

Unity Essencial Grammer

작성자 : 이 동 욱

메일 : dongwookRaynor@gmail.com

Index

벡터 연산 기초(Vector)

평행이동과 좌표계, 부모 자식 설정(Space, Translate)

회전과 쿼터니언(Quaternion)

인스턴스화(Instantiate)

제네릭(Generic)

배열

리스트(List)

레이캐스트(Raycast)

오버로드(Overload)

코루틴Coroutine)

함수 작성, 스코프(Functions, Scope)

OOP : 클래스와 오브젝트

정적 변수와 함수

싱글톤(Singleton)

상속(Inheritance)

다형성(Polymorphism)

오버라이드(Override)

인터페이스(Interface)

추상클래스(Abstract Class)

프로퍼티(Property)

유니티 이벤트(Unity Event)

델리게이트(Delegate)이벤트(Evnet)

액션과 람다함수(Action, Lambda)

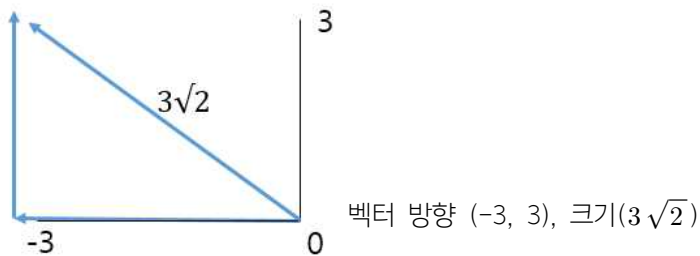
벡터 연산 기초(Vector)

Vector의 크기는 방향, 길이, 속도를 나타낸다. Vector는 Vector3(x, y, z), Vector2(x, y) 등 벡터에 있는 구성요소가 같은 개수의 원소만 가지면 Vector라고 한다.

Vector는 길이와 방향을 가진다.



예시) (-3, 3)는 어떤 점에서 왼쪽으로 3, 위쪽으로 3을 이동했다는 것과 같다.



벡터는 길이와 방향만을 가진다. 시작점은 무시한다. 어떤 방향으로 얼마나 갈 수 있는지를 나타낸다.

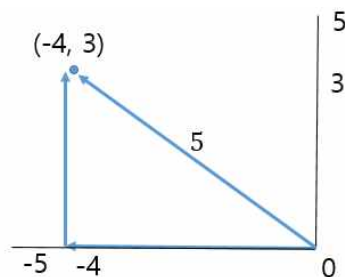
예시) (2, 3) 이란 내가(2, 3) 위치에 있다, 내 위치에서 “상대적으로” (2, 3)만큼 갔다.

벡터의 표현

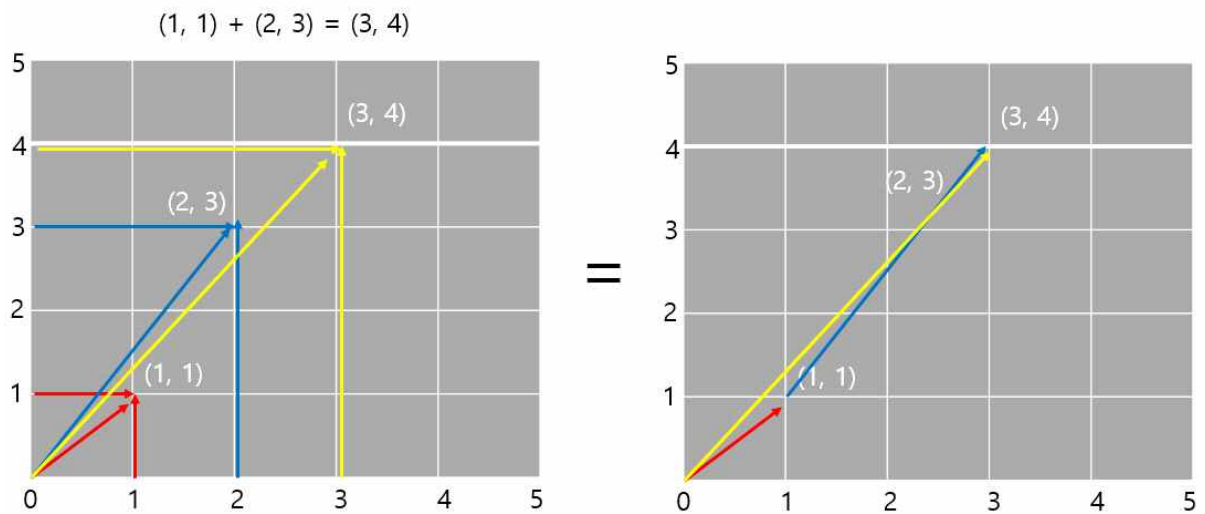
1. 벡터는 길이와 방향을 가진다.
2. 절대 좌표 (x, y)는 실제 (x, y)에 위치해있다는 뜻이다. 원점 (0, 0)에서 시작해 (x, y)만큼 떨어져 있다.
3. 상대좌표 (x, y)는 현재 위치에서 (x, y)만큼 떨어져있다.

벡터는 Unity에서 new 키워드를 사용한다. Vector3 x = new Vector3(x, y, z);

예시)



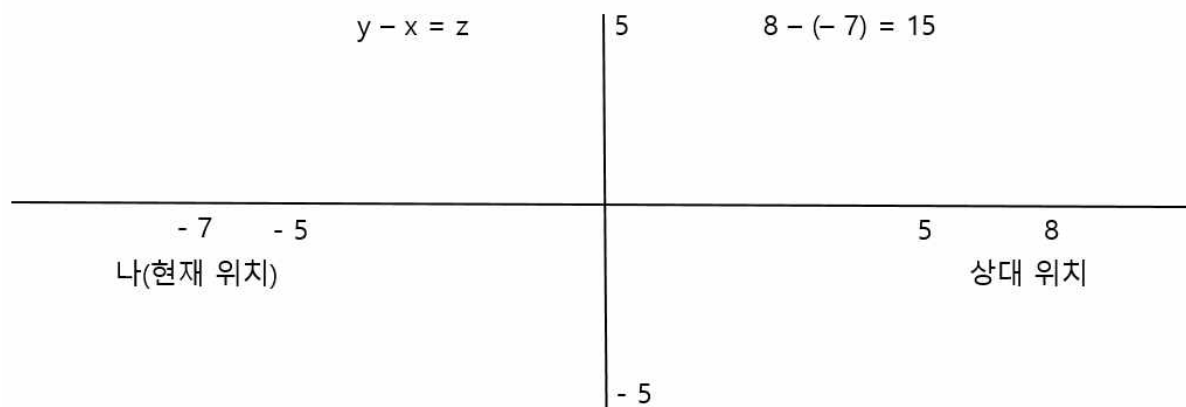
$\sqrt{3^2 + (-4)^2} = 5$ = 실제 이동한 거리(선의 실제 길이) = 벡터의 크기(magnitude)라고 부른다.



(2, 3)은 (0, 0)에서부터 오른쪽으로 2, 위쪽으로 3만큼 이동하여 (2, 3)가 되었다,
 (3, 4)는 (1, 1)에서부터 오른쪽으로 2, 위쪽으로 3만큼 이동하여 (3, 4)가 되었다.

$(1, 1) + (2, 3) = (3, 4)$ 이라는 것은 (1, 1)(현재 위치)에서 시작하여 (2, 3)만큼 이동 했다는 뜻이다.

벡터를 더한다는 것은 어떤 위치에서 더한 벡터만큼 이동하는 것을 말한다. Vector를 쓰는 이유는 내 위치에서 얼마를 가야 상대방에 도달 할지를 알 수 있기 때문이다.



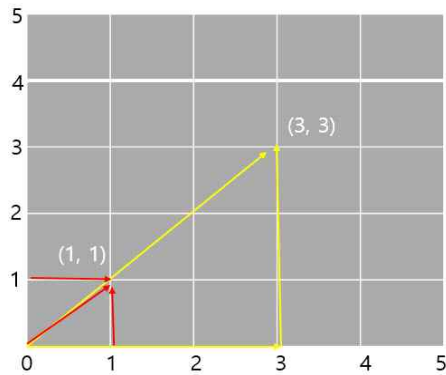
내가 상대 까지 도달하기 위한 방법은 (상대방 위치) - (내 위치) = 간격이 된다.

$8 - (-7) = 15$ 에서 부호는 중요하지가 않다. 부호는 방향을 나타낼 뿐이다. 내 위치에서 15만큼 가면 상대에게 도달 할 수 있다는 뜻이 된다.

Y	-	X	=	Z
목적지	-	현재 위치	=	현재위치에서 얼마나 가야 목적지로 갈 수 있는지 알 수 있는 거리

다시 말해 벡터는 어디서 시작한지는 중요하지 않다.

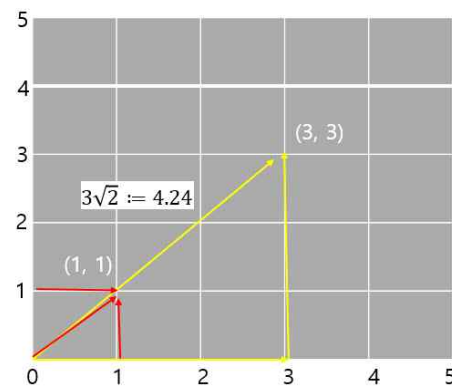
벡터는 곱셈도 가능하다.



방향 \times 크기 = $(1, 1) \times 3 = (3, 3)$ 크기가 다를 뿐 방향은 같다.

$(1, 1)$ 의 크기는 $\sqrt{2}$, $(3, 3)$ 의 크기는 $3\sqrt{2}$

벡터는 길이와 방향을 쪼개서 생각할 수 있다.(매우 중요!)



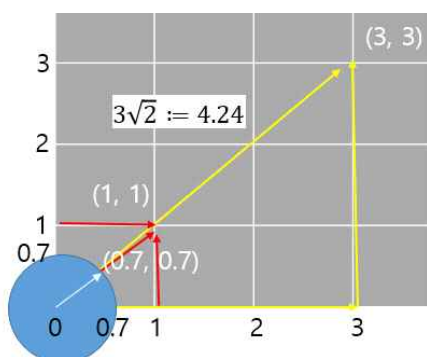
Vector는 곱셈으로 가를 수 있다. $(3, 3) = (1, 1) \times 3$

문제는 4.24를 나타내려면 어떻게해야할까? 실제 크기는 4.24이며 방향은 $(3, 3)$ 이다. 벡터를 온전히 **방향 \times 크기**로 나누어서 나타내고 싶다. 그렇게 될 경우 벡터의 크기(방향의 크기)는 1 이 되어야 한다. 1×4.24 가 되면 온전히 4.24가 되기 때문이다.

벡터를 속도 4.24(실제 크기)를 나타내려니 온전히 **방향 \times 크기**로는 식이 맞지 않는다. 그래서 어떤 수를 넣어도 그 수가 똑같이 나오도록 항등원을 만들어 적용했다. 예시) $1 \times 3 = 3$, $1 \times 9 = 9$

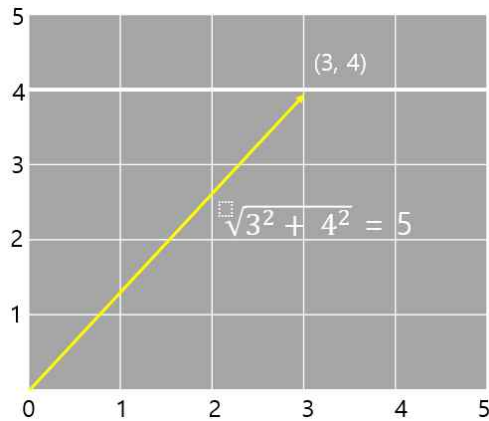
$$\begin{array}{lclclcl}
 \text{방향} & (3, 3) & = & (1, 1) \times 3 & = & \text{방향} \times \text{크기} \\
 & & & \blacktriangledown & & \\
 \text{크기} & 3\sqrt{2} (4.24) & = & \sqrt{2} \times 3 & \neq & \sqrt{2} \times 4.24 \\
 & & & \blacktriangledown & &
 \end{array}$$

즉 $(3, 3)$ 을 어떤 수 \times 4.24로 표현하고 싶다. 여기서 어떤 수는 **방향벡터**라고 하며 길이가 1이다. 벡터는 길이가 달라도 방향은 같을 수 있다. $(0.7, 0.7)$ 은 $(3, 3)$ 과 방향이 같다. 그리고 길이는 1이 된다. 즉, 단위로 사용 가능하다.



$$\begin{array}{llll} \text{방향} & (3, 3) & = & (0.7, 0.7) \times 4.24 \quad \text{성립} \\ & \blacktriangledown & & \\ \text{크기} & 3\sqrt{2}(4.24) & = & 1 \times 4.24 \end{array}$$

아까처럼 (1, 1)을 기준으로 하면 길이가 대략 1.41이 나오게 되면 1.41×4.24 는 원하는 결과를 출력하지 못한다. 예제를 적용해보자. (3, 4)은 오른쪽으로 3, 위쪽으로 4 이 된다.



실제 길이는 5, 이것을 방향 x 크기로 표현한다.

(3, 4) = 어떤 벡터 x 5, 어떤 벡터 = 길이가 1인 방향벡터(대략 0.7, 0.7)이다.

$$\sqrt{3^2 + 4^2} = \sqrt{(0.7)^2 + (0.7)^2} \times 5 = 5$$

이 처럼 방향과 크기를 갈라 표현하면 원하는 크기만큼 갈 수 있다. $10 = (0.7, 0.7) \times 10$, 이 예시 또한 (3, 4)와 방향이 같지만 10이라는 거리만큼 나아가고 갈 수 있다. $(3, 4) \times 2 = (0.7, 0.7) \times 10$, 특정 기준이 없었기 때문에 (3, 4)의 실제 길이를 구해 x 2를 해야 했다. 10을 구하고 싶은데 (3, 4)는 10이 아니기 때문에 $(3, 4) \times 2$ 같이 표현을 해야 했다. 그래서 방향과 크기를 갈라 크기를 온전히 표현하기 위해 방향벡터를 만든 것이다. 이 방향벡터를 normalized Vector 라고 부르며 일반적으로 적용 가능하다고 해서 붙었다.

평행이동과 좌표계, 부모 자식 설정(Space, Translate)

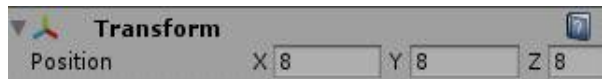
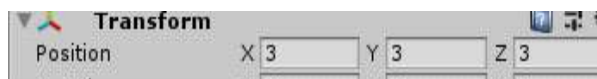
오브젝트의 위치를 어떻게 변화시키는 것에 대해 말한다. 오브젝트에 Transform 컴포넌트가 붙어 있으면 오브젝트를 선택할 수 있게 한다. transform 함수는 자주사용하기 때문에 스크립트에서 transform을 바로 사용하면 스크립트를 적용한 오브젝트에 대해서 조정하게 된다.

임의의 위치 (1, 1, 1) 을 가진 targetPosition의 값을 transform.position이 갖는다. 즉, 오브젝트의 위치는 (1, 1, 1) 위치가 된다.

```
public class Mover : MonoBehaviour {  
    private void Start()  
    {  
        Vector3 targetPosition = new Vector3(1,1,1);  
        transform.position = targetPosition;  
    }  
}
```

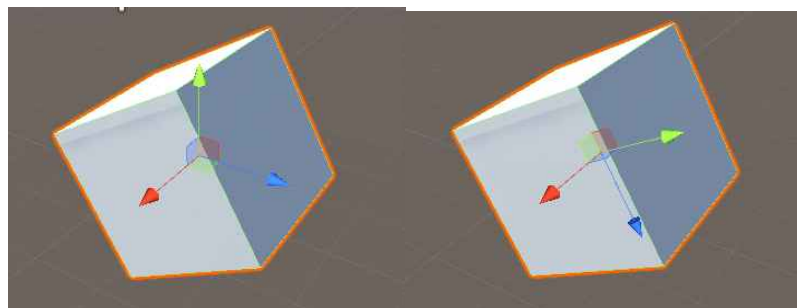
이제는 임의의 위치(3,3,3)를 지정해 놓는다. 벡터를 더한다는 의미는 현재 위치에서 특정 거리만큼 상대적으로 이동한다는 것이라고 했다. 즉, 현재위치에서 가고 싶은 거리만큼 이동하면 그 거리만큼 평행 이동을 하게 된다.

```
public class Mover : MonoBehaviour {  
    Vector3 movePosition = new Vector3(5, 5, 5);  
    private void Start()  
    {  
        transform.position = transform.position + movePosition;  
    }  
}
```



정확히 movePosition의 크기만큼 위치가 이동된 것을 볼 수 있다. 하지만 저렇게 + 연산을 사용하지 않고도 내장 기능을 이용하여 사용가능하다. transform.Translate(movePosition);을 사용하면 위와 같이 사용할 수 있다.

만약에 물체가 회전 상태라면 어떻게 평행 이동을 할까. 회전하는 동시에 좌표축이 변경된다. 좌표계(Space)에는 Local space, Global space가 있다. Local은 부모와 자신을 기준으로 한 것이며 Global은 게임 세상을 기준으로 하는 절대적인 좌표축이다. 아래는 cube를 x축으로 60도 회전한 경우이다.



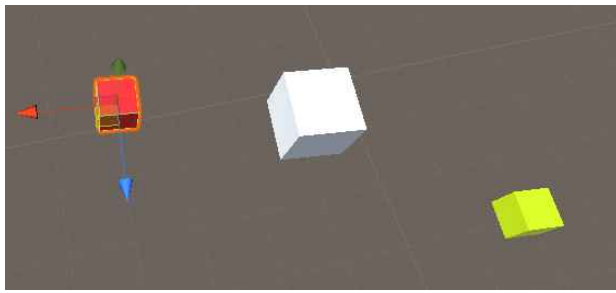
왼쪽은 Global이며 오른쪽은 Local이다. Global 경우 cube는 회전했지만 게임세상은 회전하지 않고 그대로이기 때문에 회전축은 변하지 않는다. 하지만 Local 경우 오브젝트 좌표축을 따르기 때문에 좌표축 또한 x축으로 60도 회전하는 것을 볼 수 있다.

