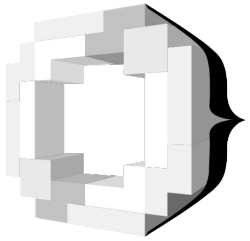
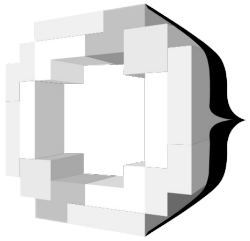
2023 유니티 중급반 3주차

오파츠 12기 여정인, 유태환



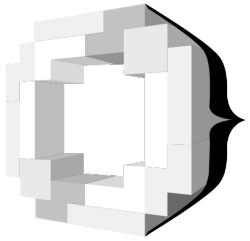
목차

- OOP (객체 지향 프로그래밍) 훑어보기
- 싱글톤 패턴과 게임매니저
- 클래스의 상속으로 UI 스크립팅하기



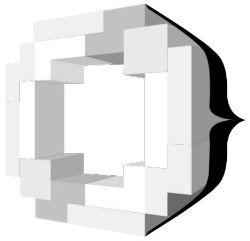
시작하기 전에

- 이미 아래 설명한 대부분은 구현되어 있어요.
- 다만 PPT 내용을 따라가면서 각 부분이 어떻게 설정되어 있는지 살펴봐요



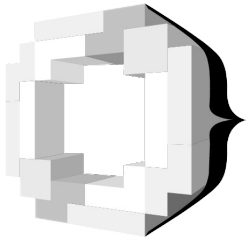
객체 지향 프로그래밍

- 객체 지향 프로그래밍 (Object Oriented Programming)
  - 데이터와 동작을 캡슐화하는 객체를 만들자!



객체 지향 프로그래밍

- 동물농장을 프로그램으로 만든다고 생각해봅시다



객체 지향 프로그래밍

- 동물농장을 프로그램으로 만든다고 생각해봅시다

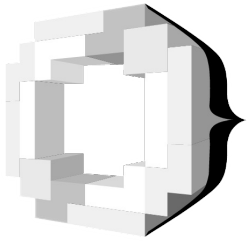
```
string name = 'none'; int id = 0
```

```
string name = 'body'; int id = 1
```

```
string name = 'human'; int id = 2
```

```
string name = 'dog'; int id = 3
```

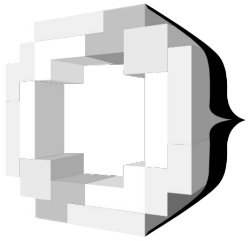
```
string name = 'cat'; int id = 4
```



객체 지향 프로그래밍

- 사람은 두발로 걷고 개는 네발로 걷는 코드를 짜주세요!

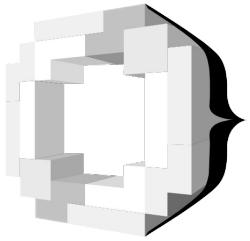
?



객체 지향 프로그래밍

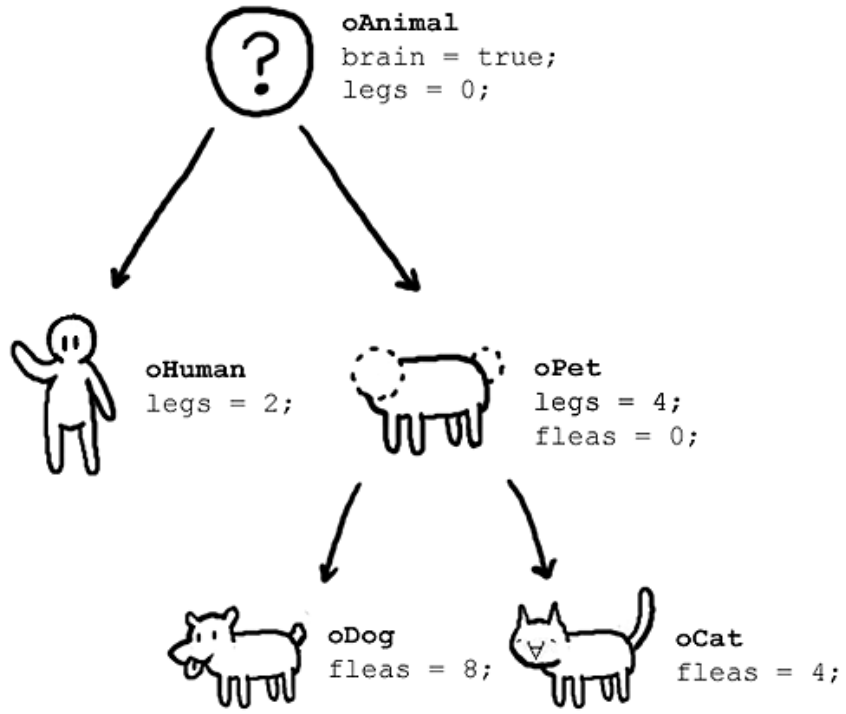
- 고양이 200마리 만들어주세요!

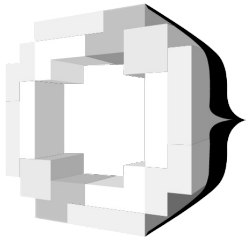
???



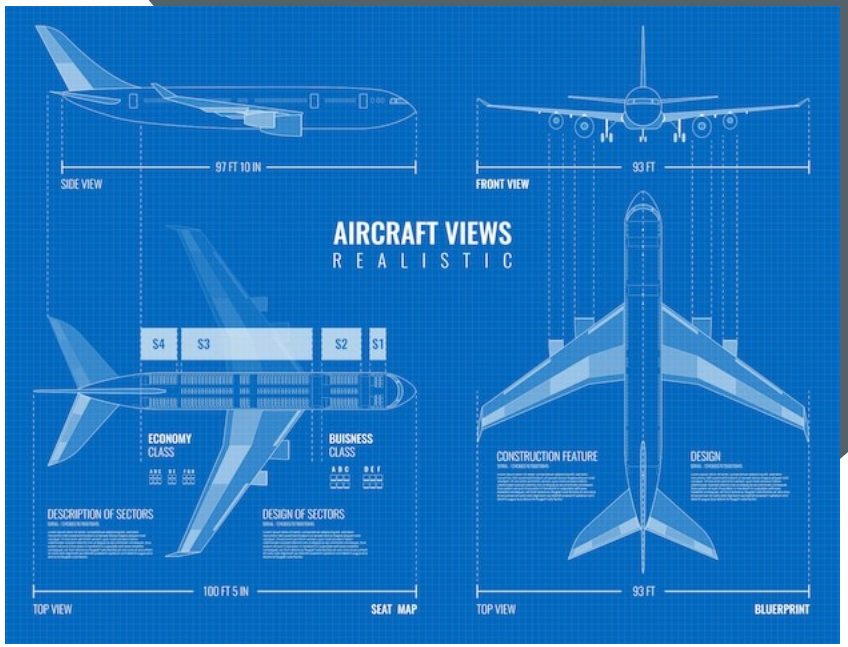
객체 지향 프로그래밍

- 객체 지향 프로그래밍 (Object Oriented Programming)





객체 지향 프로그래밍

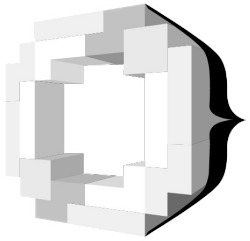


- 클래스
  - 바로 그 객체를 어떻게 만들지 그 방법을 제공하는 일종의 설계도!

    클래스명 변수명 = new 클래스명();

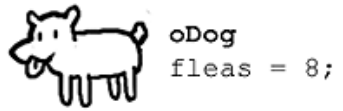
    과 같이 객체(인스턴스)를 선언하여 동일한 동작을 하는 여러 객체를 만들 수 있다!

