UNITY
-CAHPTER7-

SOUL SEEK

목차

다형성을 이용한 구성 – LIVINGENTITY CLASS

1. 다형성을 사용한 패턴

- **Monster** 클래스와 이를 상속하는 **Orc, Dragon** 클래스가 있다고 했을 때,

```
public class Monster : MonoBehaviour
{
    public float damage = 100;

    public void Attack()
    {
        Debug.Log("공격!");

        //공격 처리...
    }
}

public class Orc : Monster
{
    public void WarCry()
    {
        Debug.Log("전투함성!");
        //전투 함성 처리...
    }
}

public class Dragon : Monster
{
    public void Fly()
    {
        Debug.Log("날기");
        //공중을 나는 처리...
    }
}
```

- 사용하는 곳에서 **Monster**의 자식이기 때문에 전체를 관리하기 위해 자식을 **Monster**로 캐스팅해서 사용하려고 한다.

```
//Scene에 Load되어 있는 Orc Type이 존재하는지 찾아서 할당받는다.
Orc orc = FindObjectOfType<Orc>();
//다수일 수도 있는 Type은 아무것이나 가져올 수도 있기 때문에 주의하자.
//Orc orc = FindObjectsOfType<Orc>();

Monster monster = orc; // 몬스터 타입의 변수에 오크를 할당한다.
monster.Attack(); // 실행 가능
monster.WarCry(); // 실행 불능
```

Monster monster = orc;가 실행되면 **orc**에 할당된 **Orc** 타입의 오브젝트가 **Monster Type**으로 취급된다.
Monster.Attack();는 **Orc** 오브젝트의 **Attack() Method**를 **Monster Type**으로 실행한다.
하지만 실제로 **Orc Type**의 오브젝트를 **Monster** 타입으로 변형하는 것이 아니다.
Monster Type의 변수 **monste**에 할당된 오브젝트가 **Orc Type** 이라는 것을 추측할 수 없기 때문에 **monster.WarCry();**는 실행할 수 없다.
WarCry() 메서드는 **Monster** 타입에 없기 때문.

1. 다형성을 사용한 패턴

- **WarCry**는 모든 주의 영향을 받는 **Monster**의 능력치를 올려주는 구현이라면, 반대로 모든 **Monster Class**를 상속받는 자식 **Class**들은 **Monster**라고 인식할 수 있다. 그렇다면 다형성을 활용해서 다양한 자식 **Type**을 하나의 부모 **Type**으로 다뤄 코드를 쉽고 간결하게 만들 수 있다.
- 자식 **Type**이 할당될 때 부모의 **Type**도 할당이 된다 그러므로 모든 **Monster**를 상속받은 **Class**는 **Monster Type**을 가지고 있다 그렇기 때문에 **Monster Class**의 자원들을 이용해서 **Monster Class**를 상속받는 모든 자식 **Class**에게 간접적으로 영향을 줄 수 있다.

```
public void WarCry()
{
    Debug.Log("전투함성!");

    //모든 Monster Object를 찾아 공격력을 10 증가시킨다.
    Monster[] monsters = FindObjectsOfType<Monster>();

    for(int i = 0; i < monsters.Length; i++)
    {
        monsters[i].damage += 10;
    }
}
```

2. 오버라이드(OVERRIDE)

같은 이름으로 서로 다른 방식으로 동작하게 할 수 있는 방법.
부모 클래스에서 작성한 **Method**는 자식 클래스에서 **재정의**해서 사용할 수 있다.
일반적인 **Method**의 확장성으로 사용하는 같은 이름으로 사용하는 **오버로딩과 다르다**는 것을 알아야 한다.
부모 **Class**에서 정의한 **Method**가 각자 상속한 자식 **Class**에서 서로 다르게 동작할 필요성이 있을 때 사용하는 방법.

```
public class Monster : MonoBehaviour
{
    public virtual void Attack()
    {
        Debug.Log("공격!");
        //공격 처리...
    }
}

public class Orc : Monster
{
    public override void Attack()
    {
        base.Attack();
        Debug.Log("우리는 노예가 되지 않는다.");
    }
}

public class Dragon : Monster
{
    public override void Attack()
    {
        base.Attack();
        Debug.Log("모든 것이 불타오를 것이다!");
    }
}
```

Virtual 키워드로 지정된 **Method**는 **virtual Method**가 된다.
virtual Method는 자식 **Class**가 **override** 할 수 있도록 허용된 **Method**를 재정의 할 수 있다.

```
Orc orc = FindObjectOfType<Orc>();
Monster monster = orc;
monster.Attack(); // Orc의 Attack()이 실행됨
```

Monster.Attack();이 실행되면 **Orc Class**에서 재정의한 **Attack() Method**가 실행된다.
따라서 **Debug.Log(“공격!”), Debug.Log(“우리는 노예가 되지 않는다.”)**가 차례로 실행된다.

부모의 **Method**를 실행할 필요가 없는 경우에는 사용하지 않아도 된다.