평행이동은 기본적으로 Global 이 아닌 Local space를 기준으로 한다. 하지만 코드 상에서 옵션을 주어 변경할 수 있다. 매개변수를 추가하여 게임 기준, 오브젝트 기준으로 변경할 수 있다.

```
transform.Translate(movePosition,Space.World);
```

```
transform.Translate(movePosition, Space.Self);
```

그렇다면 부모자식 관계에서는 어떤 식으로 적용이 될까? (1, 1, 1)위치의 cube에 자식 2개를 추가한다. 이 자식 중하나의 위치는 cube로부터 (3, 1, 0) 떨어져 있고, 나머지 하나는 (-2, 1, 0) 만큼 떨어져있다. 만약 (3, 1, 0) 좌표의 자식이 부모와 떨어질 경우 어떻게 될까? 자식의 좌표는 (4, 2, 1)이 된다. 즉, 부모자식 관계에 있지 않았을 때의 그 위치는 원점(0, 0, 0)으로부터(4, 2, 1)만큼 나아간 위치이다. 하지만 부모자식 관계일 경우 부모의 위치(1, 1, 1) 기준하여 (3, 1, 0)만큼 만가면 해당 위치에 있을 수가 있다. 즉, $(4, 2, 1) = (1, 1, 1) + (3, 1, 0)$ 이 되는 것이다. transform은 Local space를 기반으로 한다. 자신의 부모에 상대적이고 자신의 회전에 상대적이다. 부모자식 관계가 있는 오브젝트들은 부모의 위치로부터 상대적으로 위치가 정해진다. 하지만 부모자식 관계가 아닌 일반 오브젝트인 경우 Local space기반임에도 상대적인 기준이 없기 때문에 Local = Global 이 된다.



scale(크기)또한 같다. 부모자식 관계 인 경우 부모가 크기가 2배로 커지면 자식도 2배로 커진다. 하지만 상대적으로 scale은 원래 크기에서 변하지 않는다. 상대적으로 부모와 달라진 것이 없기 때문이다. 부모자식 관계를 없앴을 경우 실제 크기가 2배로 변경 되었다는 것을 알 수 있다.

Scale	X 0.47874	Y 0.47874	Z 0.47874	Scale	X 0.95748	Y 0.957480	Z 0.95748
-------	-----------	-----------	-----------	-------	-----------	------------	-----------

이제는 Local space와 Global space를 구분해서 코드에서 위치를 지정하는 것을 볼 것이다. cube를 생성하고 위치를 (0, 0, 0)을 지정, sphere를 자식으로 만들어 x축으로 3을 이동한(3, 0, 0)으로 지정한다. 이 때 sphere의 위치는 Local space를 기준으로 한다. 이 때 만약 cube가 x축으로 -3 만큼 간다면 어떻게 될까? cube의 좌표는(-3, 0, 0)이며 sphere는 상대적으로 cube에서 x축으로 3만큼만 떨어져있기 때문에 그대로 (3, 0, 0)이다. 하지만 부모 자식관계를 파기한다면 Global 위치가 적용되어 sphere의 위치는 (0, 0, 0)이 된다. sphere에 MakePosition이란 스크립트를 붙인다. 씬 이 시작될 때 sphere를 (0, 0, 0) 으로 보내 보겠다. 씬 을 시작하면 (3, 0, 0)이었던 sphere의 위치가 (0, 0, 0)가 될까? 정답은 그대로이다. 기억해야 하는 것은 Inspector에서 보는 위치는 Local space 값이다. sphere을 cube에서 빼내면 위치가 (0, 0, 0) 인 것을 확인 할 수 있다. 즉, 이미 sphere가 (0, 0, 0)이었다. 코드를 실행 전 이미 sphere는 (0, 0, 0)이기 때문에 위치가 그대로인 것이다. 즉, 코드 상에서 transform.position = new Vector3(0,0,0); 는 Global space를 수정하는 하는 것이다. transform.localPosition = new Vector3(0, 0, 0);을 하면 Local space의 위치를 변경하게 되며 부모자식 관계를 없애고 보면 위치는(-3, 0, 0)이 되는 것을 확인 할 수 있다. 이 방식은 scale과 Rotation에도 적용할 수 있다.





회전과 쿼터니언(Quaternion)

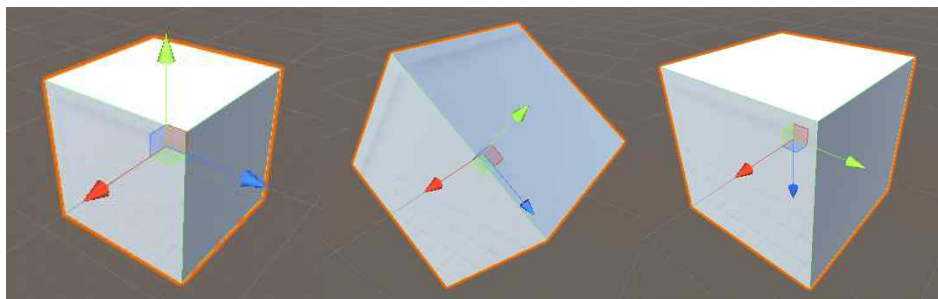
cube를 각 축의 방향에 따라 60도 회전을 하자. x, y, z를 지정하기 때문에 Vector3통해 지정하는 것처럼 보인다. 하지만 실제 각도는 Quaternion을 통해서 지정이 된다. Quaternion은 굉장히 복잡하기 때문에 Unity에서는 Vector3로 Quaternion을 지정하도록 만들었다. SetRotation 스크립트를 cube에 적용하여 각도를 지정해보자. 아래처럼 지정을 하면 예러가 나는 것을 확인 할 수 있다. 유니티에서는 Vector3를 Quaternion으로 바꾸는 기능이 있다. 이를 사용해 보자.

```
public class setRotation : MonoBehaviour {
    void Start () {
        transform.rotation = new Vector3(30, 30, 30);
    }
}
```

Vector3.Vector3(float x, float y, float z) (+ 2 오버로드)
Creates a new vector with given x, y, z components.
암시적으로 'UnityEngine.Vector3' 형식을 'UnityEngine.Quaternion' 형식으로 변환할 수 없습니다.

Quaternion 클래스의 Euler란 함수를 사용하여 Vector를 넣어주면 Quaternion 데이터 타입으로 바뀌서 저장할 수 있다. Euler(오일러)는 x, y, z Vector3을 통해 공간의 회전을 표시할 수 있는 체계를 만든 수학자이다. 실행을 하면 원하는 각도가 적용된다. x, y, z Vector3을 통해 공간의 회전을 표시할 수 있는 체계를 Euler 각이라고 부른다.

물체를 회전 할 때 x축(빨간색)으로 회전을 하는 것은 x축을 기준으로 회전을 한다는 것이다. 그리고 물체가 회전을 하면 물체의 좌표축 또한 변하는 것을 알 수 있다. Euler각을 사용하는 예시에는 문제가 있었다. 회전하기 전의 좌표축과 회전한 후의 좌표축이 겹치는 축이 존재한다면 그 정보가 소실되는 오류가 있었다. 보는 것과 같이 물체를 x축 방향으로 90도를 회전하니 y축이 z축이 되었고 z축은 y축이 되었다. 이렇게 될 경우 겹친 정보 중 하나가 소실되어 버린다. 이러한 현상을 **짐벌락 현상**이라고 부른다. Euler각에서는 x, y, z 각을 동시에 회전하는 것이 아니라 순차적으로 회전을 하게 된다. 어떤 축이 90도로 회전해버리면 다른 축과 겹쳐버려 다음 회전할 축이 어떤 축을 기준으로 회전을 해야 할지 알 수가 없게 된다. 두 축이 하나의 축으로 겹쳐버려 회전정보를 나타낼 수가 없게 된다. 즉, 90도회전은 Vector3으로 표현을 못하기 때문에 Quaternion을 사용해야한다, 아래는 각각 0, 30, 90도를 회전한 그림이다.



3차원 정보를 축이 하나 사라져 2차원 정보로 밖에 표현을 못했다. 옛날 전투기 시뮬레이션 경우 90도를 표현할 수가 없었기에 89.99..도로 표현했다고 한다. Quaternion은 4차원을 사용하며 (x, y, z, w)축을 사용한다, Quaternion은 매우 어려운 내용이므로 설명을 하지 않겠다. Quaternion 사용방법은 다음과 같다.

```
public class setRotation : MonoBehaviour {

    void Start () {

        Quaternion newRotation = Quaternion.Euler(new Vector3(30, 45, 60));
        transform.rotation = newRotation;
    }
}
```

Quaternion에는 여러 편의 기능이 있다. Quaternion.LookRotation()은 방향을 넣어주면 해당 방향으로 바라보는 기능이 있다.

```
Quaternion Look = Quaternion.LookRotation(new Vector3(0, 1, 0));
transform.rotation = Look;
```

다음 기능은 물체를 설정해주면 그 물체를 쳐다보게 하는 기능이다. 상대방의 위치에서 나의 위치를 빼면 현재위치에서 상대방까지의 방향과 거리정보가 나오게 된다.

```
public class setRotation : MonoBehaviour {

    public Transform targetTransform;

    private void Update()
    {
        Vector3 direction = targetTransform.position - transform.position;
        Quaternion Look = Quaternion.LookRotation(direction);
        transform.rotation = Look;
    }
}
```

다음기능은 Lerp라는 기능이다. 2개의 값이 주어지면 중간 값을 리턴 하는 함수이다.

```
public class setRotation : MonoBehaviour {

    void Start () {

        Quaternion aRotation = Quaternion.Euler(new Vector3(30,0,0));
        Quaternion bRotation = Quaternion.Euler(new Vector3(60, 0, 0));

        Quaternion targetRotation = Quaternion.Lerp(aRotation,bRotation,0.5f);
        transform.rotation = targetRotation;
    }
}
```

30도 60도 사이의 값을 리턴을 하는데 3번째 매개변수 0.5f는 이 두 각도 사이의 정 중간 값에 해당한다. 즉 30도는 0, 60도는 1, 45도는 0.5에 해당하며 이는 퍼센트로 수치를 나타낸다. 수치는 0 ~ 1로 나타낸다.

다음은 회전한 상태에서 추가로 회전하는 것을 알아본다. transform.Rotate는 현재 각도에서 지정한 만큼 회전한다. cube가 75도 변경 된다.

```
Quaternion targetRotation = Quaternion.Euler(45,0,0);
transform.rotation = targetRotation;
transform.Rotate(new Vector3(30,0,0));
```

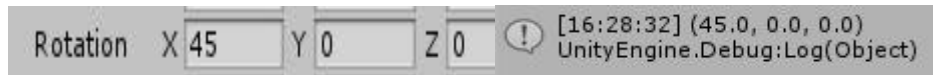
다음 방법은 Vector3각을 가지고 Vector3끼리 더한 다음 최종적으로 Quaternion에 적용하는 방법이다. 먼저 cube의 각을 특정 각도로 먼저 지정할 한다.

```
Quaternion originalRotation = transform.rotation;

Vector3 originalRotationInVector3 = originalRotation.eulerAngles;

Debug.Log(originalRotationInVector3);
```

eulerAngles는 Quaternion정보를 Vector3로 반환한다. 반환한 Vector3정보를 콘솔 창에 띄우면 처음에 지정해놓은 cube의 각도를 볼 수 있다.



그리고 Vector3로 받아온 정보와 원하는 각도를 더해 Vector3변수에 저장하여 회전을 한다.

```
Quaternion originalRotation = transform.rotation;
Vector3 originalRotationInVector3 = originalRotation.eulerAngles;
Debug.Log(originalRotationInVector3);
Vector3 targetRotationVec = originalRotationInVector3 + new Vector3(30,0,0);
Quaternion targetRotation = Quaternion.Euler(targetRotationVec);
transform.rotation = targetRotation;
```



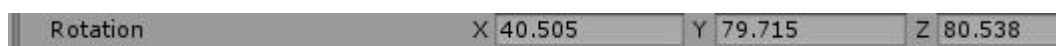
현재 각을 Quaternion으로 받아와 그 값을 Vector3로 다시 반환한다. Vector3끼리 다시 연산을 한 뒤 Quaternion으로 반환 후 각도를 적용한다.

다음 방법은 Vector3를 거치지 않고 Quaternion 끼리 더하는 방법이다. Quaternion은 두 Quaternion끼리 곱하면 된다. 더하는 것이 아니라 왜 곱하 나면 Quaternion은 행렬로 이루어져 있기 때문이다. cube를 0도로 지정해 놓고 30도와 45도를 더하면 75도가 된다.

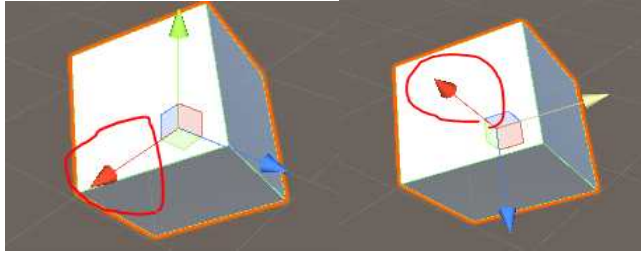
```
Quaternion originalRotation = Quaternion.Euler(new Vector3(30,0,0));
Quaternion plusRotation = Quaternion.Euler(new Vector3(45, 0, 0));
Quaternion targetRotation = originalRotation * plusRotation;
transform.rotation = targetRotation;
```

마지막으로 주의 사항이다. cube를 회전할 때 2가지 방법이 있다. 한 번에 회전하는 것이 있고 나눠서 회전하는 방법이 있다. cube가 현재 30도, 45도, 60도 회전하고 x축으로 30도 회전하면 60도, 45도, 60도가 될 것이다.

```
transform.Rotate(30,0,0);
```



하지만 의외의 전혀 이상한 값이 나온다. 한 번에 회전하는 것과 2번 나누어서 회전하는 것은 완전히 다른 이야기다. cube의 원래 각도는 30도 60도 45도 이다. 씬 을 기준으로 회전을 했다면 60도가 된다. 하지만 현재 나 자신의 상태를 기반(Local Space)으로 30도를 회전하였기 때문이다.



첫 번째는 Global, 두 번째는 Local. Global이 아닌 Local로 회전했기 때문에 다른 값이 나온다. 만약 의도한 각도가 나오지 않았을 때는 각도가 Local 기준을 변경이 되었는지 확인을 해야 한다.

참고자료

- 짐벌락과 오일러각

<http://hoodymong.tistory.com/3>

- MAX를 활용한 짐벌락 설명

<https://www.youtube.com/watch?v=yxMQKsab5TQ&feature=youtu.be>

인스턴스화(Instantiate)

씬 상에 존재하는 물체가 아니라 미리 만들어진 오브젝트를 실시간을 생성하는 것을 말한다. 오브젝트를 언제든지 사용할 수 있도록 프리팹으로 만들어 놓는다. Instantiate를 할 때 매개변수는 생성할 오브젝트, 위치, 각도 등이 있다. Instantiate는 찍어낸 오브젝트를 리턴해 줄 수 있다. Instantiate를 할 때 타입의 제한이 없다. GameObject instance 에게 반환을 할 수도 있고 Rigidbody instance도 가능하다. GameObject로 선언을 했다면 Rigidbody를 사용하려면 다시 instance의 Rigidbody를 가지고 와야 한다.

```
public class Spawner : MonoBehaviour {  
    public GameObject target;  
    public Transform spawnPosition;  
    void Start () {  
        GameObject instance = Instantiate(target,spawnPosition.position,spawnPosition.rotation);  
        instance.GetComponent<Rigidbody>().AddForce(0,1000,0);  
        Debug.Log(instance.name);  
    }  
}
```

반면 생성자체를 Rigidbody 타입으로 한다면 오브젝트의 역할을 하면서 바로 Rigidbody를 사용가능 하다.

```
public class Spawner : MonoBehaviour {  
    public Rigidbody target;  
    public Transform spawnPosition;  
    void Start () {  
        Rigidbody instance = Instantiate(target,spawnPosition.position,spawnPosition.rotation);  
        instance.AddForce(0,1000,0);  
        Debug.Log(instance.name);  
    }  
}
```

제네릭(Generic)

예를 들어 입력 값을 출력하는 함수를 만든다고 하자. 특정한 데이터 타입을 받게 되는데 만약 그 이외의 다른 타입을 입력하고 싶다면 이름이 같고 오버로딩을 사용하여 함수를 정의하거나 다른 함수를 만들면 된다. 하지만 Generic을 사용하면 여러 가지 타입을 한 번에 대응할 수 있다.

```
public void Print(int InputMessage)
{
    Debug.Log(InputMessage);
}
```

이 경우 정수형 타입에만 입력을 받을 수 있다. Generic 타입을 사용하면 함수이름에<원하는 타입 이름>을 넣고 매개변수 타입도 원하는 타입 이름 형식을 사용하면 된다. 아래처럼 입력에 따라 원하는 형태로 사용이 가능하다. Unity에서 가장 Generic을 많이 볼 때가 GetComponent<>()이다. 어떤 타입에 대해서 GetComponent를 실행할지 명시하고 사용했다. Rigidbody라면 GetComponent<Rigidbody>(); 이고 특정한 스크립트에 대해서라면 GetComponent<스크립트 명>();을 사용했었다. Generic은 클래스에 대해서도 사용한다.

```
public class Util : MonoBehaviour {
    public void Print<MyType>(MyType InputMessage)
    {
        Debug.Log(InputMessage);
    }

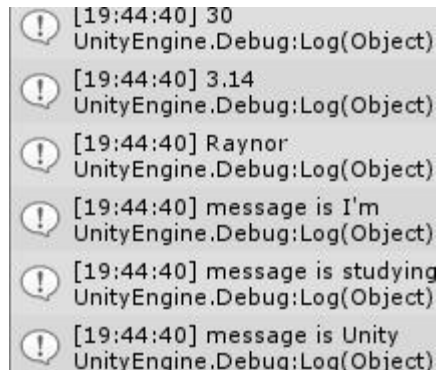
    private void Start()
    {
        Print<int>(30);
        Print<float>(3.14f);
        Print<string>("Raynor");

        container container = new container();
        container.message = new string[3];

        container.message[0] = "I'm ";
        container.message[1] = "studying ";
        container.message[2] = "Unity ";

        for(int i = 0; i<3; i++)
        {
            Debug.Log("message is " + container.message[i]);
        }
    }
}

public class container
{
    public string[] message;
}
```



