유니티 C# 스크립트

학생용 수업에 참고할 강의안 제작 유건곤 강사

1. 유니티 C# 스크립트 기초 학습

목표: 유니티에서 C# 기초 문법을 스크립트로 학습하고 이해하기

1.1 변수와 자료형

유니티 스크립트에서 변수와 자료형 사용법을 배우기

```
csharp
코드 복사
using UnityEngine;

public class VariableExample : MonoBehaviour
{
    // 변수 선언
    public int playerScore = 0;
    public float speed = 5.5f;
    public string playerName = "Hero";
    public bool isGameOver = false;

    void Start()
    {
        // 변수 출력
        Debug.Log("Player Name: " + playerName);
        Debug.Log("Player Score: " + playerScore);
        Debug.Log("Speed: " + speed);
        Debug.Log("Is Game Over?: " + isGameOver);
    }
}
```

- **연습**: 변수의 값을 바꾸고 `Debug.Log()`를 통해 확인해보기.

1.2 연산자와 조건문

산술 연산자와 조건문으로 게임 로직 구현

```
csharp
코드 복사
using UnityEngine;

public class ConditionExample : MonoBehaviour
{
    public int health = 100;

    void Update()
    {
        health -= 1; // 체력 감소
        Debug.Log("Health: " + health);

        // 조건문
        if (health <= 0)
        {
            Debug.Log("Game Over!");
        }
    }
}
```

- **연습**: `if`문을 활용하여 체력이 0 이하일 때만 게임 종료 메시지가 출력되도록 수정하기.

1.3 반복문

`for`, `while`, `foreach` 반복문을 유니티에서 활용하기

```
csharp
코드 복사
using UnityEngine;

public class LoopExample : MonoBehaviour
```

```
{
    void Start()
    {
        // for문: 1부터 10까지 출력
        for (int i = 1; i <= 10; i++)
        {
            Debug.Log("Count: " + i);
        }

        // while문: 조건이 참일 때 실행
        int counter = 0;
        while (counter < 5)
        {
            Debug.Log("While Count: " + counter);
            counter++;
        }
    }
}
```

- **연습**: `for`와 `while`을 사용해 다른 패턴의 출력을 만들어보기.

1.4 함수와 메서드

유니티에서 함수를 만들어서 호출하는 방법

```
csharp
코드 복사
using UnityEngine;

public class FunctionExample : MonoBehaviour
{
    void Start()
    {
        SayHello(); // 함수 호출
        int total = AddNumbers(3, 5);
        Debug.Log("Total: " + total);
    }
```

```
    // 함수 정의
    void SayHello()
    {
        Debug.Log("Hello, Unity!");
    }

    int AddNumbers(int a, int b)
    {
        return a + b;
    }
}
```

- **연습**: 다양한 매개변수를 받는 함수를 추가로 만들어보기.

1.5 객체지향 기초 - 클래스와 객체

유니티에서 클래스를 만들어 스크립트 간 상호작용 학습

```
csharp
코드 복사
using UnityEngine;

public class Player
{
    public string name;
    public int score;

    public Player(string playerName, int playerScore)
    {
        name = playerName;
        score = playerScore;
    }

    public void ShowInfo()
    {
        Debug.Log("Player: " + name + ", Score: " + score);
    }
}
```

```
public class ClassExample : MonoBehaviour
{
    void Start()
    {
        Player player1 = new Player("Hero", 10);
        player1.ShowInfo();
    }
}
```

- **연습**: 여러 플레이어 객체를 만들고 정보를 출력하도록 수정해보기.

1.6 MonoBehaviour 주요 함수 학습

유니티 스크립트의 **MonoBehaviour** 기본 함수 이해

```
csharp
코드 복사
using UnityEngine;

public class MonoBehaviourExample : MonoBehaviour
{
    void Start()
    {
        Debug.Log("Start: 게임이 시작될 때 호출됩니다.");
    }

    void Update()
    {
        Debug.Log("Update: 프레임마다 호출됩니다.");
    }

    void FixedUpdate()
    {
        Debug.Log("FixedUpdate: 물리 연산에 사용됩니다.");
    }
```

```
}
```

- **연습**: `Start`, `Update`, `FixedUpdate` 차이를 확인해보기.