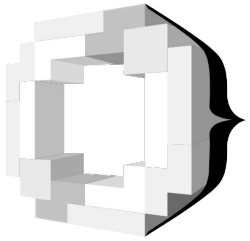
    (진부하지만 붕어빵틀과 붕어빵으로도 많이 설명합니다)



객체 지향 프로그래밍

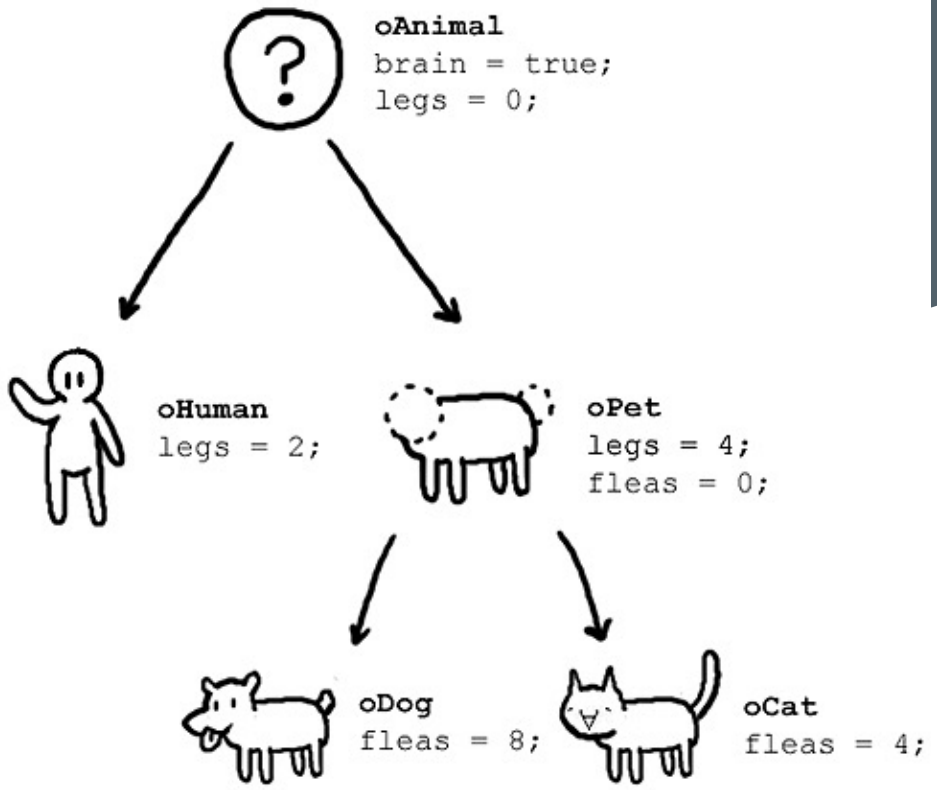
- 객체(오브젝트)? 인스턴스?
  - 객체는 사람, 개, 고양이처럼 데이터와 동작을 가지고 있는 어떠한 실체
  - 인스턴스는 클래스(설계도)에 따라서 객체를 실체화한 것!
  - '클래스를 인스턴스화하다'
  - 사실 많이 혼용되는 개념입니다.
    객체는 개념적인 거고 인스턴스는 실제 메모리에 올라간 데이터라고 생각합시다.

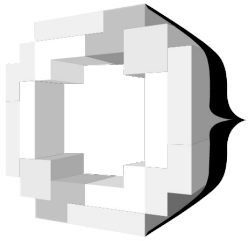




객체 지향 프로그래밍

- 상속
  - 부모의 속성과 동작을 상속하는
    자식 클래스
  - 다른 클래스의 특성을 유지하면서도
    새 속성이나 동작을 추가하고 싶을 때
  - 코드 재사용 + 계층 구조



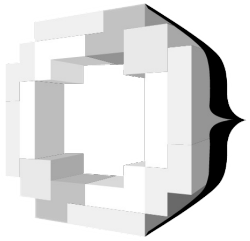


C#의 OOP

- Class

```
access-modifier class ClassName
{
    // 각종 변수와 함수들
}
```

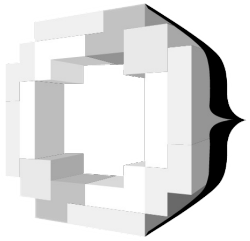
```
access-modifier?

- public, protected, default, private
```



C#의 OOP

- Abstract class
  - 추상 클래스는 다른 클래스의 기본클래스(부모)로만 사용이 가능합니다
  - 따라서 인스턴스화가 불가능합니다
  - abstract가 붙은 메소드는 반드시 자식클래스에 재정의 해야합니다.

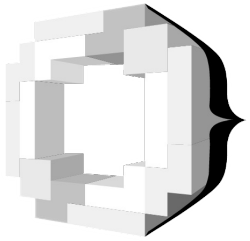
```
access-modifier abstract class ClassName
{
    // 각종 변수와 함수들
    public abstract void AbstractMethod();
}
```



C#의 OOP

- Interface
  - 인터페이스는 이를 상속하는 클래스가 반드시 구현해야 할 메소드와 프로퍼티 등등을 정의한 일종의 계약입니다.
  - 모든 필드가 abstract 입니다

```
access-modifier interface InterfaceName
{
    // 각종 변수와 함수선언 (구현 없음)
}
```

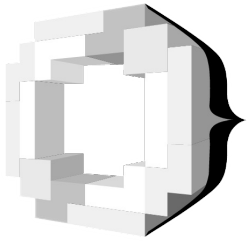


C#의 OOP

- 상속할 땐

```
access-modifier class ClassName : ParentClass,
Interface
{
    // 각종 변수와 함수선언 (구현 없음)
}
```

  - 클래스는 단 하나의 부모 클래스만을 가집니다.
  - 인터페이스는 여러 개 상속이 가능합니다.

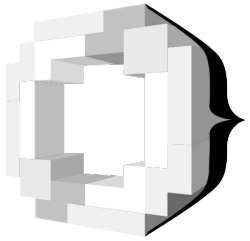


C#의 OOP

- 다형성 (런타임 다형성, 서브타입 다형성)

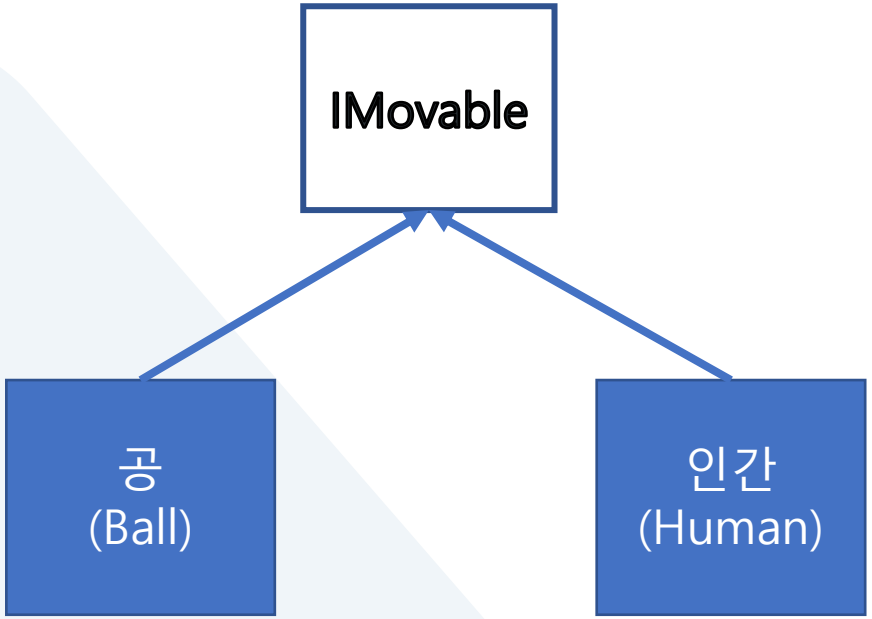
```
class Parent { void foo() { }; }
class Child1 : Parent { void foo() { print(“A”) } }
class Child2 : Parent { void foo() { print(“B”) } }

Parent pa = new Child1();
Parent pb = new Child2();
pa.foo() // A
pb.foo() // B
```

  - 같은 타입이라도 서로 다른 자식 객체가 들어있다면 다른 동작을 한다.