The screenshot shows the Unity console logs for the Util class. It displays six log entries, each preceded by a warning icon (exclamation mark in a circle). The logs are as follows:

- [19:44:40] 30
UnityEngine.Debug:Log(Object)
- [19:44:40] 3.14
UnityEngine.Debug:Log(Object)
- [19:44:40] Raynor
UnityEngine.Debug:Log(Object)
- [19:44:40] message is I'm
UnityEngine.Debug:Log(Object)
- [19:44:40] message is studying
UnityEngine.Debug:Log(Object)
- [19:44:40] message is Unity
UnityEngine.Debug:Log(Object)

단순한 클래스이지만 문자열 형태만 출력하는 것을 보았다. Generic을 사용하면 다음과 같이 사용가능하다.

```
public class Util : MonoBehaviour {
    private void Start()
    {

        Container<string> container1 = new Container<string>();
        container1.message = new string[3];

        container1.message[0] = "I'm ";
        container1.message[1] = "studying ";
        container1.message[2] = "Unity ";
    }
}
```

```

for(int i = 0; i<3; i++)
{
    Debug.Log("message is " + container1.message[i]);
}

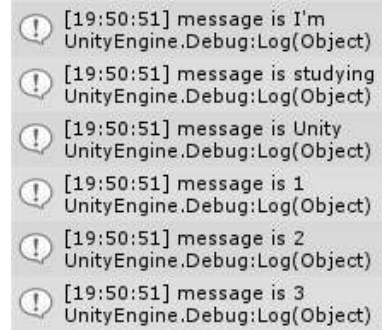
Container<int> container2 = new Container<int>();
container2.message = new int[3];

container2.message[0] = 1;
container2.message[1] = 2;
container2.message[2] = 3;

for (int i = 0; i < 3; i++)
{
    Debug.Log("message is " + container2.message[i]);
}
}

public class Container<MyType>
{
    public MyType[] message;
}

```



The screenshot shows the Unity console with six log messages, each preceded by a yellow warning icon. The messages are as follows:

- [19:50:51] message is I'm
UnityEngine.Debug:Log(Object)
- [19:50:51] message is studying
UnityEngine.Debug:Log(Object)
- [19:50:51] message is Unity
UnityEngine.Debug:Log(Object)
- [19:50:51] message is 1
UnityEngine.Debug:Log(Object)
- [19:50:51] message is 2
UnityEngine.Debug:Log(Object)
- [19:50:51] message is 3
UnityEngine.Debug:Log(Object)

배열의 확장형인 List도 이 같이 Generic 방식으로 구현된 것을 알 수 있다. Generic은 이 뿐만 아니라 특수한 타입에 대해서 제한을 줄 수가 있다. 정의된 <MyType>에 대해서 특정한 타입 만 사용이 가능하게 할 수 있다.

배열

배열은 하나의 변수에 값을 연속적으로 다룬다. 연속적으로 다룰 수 있는 공간이다.

```
public class helloArray : MonoBehaviour {  
    void Start () {  
        int[] scores = new int[10];  
  
        scores[0]=1;  
        scores[1]=2;  
        scores[2]=3;  
        scores[3]=4;  
        scores[4]=5;  
        scores[5]=6;  
        scores[6]=7;  
        scores[7]=8;  
        scores[8]=9;  
        scores[9]=10;  
  
        for(int i = 0; i < 10; i++) { Debug.Log("score["+i+"] = "+scores[i]); }  
    }  
}
```

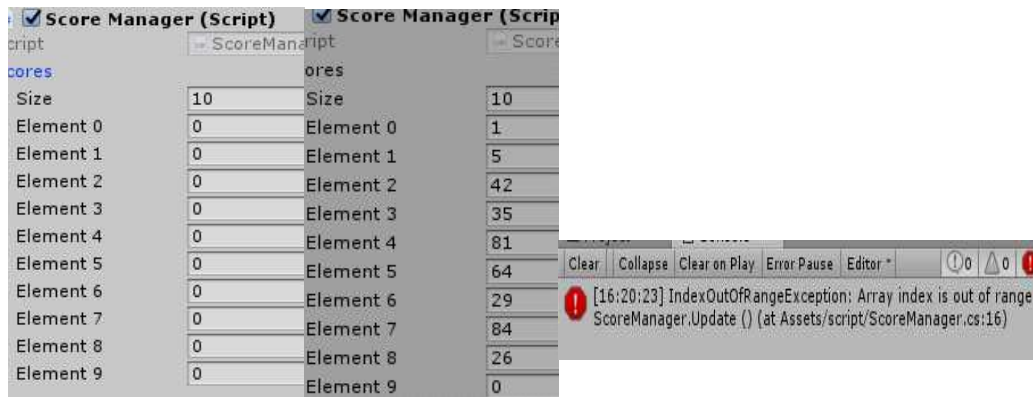
만약 score[10] =11; 할 경우 out of index라는 error 메시지가 출력.

리스트(List)

배열과 비슷하게 여러가지 항목을 한 번에 다룰 수 있지만 배열과 달리 크기와 오브젝트를 다루는데 있어 배열보다 자유롭다.

한번 클릭 할 때마다 숫자가 랜덤하게 들어가는 것을 구현하려고 한다.

```
public class ScoreManager : MonoBehaviour {  
    public int[] scores = new int[10];  
    private int index = 0;  
  
    // Update is called once per frame  
    void Update () {  
        if(Input.GetMouseButtonDown(0))  
        {  
            scores[index] = Random.Range(0,100);  
            index++;  
        }  
    }  
}
```

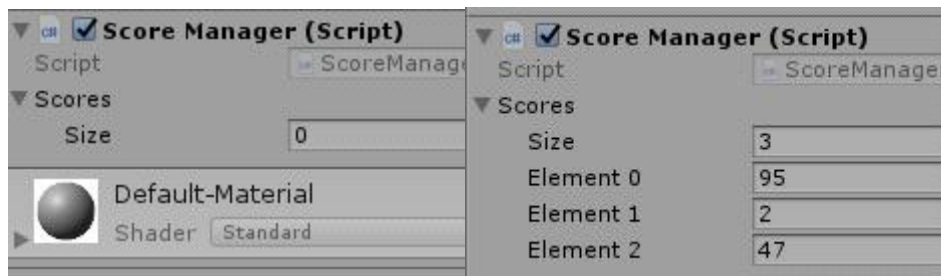


마우스 왼쪽 버튼을 클릭을 하니 배열에 랜덤한 값이 들어가게 된다. 하지만 만약 10개의 방이 다 찼는데 다시 클릭할 경우 에러가 발생한다. OutOfRange 방의 개수를 넘어가서 생기는 index 문제이다. 즉, 배열은 index의 개수가 정해지면 그 후 개수를 변화시킬 수가 없다.

배열의 경우 배열의 개수를 변화시키고 싶으면 score = new int[변화시킬 수]를 하여 기존의 배열을 없애고 새로 만드는 법밖에 없다.

List를 사용하기 위해서는 반드시 using.System.Collections.Generic;을 해주어야한다. List를 비롯한 제네릭과 관련된 것들은 C#에 나중에 추가된 것들이다. List는 여러 가지 타입에 적용되기 때문에 <원하는 데이터 타입> 형식으로 사용한다. 그리고 List는 나중에 추가된 클래스 형태이기 때문에 new 키워드를 사용하여야 한다. List는 순번을 명시해줄 필요가 없다. List는 index의 개수가 실시간으로 변화한다.

```
public class ScoreManager : MonoBehaviour {  
    public List<int> scores = new List<int>();  
    private void Update()  
    {  
        if(Input.GetMouseButtonDown(0))  
        {  
            int randomNumber = Random.Range(0,100);  
            scores.Add(randomNumber);  
        }  
    }  
}
```



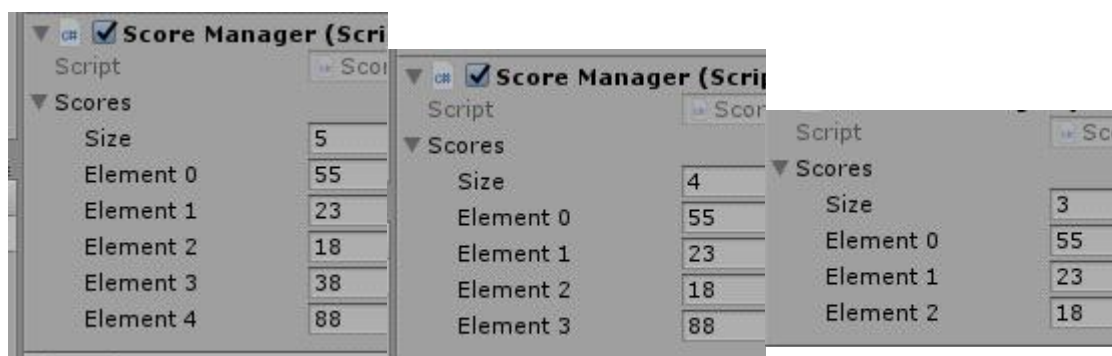
index의 개수를 지정해주지 않았기 때문에 처음에는 0개 였지만 클릭한 횟수만큼 방이 추가되는 것을 볼 수 있다.

List는 또한 index의 개수를 감소시킬 수도 있다. 배열 경우 특정 index의 값을 없앨 수 없다. 없앨 경우 기존 위치에 쓰레기 값이 들어가게 된다. 배열은 index가 지정되면 배열 자체를 없앨 수 있지만 특정 index를 제거할 수는 없다.

```
public class ScoreManager : MonoBehaviour {
    public List<int> scores = new List<int>();
    private void Update()
    {
        if(Input.GetMouseButtonDown(0))
        {
            int randomNumber = Random.Range(0,100);
            scores.Add(randomNumber);
        }

        if(Input.GetMouseButtonDown(1))
        {
            scores.RemoveAt(3);
        }
    }
}
```

RemoveAt()은 특정 index값을 제거하는 역할을 한다.



오른쪽 마우스 버튼을 누를 경우 3번째 index값이 사라지는 것을 볼 수 있고 4번째 index가 3번째로 밀리는 것을 볼 수있다. 컴퓨터에서는 0번째 부터시작하기 때문에 실질적으로는 4번째다. List개수가 총 3일 때는 오른쪽 마우스를 눌렀을 경우 OutOfRange 메시지가 출력된다.

이를 List의 중요한 특징인 trim이라고 한다. trim은 여백을 잘라주는 것을 말한다. 단, 유의할 점은 List가 index를 remove한다는 의미는 값 그자체가 제거되는 것은 아니다. 정수나 실수 등 단순 데이터 타입인 경우는 상관없지만 Call by Reference가 적용되는 class와 오브젝트관계에서는 변수가 오브젝트를 가리키는 방식이기 때문에 List에서 index가 제거된다는 것은 방이 제거된다는 것이지 해당하는 오브젝트가 제거 되는 것은 아니라는 것을 명심해야한다.

```

public class ScoreManager : MonoBehaviour {

    public List<int> scores = new List<int>();

    private void Start()
    {
        int num0 = 45;
        int num1 = 60;
        int num2 = 75;

        scores.Add(num0);
        scores.Add(num1);
        scores.Add(num2);

        //List index에서는 [45][60][75]가 있다.

        Debug.Log(scores[0]);
    }
}

```

List의 0번째 수인 45를 출력되는 것을 볼 수 있다.

만약 scores.RemoveAt(1);Debug.Log(scores[1]); 이렇게 될 경우 어떻게 될까? 60이 출력되는 것이 아닌 60이 제거되고 num2가 밀려오면서 75가 출력된다. 혹은 scores.Remove(60)을 통해 직접 값을 제거가능하다. 만약 index내에 같은 값이 여러 개가 있다면 가장 먼저 있는 index값부터 제거하게 된다.

List는 배열과 같이 특정 index의 값을 바꿀 수 있다. score[0] = 50;을 하여 45 -> 50으로 바꿀 수 있다. List는 배열보다 상위호환이며 배열과 같이 쓰이고 있다. 하지만 배열보다 무거운 단점이 있다. List를 쓰는 이유는 오브젝트의 총 개수를 알 수 없을 때 사용한다. 실시간으로 오브젝트가 계속 추가되거나 지워질 때 어떠한 변수공간에 담아 추적을 해야 할 때 사용한다.

List의 내용을 모두 삭제 하고 싶을 때는 가장 앞 index를 계속 빼주는 방법이 있다. 뒤에 있던 값이 앞으로 밀려들어오기 때문에 앞에 있던 모든 값을 빼주면 결국엔 모두 삭제가 된다.

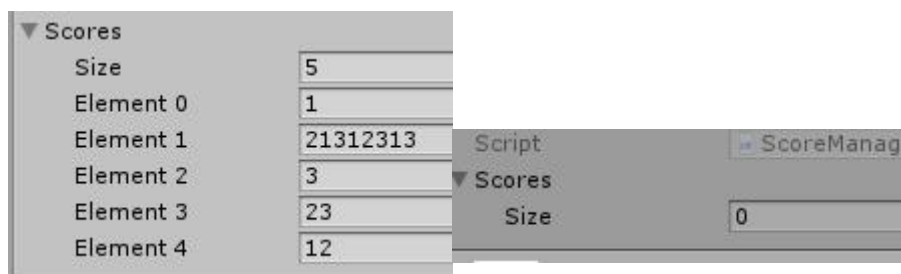
```

public class ScoreManager : MonoBehaviour {

    public List<int> scores = new List<int>();

    private void Start()
    {
        while (scores.Count > 0)
        {
            scores.RemoveAt(0);
        }
    }
}

```



scores의 개수가 1개라도 남아 있을 경우 while문은 계속 돌아간다. 값을 넣어 놓고 play를 하면 모두 삭제 되는 것을 볼 수 있다.

레이캐스트(Raycast)

1인칭 시점에서 응시하고 있는 사물을 다른 곳으로 옮기는 예시를 만들어보자. Raycast는 특정방향으로 광선을 발사하여 물체가 광선에 닿는지 검사하는 방식을 말한다. Vector3 rayOrigin은 광선이 시작될 원점, 즉 카메라상의 위치를 알려주면 실제 게임 내부에서의 위치를 알려주는 기능을 한다. 화면 위치 상, 바라보고 있는 지점에 점을 찍으면 그 점은 월드 기준으로 위치를 알려준다.

Debug.DrawRay() 경우 위치, 방향, 색깔을 정해주면 씬 뷰에서 어디를 향해 보고 있는지를 광선을 통해 나타내준다. 개발자 용도로 사용하고 있다.

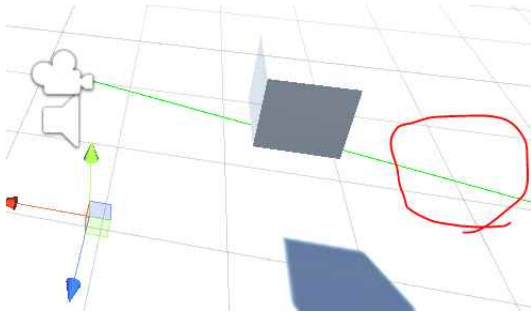
조건문에서는 rayOrigin은 위치, rayDir은 방향으로 distance만큼 광선을 쏘아 걸리는 물체가 있는지 검사한다. 아무 물체의 정보를 가지고 오면 안 되기 때문에 whatIsTarget이란 변수를 이용해서 특정 레이어마스크로 대상을 지정한다. 우리는 광선이 닿은 물체의 정보를 가지고와 위치를 옮겨주고 싶다. 그러기 위해서는 광선이 닿은 물체의 정보를 가지고 와야 하는데 RaycastHit hit 이 그 역할을 한다. 이는 raycast에 의해 정보를 담아주는 단순 정보 컨테이너이다.