---

2. 유니티 환경 이해 및 프로젝트

목표: 유니티 환경을 이해하고, C# 스크립트를 작성해 간단한 동작 구현하기

2.1 유니티 에디터 기본 이해

1. **유니티 설치 및 새 프로젝트 생성**
    - *유니티 허브(Unity Hub)**를 이용해 유니티 설치
    - 새 프로젝트 생성 시 **3D** 또는 **2D** 선택
2. **유니티 에디터 주요 화면 구성**
    - **Hierarchy**: 씬(Scene)에 있는 모든 오브젝트의 목록
    - **Scene View**: 오브젝트를 배치하고 수정하는 공간
    - **Inspector**: 선택한 오브젝트의 속성을 수정
    - **Project**: 프로젝트의 모든 파일과 리소스
    - **Game View**: 게임 실행 화면
3. **오브젝트 생성**
    - `Hierarchy` 창에서 **Create > 3D Object > Cube**를 선택
    - **Cube** 오브젝트가 생성된 것을 확인

2.2 유니티에서 첫 번째 C# 스크립트 작성

예제 1: 오브젝트를 움직이기

목표: 키보드 입력으로 Cube 오브젝트를 좌우로 움직이게 만들기

1. **스크립트 생성**

- `Project` 창에서 **Create > C# Script**로 `MoveObject` 스크립트를 생성
- 생성된 스크립트를 **Cube**에 드래그해서 붙입니다.

2. **스크립트 코드 작성**

```
using UnityEngine;

public class MoveObject : MonoBehaviour
{
    public float speed = 5.0f; // 이동 속도

    void Update()
    {
        // 키 입력에 따라 이동
        float move = Input.GetAxis("Horizontal");
        transform.Translate(Vector3.right * move * speed * Time.deltaTime);
    }
}
```

1. **설명**
   - `Input.GetAxis("Horizontal")` : 키보드의 **A/D** 또는 **왼쪽/오른쪽** 화살표 입력을 감지합니다.
   - `transform.Translate` : 오브젝트를 이동시키는 함수입니다.
   - `Time.deltaTime` : 프레임 속도에 상관없이 일정하게 이동하도록 보정합니다.
2. **실행 결과**
   - Play 버튼을 눌러 실행하면 **Cube**를 좌우로 움직일 수 있습니다.

예제 2: 오브젝트에 중력 추가하기

목표: Rigidbody를 사용해 오브젝트에 중력을 적용해 떨어지게 만들기

1. **Rigidbody 추가**
   - `Inspector` 창에서 **Add Component**를 클릭
   - `Rigidbody`를 검색하고 추가
2. **스크립트 수정**

```
using UnityEngine;

public class MoveWithGravity : MonoBehaviour
{
    public float jumpForce = 5.0f; // 점프 힘

    void Update()
    {
        // Space 키를 누르면 점프
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Rigidbody rb = GetComponent<Rigidbody>();
            rb.AddForce(Vector3.up * jumpForce, ForceMode.Impulse);
        }
    }
}
```

1. **설명**
   - **Rigidbody**: 물리 효과를 추가해 중력을 적용합니다.
   - `AddForce`: 점프를 위해 오브젝트에 힘을 줍니다.
   - `ForceMode.Impulse`: 순간적으로 힘을 가하는 옵션입니다.
2. **실행 결과**
   - **Cube**가 중력에 의해 아래로 떨어지고, 스페이스바를 누르면 위로 점프합니다.