3. LIVINGENTITY CLASS – EVENT, ACTION DELEGATE

- **Enemy AI**와 **Player**를 포함해서 게임 속 생명체들은 몇 가지 공통 기능을 가져야 한다.
  →체력을 가진다, 체력을 회복할 수 있다, 공격을 받을 수 있다, 살거나 죽을 수 있다.

```
public class LivingEntity : MonoBehaviour, IDamageable
{
    public float startingHealth = 100f; // 시작 체력
    public float health { get; protected set; } // 현재 체력
    public bool dead { get; protected set; } // 사망 상태
    public event Action onDeath; // 사망시 발동할 이벤트
```

- **startingHealth**는 **LivingEntity**가 활성화될 때 **health**에 할당될 기본 체력
- **health**는 현재 체력을 나타내고, **dead**는 죽은 상태를 나타내고 이 두가지는 프로퍼티 형태로 제공.
- **event Action onDeath**에서 **onDeath**는 사망 시 발동될 이벤트
  →사망 시 실행할 **Method**들이 등록되고 이를 이용해 **LivingEntity**를 가진 **Object**가 죽었을 때 어떤 일이 발생할지 결정할 수 있다.

Action

- 입력과 출력이 없는 **Method**를 가리킬 수 있는 **Delegate**
  → **Method**를 값으로 할당 받을 수 있는 **Type**
- 변수에는 **void SomeFunction()**처럼 입력과 출력이 없는 **Method**를 등록할 수 있다.
  →등록된 **Method**는 원하는 시점에 매번 실행할 수 있다.

3. LIVINGENTITY CLASS – EVENT, ACTION DELEGATE

- 청소를 원하는 시점에 **Delegate**를 실행하고 있다

```
public class Cleaner : MonoBehaviour
{
    Action onClean;

    void Start()
    {
        onClean += CleaningRoomA;
        onClean += CleaningRoomB;
    }

    void Update()
    {
        if(Input.GetMouseButtonDown(0))
        {
            onClean();
        }
    }

    void CleaningRoomA()
    {
        Debug.Log("A방 청소");
    }

    void CleaningRoomB()
    {
        Debug.Log("B방 청소");
    }
}
```

마우스를 클릭할 때마다 **onClean**에 등록된 방청소 **Method**가 실행된다.

Start() Method에서 **onClean**에 방을 청소하는 **Method**를 등록한다.

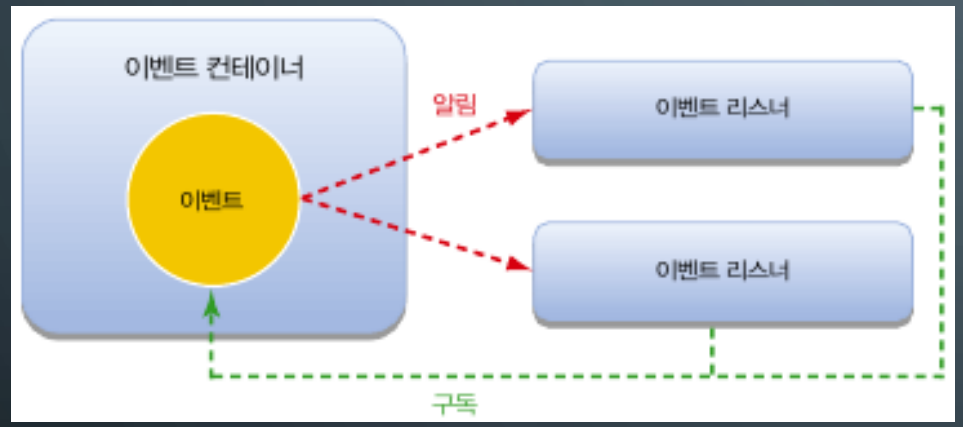
onClean에 **Method**를 등록한 후 **onClean();**을 실행하면 등록된 **Method**가 일괄 실행된다.

Update() Method에서 마우스 왼쪽 버튼을 클릭할 때마다 **onClean();**을 실행한다.
onClean();이 실행되면 **onClean**에 등록된 **CleaningRoomA()**와 **CleaningRoomB() Method**가 실행되어 **'A**방 청소**', 'B**방 청소**'**로그가 순서대로 출력된다.

3. LIVINGENTITY CLASS – EVENT, ACTION DELEGATE

Event

- 연쇄 동작을 이끌어내는 시건
  → 이벤트 자체는 어떤 일을 실행하지 않지만 이벤트가 발생하면 이벤트에 등록되어 구독하고 있는 **Method**들에게 연쇄적인 실행을 알려준다.
- 이벤트를 사용하면 어떤 클래스에서 특정 사건이 일어났을 때 다른 클래스에서 그것을 감지하고 관련된 처리를 실행할 수 있다.
- 이벤트를 구현할 때는 이벤트와 이벤트에 관심이 있는 **Event Listener**로 오브젝트를 구분한다.