조건문에서 out은 입력으로 들어간 값이 내부에서 계산되어 빠져나오는 것을 말한다. Raycast에서 계산된 값을 hit에 전달해 리턴하고 hit을 통해 물체의 정보를 가지고 올 수 있다.

```
public class RayInteraction : MonoBehaviour {  
    public LayerMask whatIsTarget;  
    private Camera playerCam;  
    public float distance = 100f;  
  
    void Start () {  
        playerCam = Camera.main; //현재활성화된 카메라를 가지고 온다.  
    }  
  
    void Update () {  
        //카메라가 보고있는 정중앙,화면에서 보고있는 곳이 게임세계상에서 어떤 위치인지를 저장  
        Vector3 rayOrigin = playerCam.ViewportToWorldPoint(new Vector3(0.5f,0.5f,0f));  
  
        //광선을 쏠때 어느 방향으로 쏠지 정함, 카메라의 앞쪽  
        Vector3 rayDir = playerCam.transform.forward;  
  
        Debug.DrawRay(rayOrigin,rayDir*100f,Color.green);  
  
        if (Input.GetMouseButtonDown(0))  
        {  
            RaycastHit hit;  
  
            //rayOrigin 위치에서 rayDir 방향으로 distance만큼 광선을 쏘아 걸리는 물체가 있는지 검사  
            if (Physics.Raycast(rayOrigin,rayDir,out hit,distance,whatIsTarget))  
            {  
                //hit을 통해서 상대방의 정보를 가지고 올 수 있다.  
                //충돌한 상대방 뿐 만 아닌 normal(충돌한상대방이 평면이 바라보고있는 방향),point(위치) 등이 많다.  
                Debug.Log(hit.collider.gameObject.name);  
                //충돌한 콜라이더의 게임오브젝트의 이름을 출력  
                Debug.Log("걸렸다!!");  
            }  
        }  
    }  
}
```

Debug.DrawRay()를 통해 생성된 씬 뷰 내에서의 가이드라인

또한 Raycast는 다양한 변수와 형식이 존재하는데 Ray ray = new Ray(rayOrigin,rayDir);를 선언하여 특정 부분을 바꿀 수 있다.



```
Debug.DrawRay(ray.origin,ray.direction*100f,Color.green);
```

```
if (Physics.Raycast(ray,out hit,distance,whatIsTarget)) 변경된 부분.
```

이제는 본격적으로 물체를 옮겨 보자. Debug 메시지를 삭제하고 대신 광선에 부딪힌 물체의 정보를 가지고 와 그 물체의 색깔을 빨간색으로 바꾼다.

moveTarget은 충돌한 물체의 위치를 가지고 오며 targetDistance 은 물체와의 거리를 저장한다.

```
public class RayInteraction : MonoBehaviour {
    public LayerMask whatIsTarget;
    private Camera playerCam;
    public float distance = 100f;
    private Transform moveTarget; //충돌한 변수를 넣어 시선에 따라 계속 움직이게 한다.
    private float targetDistance;

    void Start () {
        playerCam = Camera.main; //현재활성화된 카메라를 가지고 온다.
    }

    void Update () {
        //카메라가 보고있는 정중앙,화면에서 보고있는 곳이 게임세계상에서 어떤 위치인지를 저장
        Vector3 rayOrigin = playerCam.ViewportToWorldPoint(new Vector3(0.5f, 0.5f, 0f));

        //광선을 쏠때 어느 방향으로 쏠지 정함, 카메라의 앞쪽
        Vector3 rayDir = playerCam.transform.forward;

        //raycast는 다양한 변수와 형식이 존재한다.
        Ray ray = new Ray(rayOrigin,rayDir);

        Debug.DrawRay(ray.origin, ray.direction*100f, Color.green);

        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;

            //rayOrigin 위치에서 rayDir 방향으로 distance만큼 광선을 쏘아 걸리는 물체가 있는지 검사
            if (Physics.Raycast(ray, out hit, distance, whatIsTarget))
            {
                Game오브젝트 hitTarget = hit.collider.game오브젝트;

                hitTarget.GetComponent<Renderer>().material.color = Color.red;

                moveTarget = hitTarget.transform;
                targetDistance = hit.distance;
            }
        }
    }
}
```

왼 마우스를 누른 채 있다. 광선이 쏘는 위치에 물체가 걸리면 hitTarget에 의해 hit에 담긴 오브젝트정보를 가지고 오게 되고 material 정보를 가지고와 색상을 변경 시킨다.

```

if(Input.GetMouseButtonUp(0))
{
    if(moveTarget != null)
    {
        moveTarget.GetComponent<Renderer>().material.color = Color.white;
    }
    moveTarget = null;
}

```

왼 마우스를 땄을 경우, moveTarget이 null이 아닐 때, 즉 현재 hit정보가 있을 때 색상을 원래대로 변경하고 moveTarget을 null로 바꾼다.

```

if(moveTarget != null)
{
    moveTarget.position = ray.origin + ray.direction * targetDistance;
}

```

왼 마우스는 계속 누르고 있는데 moveTarget이 null이 아닐 때 moveTarget의 위치는 화면에서부터 쏘는 광선이 월드에 닿는 위치와 광선의 방향 그리고 target의 거리를 계산해 매순간 위치를 옮긴다.

```

}
}

```

오버로드(Overload)

함수의 여러 가지 버전을 만드는 것과도 같다. 여러 가지 상황에 따라 사용이 가능하다.

```
public class Calc : MonoBehaviour {  
    private void Start()  
    {  
        Debug.Log(Sum(1,1));  
        Debug.Log(Sum2(1, 1,1));  
        Debug.Log(Sum3(1, 1,1,1));  
    }  
  
    public int Sum(int a, int b)  
    {  
        return a + b;  
    }  
  
    public int Sum2(int a, int b, int c)  
    {  
        return a + b + c;  
    }  
  
    public int Sum3(int a, int b, int c,int d)  
    {  
        return a + b + c + d;  
    }  
}
```

만약 2개의 변수로 계산하는 것 외에 3개, 4개이상의 변수를 더한다고 할 때 Sum으로는 처리 할 수가 없다. 그렇다고 해서 여러 함수명을 선언한다고 할 경우 코드가 혼동이 오고 길어지게 된다. 이 때 사용하는 것이 method overloading이다. 함수이름을 같이하되 다른 버전, 즉 입력(매개변수 수) 혹은 출력(데이터 타입)을 달리하여야 한다.

```
public class Calc : MonoBehaviour {  
    private void Start()  
    {  
        Debug.Log(Sum(1,1));  
        Debug.Log(Sum(1, 1,1));  
        Debug.Log(Sum(1.5f,1.1f,1.3f));  
    }  
  
    public int Sum(int a, int b)  
    {  
        return a + b;  
    }  
  
    public int Sum(int a, int b, int c)  
    {  
        return a + b + c;  
    }  
  
    public float Sum(float a, float b,float c)  
    {  
        return a + b + c ;  
    }  
}
```

입력이 2개일 경우 매개변수가 2개인 Sum이 실행되고 입력이 3개일 경우 매개변수가 3개인 Sum이 실행된다. 입력하는 데이터 타입이 int 혹은 float에 따라 출력하는 데이터 타입도 맞게 출력이 된다.

코루틴(Coroutine)

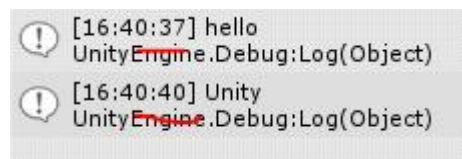
처리와 처리사이 대기시간을 넣을 수 있고 순차적으로 실행을 하는 것이 아닌 동시에 병렬로 처리할 수 있도록 한다. 예를 들어 화면 FadeInFadeOut 기능을 구현하려면 Coroutine을 사용하지 않으면 구현하기 힘들다. Ui -> Image로 들어가서 stretch를 통해 image 전체를 늘려 화면을 채운다. FadeInFadeOut기능을 구현하려면 이 Image의 알파 값을 변경하여 효과를 구현한다.

```
public class Fade : MonoBehaviour {
    public Image fadelImage;
    private void Start()
    {
        FadeIn();
    }
    void FadeIn()
    {
        Color startColor = fadelImage.color;
        for (int i = 0; i<100; i++)
        {
            startColor.a = startColor.a - 0.01f;
            fadelImage.color = startColor;
        }
    }
}
```

for문을 사용하여 image의 알파 값을 빼주면 되지만 컴퓨터의 성능이 너무 빠르기 때문에 우리는 인지하지 못한다. 그렇기 때문에 한번 for문이 돌고 대기시간을 넣어 주어야 한다. Coroutine은 Unity 내부 문법인데 IEnumerator란 형식을 사용한다. 예를 들어 A라는 구역을 처리하고 yield return 3f; 라고 한다면 Coroutine 바깥으로 나가 3초를 대기하고 다시 yield문으로 돌아와 다음을 처리한다. Coroutine 경우 Unity 내부 문법이기 때문에 이해하려하지 않아도 된다. Coroutine을 실행할 땐StartCoroutine(함수())을 사용하는데 tartCoroutine("함수")로 할 경우 인위적으로 멈출 수 있지만 함수자체를 실행시키는 방법보단 성능이 낮다.

Coroutine은 대기시간을 주는 것 외 예도 for문과 while을 섞으면 매우 유용해진다.

```
public class helloCoroutine : MonoBehaviour
{
    void Start()
    {
        StartCoroutine("HelloUnity");
    }
    IEnumerator HelloUnity()
    {
        Debug.Log("hello");
        yield return new WaitForSeconds(2f);
        Debug.Log("Unity");
    }
}
```



```
[16:40:37] hello
UnityEngine.Debug:Log(Object)
[16:40:40] Unity
UnityEngine.Debug:Log(Object)
```

약2~3초 뒤 메시지가 출력되는 것을 확인할 수 있다.

만약 yield 값을 null로 줄 경우 1/60초 즉 1fps 정도를 쉬게 된다.

그런데 만약 Coroutine을 여러 개를 실행하면 어떻게 될까? 기본적으로 프로그램은 순차적으로 위에서 아래로 수행하게 된다.


```

public class helloCoroutine : MonoBehaviour
{
    void Start()
    {
        StartCoroutine("HelloUnity");
        StartCoroutine("hiCsharp");
        Debug.Log("End");
    }

    IEnumerator HelloUnity()
    {
        Debug.Log("hello");
        yield return new WaitForSeconds(2f);
        Debug.Log("Unity");
    }

    IEnumerator hiCsharp()
    {
        Debug.Log("Hi");
        yield return new WaitForSeconds(5f);
        Debug.Log("Csharp");
    }
}

```

이 코드를 수행할 때 얼마나 시간이 걸릴까? 얼핏 보기에는 HelloUnity가 2초, hiCsharp이 5초 그리고 End 메시지를 출력하여 총 7초가 소요될 것 같다.

결과를 보면 hello와 Hi가 동시에 출력, 그 후 바로 End메시지 출력, 실행한 순간으로부터 각각 2초, 5초가 지난 뒤 Unity와 Csharp이 출력되었다. 서로 다른 Coroutine은 독립적으로 작동 되는 것을 볼 수가 있다. 즉, Coroutine은 비동기방식을 구현하는 방법 중하나가 되겠다. 무엇인가 동시에 처리를 해야 하는 것을 구현할 때 사용하는 것이다.

일반적인 함수 처리방식 A(),B(),C()가 있다고 할 때 A()는 return문을 통해서 끝났다고 다음 함수 B()에게 알린다. 그 후 B()가 실행되는 방식인데 이를 동기방식이라고 한다.

하지만 비동기 방식은 나 자신이 시작된 것만 알려주고 끝나는 시점은 알려주지 않는다.

A() ->	A() ->	A() ->		
	B() ->	B() ->	B() ->	
		C() ->	C() ->	C() ->

이런 모습이 되겠다. 3함수가 모두 겹치는 동시간대가 존재하는 것이 가능하므로 Coroutine은 멀티쓰레딩 방식을 구현할 수 있다. 그래서 총 걸리는 시간은 5초가 되겠다.

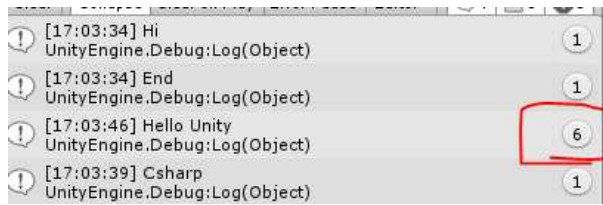
만약 코드가 아래와 같이 구성되었다고 가정해보자.

```

IEnumerator HelloUnity()
{
    while(true)
    {
        yield return new WaitForSeconds(2f);
        Debug.Log("Hello Unity");
    }
}

```

통상적으로 무한 루프로 구현할 경우 굉장히 빠른 속도로 처리를 해야하기 때문에 컴퓨터에 부하가가서 프로그램이 터질 수가 있다. 또한 무한 루프에 빠졌기 때문에 그 다음 코드를 실행하지 못하고 무한 루프에서 탈출을 하지 못하게 된다. 반면 Coroutine에서의 경우는 무한루프에 빠져도 대기시간이 존재하기 때문에 성능에 치명적인 문제를 예방할 수 있으며 비동기식 방법이기 때문 무한루프에 빠져도 다른 코드가 실행되지 않는 문제는 겪지 않는다.



마지막으로 문자열로 실행시키는 Coroutine과 함수로 실행시키는 Coroutine의 차이점은 아래와 같이 함수로 실행시킬 경우 성능이 좋지만 문자열로 실행시킬 경우 StopCoroutine을 사용하여 원할 때 멈출 수 있다는 장점이 있다.

```
private void Update()
{
    if(Input.GetMouseButtonDown(0))
    {
        StopCoroutine("HelloUnity");
    }
}
```

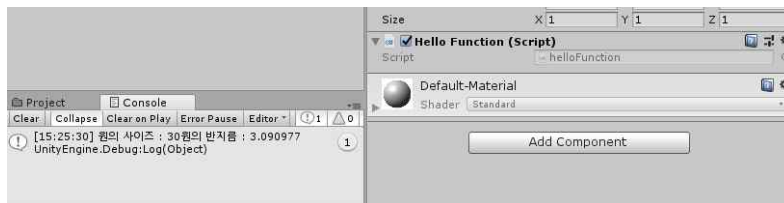
함수 작성, 스코프(Functions, Scope)

ex) 넓이가 주어졌을 때 반지름을 구하는 함수를 구한다.

넓이 = 반지름² x 3.14

```
public class helloFunction : MonoBehaviour {  
    // Use this for initialization  
    void Start () {  
        float sizeOfCircle = 30f;  
  
        float radius = getRadius(sizeOfCircle);  
  
        Debug.Log("원의 사이즈 : " + sizeOfCircle + "원의 반지름 : " + radius);  
    }  
  
    float getRadius(float size)  
    {  
        float pi = 3.14f;  
  
        float tmp = size / pi;  
  
        float radius = Mathf.Sqrt(tmp);  
  
        return radius;  
    }  
}
```

A.a에서 '.'의 의미는 A의 내부요소 a를 가지고 온다는 뜻이 된다.



통상적으로 변수는 중복이 되지 않지만 getRadius()와 start()에는 radius가 각각 있다. 이들은 서로의 영역을 침범하지 않으면 중복되지 않은 별개의 변수이다. getRadius()내의 radius는 오로지 getRadius()내에서만 사용되며 start()의 radius또한 같다. 하지만 같은 함수 내에서 radius = 1; radius = 2; 는 선언이 불가하다. 함수 내부에 정의된 변수는 함수 밖에서는 관측이 되지 않는다. 이를 scope라 하면 변수가 관측 가능한 영역이라고 한다. getRadius()와 start()는 별개의 구역이며 서로의 변수를 관측할 수 없다.

OOP : 클래스와 오브젝트

플라톤의 이데아 이론에서 비롯한다. 목수가 책상을 만들 때는 목수의 머릿속에서 가장 이상적인 책상의 모습을 생각한다. 그리고 목수가 책상을 실제로 만들 때는 머릿속에 있는 이상적인 책상을 본 따 만든다고 한다. 즉, 현실의 책상은 목수의 머릿속에 있는 책상을 가지고 와서 만들어낸다. 그리고 그것을 따라 만들었기 때문에 만든 책상은 소재가 다를 수도 있고 길이가 다를 수도 있고 무게가 다를 수도 있다. 새로 비슷한 성질을 가지고 있지만 조금씩 혹은 많이 다른 이유는 이상적인 세계에 존재하는 책상을 본 따 현실의 책상을 만들었다고 생각했기 때문이다.

그래서 이상적인 세계, 추상적인 세계에 존재하는 완벽한 원본을 이데아, 현실세계의 물건을 레플리카 라고 부른다. 추상적인 무언가를 현실에 구체화 시켜서 만든 복제본, 이것이 클래스와 오브젝트 개념의 원형이다.

사과가 하나가 있다.



이 사과를 인식할 수 있는 이유는 우리가 사과를 뭉뚱그려 인식하기 때문이다. 우리가 머릿속에 사과를 떠올리게 되면 특정 사과가 아닌 형체가 희미하며 매우 추상적이지만 사과라고 인식을 한다. 즉 우리 머릿속에 추상적인 사과라는 개념이 존재한다는 뜻이다. 우리 머릿속에는 이상적이고 추상적인 사과가 있지만 이는 구체적 이지 않다. 구체적인 구현물은 빠져있기 때문에 우리는 이 사물을 대조군으로 삼아 비교하기 때문에 현실에서의 다양하고 다른 형태의 사과를 사과라는 하나의 개념으로 인식할 수가 있다는 것이다. 각각 전부 다르지만 같은 개체로 인식하는 것이 사람을 방식이다.

만약 컴퓨터라면 사과라는 것에 구체적인 수치를 잡는다. 반지름은 3이고 무게는 10g 등. 하지만 이렇게 할 경우 여러 종류의 사과를 하나의 사과로 인식할 수가 없다. 사람은 어떠한 대상을 뭉뚱그려서(추상화를 시켜서) 머릿속에 대략적으로 집어넣는 능력이 존재한다.

실제 프로그램 세상에서는 우리머릿속의 원본을 가지고와 각각의 물체를 찍어내어 구체화 시킨다는 것이 프로그래밍에서의 핵심이다.

이상적인 세계에 개라는 구체적이지 않고 희미한 존재가 있다. 이는 class가 된다. 이상적인 세계의 "개"라는 class를 가지고와 현실세계에서 각기 다른 특징을 가지고 있는 각기 다른 "개"라는 오브젝트를 생성하게 된다.

class로 정의된 물체는 가장 추상적이고 내부가 구체적이지 않은 원형이다. class를 현실세계에 가지고와 특정한 물체를 만들어낸다. 이상적인 세계의 class를 현실세계에 반영하여 오브젝트로서 구체화시켜 만드는 행위를 인스턴스화라고 부른다. 인스턴스화를 통해서 원형을 가지고 물체를 찍어낸다는 개념이다. 오브젝트는 같은 원형을 가지고 파생되었지만 각각은 독립적으로 존재한다.

Object

- 하나의 온전한 단위로 존재한다.
- 실존하는 세상에 존재한다.
- 하나의 원본에서 파생되어도 서로 구분이 가능하다.

