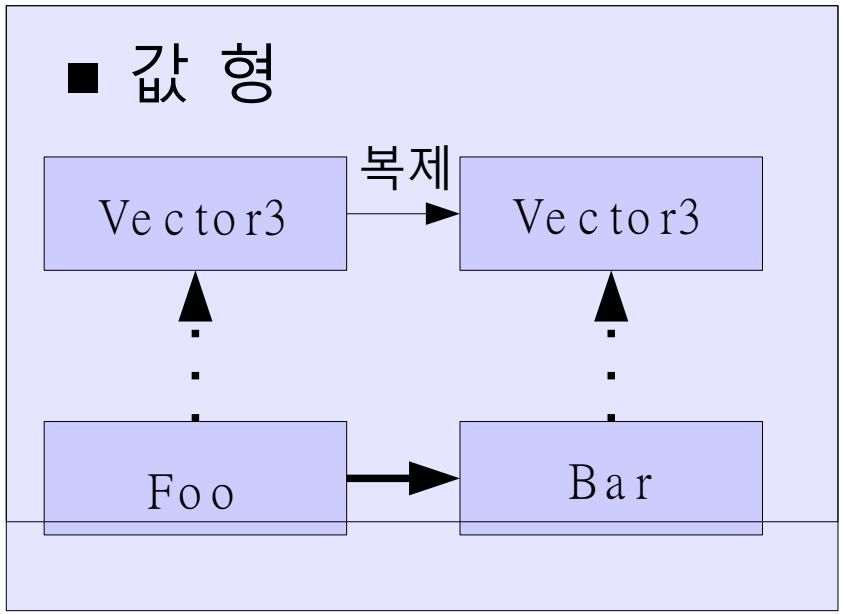
클래스 or 구 조 체

- **참 조 형 과 값 형**

  클래스는 참조형, 구조체는 값형이다. 이 2개의 차이는 메소드 인수나 반환 값으로 오브젝트가 반환 될 때의 행동에 영향이 미친다.

  예를들면 오브젝트 Foo에서 Bar에 변수가 넘겨진 경우…





클래스 or 구 조 체

- **값 형 의 참 조 전 달**
  ref 키워드나 out 키워드를 사용하여 구조체와 같은 값형의 참조를 전달하는 방법이 있다.

```
public class Foo
{
     Vector3 myVector;

     public AddVectorTo(ref Vector3 target)
     {
          target += myVector;
     }
}

public class Bar
{
     Foo foo = new Foo();
     Vector3 myVector;
     public Bar()
     {
          foo(ref myVector);
     }
}
```

클래스 or 구 조 체

- **N u l l 허 용 형**

  값 형의 변수라도 아직 초기화 되지 않았다는 것을 명시하고 싶은 경우가 있다. 이럴 때 Null 허용형을 사용한다.

```
public class Foo
{
     Vector3? myVector; // 변수의 형 뒤에 「？」을 붙이면 Null 허용형이 된다.

     public Vector3 GetVector()
     {
          if (myVector == null) myVec
               tor = Vector3.up;
          return myVector.Value;
     }
}
```

이것은 데이터베이스와 연동을 간단하게 하기 위해 준비된 것으로 Unity 에서 사용할 일은 거의 없다. 만약 이것을 사용하게 된다면 설계에 문제가 있다고 봐야 한다.

클래스 or 구 조 체

- **참 조 형**

  장점

  오브젝트의 크기에 상관없이 전달 비용이 일정. 전달 받은 곳에서 값을 변경하면 전달한 곳에서의 값도 변경

  단점

  GC에서 처리 비용이 높음

- **값 형**

  장점

  GC에서 처리할 때 비용이 낮고, 전달 받은 곳에서 값을 변경하여도 전달한 곳에 영향을 주지 않는다

  단점

  오브젝트의 크기에 비례하여 비용이 증가한다. box 화에 의한 성능 악화를 알기 어렵다

클래스 or 구 조 체

- **박 스 화 · 박 스 화 해 제**

  값형을 object 형으로 변환 하는 것을 박스화, object 형을 원래 값으로 캐스트 하는 것을 박스화 해제라고 하며, 어느쪽도 꽤 비용이 높은 조작으로 최대한 피해야 한다(박스화 되면 인스턴스 사이즈도 증가한다).

```
object o = 1;   // 박스화
int i = (int)o; // 박스화 해제

// ArrayList는 모든 요소를 object 형으로 변화해서 저장하는 컬렉션
ArrayList list = new ArrayList();
list.Add("hoge");   // 그러므로 문자열도 수치도 같은 ArrayList에 저장할 수 있지만…
list.Add(100);      // ←여기에서 박스화 발생！ 알아차리기 힘들다！
```

  뒤에 설명할 제네릭이라는 기능을 사용하면 불필요한 박스화를 줄일 수 있으므로 적극적으로 사용한다.