C#에서 이벤트를 구하는 대표적인 방법은 **Delegate**를 클래스 외부로 공개하는 방법.
→외부로 공개된 **Delegate**는 클래스 외부의 **Method**가 등록될 수 있는 명단이자 이벤트가 된다. 그리고 이벤트가 발동**(Invoke)**하면 이벤트에 등록된 메서드들이 모두 실행된다.
→등록된 이벤트들은 항상 대기하다가 이벤트 발동될 때 실행되는 **Method**들을 **Event Listener**라고 한다.

- **Event Listener**를 이벤트에 등록하는 것을 **Event Listener**가 이벤트를 구독한다고 표현한다.
- 이벤트는 자신을 구독하는 **Event Listener**들이 어떤 처리를 실행하는지 상관하지 않는다는 점에 주목하자.
  →이벤트는 자신의 명단에 등록된 **Method**들의 내부 구현을 알지 못한 채 그들을 실행한다.

3. LIVINGENTITY CLASS – EVENT, ACTION DELEGATE

Event는 견고한 커플링을 해소할 수 있다.

- 자신을 구독하는 **Method**의 구현과 상관없이 동작하므로 견고한 커플링 문제를 해소할 수 있다.
  →어떤 클래스가 다른 클래스의 구현에 강하게 결합되어 코드를 유연하게 변경할 수 없는 상태를 견고한 커플링이라고 한다.

Ex)플레이어가 죽었을 때 게임 데이터를 저장하는 기능을 구현한다고 가정해보자.

```
public class Player : MonoBehaviour
{
    public GameData gameData;

    public void Die()
    {
        // 실제 사망 처리...
        gameData.Save();
    }
}

public class GameData : MonoBehaviour
{
    public void Save()
    {
        Debug.Log("게임 저장...");
    }
}
```

```
public class Player : MonoBehaviour
{
    public Action onDeath;

    public void Die()
    {
        onDeath();
    }
}

public class GameData : MonoBehaviour
{
    private void Start()
    {
        Player player = FindObjectOfType<Player>();
        player.onDeath += Save;
    }

    public void Save()
    {
        Debug.Log("게임 저장...");
    }
}
```

3. LIVINGENTITY CLASS – EVENT, ACTION DELEGATE

event Type

Delegate Type의 변수는 **event** 키워드를 붙여 선언할 수 있다.
→어떤 **Delegate** 변수를 **event**로 선언하면 클래스 외부에서는 해당 **Delegate**를 실행할 수 없다.
→이벤트를 소유하지 않은 측에서 멋대로 이벤트를 발동하는 것을 막을 수 있다.

```
public class Player : MonoBehaviour
{
    public event Action onDeath;

    public void Die()
    {
        onDeath();
    }
}

public class GameData : MonoBehaviour
{
    private void Start()
    {
        Player player = FindObjectOfType<Player>();
        player.onDeath += Save;
        // player.onDeath(); //Error
    }

    public void Save()
    {
        Debug.Log("게임 저장...");
    }
}
```

- **Player**의 **Death** 이벤트를 등록만하고 **Player**의 사망상태에서 발동하게 된다.
- **Player**가 **onDeath**의 상태의 주체이기 때문에 이렇게 해줘야 하고 외부에서는 **Death**상태일때의 이벤트를 직접적으로 사용할 수 없게 해야 한다.

3. LIVINGENTITY CLASS – EVENT, ACTION DELEGATE

OnEnable() & OnDamage()

OnEnable()

- **LivingEntity**가 활성화가 되었을때 **OnEnable**을 이용해서 생명체의 상태를 리셋한다.
  →생존유무의 상태를 판단하고 초기화해야 하는 방법에는 여러 방법이 있다. 그 중 하나가 **Component**를 비활성화한 상태에서 **GameScene**으로 로드되어 나타났을 때 **LivingEntity**를 활성화해서 이 순간을 체크하는 **OnEnable()**을 사용하는 방법이다.

OnDamage()

입력으로 받은 **Damage**만큼 현재 **health**를 깎는다. 그리고 현재 체력이 **0**보다 작거나 같고, 아직 사망한 상태가 아니라면 **Die() Method**를 실행해 사망처리를 실행한다.

```
protected virtual void OnEnable()
{
    // 사망하지 않은 상태로 시작
    dead = false;
    // 체력을 시작 체력으로 초기화
    health = startingHealth;
}
```

```
public virtual void OnDamage(float damage, Vector3 hitPoint, Vector3 hitNormal)
{
    // 데미지만큼 체력 감소
    health -= damage;

    // 체력이 0 이하 && 아직 죽지 않았다면 사망 처리 실행
    if (health <= 0 && !dead)
    {
        Die();
    }
}
```

3. LIVINGENTITY CLASS – EVENT, ACTION DELEGATE

RestoreHealth() & Die()

RestoreHealth()

- 체력을 회복하는 **Method,** 입력 받은 회복량 **newHealth**만큼 현재 체력 **health**를 증가.
  →이미 죽은 상태에서는 체력을 회복할 수 없기 때문에 빠져나갈 수 있게 만들어야 한다.