2.3 첫 프로젝트: 간단한 오브젝트 게임

목표: 플레이어가 조작하는 Cube가 장애물을 피하면서 이동하는 미니 게임

1. **장애물 추가**
   - `Hierarchy`에서 **Create > 3D Object > Cube**로 장애물 추가
   - 위치와 크기를 조정하여 다양한 장애물을 배치
2. **플레이어 이동 스크립트**
   - 기존 **MoveObject** 스크립트를 확장해 상하좌우로 움직이도록 수정합니다.

```
using UnityEngine;

public class PlayerMovement : MonoBehaviour
{
    public float speed = 5.0f;

    void Update()
    {
        float moveX = Input.GetAxis("Horizontal"); // 좌우 이동
        float moveZ = Input.GetAxis("Vertical");   // 앞뒤 이동

        Vector3 move = new Vector3(moveX, 0, moveZ);
        transform.Translate(move * speed * Time.deltaTime);
    }
}
```

1. **설명**
   - `Input.GetAxis("Vertical")`: W/S 또는 위/아래 화살표 입력을 감지합니다.
   - `Vector3`: 3D 공간의 이동 방향을 설정합니다.
2. **게임 실행**
   - 플레이어는 Cube를 조작해 장애물을 피하며 이동합니다.

2.4 강의 및 실습 마무리

학습 목표 복습

- 유니티 에디터의 기본 구성 이해
- 오브젝트에 **스크립트**와 **컴포넌트**를 추가하는 방법 학습
- **C# 스크립트**를 활용해 오브젝트 이동 및 물리 효과 구현

추가 과제

1. **장애물 회전**: 장애물에 스크립트를 추가해 일정 속도로 회전시키기
2. **점수 시스템**: 오브젝트가 특정 영역에 도착하면 점수를 증가시키는 기능 추가

이 단계를 통해 유니티 환경에 익숙해지고, **C# 스크립트**로 게임 로직을 구현하는 기본기를 다질 수 있습니다. 이후 단계에서 이를 확장해 더 복잡한 게임을 제작할 수 있습니다. 😊

C# 심화와 유니티 활용

목표: C#의 객체지향 프로그래밍 심화 개념과 유니티 게임 제작 활용 능력 기르기

3.1 클래스와 상속

목표: 클래스의 상속을 활용해 코드 재사용성을 높이기

1. **부모 클래스 만들기**

```
using UnityEngine;

// 부모 클래스
public class Enemy : MonoBehaviour
{
    public float health = 100f;

    public virtual void TakeDamage(float damage)
    {
        health -= damage;
        Debug.Log(gameObject.name + " Health: " + health);
        if (health <= 0)
        {
            Die();
        }
    }

    protected void Die()
    {
        Debug.Log(gameObject.name + " died!");
```

```
        Destroy(gameObject);
    }
}
```

1. **자식 클래스 만들기**

```
using UnityEngine;

// 자식 클래스
public class BossEnemy : Enemy
{
    public override void TakeDamage(float damage)
    {
        // 보스는 데미지를 절반만 받음
        base.TakeDamage(damage * 0.5f);
        Debug.Log("Boss received reduced damage!");
    }
}
```

1. **설명**
   - **부모 클래스**: `Enemy` 클래스는 체력 관리와 데미지 로직을 담당합니다.
   - **자식 클래스**: `BossEnemy`는 부모 클래스의 기능을 상속받아 **데미지를 절반만 받는 기능**을 추가합니다.
   - `virtual`과 `override`: 부모 클래스의 메서드를 자식 클래스에서 재정의할 때 사용합니다.
2. **실습**
   - **Cube** 오브젝트를 하나 생성하고 `Enemy` 스크립트를 추가
   - **BossEnemy** 오브젝트를 만들고 `BossEnemy` 스크립트를 추가
   - 두 오브젝트에 다른 **데미지 값**을 적용해 차이점을 확인합니다.