클래스 or 구 조 체

- **어 느 쪽 을 선 택 할 까 ?**

기본은 클래스를 사용하고, 아래 조건을 만족한 경우에는 구조체를 사용한다(라고 Microsoft 문서에 써여 있다…).

1 · 정수나 부동 소수점과 같은 논리적으로 단일한 값을 표시
2 · 인스턴스 사이즈가 16바이트 미만인 경우
3 · 상속 에 의해 행동을 바꿀 필요가 없을 때
4 · 빈번하게 박스화 할 필요가 없다

예를드렴「이차원 좌표를 나타내는 Point」와 같은 것은 구조체에 적합하다.

C#의 편리한 기능

- **Enum : 일련의 정수를 하나로 모은 형**

```
public class Foo
{
	const int DamageTypeSlash = 1;
	const int DamageTypeBlunt = 2;
	const int DamageTypeExplosion = 3;

	// enum 으로 하는 쪽이 보기 좋다
	enum DamageType
	{

		Slash = 1,
		Blunt,
		Explosion,
	}

	DamageType damageType;

	// Enum 형의 각 값에는 「형 타입.이름」이라는 형식으로 접근
	한다.
	public Foo() { damageType = DamageType.Blunt; }

	//
	int GetDamageTypeID() { return (int)damageType; }
}
```

C#의 편리한 기능

- **namespace : 이름공간**

```
namespace MyClasses //관련된 오브젝트를 하나의 공간에 모아 놓아 검색성을 올리고, 이름 충돌을 피한다
{
     public class Foo { }
     public class Bar { }
}
// 이름 공간 안의 요소에는 이와 값이 접근한다
MyClasses.Foo f;
```

```
// 다른 필드에서도 같은 이름공간을 선언할 수 있다
namespace MyClasses
{
     public class Hoge { }
}
```

```
// 파일 선두에서 using 디렉티브를 사용하면 접근 시 이름 공간을 생략 할 수 있다
using MyClasses;

Foo f;
```

C#의 편리한 기능

- **캐스트, 형 정보 이용**

  캐스트(형의 명시적인 변환)는 C 방식과 비슷. 또 is 연산자, as 연산자라는 형 조사의 간단한 방법이 있다.

```
// 보통 캐스트는「(목적 형)」으로 한다
Collider c = (Collider)gameObject.AddComponent("BoxCollider");

// is 연산자로 인스턴스 형을 지정한 것인지 조사할 수 있다
if (c is BoxCollider)
     boxCollider = c as BoxCollider; // 인스턴스를 지정한 형으로 캐스트. 할 수 없다면 null

// 형 정보를 추출해서 자신이 판정
if (c.GetType() == typeof(BoxCollider))
     print("OK");

// 제너릭을 사용하면 캐스트는 필요 없다
c = gameObject.AddComponent<BoxCollider>();
```

C#의 편리한 기능

- **var**

  형이 명확한 로컬 변수는 형명을 var로 가능

```
public class Foo
{
	void Hoge()
	{
		// 이것을
		Transform t = transform;
		// 이렇게 써도 좋다
		var t = transform;
		// 이 1 문으로는 var가 실제 어떤 형이 될지 모르므로 에러
		var i;
		i = 0;
	}

	public var i = 0; // 이것도 에러. var를 사용할 수 있는 것은 로컬 변수만
}
```

C#의 편리한 기능

- **파티셜 클래스와 파티셜 메소드**
  하나의 클래스 정의를 복수의 파일에 나누어서 쓸 수 있다

```
public partial class Foo
{
    void Hoge()
    {
    }
}
```

```
public partial class Foo
{
    void Mage()
    {
    }
}
```

```
// 물론 클래스 Foo는 Hoge, Mage 양쪽 메소드를 멤버로 가지고 있다.
Foo f = new Foo();
f.Hoge();
f.Mage();
```

파티셜 클래스의 메소드에 partial수식자를 붙이면 구현을 다른 파일에서 정의하여도 정의 하지 않아도 (!) 좋다.

```
public partial class Foo
{
    partial void Hoge();
}
```