Die()

- **LivingEntity**의 죽음을 구현한다.
  → **onDeath** 이벤트를 발동하여 이벤트에 등록된 **Method**를 실행한다.
  → **onDeath**에 등록된 이벤트가 없으면 발동할 수 없으니 이벤트가 있는 경우만 체크하게 한다.

```
public virtual void RestoreHealth(float newHealth)
{
    if (dead)
    {
        // 이미 사망한 경우 체력을 회복할 수 없음
        return;
    }

    // 체력 추가
    health += newHealth;
}
```

```
public virtual void Die()
{
    // onDeath 이벤트에 등록된 메서드가 있다면 실행
    if (onDeath != null)
    {
        onDeath();
    }

    // 사망 상태를 참으로 변경
    dead = true;
}
```

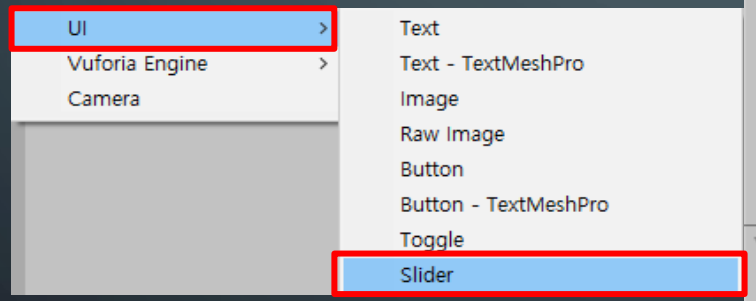
플레이어 체력 UI & PLAYERHEALTH SCRIPT

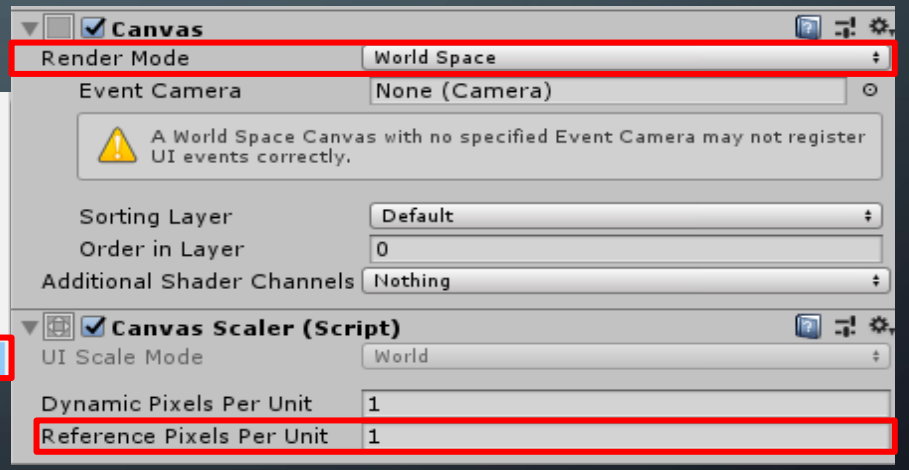
1. 플레이어 체력 UI

Player의 체력**UI**를 구현하기 먼저 **Player**와 함께 움직이는 **UI**를 만들 것이다**.**

UI Slider 만들기

1. 새로운 **Slider GameObject**를 생성한다**(Create > UI > Slider),** 하이어라키 창에서 **Canvas GameObject** 선택
2. **Canvas Component**의 **Render Mode**를 **World Space** 로 변경**, Canvas Scaler Component**의 **Reference Pixel per Unit**을 **1**로 변경**.**