3.2 인터페이스와 추상 클래스

목표: 인터페이스와 추상 클래스를 활용하여 다양한 캐릭터 동작을 설계하기

1. **인터페이스 정의**

```
public interface IDamageable
{
    void TakeDamage(float damage);
}
```

1. **인터페이스 구현**

```
using UnityEngine;

public class Player : MonoBehaviour, IDamageable
{
    public float health = 100f;

    public void TakeDamage(float damage)
    {
        health -= damage;
        Debug.Log("Player Health: " + health);

        if (health <= 0)
        {
            Debug.Log("Player Died!");
        }
    }
}
```

1. **추상 클래스 활용**

```
using UnityEngine;

public abstract class Character : MonoBehaviour
{
    public abstract void Attack();

    public void Move()
    {
        Debug.Log("Character is moving");
    }
```

```
}

public class Warrior : Character
{
    public override void Attack()
    {
        Debug.Log("Warrior attacks with a sword!");
    }
}
```

1. **설명**
    - **인터페이스**: `IDamageable`은 데미지를 받는 기능만 정의하며, 구현은 각 클래스에서 합니다.
    - **추상 클래스**: `Character`는 **공통 기능(이동)**과 **추상 메서드(공격)**를 제공합니다.
    - `abstract` 메서드는 자식 클래스에서 반드시 구현해야 합니다.
2. **실습**
    - 플레이어와 몬스터에 **인터페이스**와 **추상 클래스**를 각각 적용해 동작을 구현하세요.

3.3 이벤트 시스템 활용

목표: 게임의 다양한 동작을 이벤트로 관리해 코드 간 결합도를 낮추기

체력 UI를 업데이트하는 방식

목표: `UnityEngine.UI`를 활용해 체력 값을 화면에 표시합니다.

1. **UI 준비**
    - **Hierarchy**에서 `Canvas` 추가: **Create > UI > Canvas**
    - **Text** 추가: **Create > UI > Text - Legacy** (Unity 2020 이전) 또는 `UI > TextMeshPro` 사용 가능
    - Text의 이름을 `HealthText`로 설정하고, 글씨 크기와 색상을 조정합니다.

---

1. **스크립트 코드: HealthUI.cs**

```
csharp
코드 복사
using UnityEngine;
using UnityEngine.UI; // UI를 사용하기 위한 네임스페이스

public class HealthUI : MonoBehaviour
{
    public Text healthText; // UI Text 컴포넌트 연결

    void OnEnable()
    {
        // 이벤트에 리스너 추가
        EventExample.HealthChanged += UpdateHealthUI;
    }

    void OnDisable()
    {
        // 이벤트에서 리스너 제거
        EventExample.HealthChanged -= UpdateHealthUI;
    }

    // 체력 업데이트 함수
    void UpdateHealthUI(float health)
    {
        healthText.text = "Health: " + health.ToString("F0"); // 소수점 없이 표시
        Debug.Log("Health Updated: " + health);
    }
}
```

1. **설명**

- **Text 컴포넌트 연결**:
  - `public Text healthText`를 통해 Inspector에서 `HealthText` UI를 연결합니다.
- **이벤트 리스너 관리**:
  - `OnEnable()`에서 이벤트 구독을 설정하고, `OnDisable()`에서 구독을 해제합니다.

- 이렇게 하면 게임 오브젝트가 활성화될 때만 UI 업데이트를 듣게 됩니다.

- **UI 업데이트**:
  - 이벤트가 호출되면 `healthText`의 내용을 갱신합니다.

1. **EventExample 스크립트 (기존 코드)**

```
csharp
코드 복사
using UnityEngine;

public class EventExample : MonoBehaviour
{
    public delegate void OnHealthChange(float health);
    public static event OnHealthChange HealthChanged;

    private float health = 100f;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space)) // 스페이스바로 데미지
        {
            TakeDamage(10f);
        }
    }

    void TakeDamage(float damage)
    {
        health -= damage;
        Debug.Log("Health: " + health);

        // 이벤트 호출
        HealthChanged?.Invoke(health);
    }
}
```

5. 오브젝트 설정

1. **EventExample** 스크립트를 빈 오브젝트(예: `GameManager`)에 추가합니다.
2. **HealthUI** 스크립트를 Canvas > `HealthText`가 있는 UI 오브젝트에 추가합니다.
3. **HealthText 연결**:
    - HealthUI 컴포넌트의 `healthText` 필드에 **HealthText** 오브젝트를 드래그합니다.