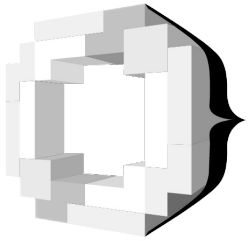
C#의 OOP

- 왜 인터페이스?
  - 관계없어보이는 클래스들에서도 동일한 동작이 있다면?




```
interface IMovable
{
    void move();
}

class Ball : IMovable
{
    void move() { … }
}

class Human : IMovable
{
    void move() { … }
}
```

```
IMovable movableObject1 = new Ball();
movableObject = Ball.move();

IMovable movableObject2 = new Human();
movableObject = Human.move();
```



다형성의 장점

c언어에선…

int to string
- itoa()

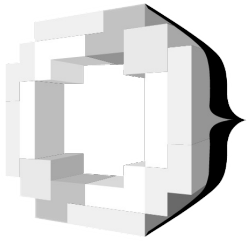
float to string
- sprintf()

bool to string
- booltostring() ???

c#에선…

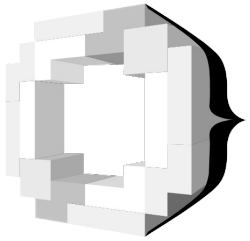
모든 클래스가 기본적으로 상속하는 Object에 ToString()이 정의되어있음.
따라서 대부분 클래스에 ToString()이 구현됨

Int32.ToString()
Float.ToString()
Boolean.ToString()
…

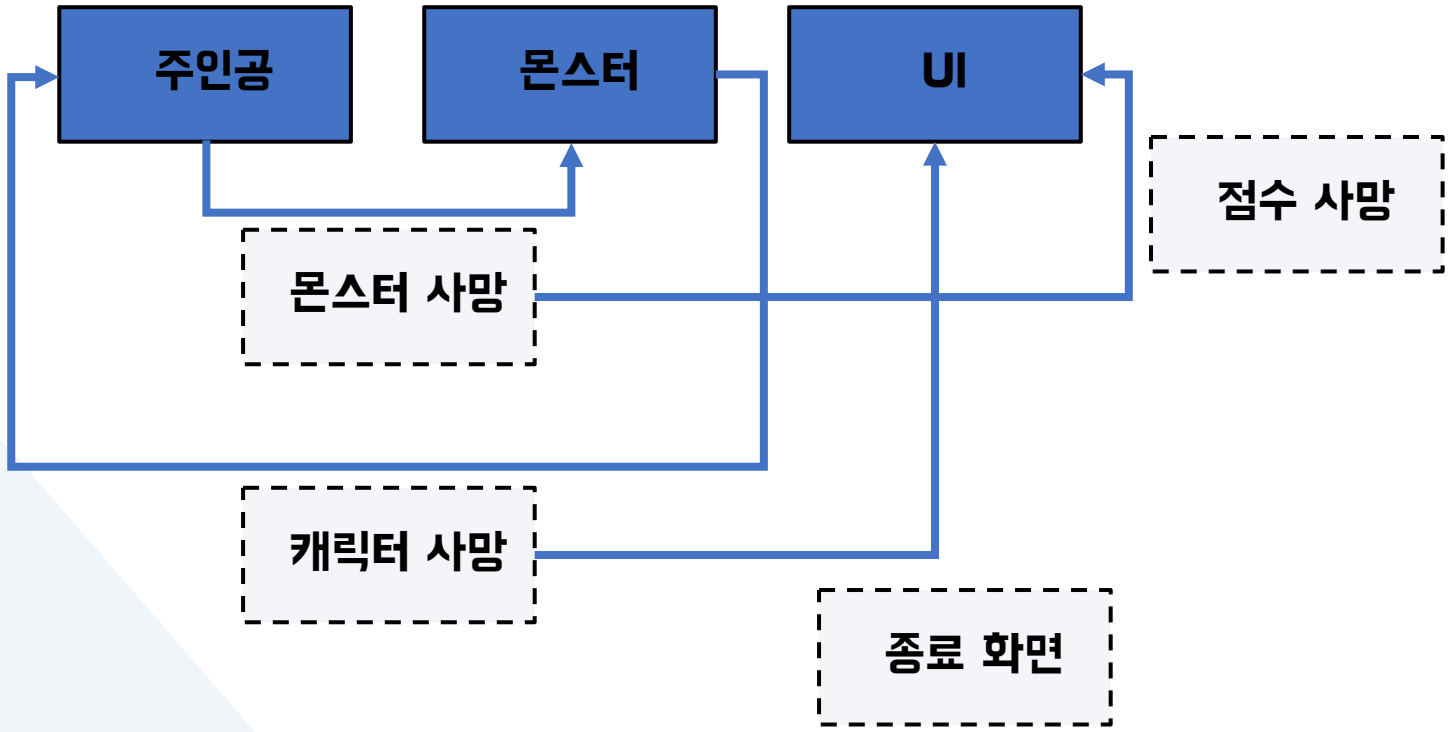


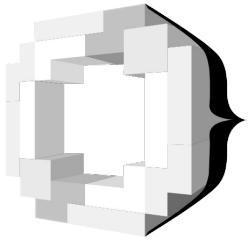
싱글톤 패턴

- 싱글톤 패턴은 인스턴스를 단 하나만 허용하는 패턴입니다.
- 특징(조건)
  - 싱글톤으로 선언된 클래스는 인스턴스를 단 하나만 가지고 있음을 보장해야합니다.
  - 싱글톤 인스턴스에 쉽게 접근할 수 있도록 Global access를 제공해야합니다
    (주로 싱글톤 클래스 내에 public static으로 인스턴스를 담아서 접근하도록 함)