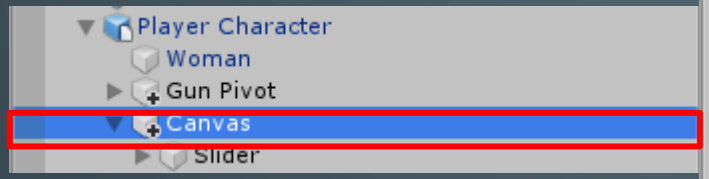


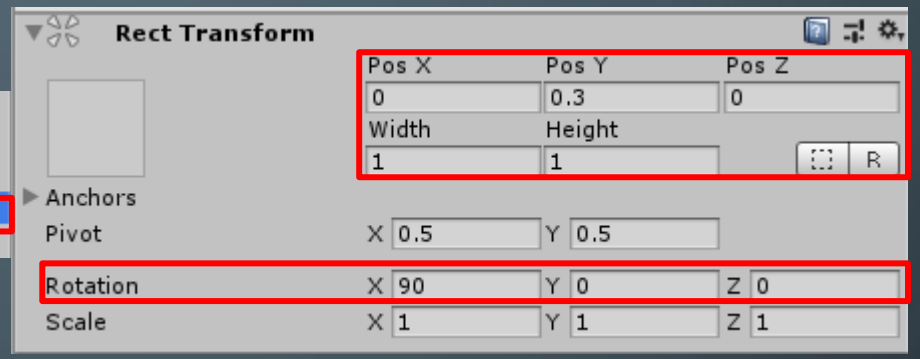
- **UGUI**의 **Canvas**의 **Render Mode**를 이용해서 **Render**가 발생할 때 어떤 변환에 적용을 받을 건지를 결정하면서 **Render** 설정을 정해준다**. 3D**의 변환을 적용 받을 것인지 아니면 깊이관계만 받을 것인지 아예 받지 않을 것인지 설정해주는 것이다**.Reference Pixels Per Unit**은 **UI**의 스프라이트와 **World** 유닛들 간의 크기비율을 말하는 것이다**. Pixel**단위의 크기는 월드단위의 크기와 개념이 다르기 때문에 비율로 설정해야 하고 그 비율 값을 설정하는 것이다**.**

1. 플레이어 체력 UI

Canvas의 위치와 크기 설정

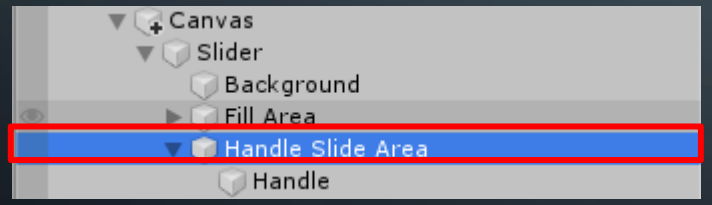
- **Canvas GameObject**를 **Player GameObject**의 자식으로 만들자**, Rect Transform Compone nt** 의 위치를 **(0, 0.3, 0), Width**와 **Height**를 **1**로 변경**, Rotation**을 **(90, 0, 0)**으로 변경

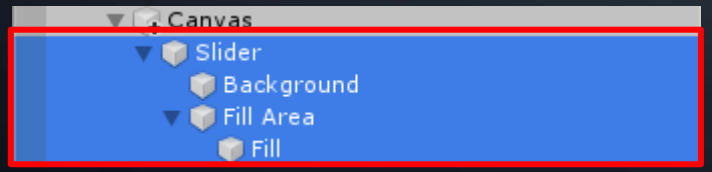


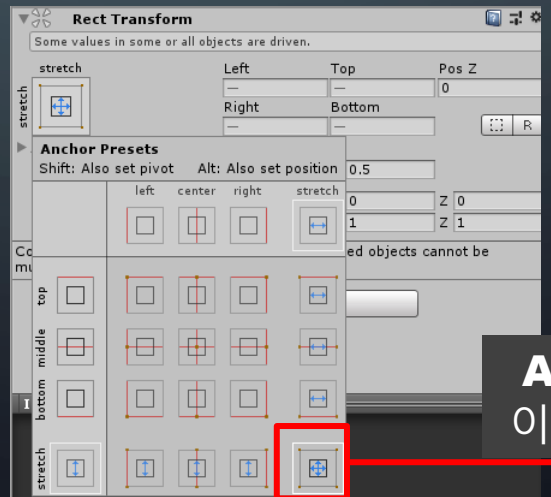


Slider 크기변경

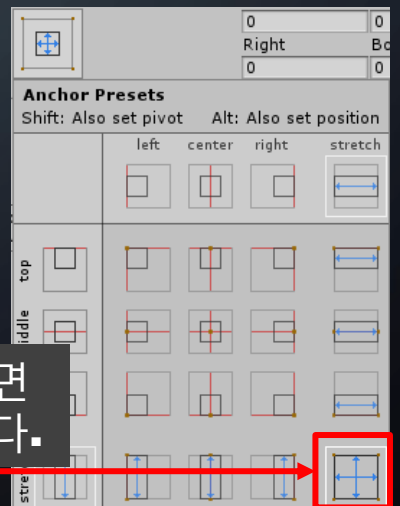
- 하이어라키 창에서 **Canvas**에서 **Handle Slide Area**를 삭제**, Slider, Background, Fill**을 모두 선택**, AnchorPresets**를 변경하기 전에 **Art**를 누르면 **Stretch**가 되는데 그 상태에서 우측 하단을 선택하면 된다**.**