실행 결과

- 게임을 실행하고 **스페이스바**를 누르면:
    - 체력이 감소하고 화면에 표시된 Health UI가 업데이트됩니다.
    - 이벤트는 오브젝트가 활성화되어 있을 때만 UI를 갱신합니다.

추가 팁

1. **UI Text 대신 TextMeshPro 사용**
    - `TextMeshPro`는 더 나은 성능과 퀄리티를 제공합니다. `TextMeshProUGUI`로 변경해서 사용하면 됩니다.
2. **비활성화 처리**
    - `HealthUI` 오브젝트가 비활성화되면 더 이상 이벤트를 듣지 않으므로 불필요한 연산이 줄어듭니다.

3.4 미니 프로젝트: 슈팅 게임

목표: 객체지향 개념을 활용해 간단한 **슈팅 게임** 제작

1. **플레이어 이동 및 총알 발사**

```
using UnityEngine;

public class PlayerShooter : MonoBehaviour
{
    public GameObject bulletPrefab;
    public Transform firePoint;

    void Update()
    {
        // 플레이어 이동
        float move = Input.GetAxis("Horizontal");
```

```
        transform.Translate(Vector3.right * move * Time.deltaTime * 5f);

        // 스페이스바로 총알 발사
        if (Input.GetKeyDown(KeyCode.Space))
        {
            Instantiate(bulletPrefab, firePoint.position, firePoint.rotation);
        }
    }
}
```

1. **총알 스크립트**

```
using UnityEngine;

public class Bullet : MonoBehaviour
{
    public float speed = 10f;

    void Update()
    {
        transform.Translate(Vector3.up * speed * Time.deltaTime);
    }

    void OnCollisionEnter(Collision other)
    {
        Destroy(gameObject); // 충돌 시 총알 제거
    }
}
```

1. **설명**
    - 플레이어는 좌우로 이동하며 **스페이스바**를 눌러 총알을 발사합니다.
    - 총알은 `Instantiate`를 사용해 복제되며 **위로 이동**합니다.
2. **실습 확장**
    - **적 오브젝트**를 추가하고 충돌 시 체력을 감소시키는 기능을 구현하세요.

강의 및 실습 마무리

학습 목표 복습

- **클래스 상속**, **인터페이스**, **추상 클래스** 개념 이해 및 활용
- 이벤트 시스템을 통해 게임의 다양한 동작 관리
- 객체지향 설계를 활용해 슈팅 게임의 기초 로직 구현

추가 과제

- 총알이 적에게 닿으면 점수를 올리는 기능 추가
- 적이 일정 시간마다 등장하는 **스폰 시스템** 구현

이 과정을 통해 학습자는 **C# 심화 개념**과 **유니티 활용 능력**을 익히고, 객체지향 설계를 게임에 적용할 수 있게 됩니다. 🚀

프로젝트와 예제를 통한 실습으로 실력을 향상시킨다.

디자인 패턴이란?

디자인 패턴이란 자주 발생하는 문제를 해결하기 위한 **재사용 가능한 코드 구조**입니다. 유지보수성과 확장성을 높이는 **객체지향 설계 원칙**을 기반으로 합니다.

---

00. 빌더 패턴 (Builder)

정의

복잡한 객체를 단계적으로 생성할 수 있도록 도와주는 패턴입니다. 생성 과정과 표현을 분리합니다.