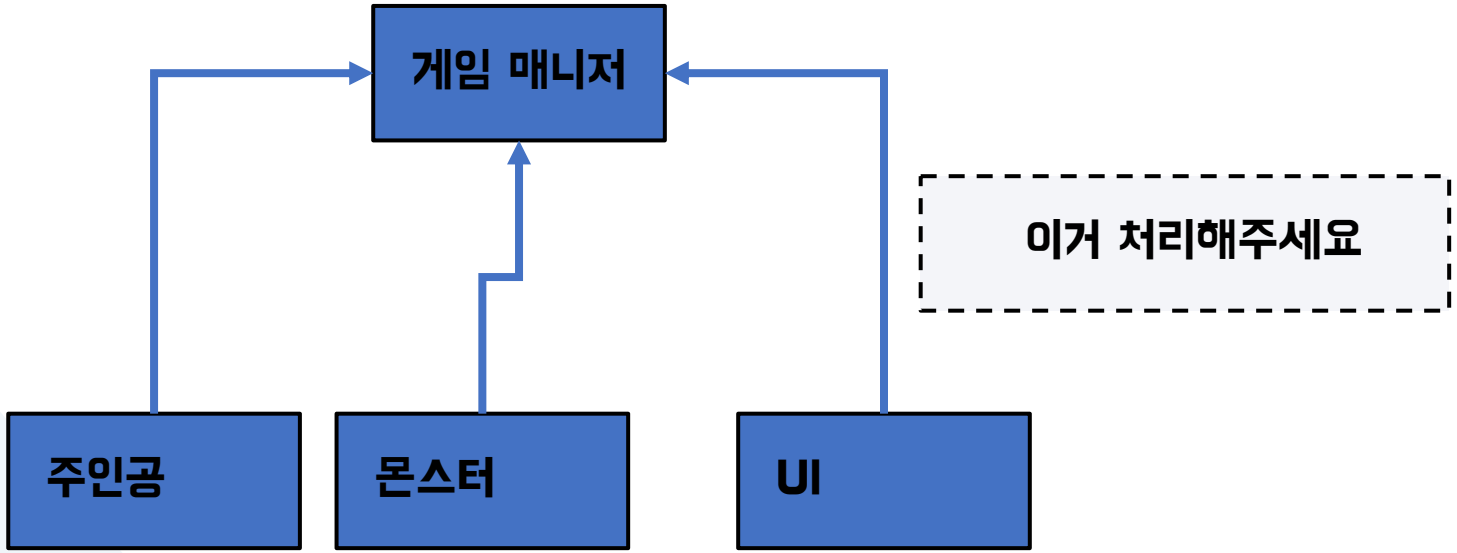


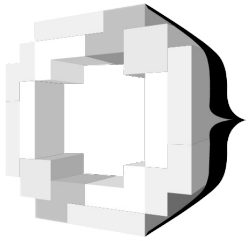
싱글톤 패턴(게임매니저)





싱글톤 패턴(게임매니저)





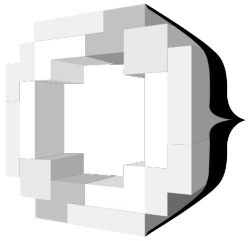
싱글톤 패턴(오디오매니저)

오디오 플레이어

오디오 플레이어

오디오 플레이어

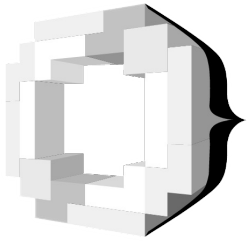
?



싱글톤 패턴(오디오매니저)

오디오 매니저

배경 음악1
배경 음악2
효과음
엔딩 음악
...

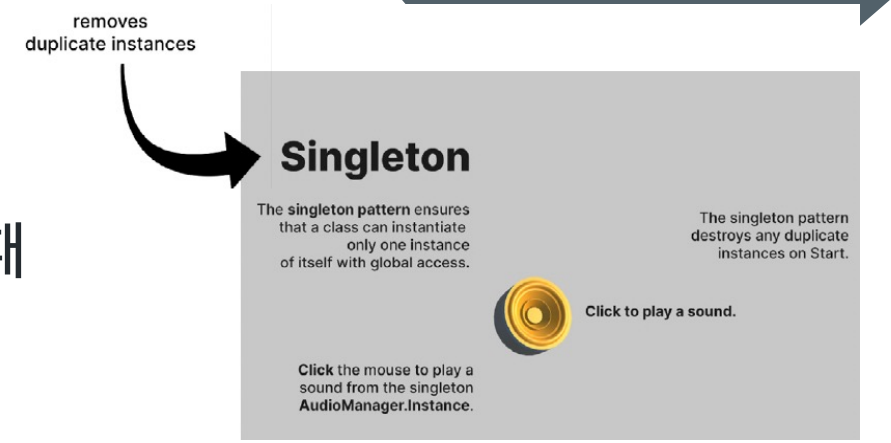


싱글톤 패턴

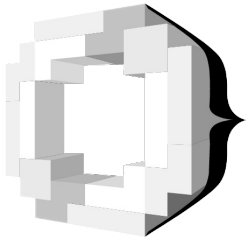
- 언제 사용할까?
  - 전역에 인스턴스가 딱 하나만 있도록 하고 싶을 때

    ex) 사용자 입력 관리, 이벤트 큐!
  - 하나의 객체가 시스템 전체에서 작업을 조정해야 할 때

    -> 유니티에서 자주 쓰이는 Manager!

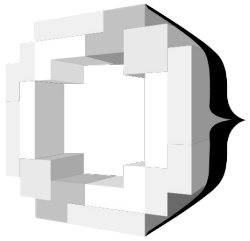


심지어 GameManager.cs 파일을 생성하면 유니티에서 따로 아이콘까지 바꿔줌



싱글톤 패턴

- 단점
  - 커플링이 심하다 (특히 매니저로 활용할 때)
    시스템의 여러 부분을 모두 제어하기 때문에 어느 매니저의 코드 하나 변경했다고 나머지 모든 부분의 코드를 변경해야 하는 대참사가 날 수도… (유지보수 어려움)
  - 멀티쓰레드 환경에서 병목이 있다.
    여러 쓰레드가 단 하나의 인스턴스를 접근하려고 하면서 병목 현상이 생긴다.
  - 싱글톤이 많아지면 메모리에 항상 올라가있는 오브젝트가 많아진다.
  - 항상 public으로 선언되어야 하기 때문에 실수하게 될 가능성이 높다
  - 그래서 유지보수의 중요성이 큰 실무 게임 프로젝트에선 지양하는 패턴



싱글톤 패턴

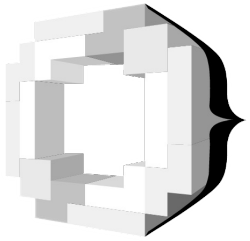
- 장점
  - **사용이 쉽고 구현이 편하다.**

    전역변수처럼 사용할 수 있어서 편리하다.
  - 메모리/성능 측면에서 이점

    인스턴스 하나만 고정된 메모리 공간을 차지하기 때문에 메모리 낭비가 없다.

    바로 접근하여 사용할 수 있기때문에 접근 속도가 빠르다.

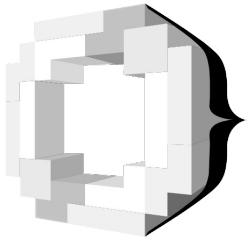
    (GetComponent 등등 불필요)



싱글톤 패턴

```
using UnityEngine;

public class SimpleSingleton : MonoBehaviour
{
    public static SimpleSingleton Instance;

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

가장 간단한 형태의 싱글톤입니다.



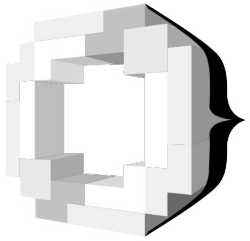
싱글톤 패턴

실제로 가장 많이 쓰는 형태입니다.

이미 싱글톤 인스턴스가 있다면 생성된 인스턴스를 파괴합니다.

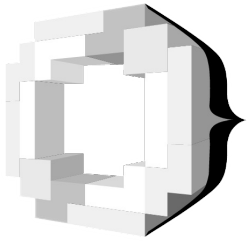
또한 씬이 변경되어도 인스턴스가 유지되도록 합니다.

```
using UnityEngine;

public class Singleton : MonoBehaviour
{
    private static Singleton instance;

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                SetupInstance();
            }
            return instance;
        }
    }

    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(this.gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }

    private static void SetupInstance()
    {
        instance = FindObjectOfType<Singleton>();
        if (instance == null)
        {
            GameObject gameObj = new GameObject();
            gameObj.name = "Singleton";
            instance = gameObj.AddComponent<Singleton>();
            DontDestroyOnLoad(gameObj);
        }
    }
}
```



싱글톤 패턴

- 싱글톤의 또다른 단점
  - 매니저를 새로 정의할 때마다 이 긴 코드를 또 작성해야된다고?
  - 그냥 싱글톤 추상 클래스를 만들자
  - 이후에는 이를 상속만 해도 싱글톤이 되도록 구현하자



제네릭

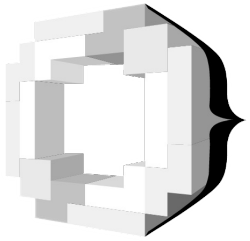
- C#에서 제네릭은 다양한 타입에 대해 재사용 가능한 코드를 만들 수 있도록 합니다.
- 컴파일 타임에 변환이 이루어집니다

제네릭 함수