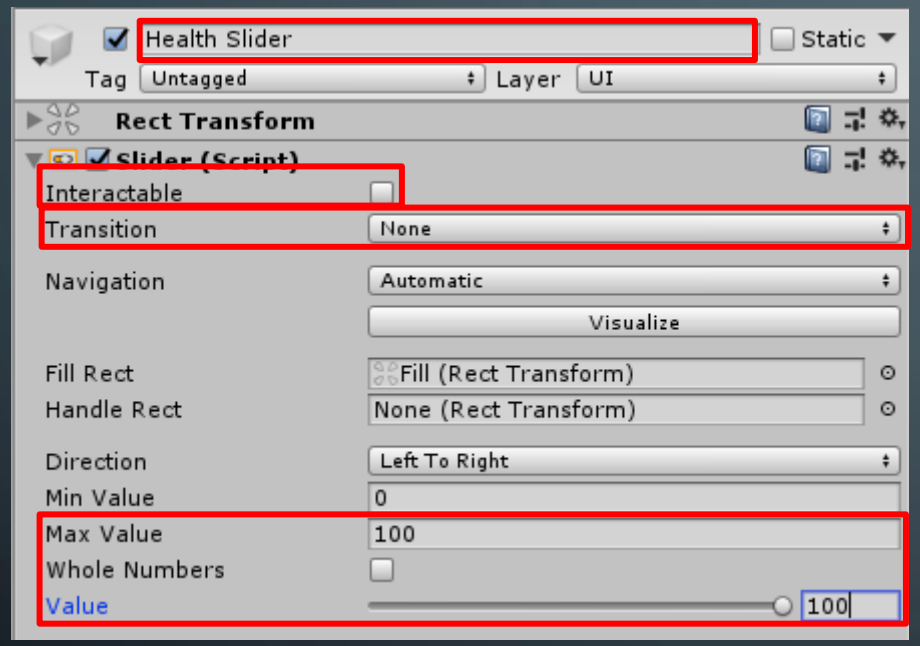
Art를 누르면
이렇게 변한다**.**



1. 플레이어 체력 UI

Slider Component 설정

1. **Slider GameObject > GameObject** 이름을 적절하게 설정, **Slider Component**에서 **Interacta ble** 체크 해제, **Transition**을 **None**으로 변경
2. **Trnasition**을 **None**으로 변경, **Max Value**와 **Value**를 **100**으로 변경



Interactable 체크를 해제하여 유저가 **Control** 해서 **Slider**가 움직이는 것을 방지한다.

Transition Field는 **UI**와 상화작용 시 일어나는 시각 피드백을 설정한다. 예를 들어 **Transition**이 **Color Tint**로 설정된 경우 **UI** 요소에 마우스를 가져다 대거나 클릭하면 색이나 투명도가 잠시 변한다. 하지만 체력바는 그런 변화가 필요 없기 때문에 **None**으로 처리한다.

Slider Component는 **Min Value, Max Value, Value**필드로 전체 크기 중 얼마만큼 채워 줄지를 결정한다.
→**Min Value**와 **Max Value** 사이에서 **Value**가 차지하는 비율에 맞춰 슬라이더를 채운다.

1. 플레이어 체력 UI

UI Slider 이미지 변경

- **Slider Component**는 **Slider**의 **background**와 **Fill** 이미지를 직접 그리지 않는다**.** 대신 **Slider Component**는 **Value** 값에 따라 **Fill Rect** 필드에 할당된 **GameObject**의 크기를 조정한다**.** 이때**, Fill Rect** 필드에 할당된 **GameObject**의 크기는 해당 **GameObject**의 부모 **GameObject**에 상대적으로 결정된다**.**

ex)Slider의 조건이**..**

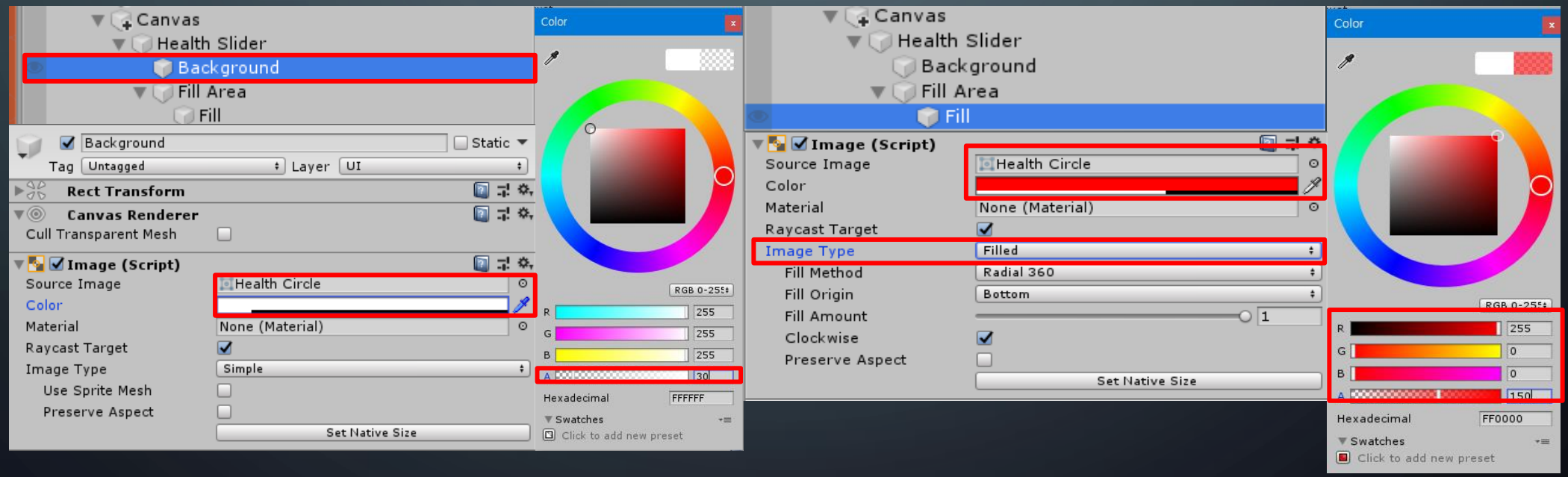
1. **Slider Component**의 **Min Value**가 **0, Max Value**가 **100, Value**가 **50**
2. **Slider Component**의 **Fill Rect**에 **A**라는 **UI GameObject**가 할당 되었다**.**
3. **GameObject A**는 **GameObjectB**의 자식이다**.**