C#의 편리한 기능

- **확장 메소드**

```
// 본래는 아래처럼 사용해야 하지만
Instantiate(prefabObject);

// 확장 메소드를 사용하면
static class GameObjectExtension
{
     // static 메소드 첫번째 인수에 앞에 this를 붙이면 확장 메소드
     public static Object Instantiate(this GameObject prefab)
     {
          return Instantiate(prefab);
     }
}

// GameObject 클래스가 Instantiate() 라는 인스턴스 메소드를 가진 것처럼 쓸 수 있다
prefabObject.Instantiate();
```

C# 제어구문

if, for, while, do ~ while문

타 언어와 비교해서 특별한 차이는 없다. continue, break, goto 등도 사용할 수 있다.

switch 문

case에 문자열을 사용할 수 있고, 기본적으로 break을 생략할 수 없는 등 타 언어와 비교해서 조금 특별하다.

```
switch (hoge)
{
case 0:
case 1:
     DoSomething();
     break;
case 2:
     DoSomething2();
     goto case 0; // 어떤 처리를 한 후 어떻게하든 다른 case로 가고 싶은 경우는 goto를 사용
default:
     DoNothing();
     break;
}
```

C# 제어구문

- **foreach문**

  배열이나 컬렉션 등의 모든 요소에 대해서 반복해서 처리를 한다.

```
foreach (AudioSource a in GetComponents<AudioSource>())
{
    a.Stop();
}
```

뒤에 설명할 Linq와 조합하면 for문이 거의 불필요해진다

```
// 어태치 되어 있는 AoudioSouce 재생 중인 것에 대해 Stop()을 호출하는 예
foreach (var a in from a in GetComponents<AudioSource>() where a.isPlaying select a)
{
    a.Stop();
}
```

루프 문 안에서 if문을 사용하여 분기하면 루프 처리 자체가 길어지므로 가능하면 Linq로 처음부터 리스트를 정형화 해서 foreach 문으로 처리하는 것이 보기 좋다.

C# 제어구문

- **try ~ catch ~ final 문**

  C#에서 예외처리를 제어하는 구문이지만 Unity가 예외를 거의 의식하지 않아도 좋은 설계로 되어 있어서 거의 사용하지 않는다.

```
StreamReader sr = null;
try
{
     sr = File.OpenText(path);
     string s = "";
     while ((s = sr.ReadLine()) != null)
          print(s);
}
catch (FileNotFoundException e)
{
     print("File Not Found!");
}
finally
{
     sr.Dispose();
}
```

C# 제어구문

- **using 문**

  이름 공간 생략이나 형의 별명 정의에도 using 이라는 키워드를 사용해서 헷갈리기 쉽지만 메소드 내에 나오는 using문은 Dispose() 호출을 보증하는 것. UnityEngine에는 IDisposable을 구현한 클래스가 없으므로 그다지 사용하지 않는다…

■using 판

```
using (StreamReader sr = File.OpenText(path))
{
     string s = "";
     while ((s = sr.ReadLine()) != null)
          print(s);
}
```

■ 비 using 판

```
StreamReader sr = null;
try
{
     sr = File.OpenText(path);
     string s = "";
     while ((s = sr.ReadLine()) != null)
          print(s);
}
finally
{
     sr.Dispose();
}
```

제너릭

- **제 너 릭 이 라 는 것 은**
  불필요한 캐스트를 줄이고, 컴파일 시에 형 조사를 할 수 있고, 코드가 최적화 되어 성능을 좋게 해주는 아주 멋진 기능이다.

```
BoxCollider c;

// 비 제너릭 판 ( 길어서 보기 않좋다 )
c = (BoxCollider)gameObject.AddComponent(typeof(BoxCollider)); c
= gameObject.AddComponent(typeof(BoxCollider)) as BoxCollider;

// 제너릭 판
c = gameObject.AddComponent<BoxCollider>();
```

그러나 Unity는 제너릭판 메소드를 그다지 준비하고 있지 않다. ;;;;

제너릭

- 직 접 제 너 릭 대 응 하 기

```
// 예를들면 이런 오브젝트를 제너릭에 대응하는 경우
Object FindObjectOf(Type type)
{
	return FindObjectOfType(type);
}

// 형 파라미터 이름을 T로 하는 것은 관습
T FindObjectOf<T>() where T: Component // 형 파라미터에 제한을 둘 수 있다(임의)
{
	return (T)FindObjectOfType(typeof(T));
}

// 위의 제너릭 메소드 호출 방법
var renderer = FindObjectOf<Renderer>(); // 형이 확실하므로 var를 사용할 수 있고 캐스트 불필요!

// 형 파라미터에 제한을 걸어서 아래는 컴파일 시 에러
int i = FindObjectOf<int>(); // int는 Component를 상속하고 있지 않다
```

제너릭

• 비 제 너 릭 컨 테 이 너 사 용 하 지 말 자 ! ! !