```
int Foo(int param);
T Bar<T>(T param);
```

제네릭 클래스

```
public class GenericClass<T>
{
  public T Property { get; set; }
}

GenericClass<int> intObject = new GenericClass<int>();
```



제네릭

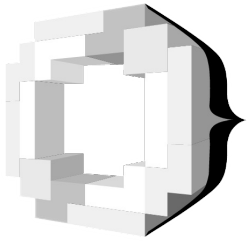
- 사실 여러분은 유니티에서 제네릭을 매우 자주 쓰고 있습니다.

GetComponent<컴포넌트 타입>();

```
animator = GetComponent<Animator>();
rgbody = GetComponent<Rigidbody2D>();
spriteRenderer = GetComponent<SpriteRenderer>();
```

GetComponent 함수

```
public unsafe T GetComponent<T> ()
{
    CastHelper<T> castHelper = default(CastHelper<T>);
    GetComponentFastPath (typeof(T), new IntPtr (&castHelper.onePointerFurtherThanT));
    return castHelper.t;
}
```



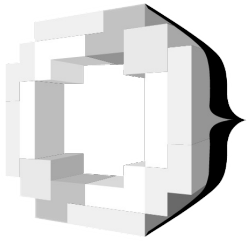
싱글톤 패턴

- 싱글톤을 제네릭으로 구현해봅시다.

- 사용할 땐 그냥 상속하면 됩니다

```
public class SomeManager : Singleton<SomeManager>
{

}
```

```
public class Singleton<T> : MonoBehaviour where T : MonoBehaviour
{
    private static T instance;
    public static T Instance
    {
        get
        {
            if (instance == null)
            {
                SetupInstance();
            }
            return instance;
        }
    }
    private void Awake()
    {
        if (instance == null)
        {
            instance = this as T;
            DontDestroyOnLoad(this.gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }
    private static void SetupInstance()
    {
        instance = FindObjectOfType<T>();
        if (instance == null)
        {
            GameObject gameObj = new GameObject();
            gameObj.name = typeof(T).Name;
            instance = gameObj.AddComponent<T>();
            DontDestroyOnLoad(gameObj);
        }
    }
}
```



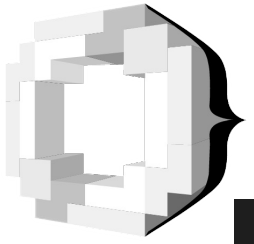
싱글톤 패턴

- 이 프로젝트에서는…

```
// UI 매니저
private static UIManager _uiManager = new UIManager();
public static UIManager UI
{
    get
    {
        return _uiManager;
    }
}
```

게임 매니저 하나만 싱글톤으로 구현하고 게임매니저 내에 다른 매니저들을 static 멤버로 들고 있겠습니다.
전역변수가 중구난방으로 있으면 다른 코드에서 사용할 때 관리가 힘들어집니다.

위와 같이 구현하면 GameManager.Instance.UI 와 같은 식으로 비교적 깔끔하게 접근이 가능합니다.



코드 살펴보기

```
private static UIManager _uiManager = new UIManager();
public static UIManager UI
{
    get
    {
        return _uiManager;
    }
}

private static GamePlayManager _gamePlayManager = new GamePlayManager();
public static GamePlayManager Play
{
    get
    {
        return _gamePlayManager;
    }
}

private void InitOtherManagers()
{
    // 다른 매니저 추가됐을 때 여기에 초기화 코드 작성
    _uiManager.Init();
}

// 게임 종료
public void ExitGame()
{
    Application.Quit();
}

// 씬 관리
public enum SceneType
{
    MainMenu,
    Option,
    Game
}

private SceneType currentScene;

public void ChangeScene(SceneType sceneType)
{
    currentScene = sceneType;
    SceneManager.LoadScene(sceneType.ToString());
    OnSceneChanged?.Invoke(sceneType);
}