- 이 상태에서 **Slider Component**는 **GameObject A**의 크기를 **B** 크기의 **50%**로 줄입니다**.** 만약 현재 값이 **100**이라면 **Slider Component**는 **GameObject A**의 크기가 **GameObject B**와 **100%** 일치하도록 늘린다**.**
- 현재 **Health Slider GameObject**의 **Slider Component**의 **Fill Rect** 필드에는 **Fill GameObject** 가 할당되어 있다**,** 그리고 **Fill GameObject**의 부모는 **Fill Area GameObject**이다**.** 즉**, Slider Comp onent**는 **Fill GameObject**의 크기를 **Fill Area GameObject**에 상대적으로 잡아 늘리거나 줄임으로 써 **Slider**가 줄어들거나 채워지는 것을 구현한다**.**
- 마찬가지로 **Slider**의 배경은 **Background GameObject**의 **Image Component**가 그린다**.**

1. 플레이어 체력 UI

Slider Bacground Image 변경, **Fill Image** 변경

1. **Background GameObject**를 선택해서 **Image Component**의 **Source Image**에 **Health Circle Sprite**를 할당한다.
2. **Image Component**의 **Color Field** 클릭 **>** 알파 값을 **30**으로 변경
3. **Fill GameObject**를 선택해서 **Image Component**를 **Source Image**에 **Health Circle Sprite**를 할당한다.
4. **Image Component**의 **Color Field** 클릭 **> Color(255, 0, 0, 150)**으로 변경, **ImageType**을 **Filled** 로 변경



2. PLAYERHEALTH SCRIPT

- **LivingEntity Class**를 확장하여 **Player**의 체력을 구현하는 **PlayerHealth Script**를 구현한다. **Play erHealth Script**는 다음 기능을 가져야 한다.
  →**LivingEntity**의 생명체 기본 기능, 체력이 변경되면 체력 **Slider**에 반영, 공격받으면 피격 효과음 재생, 사망 시 플레이어의 다른 **Component**를 비활성화, 사망 시 사망 효과음과 사망 애니메이션 재생, 아이템을 감지하고 사용

PlayerHealth의 **Field**

- **UI HealthSlider**를 할당할 **healthSlider**가 선언되어 있다. **healthSlider**에는 제작한 **HealthSlider GameObject**의 **Slider Component**가 할당.
- 상황에 따라 재생할 오디오 클립 변수가 선언.

```
public Slider healthSlider; // 체력을 표시할 UI 슬라이더

public AudioClip deathClip; // 사망 소리
public AudioClip hitClip; // 피격 소리
public AudioClip itemPickupClip; // 아이템 습득 소리
```

- 사용할 **Component**에 대한 변수 선언

```
private AudioSource playerAudioPlayer; // 플레이어 소리 재생기
private Animator playerAnimator; // 플레이어의 애니메이터

private PlayerMovement playerMovement; // 플레이어 움직임 컴포넌트
private PlayerShooter playerShooter; // 플레이어 슈터 컴포넌트
```

2. PLAYERHEALTH SCRIPT

Awake() Method

Player GameObject에서 필요한 **Component**를 찾아 변수에 할당하고 수치를 초기화 한다.

```
private void Awake()
{
    // 사용할 컴포넌트를 가져오기
    playerAnimator = GetComponent<Animator>();
    playerAudioPlayer = GetComponent<AudioSource>();

    playerMovement = GetComponent<PlayerMovement>();
    playerShooter = GetComponent<PlayerShooter>();
}
```

OnEnable() method

```
protected override void OnEnable()
{
    // LivingEntity의 OnEnable() 실행 (상태 초기화)
    base.OnEnable();

    // 체력 슬라이더 활성화
    healthSlider.gameObject.SetActive(true);
    // 체력 슬라이더의 최댓값을 기본 체력값으로 변경
    healthSlider.maxValue = startingHealth;
    // 체력 슬라이더의 값을 현재 체력값으로 변경
    healthSlider.value = health;

    // 플레이어 조작을 받는 컴포넌트 활성화
    playerMovement.enabled = true;
    playerShooter.enabled = true;
}
```

LivingEntity Class의 **OnEnable() method**를 오버라이드하여 구현한다.