유니티 예제: 캐릭터 생성기

```
using UnityEngine;

// Builder 인터페이스
public interface ICharacterBuilder
```

```
{
    void SetName(string name);
    void SetHealth(int health);
    void SetDamage(int damage);
    Character Build();
}

// Concrete Builder
public class CharacterBuilder : ICharacterBuilder
{
    private Character character = new Character();

    public void SetName(string name) { character.Name = name; }
    public void SetHealth(int health) { character.Health = health; }
    public void SetDamage(int damage) { character.Damage = damage; }

    public Character Build() { return character; }
}

// Product
public class Character
{
    public string Name;
    public int Health;
    public int Damage;

    public void Display()
    {
        Debug.Log($"Character: {Name}, Health: {Health}, Damage: {Damag
e}");
    }
}

// Director
public class CharacterDirector
{
    public void ConstructKnight(ICharacterBuilder builder)
    {
```

```
        builder.SetName("Knight");
        builder.SetHealth(150);
        builder.SetDamage(30);
    }
}

// 유니티 스크립트
public class BuilderPatternExample : MonoBehaviour
{
    void Start()
    {
        ICharacterBuilder builder = new CharacterBuilder();
        CharacterDirector director = new CharacterDirector();

        // Knight 생성
        director.ConstructKnight(builder);
        Character knight = builder.Build();
        knight.Display();
    }
}
```

설명

- **Builder**: 객체 생성 과정을 단계적으로 수행합니다.
- **Director**: 빌더를 조합해 구체적인 객체를 생성합니다.
- **실습**: 다양한 캐릭터(예: Archer, Mage)를 추가로 만들어 보세요.

---

01. 상태 패턴 (State)

정의

객체의 상태에 따라 행동을 변경하는 패턴입니다. 상태를 클래스로 분리합니다.

유니티 예제: 캐릭터 상태 전환

```
using UnityEngine;

// 상태 인터페이스
```

```
public interface IState
{
    void Handle();
}

// 구체적인 상태들
public class IdleState : IState
{
    public void Handle() => Debug.Log("Character is Idle.");
}

public class MoveState : IState
{
    public void Handle() => Debug.Log("Character is Moving.");
}

public class AttackState : IState
{
    public void Handle() => Debug.Log("Character is Attacking.");
}

// Context
public class Character : MonoBehaviour
{
    private IState currentState;

    public void SetState(IState state)
    {
        currentState = state;
        currentState.Handle();
    }

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.I)) SetState(new IdleState());
        if (Input.GetKeyDown(KeyCode.M)) SetState(new MoveState());
        if (Input.GetKeyDown(KeyCode.A)) SetState(new AttackState());
```

```
    }
}
```

설명

- 각 **상태**는 `IState`를 구현하며 **행동**을 정의합니다.
- **Context**는 상태를 변경하고 해당 상태에 맞는 행동을 호출합니다.
- **실습**: 점프 상태, 죽음 상태 등을 추가해 보세요.

---

02. 어댑터 패턴 (Adapter)

정의

호환되지 않는 인터페이스를 연결해주는 패턴입니다.

유니티 예제: 유닛 속도 변환기

```
using UnityEngine;

// 기존 시스템
public class OldSpeedSystem
{
    public float GetSpeedInKmH() => 100f; // km/h
}

// 타겟 인터페이스
public interface INewSpeedSystem
{
    float GetSpeedInMilesPerHour();
}

// 어댑터 클래스
public class SpeedAdapter : INewSpeedSystem
{
    private OldSpeedSystem oldSystem;

    public SpeedAdapter(OldSpeedSystem system)
    {
```

```
        oldSystem = system;
    }

    public float GetSpeedInMilesPerHour()
    {
        return oldSystem.GetSpeedInKmH() * 0.621371f; // 변환
    }
}

// 유니티 스크립트
public class AdapterPatternExample : MonoBehaviour
{
    void Start()
    {
        OldSpeedSystem oldSystem = new OldSpeedSystem();
        INewSpeedSystem adapter = new SpeedAdapter(oldSystem);

        Debug.Log("Speed in Miles per Hour: " + adapter.GetSpeedInMilesPer
Hour());
    }
}
```

설명

- **어댑터**가 서로 다른 속도 단위를 변환합니다.
- **실습**: 다른 단위 변환 (예: 온도, 길이)을 추가해 보세요.