박스화에서 설명 했듯이 ArrayList 등의 비 제너릭은 몰래 성능을 악화시키고 본래 필요 없는 캐스트가 증가해서 좋지 않다(캐스트가 증가하면 버그도 증가). 명확한 이유가 없는한 System.Collections.Generic 이름공간에 있는 제너릭판 컨테이너를 사용하자.

```
using System.Collections.Generic;

List<int> list;
list.Add(100); // 박스화 없음!
list.Add("hoge"); // 컴파일 시 에러!
int i = list[0]; // 물론 박스화 해제도 없다!
```

델리게이트

- **메 소 드 를 참 조 할 수 있 는 형**

메소드의 참조를 보존해 두고 뒤에 호출하거나 복수의 메소드를 하나의 변수에 넣어서 호출 할 수 있는 기능.

```
// 델리게이트 형 정의
delegate void SomeDelegate(float f);
// 정의한 형의 변수를 선언
SomeDelegate someDelegate;

// 예를들면 이런 메소드를 앞의 변수에 대입할 수 있다
void Hoge(float value) { } // 형만 맞으면 인수의 이름은 같지 않아도 괜찮다
someDelegate = Hoge;

// 인스턴스 메소드도 대입할 수 있다
public class Foo
{
     public void Bar(float amount) { }
}
Foo f = new Foo();
someDelegate += f.Bar; // 「+=」로 추가 대입 할 수 있다

// 델리게이트 변수에 저장 되어 있는 메소드를 호출（Hoge와 f.Bar가 순서대로 불려진다）
someDelegate(1.0f);
```

델리게이트

• 이 미 준 비 되 어 있 는 델 리 게 이 트

인수도 없고 반환 값도 없는 델리게이트나 인수를 하나 취하고 bool 값을 반환하는 델리게이트 등 잘 사용하므로 System 아래에 처음부터 준비해 두고 있다.

```
System.Func<TResult> // TResult Func()

System.Predicate<T> // bool Predicate(T value)

System.Action<T> // void Action(T value)

System.Action<T1, T2> // void Action(T1 p1, T2 p2)
```

이벤트

• 델리게이트를 밖에서 실행하지 않도록 한다

예를들면 데미지를 먹은 타이밍을 델리게이트를 사용하여 다른 오브젝에 통지하고 싶은 경우 델리게이트를 노출하면 밖에서 실행 할 수 있게 되어버려서 곤란하다.

```
public class Foo
{
	// 델리게이트 형 정의
	public delegate void OnDamageDelegate(GameObject victim, float amount);
	// 정의한 델리게이트 형의 변수 선언(private로 해버리면 등록조차 할 수 없으므로 의미 없다)
	public OnDamageDelegate onDamage;
	// 데미지 처리
	float hp { get; private set; }
	public void DoDamage(float amount) {
		hp -= amount;
		onDamage(gameObject, amount);
	}
}
Foo f = new Foo();
f.onDamage(1.0f); // 실제로 데미지를 먹지 않았는다 onDamage를 직접 호출 해버린다
```

이런한 때에는 이벤트라는 기능을 사용하자

이벤트

• 이벤트는 밖에서는 추가와 삭제만 할 수 있다

델리게이트를 노출할 필요 없이 이벤트로 하면 추가와 삭제는 클래스 밖에서 할 수 있지만 실행은 클래스 내에서만 할 수 있다. 이것으로 이상한 타이밍에서 호출 되는 것을 방지.

```
public class Foo
{
     // 델리게이트 형 정의
     public delegate void OnDamageDelegate(GameObject victim, float amount);
     // 델리게이트 형 변수에 「event」를 붙인다
     public event OnDamageDelegate onDamage;
     // 데미지 처리
     float hp { get; private set; } publi
     c void DoDamage(float amount) {
          hp -= amount;
          onDamage(gameObject, amount);
     }
}
Foo f = new Foo();
f.onDamage(1.0f); // 컴파일 에러
f.onDamage += PrintDamage; // 추가는 문제 없이 할 수 있다.
```