public event Action<SceneType> OnSceneChanged;
```

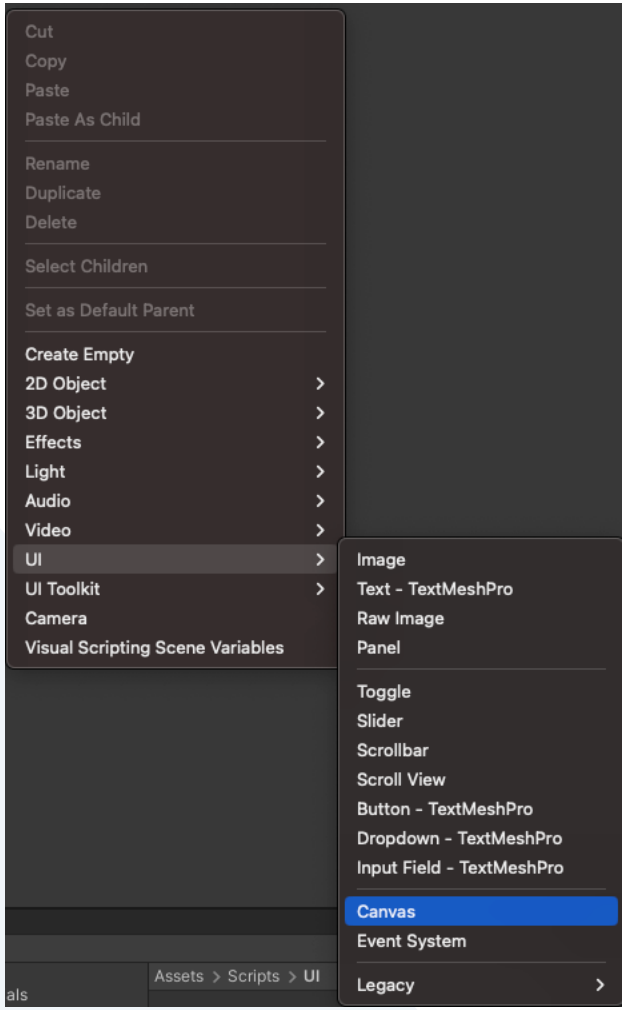
Scripts/Manager/GameManager.cs

나머지는 싱글톤 기본 구현과 같습니다.

다른 매니저들을(일단은 UI, GamePlay만)
멤버로 선언해주고,

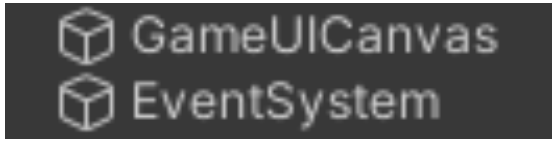
게임매니저에서 씬도 관리해주겠습니다.

게임 내 UI

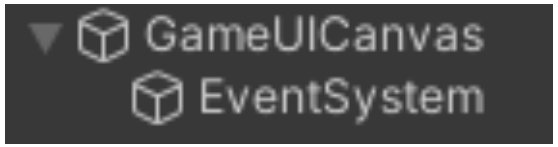


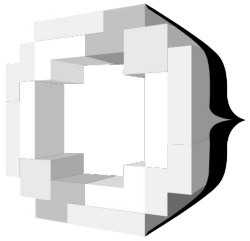
하이어라키에서 우클릭하여 Canvas를 추가해줍니다.

그럼 아래와 같이 캔버스와 이벤트시스템이 추가됩니다.



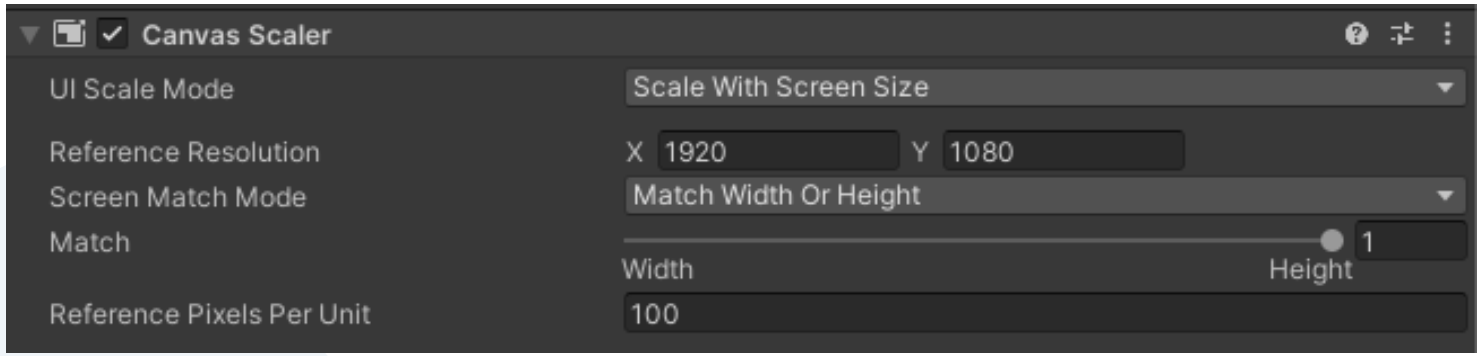
이벤트 시스템을 캔버스의 자식으로 넣어주겠습니다.



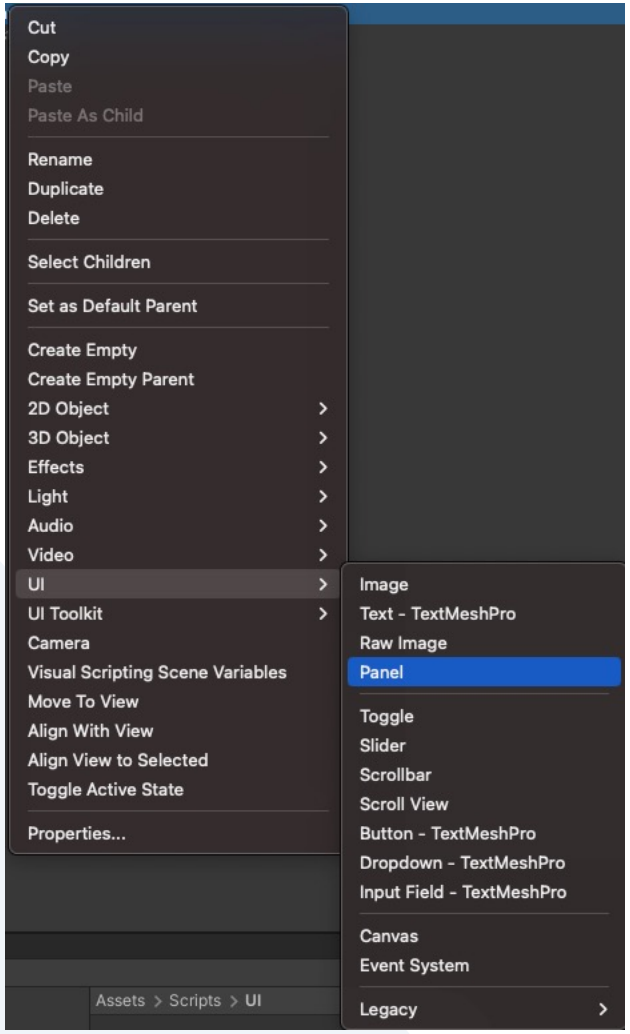


게임 내 UI

멀티 해상도에 대응하기 위해 Canvas 내에
Canvas Scaler에서 UI Scale Mode를 Scale With Screen Size로 설정해주었습니다.



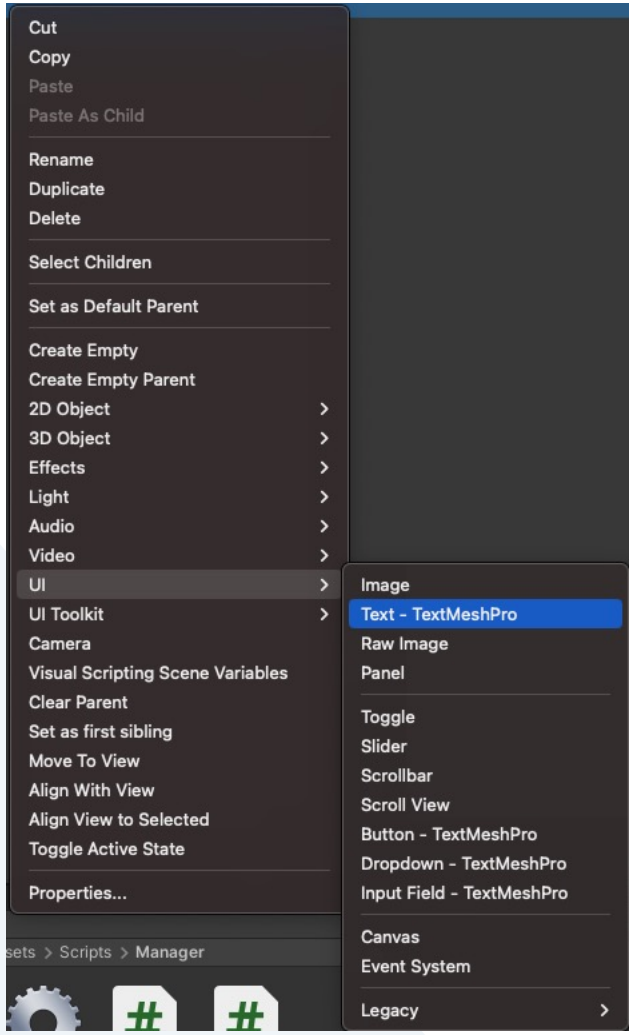
게임 내 UI



캔버스 내에 Panel을 생성해줍니다.

레이아웃을 구성하는 방법은 여러가지가 있지만 저는 Panel로 구성하는 것을 선호합니다.

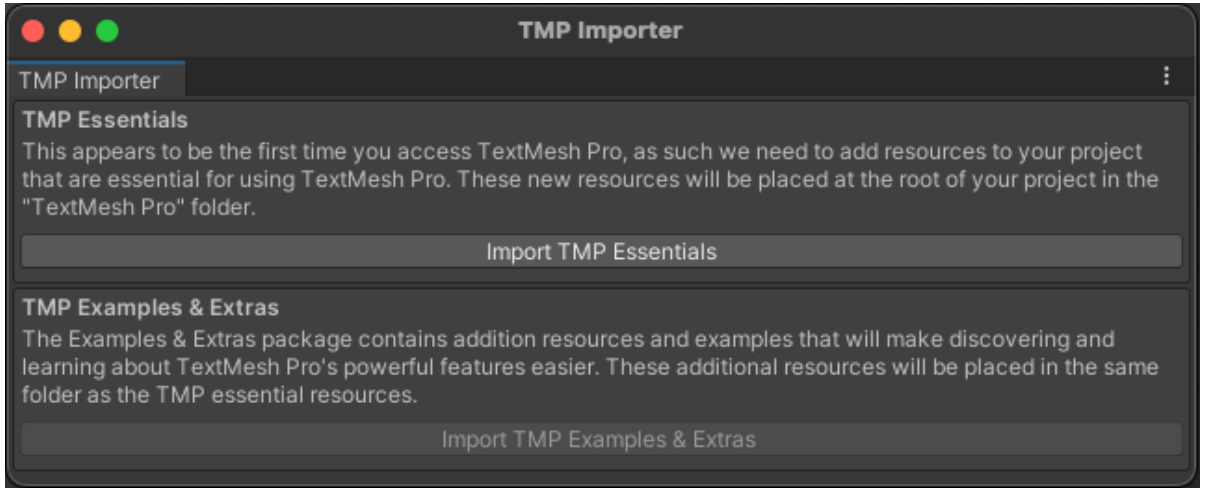
게임 내 UI

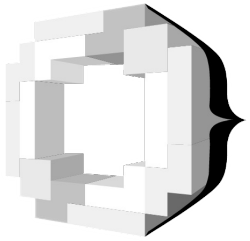


패널 내에 Text - TextMeshPro를 넣어주겠습니다.

다양한 폰트를 지원하고 싶다면, 특히 한글을 지원하고 싶다면 TextMeshPro는 필수 에셋입니다.

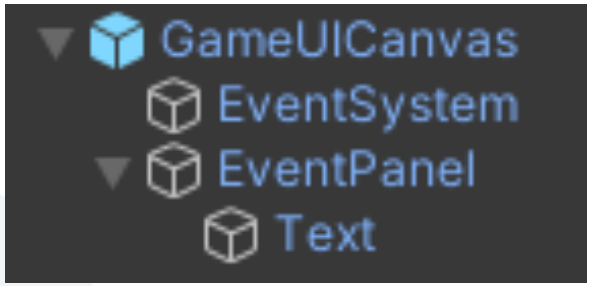
처음 생성한다면 아래와 같이 Importer가 뜨는데,
Import TMP Essentials를 통해 불러오도록 하겠습니다.



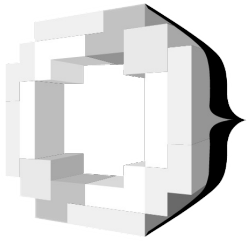


게임 내 UI

이 캔버스는 동적으로 로드할 계획입니다. 이를 위해서 Resources/Prefabs/UI 폴더에 캔버스 오브젝트를 드래그 앤 드랍을 해주겠습니다.



이렇게 나타나는 것이 프리팹입니다.
미리 생성해둔 게임 오브젝트라고 생각하시면 됩니다.



코드 살펴보기

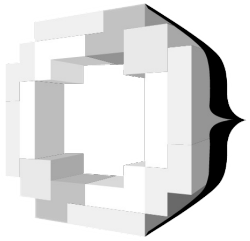
```
public class UIManager
{