---

03. 컴포지트 패턴 (Composite)

정의

트리 구조를 구성해 객체들을 계층적으로 관리하는 패턴입니다.

유니티 예제: 오브젝트 그룹 관리

```
using System.Collections.Generic;
using UnityEngine;
```

```
// 컴포넌트 인터페이스
public interface IComponent
{
    void Operation();
}

// 리프 클래스
public class Leaf : IComponent
{
    private string name;

    public Leaf(string name) => this.name = name;

    public void Operation() => Debug.Log("Leaf: " + name);
}

// 컴포지트 클래스
public class Composite : IComponent
{
    private List<IComponent> children = new List<IComponent>();

    public void Add(IComponent component) => children.Add(component);
    public void Operation()
    {
        foreach (var child in children)
            child.Operation();
    }
}

// 유니티 스크립트
public class CompositePatternExample : MonoBehaviour
{
    void Start()
    {
        Composite root = new Composite();
        root.Add(new Leaf("Leaf 1"));
        root.Add(new Leaf("Leaf 2"));
```

```
        Composite subTree = new Composite();
        subTree.Add(new Leaf("Leaf 3"));
        root.Add(subTree);

        root.Operation(); // 전체 구조 출력
    }
}
```

설명

- **Leaf**는 개별 객체를 나타내고 **Composite**는 그룹화된 객체입니다.
- **실습**: 게임 오브젝트를 계층 구조로 관리해보세요.

---

04. 싱글톤 패턴 (Singleton)

정의

한 클래스의 객체가 **하나만 존재하도록** 보장하는 패턴입니다.

유니티 예제: 게임 매니저

```
using UnityEngine;

public class GameManager : MonoBehaviour
{
    private static GameManager instance;

    public static GameManager Instance
    {
        get
        {
            if (instance == null)
            {
                instance = new GameObject("GameManager").AddComponent<G
ameManager>();
            }
            return instance;
        }
```

```
    }

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

    public void DisplayMessage() ⇒ Debug.Log("Singleton Instance Runnin
g");
}
```

설명

- `GameManager.Instance`로 어디서나 접근 가능합니다.
- **실습**: 점수 관리, 씬 전환 등을 싱글톤으로 관리해보세요.

---

05. 옵저버 패턴 (Observer)

---

1. 정의

옵저버 패턴은 **한 객체의 상태 변화**를 다른 객체들이 감지하고 **자동으로 동작**하도록 만드는 패턴입니다.

- **Subject**: 상태 변화를 감지할 대상
- **Observer**: 상태 변화를 감지하고 반응하는 객체

유니티에서는 **이벤트(Event)**와 **델리게이트(Delegate)**를 활용해 옵저버 패턴을 쉽게 구현할 수 있습니다.

---

2. 유니티 C# 스크립트 예제: 체력 시스템 & UI 업데이트

예제 목표

- 플레이어의 체력이 감소할 때 **UI**가 자동으로 업데이트되도록 옵저버 패턴을 구현합니다.

2.1 Subject: 체력 관리 클래스

Subject는 이벤트를 발생시키고 상태 변화를 알립니다.

```
using UnityEngine;

public class HealthManager : MonoBehaviour
{
    public delegate void OnHealthChanged(int currentHealth); // 이벤트 델리
게이트 선언
    public static event OnHealthChanged HealthChanged; // 이벤트

    private int health = 100;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space)) // 스페이스바를 누르면 체력 감소
        {
            TakeDamage(10);
        }
    }

    void TakeDamage(int damage)
    {
        health -= damage;
        Debug.Log("Player Health: " + health);

        // 이벤트 발생 (옵저버에게 상태 변화 알림)
        HealthChanged?.Invoke(health);

        if (health <= 0)
        {
            Debug.Log("Player is Dead!");
        }
    }
}
```