Class

- 이상적인 세계에서 존재하는 단 하나의 기준
- 실제로는 존재하지 않는다.
- 가장 중요한 특성만 대략적으로 알려준다.
- 구체적인 수치가 없다.

```
public class helloClass : MonoBehaviour {
```

```
    // Use this for initialization
```

```
    void Start () {  
        Animal jack = new Animal();  
        jack.name = "jack";  
        jack.sound = "bark";  
        jack.weight = 4.5f;
```

```
        Animal nate = new Animal();  
        nate.name = "nate";  
        nate.sound = "nya";  
        nate.weight = 10f;
```

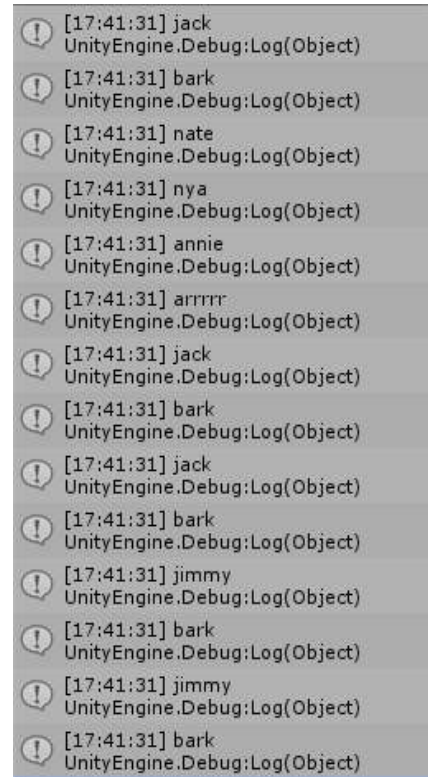
```
        Animal annie = new Animal();  
        annie.name = "annie";  
        annie.sound = "arrrrr";  
        annie.weight = 100f;
```

```
        Debug.Log(jack.name);  
        Debug.Log(jack.sound);  
        Debug.Log(nate.name);  
        Debug.Log(nate.sound);  
        Debug.Log(annie.name);  
        Debug.Log(annie.sound);
```

```
        nate = jack; //nate에 jack이 덮어 씌우기가 된다.  
        Debug.Log(jack.name);  
        Debug.Log(jack.sound);  
        Debug.Log(nate.name);  
        Debug.Log(nate.sound);
```

```
        nate.name = "jimmy"; //nate의 이름을 jimmy로 바꾼다.  
        //서로 독립적이기 때문에 nate의 이름은 바뀌지만 jack의 이름은 그대로 라고 생각가능.  
        Debug.Log(jack.name);  
        Debug.Log(jack.sound);  
        Debug.Log(nate.name);  
        Debug.Log(nate.sound);  
    }
```

```
public class Animal //동물이라는 새로운 원형을 만든다.  
{  
    public string name;  
    public string sound;  
    public float weight;  
}
```



```
[17:41:31] jack  
UnityEngine.Debug:Log(Object)  
[17:41:31] bark  
UnityEngine.Debug:Log(Object)  
[17:41:31] nate  
UnityEngine.Debug:Log(Object)  
[17:41:31] nya  
UnityEngine.Debug:Log(Object)  
[17:41:31] annie  
UnityEngine.Debug:Log(Object)  
[17:41:31] arrrrr  
UnityEngine.Debug:Log(Object)  
[17:41:31] jack  
UnityEngine.Debug:Log(Object)  
[17:41:31] bark  
UnityEngine.Debug:Log(Object)  
[17:41:31] jack  
UnityEngine.Debug:Log(Object)  
[17:41:31] bark  
UnityEngine.Debug:Log(Object)  
[17:41:31] jimmy  
UnityEngine.Debug:Log(Object)  
[17:41:31] bark  
UnityEngine.Debug:Log(Object)  
[17:41:31] jimmy  
UnityEngine.Debug:Log(Object)  
[17:41:31] bark  
UnityEngine.Debug:Log(Object)
```

출력 결과는 우리가 생각하던 개념과는 많이 다르다. 변수는 값 그자체가 아니라 값을 가리키는 이름표.

Animal jack = new Animal(); 의 의미는 “jack”이라는 이름이 새로 탄생된 Animal 오브젝트를 가리킨다는 의미가 된다. 새로 생긴 오브젝트는 실존하는 존재지만 이름이 없다. 그래서 아래와 같은 방법으로 이름을 붙여주는 것이다.

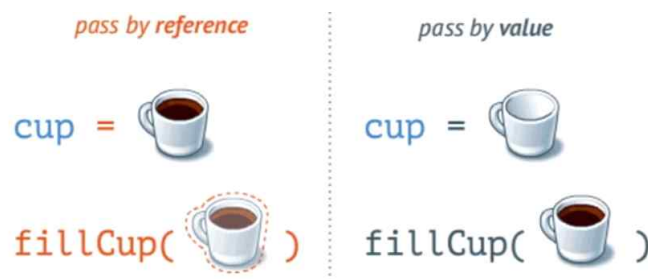
```
jack.name = "jack";
jack.sound = "bark";
jack.weight = 4.5f;
```

Class & Object 개념에서 변수는 오브젝트 그자체가 아니라 오브젝트를 가리키는 이름표이다.

쉽게 말하자면 `Animal jack = new Animal();`에서 `new Animal();`는 현실에 새로운 `Animal`이 생성된다. `Animal jack =`은 `jack`이라는 이름표에 새로 생긴 `Animal`을 붙여준 것이다. 새로 생긴 `Animal`(오브젝트)의 호칭일 뿐이다.

`nate = jack;`을 하는 순간 `nate`는 기존의 `nate`가 가리키던 오브젝트를 가리키는 것이 아니라 `jack`이 가리키는 오브젝트를 같이 가리키게 되는 것이다. 그래서 변수는 3개지만 실제 존재하는 오브젝트는 2개인 셈이다. `jack`과 `nate`는 변수이름은 다르지만 같은 `Animal` 오브젝트를 가리키고 있게 된다. 그렇다면 `nate`가 가리키고 있던 원래 오브젝트 경우 누구도 접근이 불가하기 때문에 가비지컬렉터에서 메모리는 반환하게 한다.

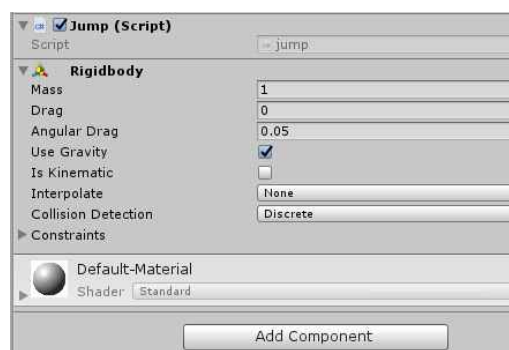
이러한 개념을 Call by Reference라고 부른다. 변수는 실제 오브젝트를 가리키는 이름표일 뿐이다. 변수는 실제 오브젝트가 아니라 실제 오브젝트를 찾아가기 위한 참고자 일뿐이다.



`cup`이라는 변수를 다른 곳에 전달했을 때 call by reference에서는 변수는 실존하는 오브젝트의 이름표일 뿐이기 때문에 다른 곳에서 수정하면 원본 또한 수정이 된다. 실존하는 오브젝트는 하나고 변수는 오브젝트를 가리키는 이름표이기 때문이다. call by value는 변수는 값 그 자체가 된다. 둘의 차이는 왼쪽은 실존하는 `cup`은 단 하나이며 단지 `cup`을 가리키는 이름이 2개일 뿐이다. 오른쪽은 실제 존재하는 `cup` 오브젝트가 2개이다.

Call by Reference 개념은 “가져와서 쓴다.”는 개념을 확립하기 위함이다. 이 개념을 이용하여 유니티에서는 Component를 가지고 와서 사용하는 것을 구현할 수 있는 것이다.

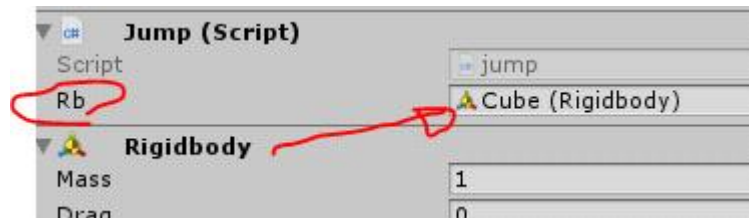
ex)cube라는 개체에 힘을 주고 싶다. cube라는 개체 rigidbody라는 이미 구현된 Component를 붙였다. rigidbody라는 Component에 힘을 주고 싶다.



그리고 jump라는 script를 만들었다.

```
public class jump : MonoBehaviour {  
    public Rigidbody rb;  
}
```

여기서 rb는 변수이다. 오브젝트 그자체가 아니라 오브젝트를 따라다니는 이름표이다.



rb라는 이름표에 실제 Rigidbody를 드래그앤드롭을 한다. 우리는 코드상에서 rb를 수정한다고 생각하지만 rb는 오브젝트가 아닌 그를 따라 다니는 이름표일 뿐이다. rb를 수정하는 것은 rb가 가리키고 있는 실제 존재하는 Rigidbody를 수정하는 것과도 같다.

```
public class jump : MonoBehaviour {  
    public Rigidbody rb;  
    private void Start()  
    {  
        rb.AddForce(100,100,100);  
    }  
}
```

Call by Reference는 변수는 실제 오브젝트를 가리키는 이름표라는 개념이 없게 되면 코드 상에서 Rigidbody에 접근할 수가 없다. 유니티는 어떤 기능을 가진 Component를 합친 다음 미리 만들어진 Component를 가져와서 사용한다는 개념이다.

변수를 실제 오브젝트라고 착각하지 말고 또 다른 오브젝트를
가리키는 이름표라는 사실을 명심하자.

정적 변수와 함수

정적 변수 혹은 함수(static)이란 모든 오브젝트에서 접근할 수 있는 변수 또는 함수를 이야기한다. static은 모든 오브젝트가 공유하며 개별 오브젝트가 가지기에는 어색한 정보를 나타낼 때 사용한다.

Dog이라는 기본클래스를 만들고 Unity Editor 상 계층뷰에 3개의 cube 오브젝트를 생성한 뒤 각각 다른 이름을 정하고 Dog클래스를 적용했다. 각 다른 Dog 오브젝트는 독립적이며 메모리도 독립적으로 할당받는다.

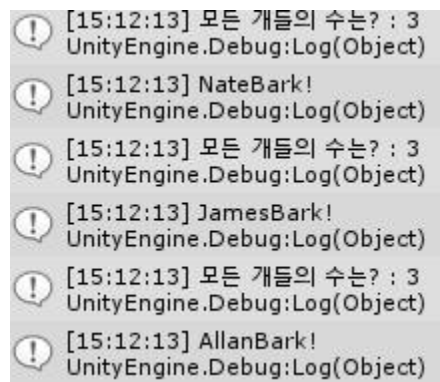
Dog 클래스에는 name, sound 등 멤버변수가 있고 각 오브젝트는 이 멤버변수를 가지고 서로 다른 값을 가진다. 하지만 만약에 count 변수를 통해서 모든 Dog의 수를 알고 싶다면?

```
public class Dog : MonoBehaviour {  
    public string nickName;  
    public float weight;  
    public int count = 0;  
    public void Awake()  
    {  
        count = count + 1;  
    }  
    public void Start()  
    {  
        Bark();  
    }  
    public void Bark()  
    {  
        Debug.Log("모든 개들의 수는? : " + count);  
        Debug.Log(nickName + "Bark!");  
    }  
}
```



위처럼 Dog들의 수를 세기 위해 count 변수를 선언하고 Bark()함수를 통해 수를 출력하도록 했지만 3이 아닌 모두 1이라는 수만 출력했다. Dog 클래스에 count라는 변수를 추가했다. 이 변수는 모든 오브젝트에 count변수가 추가된다. count = 0 이다. 그리고 각자가 count를 1씩 증가 시킨다. 0 ->1, 1->2, 2->3이 아니라 오브젝트는 서로 독립적이기 때문에 0->1, 0->1, 0->1 가 되어 버린다. count 변수는 3번 증가했지만 전부 개별 count 변수이다. 이 때 필요한 변수가 static(정적 변수)이다. count는 모든 오브젝트가 공유한다. count변수가 모든 오브젝트에 count변수를 찍어내지 않고 오브젝트가 count변수를 사용하게 하고 공유하게 된다.

```
public class Dog : MonoBehaviour {  
    public string nickName;  
    public float weight;  
    public static int count = 0;  
    public void Awake()  
    {  
        count = count + 1;  
    }  
    public void Start()  
    {  
        Bark();  
    }  
    public void Bark()
```




```
{
    Debug.Log("모든 개들의 수는? : " + count);
    Debug.Log(nickName + "Bark!"); } }
```

Awake()단에서 각 오브젝트들은 count를 0->1, 1->2, 2->3으로 변경 시킨다. 모든 오브젝트가 공유하는 count 변수는 현재 3이되고 Start()함수가 실행되고 Bark()함수가 실행되면서 count된 수를 말하게 되면 모든 오브젝트는 3을 외친다.

static(정적변수)를 사용하는 때는 오브젝트와 관련은 있지만 단일 오브젝트가 사용하기에는 어색한 정보를 쓸 때 사용한다. 개별 오브젝트가 어떤 것이든 Dog 클래스에서 파생된 오브젝트들은 '개'이다. 즉, 개별 오브젝트에게 의존적이지 않다. static 함수나 변수는 오브젝트에게 묶여있지 않기 때문에 오브젝트를 통하지 않고 사용할 수 있다. static 변수나 함수는 전역 구역에 있기 때문에 독립적으로 사용 및 호출이 가능하다.

```
public class Dog : MonoBehaviour {

    public static void ShowAnimalType()
    {
        public static int count = 0;

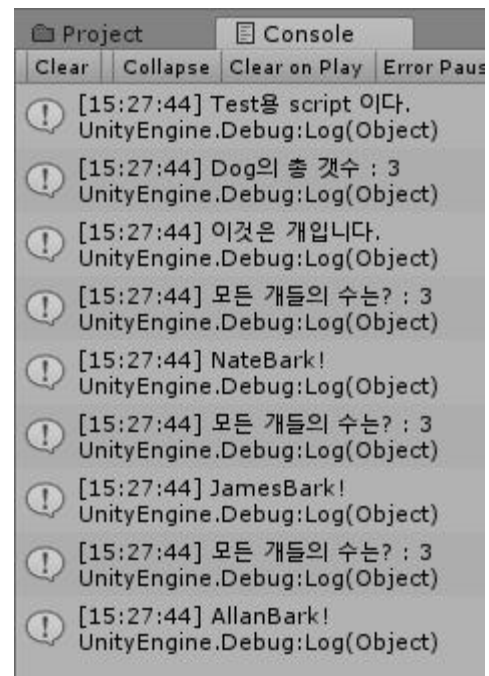
        Debug.Log("이것은 개입니다.");
    }
}

public class Test : MonoBehaviour {

    void Start () {
        Debug.Log("Test용 script 이다. ");
        Debug.Log("Dog의 총 갯수 : " + Dog.count);

        Dog.ShowAnimalType();
    }
}
```

Testscript란 빈 오브젝트를 만들고 test 스크립트를 적용했다. static으로 선언된 변수나 함수는 특정 오브젝트를 통하지 않고도 사용가능 하다.



싱글톤(Singleton)

디자인 패턴 중 하나. 게임이나 메모리상에 단 하나만 존재하고 언제 어디서 접근 가능한 오브젝트를 만들 때 사용하는 개념이다. 예시로 3개의 오브젝트를 생성하고 그 중하나만 닌자왕이 될 수 있다고 가정한다. naruto를 닌자왕으로 혼자 체크를 해놓는다. 실행하면 매순간 세 오브젝트는 자신의 이름을 외친다.

```
public class Ninja : MonoBehaviour {  
    public string ninjaName;  
    public bool isKing;  
    private void Update()  
    {  
        Debug.Log("My name : " + ninjaName);  
    }  
}
```

하지만 지금 원하는 것이 단 하나의 닌자왕이고 하나만 알았으면 할 땐 어떡할까? 닌자왕 만의 자리를 만들고 닌자왕이 그곳에 앉는다. Ninja 라는 클래스에는 name이 있다. 각 세 오브젝트들 또한 name을 가지고 있다. Naruto란 친구가 닌자왕이라고 가정을 한다. static이 아닌 name은 각각의 오브젝트에 주어진다. 하지만 static으로 선언된 NinjaKing 변수는 단 하나만 존재한다. 아직 할당 하지 않았기 때문에 단 하나의 공간이 비어있고 모든 오브젝트가 접근 가능하다. 비어있는 공간에 Naruto가 들어간다. 그렇게 되면 나머지 닌자들은 일일이 오브젝트를 찾아 NinjaKing을 구별하는 것이 아니라 static 변수 NinjaKing만 보면 누가 닌자왕 인지 알 수 있다. Singleton을 구현하는 핵심이다.

```
public class Ninja : MonoBehaviour {  
    public static Ninja NinjaKing;  
    public string ninjaName;  
    public bool isKing;  
    private void Start()  
    {  
        if(isKing)  
        {  
            NinjaKing = this; //isKing이 true면 자기자신을 NinjaKing으로 한다.  
        }  
    }  
    private void Update()  
    {  
        Debug.Log("My name : " + ninjaName + "NinjaKing : " + NinjaKing);  
    }  
}
```



Singleton은 주로 GameManager, FileManager 등 단 하나만 존재하며 모든 오브젝트가 손쉽게 접근 할 수 있는 것을 만들 때 구현한다. 예시로 scoreManager를 만든다. 점수를 관리하는 오브젝트는 1개만 있어도 충분하다.

```

public class ScoreManager2 : MonoBehaviour {
    private int score = 0;

    public int GetScore()
    {
        return score;
    }

    public void AddScore(int newScore)
    {
        score = score + newScore;
    }
}

```

그리고 왼쪽마우스를 누르면 점수가 오르는 ScoreAdder라는 함수를 만든다.

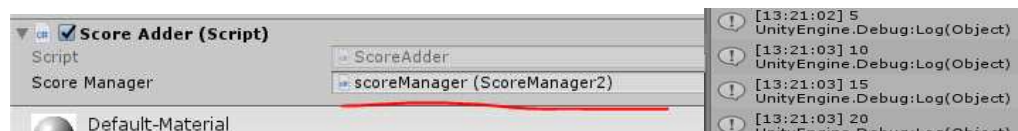
```

public class ScoreAdder : MonoBehaviour {
    public ScoreManager2 scoreManager;

    void Update () {
        if(Input.GetMouseButtonDown(0))
        {
            scoreManager.AddScore(5);
            Debug.Log(scoreManager.GetScore());
        }
    }
}

```

ScoreAdder가 게임에서의 몬스터라고 할 때 ScoreAdder는 여러 개가 생겨날 수 있다. 점수를 주고 빼고 스스로 파괴되고 하는 Adder는 여러 개가 있어도 되지만 그것을 관리하는 Manager은 1개만 있으면 된다. 스크립트 상에 `public ScoreManager2 scoreManager;`를 사용하려면 아래와 같이 실제 오브젝트를 적용시켜야한다. 명심해라 코드 상에 변수는 이름표일 뿐이지 실제 오브젝트가 아니다.



왼쪽 마우스를 누르자 점수를 추가되는 것을 볼 수 있다.