}
```

Scripts/Manager/UIManager.cs

UIManager 클래스를 작성해보겠습니다.

이 클래스는 MonoBehaviour를 상속하지 않습니다.

오브젝트에 붙을 컴포넌트가 아니라
GameManager에서 멤버로 관리하기 때문이에요

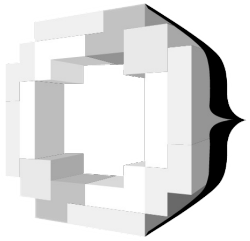


코드 살펴보기

```
// 캔버스 정보
public readonly CanvasPrefab[] CANVAS_INFO_LIST =
{
    new CanvasPrefab() {type=CanvasType.GAME_PLAY, canvasName="GameUICanvas"}
};

// 캔버스와 씬 바인딩 정보
private readonly Dictionary<GameManager.SceneType, CanvasType> SCENE_CANVAS_INFO
    = new Dictionary<GameManager.SceneType, CanvasType>()
{
    {GameManager.SceneType.Game, CanvasType.GAME_PLAY}
};
```

```
public enum CanvasType
{
    NONE,
    MAIN_MENU,
    OPTION,
    GAME_PLAY
}

[System.Serializable]
public struct CanvasPrefab
{
    public CanvasType type;
    public string canvasName;
}
```

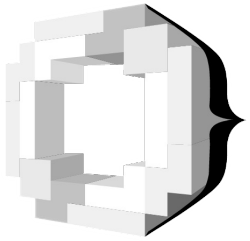
읽기 전용으로 캔버스 프리팹 정보를 담은 배열과
어떤 씬에 어떤 캔버스를 띄울지 그 정보를 담은 딕셔너리 하나를 선언해주겠습니다.



코드 살펴보기

```
// 생성된 캔버스 오브젝트 담을 딕셔너리
private Dictionary<CanvasType, GameObject> _canvasDict = new Dictionary<CanvasType, GameObject>();
```

동적으로 로드한 캔버스 오브젝트를 담을 딕셔너리도 선언해주었습니다.



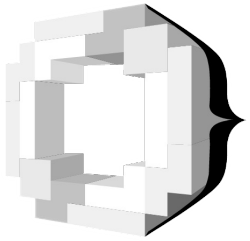
코드 살펴보기

```
// 매니저 초기화
public void Init()
{
    foreach (CanvasPrefab c in CANVAS_INFO_LIST)
    {
        GameObject canvasObject = Object.Instantiate(GetPrefabResource(c.canvasName), null);
        Object.DontDestroyOnLoad(canvasObject);
        AddCanvas(c.type, canvasObject);
    }

    GameManager.Instance.OnSceneChanged += (sceneType) => ShowCanvas(SCENE_CANVAS_INFO[sceneType]);
}
```

매니저를 초기화해주는 함수도 만들었습니다.

Object.Instantiate 함수는 유니티에서 제공하는 함수입니다.
파라미터로 프리팹을 전달하면 해당 프리팹으로 게임오브젝트를 생성하여 하이어라키에 올려줍니다.

GetPrefabResource와 ShowCanvas는 UIManager에 정의된 메소드입니다.



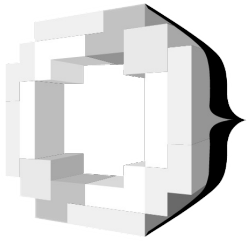
코드 살펴보기

```
// 캔버스 딕셔너리에 캔버스 오브젝트 추가
public void AddCanvas(CanvasType type, GameObject canvas)
{
    _canvasDict.Add(type, canvas);
}

// 화면에 캔버스 보여주기
public void ShowCanvas(CanvasType type)
{
    HideAllCanvas();
    _canvasDict[type].SetActive(true);
}

// 딕셔너리에 있는 모든 캔버스 숨기기
public void HideAllCanvas()
{
    foreach (var c in _canvasDict)
    {
        c.Value.SetActive(false);
    }
}
```

AddCanvas는 캔버스 타입과 캔버스 오브젝트를 받아 딕셔너리에 추가해줍니다.

ShowCanvas는 캔버스 타입을 받아 해당 캔버스만 화면에 보이도록 합니다. (SetActive; 활성화)

HideAllCanvas는 모든 캔버스를 화면에서 보이지 않도록 합니다. (SetActive; 비활성화)

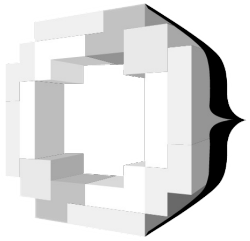


코드 살펴보기

```
// 리소스 폴더에서 프리팹 로드하기
public GameObject GetPrefabResource(string name, string path = null)
{
    string targetPath = "Prefabs/UI/" + (path == null ? name : path + name);
    GameObject go = Resources.Load<GameObject>(targetPath);

    if (go == null)
    {
        Debug.LogError($"Can't find prefab: {name}");
    }
    return go;
}
```

GetPrefabResource는 Resources/Prefabs/UI에 있는 프리팹 파일을 불러오는 함수입니다.

Resources.Load<프리팹 타입>(파일경로) 는 유니티에서 제공하는 함수로, Resources에 있는 프리팹들을 로드해주는 함수입니다. 프리팹은 반드시 Resources 폴더 내에 존재해야 합니다.



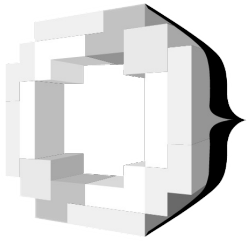
코드 살펴보기

```
public class GamePlayManager
{

}
```

Scripts/Manager/GamePlayManager.cs

GamePlay 클래스를 작성해보겠습니다.

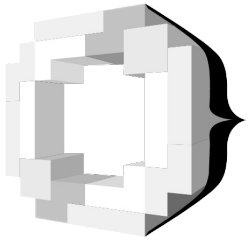
여기서는 캐릭터가 죽었는지, 살았는지 뭐 그런 것을 관리해보도록 하겠습니다.



코드 살펴보기