람다식

- 이 름 없 는 메 소 드

```
public class Foo
{
     // 델리게이트 형 정의
     public delegate void OnDamageDelegate(GameObject victim, float  amount);
     // 델리게이트 형 변수에「event」를 붙인다.
     public event OnDamageDelegate onDamage;
     /* 생략 */
}
Foo f = new Foo();
// 이벤트에 람다식을 대입
f.onDamage = (victim, amount) => print(victim.name + " got damage "  + amount);
// 이것은 아래와 같다
void PrintDamage(GameObject victim, float amount)
{
     print(victim.name + " got damage " + amount);
}
f.onDamage += PrintDamage;
```

람다식

List 클래스 등은 조건 판정용 델리게이트(리스트 요소 하나를 인수로 하여 bool 값을 반환하는 메소드)를 인수로 잡는 Find 메소드 등을 멤버로 가지고 있다.여기에 람다식을 넘기는 것도 가능.

```
List<GameObject> gameObjects = new List<GameObject>(src);

// 이름에「Enemy」를 포함한 오브젝트를 찾는다
GameObject go = gameObjects.Find(o => o.name.Contains("Enemy"));

// gameObjects 리스트 중 Y 좌표가 0 미만 오브젝트를 모두 삭제
gameObjects.RemoveAll(o => o.transform.x < 0);
```

LINQ

- **데이터베이스 용으로 도입된 기능이지만**

  대량의 데이터 중에서 필요한 것을 찾거나, 복수의 리스트를 합치거나, 데이터 뭉치를 정형화 하기 위한 기능. 잘 사용하면 코드를 깔끔하게 할 수 있다.

```
// 예를들면 이런 루푸는
var renderers = GetComponentsInChildren<Renderer>();
foreach (var r in renderers)
{
     if (r != null && r.material != null) r
          .material.mainColor = Color.red;
}

// LINQ로 이렇게 할 수 있다
var materials = from r in GetComponentsInChildren<Renderer>()
                where r != null && r.material != null
                select r.material;
foreach (Material m in materials)
     m.mainColor = Color.red;
```

  루프 안에서 분기하는 것보다 리스트를 정형화한 쪽이 실수 하기가 어렵다.

LINQ

주목하고 싶은 멤버만을 추출하여 익명 클래스를 만들어서 반환하면 관심 없는 멤버에 접근해서 버그를 만드는 것을 방지.

```
var materialSets = from Renderer r in FindObjectsOfType(typeof(Renderer))
                   where r.material != null && r.sharedMaterial != null
                   select new { r.material, r.sharedMaterial };
foreach (var m in materialSets)
{
     m.material.mainTexture = texture; m.sha
     redMaterial.mainTexture = texture;
}
```

단 가벼운 처리는 아니므로 사용할 때 주의가 필요

코루틴

- **C#의 이터레이터 구문을 유용하고 있다**

C#의 yield return의 본래 목적은 이터레이터를 가볍게 구현하기 위한 것이다. 도중까지 처리하고 일시 메소드를 빠져서 다음에 또 그기에서(메소드 도중) 재개할 수 있다 라는 특성이 멀티 태스크 처리에 어울려서 Unity에서는 코루틴에 사용되고 있다. 코루틴은 C#의 기능은 아니다.

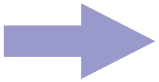
```
// 본래 사용 방법
IEnumerator Count10()
{
     for (int i = 10; i >= 0; --i)
          yield return i;
}

foreach (var i in Count1to10())
{
     print(i); // 10, 9, 8, 7...
}
```

코루틴

- **컴파일 시에 암묵적으로 클래스가 만들어진다**

yield 문이 포함된 메소드는 실은 컴파일 시에 암묵의 열거용 클래스로 변환되고 있다.

```
IEnumerator CountUp()
{
      print(1);
      yield return null;
      print(2);
      yield return null;
      print(3);
}
```



```
class CountUpEnumerator : IEnumerator
{
      int state = 0;
      bool MoveNext()
      {
            switch (state++)
            {
            case 0:
                  print(1); re
                  turn null;
            case 1:
                  print(2); re
                  turn null;
            case 2:
                  print(3);
            }
      }
}
IEnumerator CountUp()
{
      return new CountUpEnumerator();
}
```

코루틴

- **코 루 틴 은 MoveNex를 을 매 프 레 임 호 출**

아마 GameObject가 지금 실행 중의 코루틴의 리스트를 가지고 있고 매 프레임 이것들의 MoveNext() 메소드를 호출하고 있을 뿐. 비슷한 처리를 직접 쓸 수 있다. 코루틴 마다 정지 재개를 관리하고 싶은 경우는 이쪽이 유연성이 더 높다.

```
public class Foo : MonoBehaviour
{
     List<IEnumerator> coroutines = new List<IEnumerator>();
     void Start()
     {
          coroutines.Add(MyCoroutine1());
          coroutines.Add(MyCoroutine2());
     }
     void Update()
     {
          coroutines.RemoveAll(c => !c.MoveNext());
     }
}
```