문제는 실제 프로그램에서는 점수를 읽고 쓰는 것들이 한 두 개가 아니라는 것이다. 예시로 ScoreSubtractor를 만든다.

```

public class ScoreSubtractor : MonoBehaviour {
    public ScoreManager2 ScoreManager;

    private void Update()
    {
        if(Input.GetMouseButtonDown(1))
        {
            ScoreManager.AddScore(-2);
            Debug.Log(ScoreManager.GetScore());
        }
    }
}

```



문제는 Adder와 Subtractor가 다수일 때 일일이 드래그 앤 드롭을 통해 적용하기가 매우 번거롭다는 것이다. 이제는 ScoreManager를 static으로 만들어서 사용한다. 유일한 개체가 되면서 모든 오브젝트가 자신을 접근 할 수 있게 된다. 그 후 Adder와 Subtractor에서는 scoreManager를 사용하기 위해 일일이 드래그 앤 드롭을 할 필요가 없다.

```

public class ScoreManager2 : MonoBehaviour {
    public static ScoreManager2 instance;

    private void Awake()
    {
        instance = this;
    }

    private int score = 0;

    public int GetScore()
    {
        return score;
    }

    public void AddScore(int newScore)
    {
        score = score + newScore;
    }
}

public class ScoreSubtractor : MonoBehaviour {
    private void Update()
    {
        if(Input.GetMouseButtonDown(1))
        {
            ScoreManager2.instance.AddScore(-2);
            Debug.Log(ScoreManager2.instance.GetScore());
        }
    }
}

public class ScoreAdder : MonoBehaviour {
    void Update () {
        if(Input.GetMouseButtonDown(0))
        {
            ScoreManager2.instance.AddScore(5);
            Debug.Log(ScoreManager2.instance.GetScore());
        }
    }
}

```

ScoreAdder와 ScoreSubtractor는 ScoreManager2의 ScoreManager변수에 접근하려고 일일이 inspector 창에서 드래그앤드롭을 하지 않고 클래스자체의 static변수 instance에게 사용하고 싶은 씬 상에 유일하게 존재하는 ScoreManager가 들어있기 때문이다. 이제는 코드 상에서 바로 접근할 수 있다는 것을 생각하자.



결과는 이전과 동일하다. Singleton은 단하나만 존재하는 유일한 변수에 자기를 넣음으로써 자신을 사용할 때 복잡하게 드래그드롭, 검색 등의 번거로운 작업을 막을 수가 있다. 단하나만 존재하고 모든 오브젝트가 접근 가능할 때 사용하는 디자인패턴이다. 하지만 이러한 형식의 Singleton은 매우 간단한 형태의 Singleton이다. Singleton은 씬 상에 ScoreManager가 무조건 1개는 있어야한다고 확정해야한다. instance에 오브젝트가 할당되어있지 않다면 오브젝트를 할당 해주어야 하고 1개 이상일 경우 그것을 파괴하는 safety code를 넣어야한다. 지금 작성하는 방법은 instance에 아직 할당 된 것이 없다면 씬 상의 모든 ScoreManager를 찾아서 GetInstance()통해서 할당하게 한다.

```

public class ScoreManager2 : MonoBehaviour {
    public static ScoreManager2 GetInstance()
    {
        if (instance == null)
        {
            instance = Find오브젝트OfType<ScoreManager2>();
        }

        return instance;
    }

    private static ScoreManager2 instance;

    private int score = 0;

    public int GetScore()
    {
        return score;
    }

    public void AddScore(int newScore)
    {
        score = score + newScore;
    }
}

```

Find오브젝트Type() 경우 성능상의 문제를 줄 수가 있기 때문에 게임 시작처음에 매우 적은 수로 사용한다. 또한 이대로 실행 경우 NullReference error메세지가 출력되는데 수정된 ScoreManager2에서 instance는 private으로 다시 선언되었기 때문에 Adder,와 Subtractor 스크립트에서도 수정해 주어야한다.

ScoreManager2.instance.AddScore(5); -> ScoreManager2.GetInstance().AddScore(5); 로 변경한다.

```

public class ScoreSubtractor : MonoBehaviour {
    private void Update()
    {
        if(Input.GetMouseButtonDown(1))
        {
            ScoreManager2.GetInstance().AddScore(-2);
            Debug.Log(ScoreManager2.GetInstance().GetScore());
        }
    }
}

public class ScoreAdder : MonoBehaviour {
    void Update () {
        if(Input.GetMouseButtonDown(0))
        {
            ScoreManager2.GetInstance().AddScore(5);
            Debug.Log(ScoreManager2.GetInstance().GetScore());
        }
    }
}

```

하지만 여기서 ScoreManager2는 ScoreManager 오브젝트가 적어도 1개가 있어야 작동하기 때문에 오브젝트가 없을 경우 error 발생할 수 있다. 다시 ScoreManager2를 수정하면 이렇게 된다.

```

public class ScoreManager2 : MonoBehaviour {
    public static ScoreManager2 GetInstance()
    {
        if (instance == null)
        {
            instance = Find오브젝트OfType<ScoreManager2>();

            if(instance == null)
            {
                Game오브젝트 container = new Game오브젝트("ScoreManager");
            }
        }
    }
}

```

```

        instance = container.AddComponent<ScoreManager2>();
    }
    return instance;
}

private static ScoreManager2 instance;
private int score = 0;
public int GetScore()
{
    return score;
}

public void AddScore(int newScore)
{
    score = score + newScore;
}
}

```

만약 instance를 찾았는데도 없다고 한다면 새로하나 만들기로 한다. 새로운 Game오브젝트를 만들기로 한다. 일반적으로 Unity에서 Game오브젝트는 Instantiate()를 통해서 생성하게 되는데 empty 오브젝트인 경우 new 키워드를 통해서 생성할 수 있다. Game오브젝트 container = new Game오브젝트("ScoreManager");을 통해서 ScoreManager란 이름의 empty 오브젝트를 생성한다.

container.AddComponent<ScoreManager2>()을 통해서 방금 생성한 empty 오브젝트에 ScoreManager2를 붙이고 이를 instance로 return한다. 이렇게 되면 ScoreManager 오브젝트가 씬 상에 없어도 실행하면 해당 오브젝트가 생성이 되고 삭제를 하더라도 생성이 된다. 이러한 것을 '자연생성'이라고 하며 Singleton의 특징 중 하나는 사용하려 할 때 생성이 된다. Singleton은 code를 통해 오브젝트를 생성해서 간편할 수도 있다. 마지막으로 Singleton에서의 변수는 단 하나만 존재해야하기 때문에 2개 이상이 될 경우 하나를 파괴하는 식으로 구현을 한다.

만약에 instance가 null이 아니면서 instance가 나 자신 또한 아니게 된다면 즉 이미 다른 누군가가 instance에 할당이 되었다는 뜻이다. 즉, 나는 유일한 instance 외의 것이기 때문에 나 자신을 제거하게 된다.

```

public class ScoreManager2 : MonoBehaviour {
    public static ScoreManager2 GetInstance()
    {
        if (instance == null)
        {
            instance = Find오브젝트OfType<ScoreManager2>();

            if(instance == null)
            {
                Game오브젝트 container = new Game오브젝트("ScoreManager");
                instance = container.AddComponent<ScoreManager2>();
            }
        }
        return instance;
    }

    private static ScoreManager2 instance;
    private int score = 0;

    private void Start()
    {
        if(instance != null)
        {
            if(instance != this)
            {
                Destroy(game오브젝트);
            }
        }
    }

    public int GetScore()

```

```

    {
        return score;
    }

    public void AddScore(int newScore)
    {
        score = score + newScore;
    }
}

```

그리고 게임이 실행될 때 중복체크를 할 때 instance가 미리 1개는 존재해야 중복체크가 가능하다. 그래서 Adder스크립트에서 1번은 사용하기로 했다. 이런 식으로 safety code를 작성가능하다. 정해진 방법은 아니며 디자인패턴 방식 중 하나이기 때문에 자유롭게 작성가능하다.

```

public class ScoreAdder : MonoBehaviour {
    private void Awake()
    {
        Debug.Log("Start Score : " + ScoreManager2.GetInstance().GetScore());
    }
    void Update () {
        if(Input.GetMouseButtonDown(0))
        {
            ScoreManager2.GetInstance().AddScore(5);
            Debug.Log(ScoreManager2.GetInstance().GetScore());
        }
    }
}

```

상속(Inheritance)

기존 클래스의 정보를 가지고와 내용을 확장시키는 개념이다, Unity 스크립트를 생성하면 A : MonoBehaviour라는 클래스를 상속받는데 이는 A 스크립트에는 전혀 작성하지 않아도 MonoBehaviour의 클래스를 상속받았기 때문에 Unity 내장 기능을 가져와 쓸 수 있다는 뜻이다.

```
public class Animal {
    public string name;
    public float weight;
    public int year;
    public void Print()
    {
        Debug.Log(name + "| 몸무게 : " + weight + "| 나이 : " + year);
    }

    public float getSpeed()
    {
        float speed = 100f / (weight * year);

        return speed;
    }
}

public class Wolf : Animal
{
    public void Hunt()
    {
        float speed = getSpeed();
        Debug.Log(speed + "의 속도로 달려가 사냥했다.");

        weight = weight + 10f;
    }
}

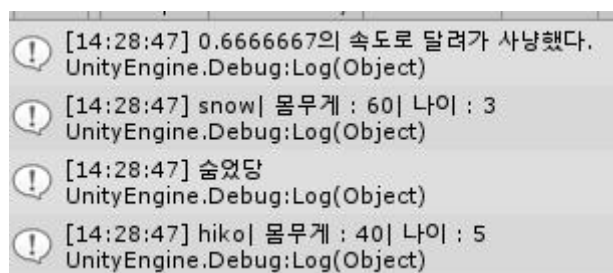
public class Lion : Animal
{
    public void hide()
    {
        Debug.Log("숨었당");
    }
}
```

이 스크립트를 그대로 오브젝트에 적용할 수는 없다. MonoBehaviour를 상속받고 있지 않기 때문에 Unity의 기본적인 기능을 사용할 수가 없기 때문이다. MonoBehaviour에는 Unity의 Component 기능이 들어있다. 그렇기 때문에 GetComponent()함수를 사용하거나 Inspector창에서 스크립트를 적용할 수 있게 된다. Animal 클래스는 메모리상에 공간이 할당 되지만 Unity Editor상에서 Component로 적용할 수 없다.

```
public class AnimalTest : MonoBehaviour {
    private void Start()
    {
        Wolf snow = new Wolf();
        snow.name = "snow";
        snow.weight = 50f;
        snow.year = 3;

        Lion hiko = new Lion();
        hiko.name = "hiko";
        hiko.weight = 40f;
        hiko.year = 5;

        snow.Hunt();
        snow.Print();
    }
}
```



```
[14:28:47] 0.6666667의 속도로 달려가 사냥했다.
UnityEngine.Debug:Log(Object)
[14:28:47] snow| 몸무게 : 60| 나이 : 3
UnityEngine.Debug:Log(Object)
[14:28:47] 숨었당
UnityEngine.Debug:Log(Object)
[14:28:47] hiko| 몸무게 : 40| 나이 : 5
UnityEngine.Debug:Log(Object)
```



```

        hiko.hide();
        hiko.Print();
    }
}

```

Test스크립트에서 Wolf와 Lion 클래스의 객체를 각각 만들었다. name, year 등의 멤버변수를 가지고 있지 않지만 Animal 클래스를 상속받고 있기 때문에 사용가능하며 Print()도 사용가능하다. 또한 자기가 가지고 있는 고유한 함수인 Hunt()와 hide()도 같이 사용할 수 있다. 그렇다는 것은 나중에 특정 캐릭터나 오브젝트의 성질을 미리 따로 만들어 놓고 MonoBehavior가 상속된 Test 스크립트에서 이를 불러와 사용하면 코드가 짧아지는 효과를 보지 않을까?

자식클래스는 부모클래스에 얼마나 접근가능한지 지정해줄 수 있다. 자식은 상속받은 내용을 사용하기만 하면 되지 굳이 사용함수의 작동방법을 알 필요는 없다. 그래서 부모는 필요한 기능만 제공하고 세부 내용을 숨기는데 이렇게 되면 코드를 간략하게 할 수 있다.(나중에 다른 코드에서 접근을 할 때 필요 없는 기능인데도 public으로 선언해놓으면 .을 찍었을 때 매우 복잡하게 변수와 함수가 보여 질 수 있다.)

```

public float getSpeed()
{
    return CalcSpeed();
}

private float CalcSpeed()
{
    return 100f / (weight * year);
}

```

코드를 이렇게 변경할 경우 자식클래스 경우 변경 전과 같이 getSpeed()를 사용할 수 있다. 하지만 속도를 계산하는 CalcSpeed()에는 private이 때문에 접근이 불가하다. 만약 상속받은 자식은 getSpeed()를 사용가능 하지만 완전 외부의 오브젝트가 getSpeed()에 접근하지 못하게 하려면 어떻게 할까? 그럴 때는 protected 키워드를 사용한다.

다형성(Polymorphism)

기본 형태에서 파생된 여러 물체들은 기본 형태로써 관리가 가능하다. 예를 들어 생물이라는 집합 군이 있다. 특징은 숨을 쉰다. 라고 하자. 생물은 동물과 식물로 나뉜다. 동물은 숨을 쉬고 움직이지만 식물은 숨을 쉬지만 움직이지 않는 특징이 있다. 조건은 늘어날수록 집합의 크기는 작아진다. 작은 집합(동물, 식물)은 생명이라고 부를 수 있지만 큰 집합(생물)은 동물 혹은 식물이라고 일치시킬 수는 없다. 가장 큰 집합은 조건이 매우 간단하기 때문에 모든 것을 포용할 수 있다. 하지만 조건이 많아질수록 세부적이며 집합의 범위는 작아지게 된다.

Animal클래스의 someAnimal이라는 빈 변수에 snow를 할당해보자. Base class 로써 snow를 가지고 올 수 있다. 이렇게 될 경우 Wolf의 고유 기능인 Hunt()를 쓸 수가 없다. Animal에서는 Hunt()라는 함수가 없기 때문이다. 여기서 someAnimal.Hunt()를 하면 error가 발생하지만 someAnimal.Print()는 가능하다. 적어도 Animal 클래스로서의 역할을 가능하기 때문이다.

Animal someAnimal = snow; 가 될 경우 snow가 Animal로써만 사용할 수 있지만 그렇다고 snow의 고유 기능이 메모리에서 사라지는 것은 아니다. 단지 snow를 Animal로 사용할 때는 사용할 수 없을 뿐이다.

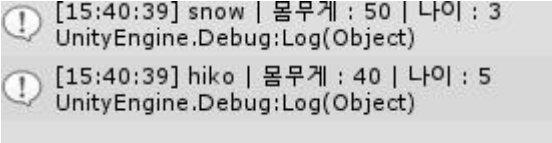
```
Animal someAnimal = snow;
```

```
Wolf ComeBackSnow = (Wolf)someAnimal;
```

```
ComeBackSnow.Hunt();
```

다시 Wolf로 캐스팅해서 가지고 오면 다시 원래대로 사용할 수 있다. 이러한 특징은 매우 큰 장점이 있다. 그래서 이 변수들을 Animal로써 한 번에 관리가 가능하다. 즉 다양한 파생형들을 Base class로써 가지고와 관리가 가능하다는 것이다. 고유의 기능을 사용할 수는 없지만 이러한 상속의 특성을 이용하여 여러 가지 타입(직업, 몬스터, 아이템)등을 관리 할 수 있다.

```
public class AnimalTest : MonoBehaviour {  
    private void Start()  
    {  
        Wolf snow = new Wolf();  
        snow.name = "snow";  
        snow.weight = 50f;  
        snow.year = 3;  
  
        Lion hiko = new Lion();  
        hiko.name = "hiko";  
        hiko.weight = 40f;  
        hiko.year = 5;  
  
        Animal[] animals = new Animal[2];  
  
        animals[0] = snow;  
        animals[1] = hiko;  
  
        for(int i =0; i<animals.Length;i++)  
        {  
            animals[i].Print();  
        }  
    }  
}
```



```
[15:40:39] snow | 몸무게 : 50 | 나이 : 3  
UnityEngine.Debug:Log(Object)  
[15:40:39] hiko | 몸무게 : 40 | 나이 : 5  
UnityEngine.Debug:Log(Object)
```

오버라이드(Override)

부모클래스에서 상속받은 것을 자식클래스에서 재해석하여 사용하는 것을 말한다.
예시로 X축으로 1초에 60도씩 회전하는 코드이다.

```
public class BaseRotator : MonoBehaviour {  
    public float speed = 60f;  
    protected void Rotate()  
    {  
        transform.Rotate(speed * Time.deltaTime,0,0);  
    }  
    private void Update()  
    {  
        Rotate();  
    }  
}
```

만약 x축 외에 y, z축으로도 회전하는 다양한 코드를 만들려면 어떡해야할까? 단순히 원하는 방향으로 각도를 조정하고 싶은 스크립트를 여러 개 만들 수도 있지만 이는 비효율적이다. BaseRotator는 회전함수만 호출하고 회전 구현을 상속받은 자식클래스가 구현을 하면 어떨까? BaseRotator는 회전을 하는 모든 자식클래스에 상속을 한다. 회전이 어떻게 구현되는지는 알 필요가 없다. BaseRotator는 1개이지만 이를 상속받는 여러 자식들이 Rotate()를 자기 스타일에 맞게 재 정의한다.

virtual이라는 키워드는 가상함수라는 키워드로 자식들이 Rotate()를 재 정의를 가능하게 한다.

```
protected virtual void Rotate()  
{  
    transform.Rotate(speed * Time.deltaTime,0,0);  
}
```

즉, virtual void Rotate()는 가상으로 존재하며 자식클래스들이 자식들이 상속받아 재 정의를 한다. 예시로 x, y, z Rotator 스크립트를 각각 생성한다. Override를 할 때는 함수의 형식이 같아야한다. 부모 함수가 protected인데 자식 함수가 다른 접근 지정자일 수는 없다. 부모함수와 자식함수의 이름은 같아야한다.

override를 하게 되면 부모 Rotate()의 transform.Rotate(speed * Time.deltaTime,0,0);가 사라지고 재정의 된 내용으로 적용된다. 만약 부모 함수의 내용을 쓰면서 덧붙이고 싶다면 Base.함수이름, Base.Rotate; 형식으로 사용하면 된다.

```
public class zRotator : BaseRotator {  
    protected override void Rotate()  
    {  
        transform.Rotate(0, 0, speed * Time.deltaTime);  
    }  
}
```

zRotator에서는 Rotate()하나만 작성했지만 BaseRotator에서는 Update()함수에서 Rotate()가 호출되고 있다. 즉, BaseRotator에서는 Rotate()내부는 상관하지 않고 함수만 계속 실행한다. zRotator에서는 Update()를 사용하지 않고 구현부분만 만들어주면 된다. 이렇게 자식클래스들이 수정해야하는 동작 방식만 virtual로 정의하며 그 내부 구현은 상관없이 알아서 함수를 호출 하는 방법이 있다.