Base.OnEnable()에 의해 **LivingEntity**의 **OnEnable Method**가 실행되고, **LinvingEntity**의 **OnEnable Method**는 **healt**의 값을 **startingHealth**의 값으로 초기화.

Slider의 **Max = startingHealth**
Slider의 **Value = health**

2. PLAYERHEALTH SCRIPT

OnEnable() Method를 이용해 확장성을 고려

PlayerHealth의 **OnEnable() Method**는 '부활 기능'을 염두에 둔 구현이라는 점에 주목하자.
Player가 사망하면 그대로 게임오버가 되는 구조이다, 현재는 **OnEnable()**의 처리를 **Awake() Method**로 옮기거나, **OnEnable()**에서 **Health Slider GameObject**의 **PlayerShooter, PlayerMovement Component**를 활성화하는 처리를 삭제해도 된다.
굳이 **OnEnable() Method**에서 활성화하지 않아도 **Player**의 **Health Slider GameObject**와 **PlayerShooter, PlayerMovement Component**는 미리 활성화된 상태이기 때문이다.
우리는 **PlayerHealth**의 **Die() Method**에서 **Player**가 사망할 경우 체력 슬라이더와 **PlayerShooter, PlayerMovement Component**를 모두 비활성화하게 된다.
만약, 프로토타입 완성 후 사망한 **Player**가 부활하는 기능을 확장 구현한다면 **Component**가 다시 활성화될 때마다 매번 실행되는 **OnEnable() Method**는 **Player** 캐릭터가 부활하면서 비활성화된 **GameObject**와 **Component**를 다시 활성화 하는 '자동 리셋' 기능을 담당한다.

RestoreHealth() Method

LivingEntity Class의 **RestoreHealth() Method**를 오버라이드 한다.

```
public override void RestoreHealth(float newHealth)
{
    // LivingEntity의 RestoreHealth() 실행 (체력 증가)
    base.RestoreHealth(newHealth);
    // 갱신된 체력으로 체력 슬라이더 갱신
    healthSlider.value = health;
}
```

2. PLAYERHEALTH SCRIPT

OnDamage() Method

LivingEntity Class의 **OnDamage() Method**를 **Override**한다.

```
public override void OnDamage(float damage, Vector3 hitPoint, Vector3 hitDirection)
{
    if(!dead)
    {
        // 사망하지 않은 경우에만 효과음 재생
        playerAudioPlayer.PlayOneShot(hitClip);
    }

    // LivingEntity의 OnDamage() 실행(데미지 적용)
    base.OnDamage(damage, hitPoint, hitDirection);

    // 갱신된 체력을 체력 슬라이더에 반영
    healthSlider.value = health;
}
```

- 사망상태가 아니라면 피격 사운드를 출력한다.
- 그리고 **LivingEntity**의 **OnDamage**를 실행해서 데미지를 적용받고
- **Slider**에 체력값을 갱신한다.

2. PLAYERHEALTH SCRIPT

Die() Method

LivingEntity Class의 **Die() Method**를 **Override**한다.
→ **Livingentity**의 **Die()**의 원래 동작을 처리한다.
→ 이곳에서는 체력 슬라이더와 다른 **Component**들을 비활성화 처리한다.

```
public override void Die()
{
    // LivingEntity의 Die() 실행(사망 적용)
    base.Die();

    // 사망음 재생
    healthSlider.gameObject.SetActive(false);
    // 애니메이터의 Die 트리거를 발동시켜 사망 애니메이션 재생
    playerAnimator.SetTrigger("Die");

    // 플레이어 조작을 받는 컴포넌트 비활성화
    playerMovement.enabled = false;
    playerShooter.enabled = false;
}
```

OnTriggerEnter()

- **Trigger** 충돌한 상대방 **GameObject**가 **Item**인지 판단하고 아이템을 사용하는 처리를 한다.

```
private void OnTriggerEnter(Collider other)
{
    // 아이템과 충돌한 경우 해당 아이템을 사용하는 처리
    // 사망하지 않은 경우에만 아이템 사용 가능
    if(!dead)
    {
        // 충돌한 상대방으로부터 IItem Component 가져오기 시도
        IItem item = other.GetComponent<IItem>();

        // 충돌한 상대방으로부터 IITem Component 가져오는 데 성공했다면
        if(item != null)
        {
            // Use Method를 실행하여 아이템 사용
            item.Use(gameObject);
            // 아이템 습득 소리 재생
            playerAudioPlayer.PlayOneShot(itemPickupClip);
        }
    }
}
```

2. PLAYERHEALTH SCRIPT

PlayerHealth Component 설정