```
public bool isWin { get; private set; }

public bool isDead { get; private set; }
```

죽었는지, 또는 승리했는지를 저장할 수 있는 bool 타입 프로퍼티를 선언해주겠습니다.

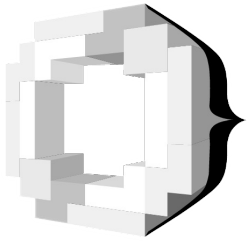


코드 살펴보기

```
// OnPlayerWin(float score)
public event Action<float> OnPlayerWin;
// OnPlayerDead()
public event Action OnPlayerDead;
// OnRestart()
public event Action OnRestart;
```

event와 Action이 등장했습니다.

Action은 델리게이트고
event는 델리게이트와 같이 사용할 수 있는 키워드입니다.



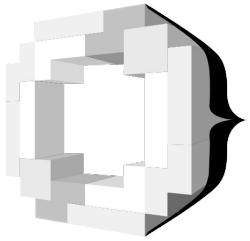
C# 델리게이트

```
delegate void TestDelegate(int a);

void foo(int a)
{

}

void bar()
{
    TestDelegate d = foo;
    foo(1);
}
```

델리게이트는 C/C++에서 함수 포인터와 비슷하게,
파라미터와 반환형식을 가지는 메서드를 가리킬 수 있는 타입입니다.

delegate <반환형식> <델리게이트 이름>(파라미터)
형식으로 델리게이트를 선언해주고

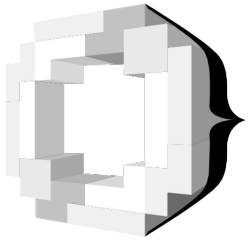
TestDelegate로 선언해준 델리게이트에 함수를 전달하면
델리게이트를 호출하여 해당 함수를 호출할 수 있습니다.



C# 델리게이트

```
delegate void TestDelegate(int a);

void A(int a)
{

}

void B(int a)
{

}

TestDelegate d;

void foo()
{
    d += A;
    d += B;
    d -= A;
    d(1);
}
```

왼쪽과 같이 +=, -= 연산자를 이용해서 하나의 델리게이트에 여러 함수들을 할당하거나 할당했던 함수를 제거할 수 있습니다.



C# 익명 메소드 / 람다 식

```
void foo()
{
    d = delegate (int a) { };
    d(1);
}
```

델리게이트에는 왼쪽과 같이 익명 메소드를 전달할 수도 있습니다.

delegate (파라미터) { 내용 }

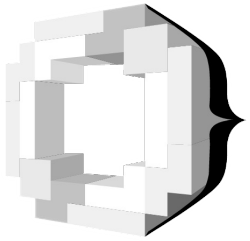
과 같은 식으로 사용합니다.

```
void foo()
{
    d = (int a) => { };
    d(1);
}
```

또는 람다식을 사용할 수도 있습니다.

(파라미터) => { 내용 };

과 같으며, 중괄호는 내부 expression이 하나일 경우 생략 가능합니다. 보통 델리게이트는 람다식으로 많이 사용합니다.



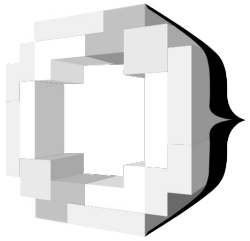
C# event

```
event TestDelegate d;

void foo()
{
    d = (int a) => { };
    d(1);
}
```

event는 델리게이트 선언 시에 event를 붙여 선언할 수 있습니다.

델리게이트와 겉보기에도 차이가 없어보이고 실제로도 사용법도 같은데, 왜 구분을 해뒀을까요?

event는 델리게이트와 달리 클래스 외부에서 호출이 불가능합니다.



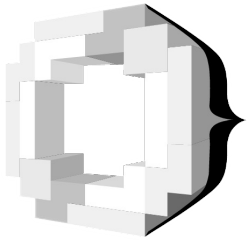
C# Action

```
// OnPlayerWin(float score)
public event Action<float> OnPlayerWin;
// OnPlayerDead()
public event Action OnPlayerDead;
// OnRestart()
public event Action OnRestart;
```

다시 돌아가볼까요?

Action은 C#에 미리 정의된 델리게이트 타입이에요. (매번 delgate void 어쩌고(); 하는건 힘들잖아요!)

Action<파라미터> 형식으로 사용할 수 있고, 리턴타입이 없는 함수만 담을 수 있어요

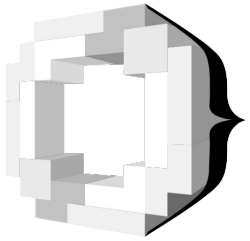


코드 살펴보기

```
public void Init()
{
    GameManager.Instance.OnSceneChanged +=
        ((sceneType) =>
        {
            if (sceneType == GameManager.SceneType.Game)
            {
                ResetState();
            }
        });

    ResetState();
}

public void ResetState()
{
    Debug.Log("Reset Play");
    isWin = false;
    isDead = false;
}

public void Replay()
{
    ResetState();
    OnRestart?.Invoke();
}

public void PlayerWin(float score)
{
    isWin = true;
    OnPlayerWin?.Invoke(score);
}

public void PlayerDead()
{
    isDead = true;
    OnPlayerDead?.Invoke();
}
```

GamePlayManager 나머지 코드입니다.

PlayerWin 함수에서 OnPlayerWin,
PlayerDead 함수에서 OnPlayerDead
를 호출하는 걸 볼 수 있어요

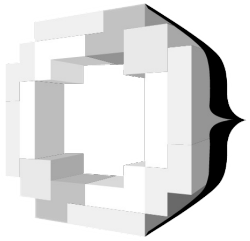


코드 살펴보기

```
public void ShowWin(float score)
{
    ShowCanvas(CanvasType.GAME_PLAY);
    _canvasDict[CanvasType.GAME_PLAY].GetComponentInChildren<TextMeshProUGUI>().text = "Win";
}

public void ShowDead()
{
    ShowCanvas(CanvasType.GAME_PLAY);
    _canvasDict[CanvasType.GAME_PLAY].GetComponentInChildren<TextMeshProUGUI>().text = "Dead";
}
```

UIManager에 ShowWin과 ShowDead 함수를 만들어주겠습니다.

아까 만들어줬던 패널을 띄우고, 텍스트를 넣어주는 함수에요.

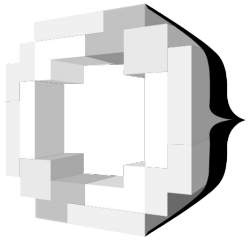


코드 살펴보기

```
// 매니저 초기화
public void Init()
{
    foreach (CanvasPrefab c in CANVAS_INFO_LIST)
    {
        GameObject canvasObject = Object.Instantiate(GetPrefabResource(c.canvasName), null);
        Object.DontDestroyOnLoad(canvasObject);
        AddCanvas(c.type, canvasObject);
    }

    GameManager.Instance.OnSceneChanged += (sceneType) => ShowCanvas(SCENE_CANVAS_INFO[sceneType]);

    GameManager.Play.OnPlayerWin += ShowWin;
    GameManager.Play.OnPlayerDead += ShowDead;
}
```

UIManager Init 함수입니다.

GamePlayerManager의
OnPlayerWin에는 ShowWin을,
OnPlayerDead에는 ShowDead를 넣어줘야겠죠?



해보면 좋아요

- 코드를 살펴보면서 직접 코드를 수정해봐요

- 유니티 기능들은 자세히 안 다뤘어요
  - 유니티 공식 매뉴얼을 살펴봐요
  - https://docs.unity3d.com/kr/2021.3/Manual/UnityManual.html