상속부분에서 Animal과 달리 zRotator는 MonoBehaviour클래스를 상속받고 있지 않지만 BaseRotator가 대신 상속받고 있기 때문에 zRotator는 오브젝트에 Component로 적용이 될 수 있다. 이 방법 외에도 다양하게 override를 하는 방법이 존재한다.

```
public class BaseRotator : MonoBehaviour {  
  
    public float speed = 60f;  
    protected virtual void Rotate(){}  
    private void Update()  
    {  
        Rotate();  
    }  
}  
  
public class xRotator : BaseRotator  
{  
    protected virtual void Rotate()  
    {  
        transform.Rotate(speed * Time.deltaTime, 0, 0);  
    }  
}  
  
public class yRotator : BaseRotator  
{  
    protected override void Rotate()  
    {  
        transform.Rotate(0, speed * Time.deltaTime, 0);  
    }  
}  
  
public class zRotator : BaseRotator  
{  
    protected override void Rotate()  
    {  
        transform.Rotate(0, 0, speed * Time.deltaTime);  
    }  
}
```

인터페이스(Interface)

상속받은 클래스에게 무조건 어떤 함수나 프로퍼티를 만들도록 강제하는 장치이다, 오브젝트의 타입을 한 번에 인터페이스로써 다룰 수 있다. 모든 오브젝트가 공유하는 그 공통된 기능을 타입을 식별할 필요할 필요 없이 한 번에 사용할 수 있다. 인터페이스를 사용하는 가장 강력한 경우는 아이템, 직업, 몬스터를 구현할 때 공통적인 기능을 구현할 때 사용한다. Interface를 사용하지 않을 땐 예를 들어 캐릭터가 있는데 hp = 50 이고 gold = 1000이라고 하자. medickit이 있고 gold가 있다. 이 둘을 습득 하는 게 목적이다. 두 물체 내에는 캐릭터에게 수치를 적용시킬 함수가 필요한데 기능이 같다. 캐릭터 스크립트에서는 기능이 같은데도 이 두 물체를 적용하기 위해 각각에 코드를 따로 작성한다. 아이템 개수가 적으면 상관없지만 무수한 아이템이 생길 때마다 이 작업을 반복하게 되면 불필요하게 코드가 길어지게 되며 보기도 불편하게 된다. 예시를 보겠다.

기본으로 제공하는 3인칭 캐릭터 스크립트 외에 골드와 hp를 관리하는 스크립트를 만들었다.

Gold와 healPack의 collider에 isTrigger 옵션을 켜고. 각 오브젝트에 GoldItem, HealItem 스크립트를 붙인다. 충돌한 상대방에게서 GoldItem 스크립트가 있는지 확인 후 있으면 goldItem 변수에 저장한다. 충돌한 상대방이 해당 정보를 가지고 있다면 그 정보를 통해 적용하는 간단한 방식이다. 문제는 아이템의 종류가 많다는 것이다. healPack또한 Gold와 동일한 기능의 스크립트를 가진다.

```
public class Player : MonoBehaviour {  
    public float Hp = 50f;  
    public int gold = 1000;  
    private void OnTriggerEnter(Collider other)  
    {  
        GoldItem goldItem = other.GetComponent<GoldItem>();  
        if(goldItem != null)//충돌한 아이템에서 goldtime을 가지고 왔다면  
        {  
            goldItem.Use();  
        }  
        healPack healpack = other.GetComponent<healPack>();  
        if (healpack != null)  
        {  
            healpack.Use();  
        }  
    }  
}
```

```
public class GoldItem : MonoBehaviour {  
    public int goldAmount = 100;  
    public void Use()  
    {  
        Debug.Log("골드를 얻었다.");  
        Player player = FindObjectOfType<Player>();  
        player.gold += goldAmount;  
        gameObject.SetActive(false);  
    }  
}
```

```
public class healPack : MonoBehaviour {  
    public int HealAmount = 10;  
    public void Use()
```

```

{
    Debug.Log("체력이 늘었다.");

    Player player = FindObjectOfType<Player>();
    player.Hp += HealAmount;

    gameObject.SetActive(false);
}
}

```



이처럼 아이템의 개수가 늘어날 때마다 같은 기능에도 불구하고 코드가 계속 늘어나는 것을 확인 할 수 있다. 인터페이스를 상속하는 것들이 무조건 어떤 함수를 구현을 하라고 명시를 해준다. 인터페이스를 사용하면 타입과 상관없이 공통된 기능을 사용할 수 있게 해준다. 인터페이스는 내부를 구현하지 않는다. 또한 C#에서는 다중 상속이 가능하지 않지만 인터페이스는 예외이다. 만약 이 상태에서 Use()를 지우게 되면 에러가 뜬다. 'GoldItem'은 'PlayerItem'의 Use() 인터페이스 멤버를 구현하지 않는다고 나온다. 인터페이스의 첫 번째 역할은 인터페이스를 상속받는 클래스에게 무조건 인터페이스에 정의된 기능을 구현해야한다. 인터페이스에 정의 된 함수 껍데기는 상속받은 클래스가 어떤 식으로 구현하는 아무런 상관이 없다.

goldItem과 HealPack은 Use()를 똑같이 구현하지만 내용은 다르게 정의한다. 다시 말해 인터페이스는 자기 가 정의한 기능만 구현하면 되지 기능을 어떤 타입, 방식으로 구현하는지는 관심이 없다. Player 스크립트를 변경해보자. 기존의 각 타입 별로 정의된 스크립트가 인터페이스 PlayerItem에 대해서 한 번에 정의 된 것을 볼 수 있다. goldItem의 healPack의 다른 고유의 기능은 사용할 수 없지만 적어도 인터페이스에서 정의 된 Use()함수는 공통으로 사용가능하다는 점이 있다. 이렇게 타입이 다른 아이템이 많은 종류가 있어도 인터페이스를 사용하면 코드를 매우 간결하게 할 수 있다.

```

public class Player : MonoBehaviour {
    public float Hp = 50f;
    public int gold = 1000;
    private void OnTriggerEnter(Collider other)
    {
        PlayerItem playerItem = other.GetComponent<PlayerItem>();
        if(playerItem != null)
        {
            playerItem.Use();
        }
    }
}

public interface PlayerItem
{
    void Use();
}

public class healPack : MonoBehaviour, PlayerItem {
    public class GoldItem : MonoBehaviour, PlayerItem {

```

추상클래스(Abstract Class)

추상클래스는 일반적인 클래스와 인터페이스의 중간 역할을 한다. 자식클래스에게 자신이 가지고 있는 멤버변수와 함수를 상속시켜주지만 동시에 자신은 기능을 구현하지 않고 기본형만 있다. 그렇기 때문에 인스턴스가 실시간으로 생성을 할 수 없는 클래스를 말한다. 인터페이스와 다른 점은 인터페이스는 멤버변수 등, 내부의 값, 구현이 가능하지 않지만 추상클래스는 내부구현은 어느 정도 가능하지만 생성할 수 없다는 점이 있다.

예시 스크립트 BaseMonster를 보자. 이 스크립트는 모든 몬스터의 기본이 되는 클래스이고 상속을 한다. BaseMonster에는 몬스터가 가지고 있는 기본적인 내용과 공격함수를 상속해준다. 단, Attack()는 자유자재로 몬스터 타입에 맞게 정의해주면 된다. virtual키워드를 사용하는데 이 키워드를 사용하면 자식이 함수를 재 정의할 수가 있다. 우리는 BaseMonster가 인터페이스역할을 해주었으면 한다. 상속받는 자식들에게 무조건 Attack()을 정의 함으로써 BaseMonster에서는 틀만 남기고 내부는 구현하지 않는다. virtual 키워드를 붙여도 내부를 구현해야한다. 최소한 { }는 붙여야한다.

인터페이스의 한계는 함수의 틀만 정의 할 수 있었다. 내부의 멤버변수나 구현물이 들어가 있는 함수가 존재할 수 없다. BaseMonster는 오브젝트로 찍어낼 수 없기 때문에 기본적인 정보를 상속을 통해 자식에게 전달하고 인터페이스 역할을 하는 것이다. 추상클래스는 클래스 키워드와 데이터 타입 앞에 abstract 키워드를 붙인다. 추상클래스는 자신을 상속받는 자식에게 인터페이스처럼 무조건 정의된 함수를 구현하도록 한다. BaseMonster의 정보로 오브젝트를 생성하고 싶으면 BaseMonster를 상속받은 클래스의 오브젝트를 생성해야한다.

```
public abstract class BaseMonster : MonoBehaviour {
    public float damage = 100f;
    private void Update()
    {
        if(Input.GetKeyDown(KeyCode.Space))
        {
            Attack();
        }
    }
    public abstract void Attack();
}

public class Goblin : BaseMonster {
    public override void Attack()
    {
        Debug.Log("한 캐릭터를 공격했다. 공격력 : " + damage);
    }
}

public class Orc : BaseMonster {
    public override void Attack()
    {
        Debug.Log("광역 공격했다. 공격력 : " + damage * 100);
    }
}
```

또한 다형성이라는 성질 덕분에 Orc와 Goblin을 한 번에 다룰 수 있다. BaseMonster에 대해서 배열을 만들었지만 BaseMonster의 자식클래스인 Orc와 Goblin가 출력하는 등 파생된 것들을 관리할 수가 있다.

```
public class test1 : MonoBehaviour {  
    public BaseMonster[] monster;  
    private void Start()  
    {  
        for(int i =0; i<monster.Length;i++)  
        {  
            Debug.Log(monster[i].gameObject.name);  
        }  
    }  
}
```


프로퍼티(Property)

외부에서는 변수처럼 사용하지만 내부에서는 특수한 처리를 넣을 수 있는 기능이다. 매개변수를 전달하는데 있어 관리하기 위해서 C++에서는 set(),get()를 별도로 지정한다. 하지만 Unity에서는 set(),get()처럼 내부 기능을 사용하여 쉽게 관리할 수 있다.

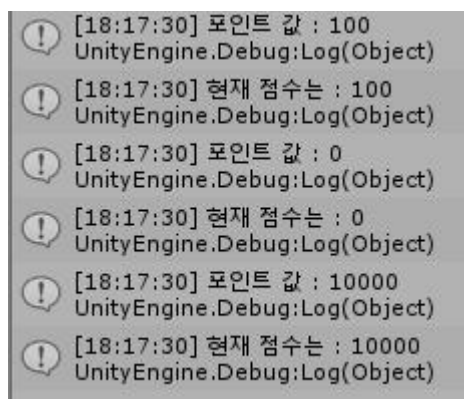
예시로 점수를 관리하는 PointManager 스크립트를 만들었다. 여기서 점수가 음수가 될 수 없다고 가정한다. 불법적인 방식으로 점수를 증가시킬 수 있기 때문에 이를 막기 위해 점수에 제한도 둔다. 그 것들을 막기 위해서 set(),get()을 이용해 함수를 거치도록 만들 수 있다. 만약 함수를 거치지 않고 그냥 변수를 쓴다고 가정한다. test2 스크립트에서 point 변수를 덮어씌운다. 여기서 문제가 생긴다. test2에서 point 값을 음수로 지정해버린다. 이렇게 될 경우 시스템에 어긋나게 되지만 막을 방법이 없다. 누군가 치트를 사용하여 엄청 높은 수를 입력해도 막을 수가 없다.

그래서 멤버변수를 private로 바꾼 뒤 함수를 거쳐서 사용하도록 한다.

```
public class PointManager : MonoBehaviour {  
    private int point = 0;  
    public void SetPoint(int newPoint)  
    {  
        if (newPoint < 0)  
        {  
            point = 0;  
        }  
  
        else if (newPoint > 10000)  
        {  
            point = 10000;  
        }  
  
        else  
            point = newPoint;  
    }  
    public int GetPoint()  
    {  
        Debug.Log("포인트 값 : " + point);  
        return point;  
    }  
}
```

이렇게 되면 test2에서는 point 변수에 접근하기 위해 SetPoint()와 GetPoint()를 사용해야한다. 이를 사용함으로써 SetPoint()에서 정의된 조건문에 반한 값이 들어올 경우 값이 변경되어 적용된다.

```
public class test2 : MonoBehaviour {  
    public PointManager pointManager;  
    void Start () {  
        pointManager.SetPoint(100);  
        int myPoint = pointManager.GetPoint();  
        Debug.Log("현재 점수는 : " + myPoint);  
  
        pointManager.SetPoint(-100);  
        myPoint = pointManager.GetPoint();  
        Debug.Log("현재 점수는 : " + myPoint);  
  
        pointManager.SetPoint(20000);  
        myPoint = pointManager.GetPoint();  
        Debug.Log("현재 점수는 : " + myPoint);  
    }  
}
```



```
[18:17:30] 포인트 값 : 100  
UnityEngine.Debug:Log(Object)  
[18:17:30] 현재 점수는 : 100  
UnityEngine.Debug:Log(Object)  
[18:17:30] 포인트 값 : 0  
UnityEngine.Debug:Log(Object)  
[18:17:30] 현재 점수는 : 0  
UnityEngine.Debug:Log(Object)  
[18:17:30] 포인트 값 : 10000  
UnityEngine.Debug:Log(Object)  
[18:17:30] 현재 점수는 : 10000  
UnityEngine.Debug:Log(Object)
```

문제는 이 과정이 매우 번거롭고 복잡하다는 것이다. Unity에서는 이 과정을 set(),get()으로 구현을 했다. property를 이용하면 외부에서는 point를 사용하는 것 같지만 내부에서는 get과 set의 과정을 거친다. int a = point 라고 하면 get이 적용되며 point = 100; 이라고 하면 set이 적용된다. 다음과 같은 처리를 통해서 같은 결과를 낼 수가 있다.

```
public class PointManager : MonoBehaviour {
    public int point
    {
        get
        {
            Debug.Log(m_point);
            return m_point;
        }
        set
        {
            if (value < 0)
            {
                m_point = 0;
            }
            else
                m_point = value;
        }
    }
    private int m_point = 0;
}
```

```
public class test2 : MonoBehaviour {
    public PointManager pointManager;

    void Start () {
        pointManager.point = 100;
        Debug.Log("현재 점수 : " + pointManager.point);

        pointManager.point = -100;
        Debug.Log("현재 점수 : " + pointManager.point);
    }
}
```

다른 예시로 몬스터를 관리하는 MonsterManager를 만들어 보자. 만약 count란 변수가 public int count = 0; 라고 할 때 test2 스크립트에서 가져다 쓸 수 있다. 문제는 쓰는 도중 실수로 이를 다른 방식으로 덮어 씌워 버릴 수 있기 때문에 위험하다. 그렇기 때문에 get() 통해서만 count를 알 수 있다. 또한 set()을 임의로 지정하지 않게 되게 되면 값을 지정할 수 없기 때문에 이러한 오류를 막을 수 있다.

```
public class MonsterManager : MonoBehaviour {
    public int count
    {
        get {
            Monster[] monster = FindObjectsOfType<Monster>();
            return monster.Length;
        }
    }
}

public class test2 : MonoBehaviour {
    public MonsterManager monsterManager;

    void Start () {
        Debug.Log(monsterManager.count); //변수같지만 사실 get() 통해 얻는다.
    }
}
```

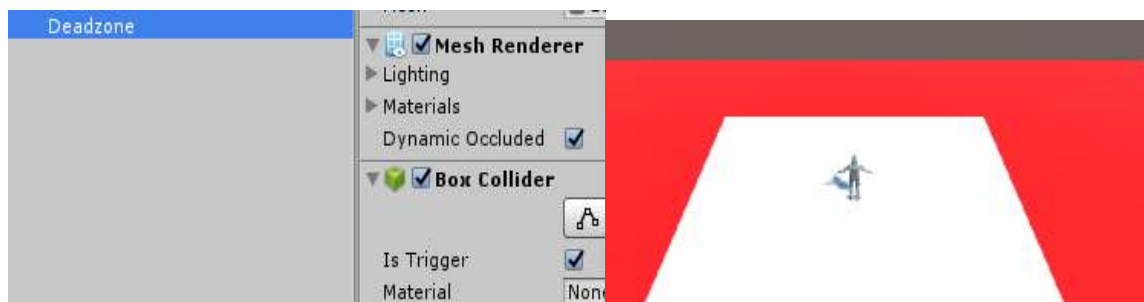
유니티 이벤트(Unity Event)

이벤트는 이벤트가 실행이 되면 이벤트에 등록이 된 기능들이 자동으로 실행되는 것을 말한다. 특징은 등록하는 이벤트 주체와 등록되는 기능들은 아무런 상관이 없는 독립적인 관계이다. 이벤트에 등록해놓은 오브젝트들은 기능을 등록을 해놓았지만 언제 실행되는지는 관여하지 않는다. 등록하는 쪽에서도 어떤 기능이 등록되어있는지 상관하지 않는다. 이러한 특징은 스파게티코드가 되지 않게 하는 강력한 장점이 있다.

예시로 플랫폼 게임이 있다고 하자. 캐릭터가 죽으면 여러 기능들이 동작을 한다. UI에 죽었다고 표시를 하거나 도전과제가 클리어가 되었다고 하거나 등 말이다. 이 기능들을 플레이어가 죽을 때 좀 더 깔끔하게 구현하고 싶다. 중요한 것은 캐릭터의 죽음에 여러 기능들이 연쇄적으로 엮여서 실행된다는 점이다. 예시로 3인칭 캐릭터가 맵의 낭떠러지에 떨어져 특정 오브젝트에 부딪힐 경우 죽는다고 가정한다. 캐릭터가 떨어지면 UI를 갱신하고 도전과제를 클리어 하며 게임을 재시작 하도록 한다. 3인칭 캐릭터 컨트롤러에 스크립트를 붙인다.

```
public class PlayerHealth : MonoBehaviour {  
    private void OnTriggerEnter(Collider other)  
    {  
        Debug.Log("죽었당");  
        Destroy(game오브젝트);  
    }  
}
```

부딪치는 물체에 isTrigger를 체크하면 OnTriggerEnter에서 충돌을 감지한다.