C# 학습 소개

- http://www.csharpstudy.com/Default.aspx
- http://www.aladin.co.kr/shop/wproduct.aspx?isbn=8998139340

시작하세요! C# 프로그래밍 - 기본 문법부터 실전 예제까지 | 위키북스 프로그래밍 & 프랙티스 시리즈 5 무료배송

정성태 (지은이) | 위키북스 | 2013-11-07

[구매 금액별 특별사은품 : 명화 부채+스프링 노트+스프링 분철 1천원 쿠폰]

정가 : 35,000원

판매가 : 31,500원 (10%, 3,500원 할인)

알라딘 Magic 롯데카드 최대 30% 할인

마일리지 : 1,750점(5%) + 멤버십(3~1%) + 5만원이상 구매시 2,000점

↳ 비국내도서 상품 포함 구매에 한함

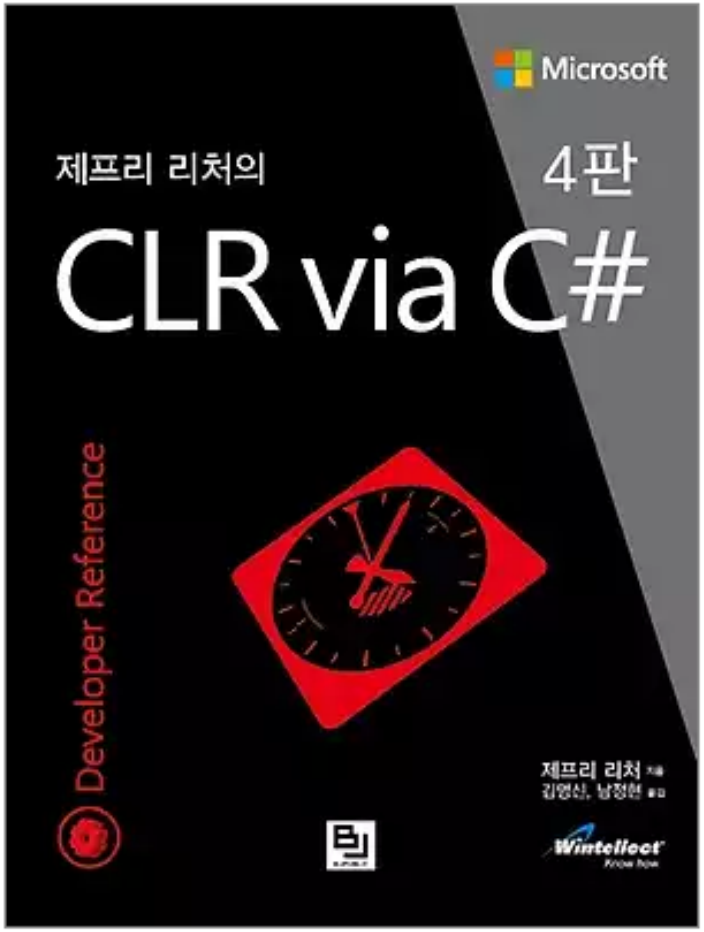
복불복! 5만원 이상 구매 후 행운의 마일리지에 응모하세요! >

반양장본 | 920쪽 | 240*188mm | 1665g | ISBN : 9788998139346

배송료 : 무료

절판 품절센터도서 재판매 알림신청 보관함 담기

C# 학습 소개



제프리 리처의 CLR via C# 4판

- 저 : 제프리 리처 , 남정현
- 역 : 김명신
- 출판사 : 비제이퍼블릭

- 판매가 : ~~50,000원~~ → **45,000원**

CLR과 .NET 개발을 철저하게 마스터하기 위한 완벽 가이드선도적인 프로그래밍 전문가로서 오랫동안 마이크로소프트 .NET팀을 컨설팅해온 제프리 리처와 함께 CLR, C#, .NET 개발의 난해함을 깊이 있게 조명하고 마스터해 보자. 이 책을 통해서 안정적이며, 신뢰할 수 있고, 빠르게 동작하는 응용프로그램과 컴포넌트를 개발하기 위한 실용적인 통찰력을 얻...

상세정보 > 북카트 담기 >