2.2 Observer: 체력 UI 업데이트 클래스

옵저버는 Subject의 이벤트를 구독하고 반응합니다.

```
using UnityEngine;
using UnityEngine.UI;

public class HealthUI : MonoBehaviour
{
    public Text healthText; // 체력을 표시할 UI 텍스트

    void OnEnable()
    {
        // Subject의 이벤트를 구독
        HealthManager.HealthChanged += UpdateHealthUI;
    }

    void OnDisable()
    {
        // 이벤트 구독 해제
        HealthManager.HealthChanged -= UpdateHealthUI;
    }

    void UpdateHealthUI(int currentHealth)
    {
        healthText.text = "Health: " + currentHealth; // UI 업데이트
        Debug.Log("UI Updated: " + currentHealth);
    }
}
```

2.3 씬 설정

1. **UI 설정**
    - **Canvas**를 생성하고 **Text** UI를 추가합니다.
    - `HealthUI` 스크립트를 Text 오브젝트에 추가하고 `healthText` 필드에 연결합니다.
2. **플레이어 오브젝트 설정**
    - 빈 오브젝트를 만들고 `HealthManager` 스크립트를 추가합니다.

3. 실행 결과

1. 게임 실행 후 **스페이스바**를 누르면:
    - **HealthManager**가 체력을 감소시키고 이벤트를 발생시킵니다.
    - **HealthUI**는 이벤트를 감지하고 UI를 업데이트합니다.
2. 체력 값이 0 이하로 감소하면 콘솔에 "Player is Dead!" 메시지가 출력됩니다.

---

4. 설명

- **HealthManager** (Subject): 체력 상태를 관리하고 상태가 변하면 이벤트를 발생시킵니다.
- **HealthUI** (Observer): 이벤트를 구독하고 상태 변화에 반응해 UI를 업데이트합니다.
- **OnEnable() / OnDisable()**: 옵저버의 이벤트 구독과 해제를 관리해 **메모리 누수**를 방지합니다.
- **Delegate와 Event**: 옵저버 패턴을 구현하기 위한 핵심 도구입니다.

---

5. 실습 확장

1. **여러 옵저버 추가**
    - 체력이 변경될 때 **사운드 효과**를 재생하는 클래스를 추가해 보세요.
2. **적군 체력 시스템 구현**
    - 여러 적군이 동일한 이벤트를 구독하고 개별 체력을 관리하도록 확장하세요.

이 예제를 통해 옵저버 패턴의 핵심인 **이벤트 기반 상태 알림**을 쉽게 이해하고 유니티 프로젝트에 적용할 수 있습니다. 😊

---

06. 플라이웨이트 패턴 (Flyweight)

정의

공유 객체를 사용해 메모리 사용량을 줄이는 패턴입니다.

유니티 예제: 적 오브젝트 재사용

```
using System.Collections.Generic;
using UnityEngine;
```

```
public class EnemyFactory
{
    private Dictionary<string, GameObject> enemyPool = new Dictionary<str
ing, GameObject>();

    public GameObject GetEnemy(string type)
    {
        if (!enemyPool.ContainsKey(type))
        {
            GameObject enemy = new GameObject(type);
            enemyPool[type] = enemy;
        }
        return enemyPool[type];
    }
}

public class FlyweightPatternExample : MonoBehaviour
{
    void Start()
    {
        EnemyFactory factory = new EnemyFactory();
        factory.GetEnemy("Orc");
        factory.GetEnemy("Goblin");
        factory.GetEnemy("Orc"); // 이미 존재하는 객체를 반환
    }
}
```

설명

- 동일한 객체를 재사용해 메모리 사용량을 줄입니다.
- **실습**: 총알, 적 캐릭터 객체를 재사용하도록 확장해 보세요.

이 예제들을 통해 각 디자인 패턴의 개념과 유니티에서의 활용을 익힐 수 있습니다! 😊