이제는 Debug 메시지가 아닌 플레이어가 죽었을 때 '죽었당'라는 UI를 출력하려한다. UI text를 만들고 UI Manager 스크립트를 적용한다.

```
using UnityEngine.UI;  
  
public class UIManager : MonoBehaviour {  
    public Text playerStateText;  
    public void OnPlayerDead()  
    {  
        playerStateText.text = "넌 죽었당!";  
    }  
}
```

도전 과제 시스템도 만든다.

```
public class AchievementSystem : MonoBehaviour {  
    public Text achivementText;  
    public void UnLockAchivement(string title)  
    {  
        Debug.Log("도전과제 해제! -" + title);  
        achivementText.text = "도전과제 해제! -" + title;  
    }  
}
```

마지막으로 GameManager를 만든다. 씬을 재시작하기 위해서는 `using UnityEngine.SceneManagement;` 라는 namespace를 사용한다. `Invoke()`는 지연시간이 흐르고 해당 함수를 실행하는 기능이 있다.

```
using UnityEngine.SceneManagement;

public class GameManager : MonoBehaviour {

    public void OnPlayerDead()//UIS크립트의 동명함수와 전혀 다름
    {
        Invoke("Restart",5f);
    }

    private void Restart()
    {
        SceneManager.LoadScene(0); //현재 씬 재시작
    }
}
```

이제는 플레이어가 죽었을 때 죽었다는 함수를 발동 시킬 때 3가지 함수를 실행시켜야한다. 이 때 Event가 없으면 PlayerHealth가 다른 스크립트들의 3가지 함수를 모두 알고 있어야한다. 아래처럼 해당 클래스를 접근할 변수를 만들어주어 변수에 접근해야한다.

```
public class PlayerHealth : MonoBehaviour {

    public UIManager uiManager;

    public AchivementSystem achivementSystem;

    public GameManager gameManager;

    private void Dead()
    {
        Debug.Log("죽었따");
        Destroy(game오브젝트);

        uiManager.OnPlayerDead();
        achivementSystem.UnLockAchivement("뉴턴의 법칙");
        gameManager.OnPlayerDead();
    }

    private void OnTriggerEnter(Collider other)
    {
        Dead();
    }
}
```



Event없이 Call by Reference 방식으로 다른 클래스를 가지고와서 사용하는 예시이다. 문제는 플레이어가 죽을 때 관여하는 모든 것을 PlayerHealth가 알고 있어야 한다. 복잡할 수 있고 비효율적이다. Achivement 시스템 경우 필수적인 기능이 아님에도 PlayerHealth가 개발 처음부터 알고 있어야한다면 매우 비효율적이다. 꼭 필요한 기능이 아님에도 공간을 차지하고 있다면 시각적으로 불편하며 성능에도 지장을 줄 수 있게 된다. 플레이어는 자기 자신에 관한 것만 관리하면 된다. 플레이어가 죽었을 때 다른 기능이 실행되게 하는데 있어 기능들 간에 느슨하게 연결할 수 있다.

이는 소프트웨어 유지보수 측면에서도 모듈간의 응집도와 매우 밀접한데 Event를 사용함으로써 모듈간의 응집도를 낮출 수 있는 장점이 된다.

UnityEvent를 사용하려면 `using UnityEngine.Events;` namespace를 사용해야한다.

```
using UnityEngine.Events;

public class PlayerHealth : MonoBehaviour {

    public UnityEvent onPlayerDead;

    private void Dead()
```

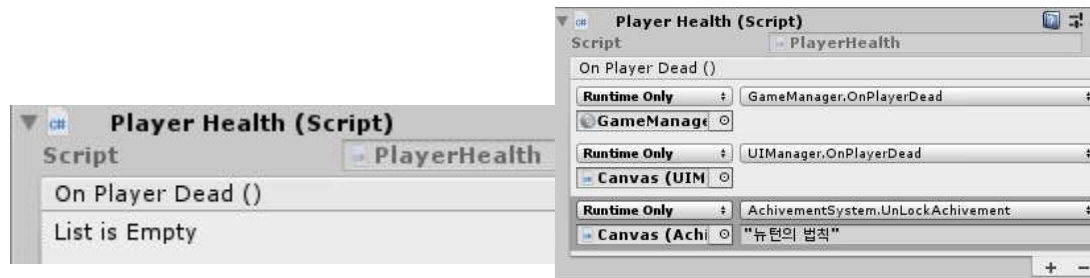
```

{
    onPlayerDead.Invoke();//UnityEvent 실행

    Debug.Log("죽었따");
    Destroy(game오브젝트);
}

private void OnTriggerEnter(Collider other)
{
    Dead();
}
}

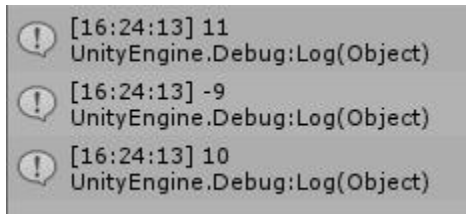
```



쉽게 생각한다면 main()에서 함수를 사용하고 매개변수를 넣고 하는 작업을 이 UnityEvent에서 대신 할 수 있다고 생각하면 편하다. AchievementSystem() 경우 매개변수가 String이기 때문에 매개변수를 쓸 수 있는 란도 생긴 것을 볼 수 있다.

델리게이트(Delegate)

‘위임 하다’라는 뜻으로 어떤 기능을 목록에 추가해놓으면 Delegate가 대신 실행 해주는 역할을 한다. 예시로 계산기를 만들어본다. 계산기는 계산을 대행해준다. 우리의 계산을 해주지만 어떤 내용을 계산하는지는 모른다. 단지 계산을 할 뿐이다. 일단 Delegate를 사용하지 않고 만들어보겠다. 실행하는 순간 어떤 계산을 할지 명시를 해주어야한다. 계산기가 어떤 계산을 하는지 알고 있어야한다는 뜻이다. 계산기가 자기가 가진 기능 중 어떤 기능을 쓰는지 알고 있다.



space를 누를 때마다 계산이 실행된다.

계산기가 어떤 계산을 하는지가 왜 그렇게 문제인가 하면 계산기는 계산만 하면 되지 무엇을 계산하는지까지는 알 필요가 없기 때문이다. Delegate는 리스트에 추가된 항목이 무엇인지는 상관하지 않고 실행만 한다. Delegate는 ‘~한 형식을 지정해놓으면 대신해주겠다.’ 라는 뜻이 된다. Delegate는 함수 포인터를 가지고 있다. 즉, Delegate에는 리스트에 있는 함수를 가리키는 포인터를 가지고 있다. 코드 상에서는 변수가 아닌 새로운 형을 정의한다. 새로운 함수나 변수를 만드는 것이 아니라 새로운 Delegate를 정의한 것이다. Calculate란 형의 delegate(delegate float Calculate(float a, float b))는 출력이 float을 리턴하고 float 형의 변수 2개를 입력으로 하는 함수만 대행하여 처리해준다는 것이다. Delegate는 괄호를 붙이지 않는다. 괄호를 붙인다는 것은 함수를 호출한다는 의미이다. 괄호를 붙이지 않는 것은 sum()을 리스트에 등록한다는 것이지 호출한다는 의미는 아니다. Debug 메시지를 통해서 결과 값을 받아 볼 수 있다.

```
public class Calculator : MonoBehaviour {
    delegate float Calculate(float a, float b);

    Calculate onCalculate;

    private void Start()
    {
        onCalculate = sum;
    }

    public float sum(float a, float b)
    {
        Debug.Log(a+b);
        return a + b;
    }

    public float sub(float a, float b)
    {
        Debug.Log(a - b);
        return a - b;
    }

    public float mul(float a, float b)
    {
        Debug.Log(a * b);
        return a * b;
    }

    private void Update()
    {
        if(Input.GetKeyDown(KeyCode.Space))
        {
            //sum(1,10)이 실행된다. 하지만 Calculator는 이를 알필요가 없다.
            onCalculate(1,10);
        }
    }
}
```

```
        Debug.Log("결과값 : " + onCalculate(1, 10));
    }
}
```

onCalculate = sum;
onCalculate += sub;을 하면 sum 결과에 sub결과 까지 같이 출력할 수 있게 한다. 주의할 점은 결과 값을 출력할 때 대행 해준 결과 값의 마지막 결과 값만 출력한다는 점이 있다.

이벤트(Evnet)

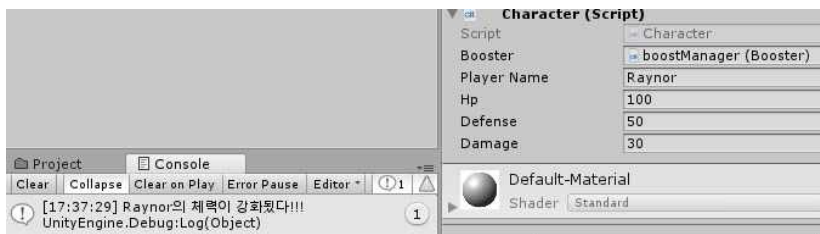
이벤트를 사용하는 이유는 두 오브젝트 사이의 응집도(coupling)를 낮추기 위해서이다. 응집도가 높을 경우 오브젝트간의 수정이 어렵고 제거나 생성 등이 용이하지 않다. 이벤트에서 구분하는 용어가 있는데 이벤트 리스트를 가지고 실행시키는 쪽을 publisher 라고 하고 이벤트에 자신의 기능을 등록시키고 대기하는 쪽을 subscriber라고 한다. 이벤트를 구현하는데 delegate를 사용한다.

예시를 들어보자. 플레이어가 게임시작 전 아이템을 사용해서 부스터가 되었다. 부스터팩은 동시에 여러 개를 중첩 사용할 수 있다. 플레이어는 중첩된 효과의 경우를 생각할 수가 없다. 효과의 내용을 모른 채로 부스터팩을 사용한다. 이럴 때 event가 필요하다.

Character 스크립트와 Booster 스크립트를 만들었다. 문제는 어떻게 Booster의 기능을 적용시킬 것 인가 이다. 이벤트를 사용하지 않으면 Chracter 스크립트에서 Booster의 사용할 기능을 알고 있어야한다.

```
public class Character : MonoBehaviour {  
    public Booster booster;  
    public string playerName = "Raynor";  
    public float hp = 100f;  
    public float defense = 50f;  
    public float damage = 30f;  
    private void Awake()  
    {  
        booster.HealthBoost(this);  
    }  
}
```

```
public class Booster : MonoBehaviour {  
    public void HealthBoost(Character target)  
    {  
        Debug.Log(target.playerName + "의 체력이 강화됐다!!!");  
        target.hp += 10;  
    }  
    public void shieldBoost(Character target)  
    {  
        Debug.Log(target.playerName + "의 방어력이 증가했다!!!");  
        target.defense += 50;  
    }  
    public void DamageBoost(Character target)  
    {  
        Debug.Log(target.playerName + "의 공격력이 증가했다!!!");  
        target.damage += 10;  
    }  
}
```



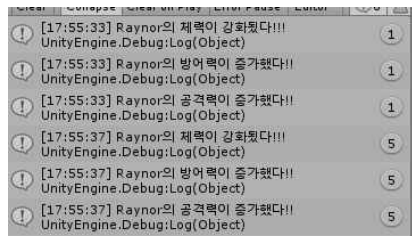
문제는 Event를 사용하지 않으면 여러 조합에 대응하기가 힘들다. 미리 스크립트를 변경하는 것 외에는 변경하는 것이 불가능하기 때문이다. 우리는 캐릭터가 기능만 실행시키고 어떻게 실행시키는지 신경 쓰고 싶지 않다.

```
public class Character : MonoBehaviour {
    public delegate void Boost(Character target);
    public Boost playerBoost;
    public string playerName = "Raynor";
    public float hp = 100f;
    public float defense = 50f;
    public float damage = 30f;
    private void Start()
    {
        playerBoost(this);
    }
    private void Update()
    {
        if(Input.GetKeyDown(KeyCode.Space))
        {
            playerBoost(this);
        }
    }
}
```

이제 Chracter 입장에서는 Booster가 어떤 기능을 가지고 있는지 알 필요가 없다.

```
public class Booster : MonoBehaviour {
    public void HealthBoost(Character target)
    {
        Debug.Log(target.playerName + "의 체력이 강화됐다!!!");
        target.hp += 10;
    }
    public void shieldBoost(Character target)
    {
        Debug.Log(target.playerName + "의 방어력이 증가했다!!");
        target.defense += 50;
    }
    public void DamageBoost(Character target)
    {
        Debug.Log(target.playerName + "의 공격력이 증가했다!!");
        target.damage += 10;
    }
    private void Awake()
    {
        Character player = Find오브젝트OfType<Character>();
        player.playerBoost += HealthBoost;
        player.playerBoost += shieldBoost;
        player.playerBoost += DamageBoost;
    }
}
```

중요한 점은 플레이어가 booster하는 내용물이 달라졌다. 예를들어 Chracter가 shieldBoost를 쓰지 않을 경우 Chracter 스크립트를 변경시켜야하지만 지금은 Chracter 스크립트를 변경하지 않는다. playerBoost만 바꾸면 된다. 시작 시 boost가 적용되고 space를 누를 때마다 적용된다.



그런데 이런 의문이 들 수 있다. delegate로 이벤트를 만들 수 있는데 왜 event라는 키워드가 있는가? delegate앞에 event라는 키워드를 붙일 수 있는데(`public event Boost playerBoost;`) 이는 delegate가 event가 아닌 방향으로 잘못 사용되는 것을 제한하는 역할을 한다.

delegate Boost같은 경우 public 변수이다. `player.playerBoost = DamageBoost;` 이렇게 된 경우 덮어 씌우기가 된다. 여태 적용된 모든 기능들이 사라져버린다. 이 상태에서 실행할 경우 DamageBoost만 적용되는 것을 볼 수 있다. event는 subscriber가 자유롭게 등록하고 뺄 수 있어야하는데 누군가 덮어 씌워서 기능을 망치면 안 된다. event라는 키워드를 붙이면 `player.playerBoost = DamageBoost;`를 했을 경우 컴파일 에러가 난다. 이렇게 event키워드는 굳이 필요는 없지만 실수로 인한 에러를 막아 줄 수 있다.



액션과 람다함수(Action, Lambda)

Action은 c#에서 미리 사용하게 쉽게 만들어진 delegate이며 Lambda는 오브젝트를 생성하듯이 코드도중에 이름이 없는 함수를 만들고 매개변수처럼 사용할 수 있다.

```
public class Worker : MonoBehaviour {  
    delegate void Work();  
  
    Work work;  
  
    void MoveBricks()  
    {  
        Debug.Log("벽돌을 옮겼다");  
    }  
  
    void DigIn()  
    {  
        Debug.Log("땅을 팠다");  
    }  
  
    private void Start()  
    {  
        work += MoveBricks;  
        work += DigIn;  
    }  
  
    private void Update()  
    {  
        if(Input.GetKeyDown(KeyCode.Space))  
        {  
            work();  
        }  
    }  
}
```

이처럼 void 형식의 입력은 없는 delegate함수는 정말 많다. Return 타입이 없는 형식이 매우 많아서 C# 내부에 기능을 구현을 해놓았다. using System;을 선언하고 delegate void Work();을 지우고 Action work; 만 선언해도 전과 같은 기능을 할 수 있다. Action work;은 입력이 없고 리턴타입이 void인 delegate, delegate void Action();과 같다.

```
public class Worker : MonoBehaviour {  
    Action work;  
  
    void MoveBricks()  
    {  
        Debug.Log("벽돌을 옮겼다");  
    }  
  
    void DigIn()  
    {  
        Debug.Log("땅을 팠다");  
    }  
  
    private void Start()  
    {  
        work += MoveBricks;  
        work += DigIn;  
    }  
  
    private void Update()  
    {  
        if(Input.GetKeyDown(KeyCode.Space))  
        {  
            work();  
        }  
    }  
}
```

```
}  
}
```

Lambda함수를 살펴 보자. 먼저 일반적인 delegate 함수를 보자. onSend("Raynor")을 통해 Raynor에게 돈과 메일이 보내는 것을 출력할 수 있다.

```
public class messenger : MonoBehaviour {  
    public delegate void Send(string receiver);  
    Send onSend;  
    private void Start()  
    {  
        onSend += SendMail;  
        onSend += SendMoney;  
    }  
    private void Update()  
    {  
        if(Input.GetKeyDown(KeyCode.Space))  
        {  
            onSend("Raynor");  
        }  
    }  
    void SendMail(string receiver)  
    {  
        Debug.Log("Mail sent to" + receiver);  
    }  
    void SendMoney(string receiver)  
    {  
        Debug.Log("Mail sent to" + receiver);  
    }  
}
```

만약 이 스크립트에 정의되지 않은 새로운 기능을 실시간으로 추가한다고 하자. 'man'이라고 하는 대상을 없앤다는 함수를 즉석으로 만들어보자. Lambda함수는 이름이 없는 함수를 뜻한다. man => Debug.Log("Assasinate" + man);는 함수인 동시에 오브젝트이기도 하다.

예를 들어 float a = 3.14f라고 하자. 우리는 3.14f에 a라는 이름을 붙인 것이지 3.14f가 a라는 것은 아니다. 값 그 자체는 이름이 없다. 즉, 이름이 없다면 그 함수를 값 혹은 오브젝트처럼 다른 곳에게 적용할 수 있지 않을까?

```
void Assasinate(string man)  
{  
    Debug.Log("Assasinate" + man);  
}
```

이런 식으로 사용가능 하지만 실시간으로 만들 수 있게 표현할 수 있는 것이 Lambda함수며 익명함수라고도 한다. man => 부분 경우 입력인데 데이터 타입을 적지 않아도 된다. C#컴파일러가 알아서 다 처리해주기 때문이다. 이름이 없는 것들을 만들면 미리 등록된 것이 아니라 이리저리 전달할 수 있는 것이 값이다. 같은 표현으로 onSend += (string man) => { Debug.Log("Assasinate " + man); }; 쓸 수도 있다. Lambda함수를 사용하면 즉석해서 delegate에 적용하여 사용할 수 있다.