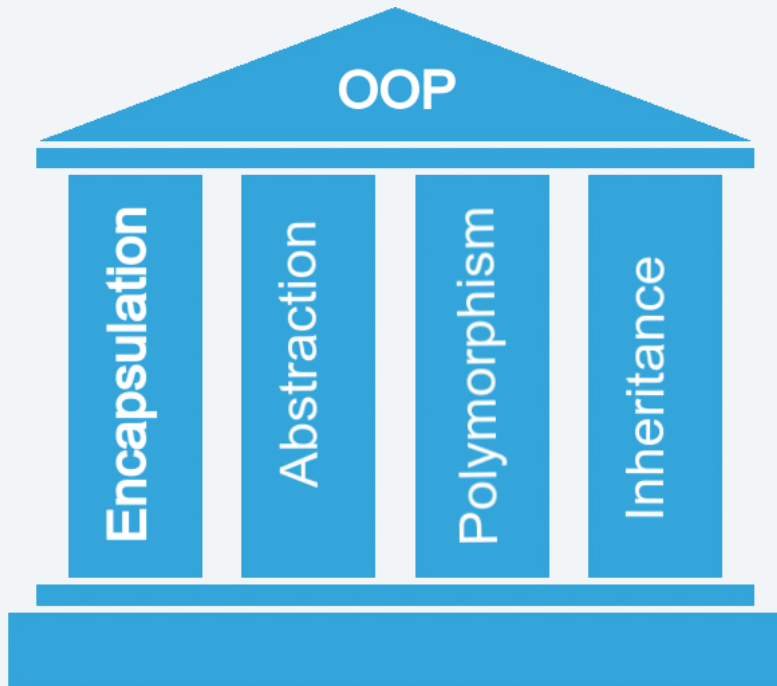


메서드, 프로퍼티, 상속

# 메서드, 프로퍼티, 상속

메서드, 프로퍼티, 상속 : 객체지향의 기둥이다!



객체는 자신만의 행동(메서드)을 가지고,  
자신의 상태(프로퍼티)를 스마트하게 관리하며,  
다른 객체와의 관계(상속)를 통해 확장

# 메서드: 객체의 행동

## 메서드: 객체에 생명을 불어넣는 행동

- 객체 = 상태(데이터) + 행동(메서드)
  - 현실 세계의 객체들
    - 자동차: 달린다, 멈춘다, 방향을 바꾼다
    - 캐릭터: 걷는다, 뛴다, 공격한다, 회복한다
- 메서드가 없다면? 단순한 데이터 구조일 뿐

# 메서드의 종류

- **인스턴스 메서드**

- 객체의 상태(필드)를 사용
- 객체마다 다른 결과 가능
- 예시: 캐릭터의 공격, 이동, 회복

## 인스턴스 메서드

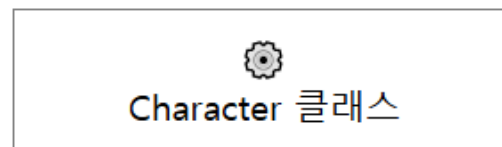


각자 다른 상태와 행동

- **정적(Static) 메서드**

- 객체의 상태와 무관
- 클래스에 속하며 모든 객체가 공유
- 예시: 유틸리티 기능, 팩토리 메서드

## 정적 메서드



모든 객체가 공유하는 기능

# 메서드 오버로딩

## 메서드 오버로딩: 같은 이름, 다른 매개변수

- 하나의 메서드 이름으로 다양한 동작 구현
- 매개변수의 개수나 타입이 달라야 함
- 반환 타입만 다른 경우는 오버로딩 불가

```
// 기본 공격
void Attack() { ... }

// 대상을 지정한 공격
void Attack(Enemy target) { ... }

// 대상과 공격력을 지정한 공격
void Attack(Enemy target, int power) { ... }

// 범위 공격
void Attack(float radius) { ... }
```


# 프로퍼티: 스마트한 필드


- 단순 필드 vs 프로퍼티
  - 필드: 단순 데이터 저장소
  - 프로퍼티: 접근자(get/set)로 제어되는 스마트한 데이터

## 단순 필드



```
public int health = 100;
```


 값 변경 제한 없음


 `health = -999;` (문제 발생!)

## 프로퍼티



```
private int health;  
public int Health { get; set; }
```

 유효성 검사 가능

 `Health = Math.Max(0, value);`

# 프로퍼티의 구성요소

## 기본 구조

```
private int health; // 백킹 필드

public int Health // 프로퍼티
{
    get { return health; } // getter 접근자
    set { health = value; } // setter 접근자
}
```

## 응용

```
// 읽기 전용 프로퍼티
public bool IsAlive { get { return health > 0; } }

// 자동 구현 프로퍼티
public string Name { get; set; }

// 접근 제한 프로퍼티
public int Level { get; private set; }
```

### 🔑 프로퍼티 구조

#### 📁 백킹 필드 (실제 데이터 저장)

```
private int health;
```

#### 🔍 프로퍼티 선언

```
public int Health
```

#### 📡 getter

```
get { return health; }
```

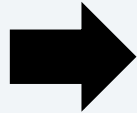
#### 📡 setter

```
set { health = value; }
```

*value*는 프로퍼티에 할당하는 값을 나타내는 암시적 변수

# 프로퍼티의 활용

## 1. 데이터 유효성 검사

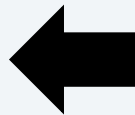


```
public int Experience
{
    get { return experience; }
    set
    {
        int oldValue = experience;
        experience = value;

        // 값 변경 감지
        if (experience != oldValue)
        {
            // 레벨업 체크
            CheckLevelUp();
            // UI 업데이트
            UpdateExperienceUI();
        }
    }
}
```

```
public int Health
{
    get { return health; }
    set
    {
        // 0-100 사이 값으로 제한
        health = Math.Clamp(value, 0, 100);
    }
}
```

## 2. 값 변경 감지 및 연계 작업





# 메서드와 프로퍼티의 협력

## 객체지향 설계의 핵심

- 프로퍼티: 객체의 상태 관리
- 메서드: 객체의 행동 정의
- \*함께 작동하여 완전한 객체 모델 구현\*

```
// 프로퍼티: 상태 관리
public int Experience
{
    get { return experience; }
    set
    {
        experience = value;
        CheckLevelUp(); // 메서드 호출
    }
}

public int Level { get; private set; } = 1;
```

```
// 메서드: 행동 정의
private void CheckLevelUp()
{
    int requiredExp = Level * 100;

    while (Experience >= requiredExp)
    {
        Experience -= requiredExp;
        Level++;
        LevelUp(); // 다른 메서드 호출
    }
}

private void LevelUp()
{
    // 레벨업 효과 (능력치 상승, 효과음, 파티클 등)
    Health = MaxHealth; // 프로퍼티 사용
    OnLevelUp?.Invoke(Level); // 이벤트 발생
}
```

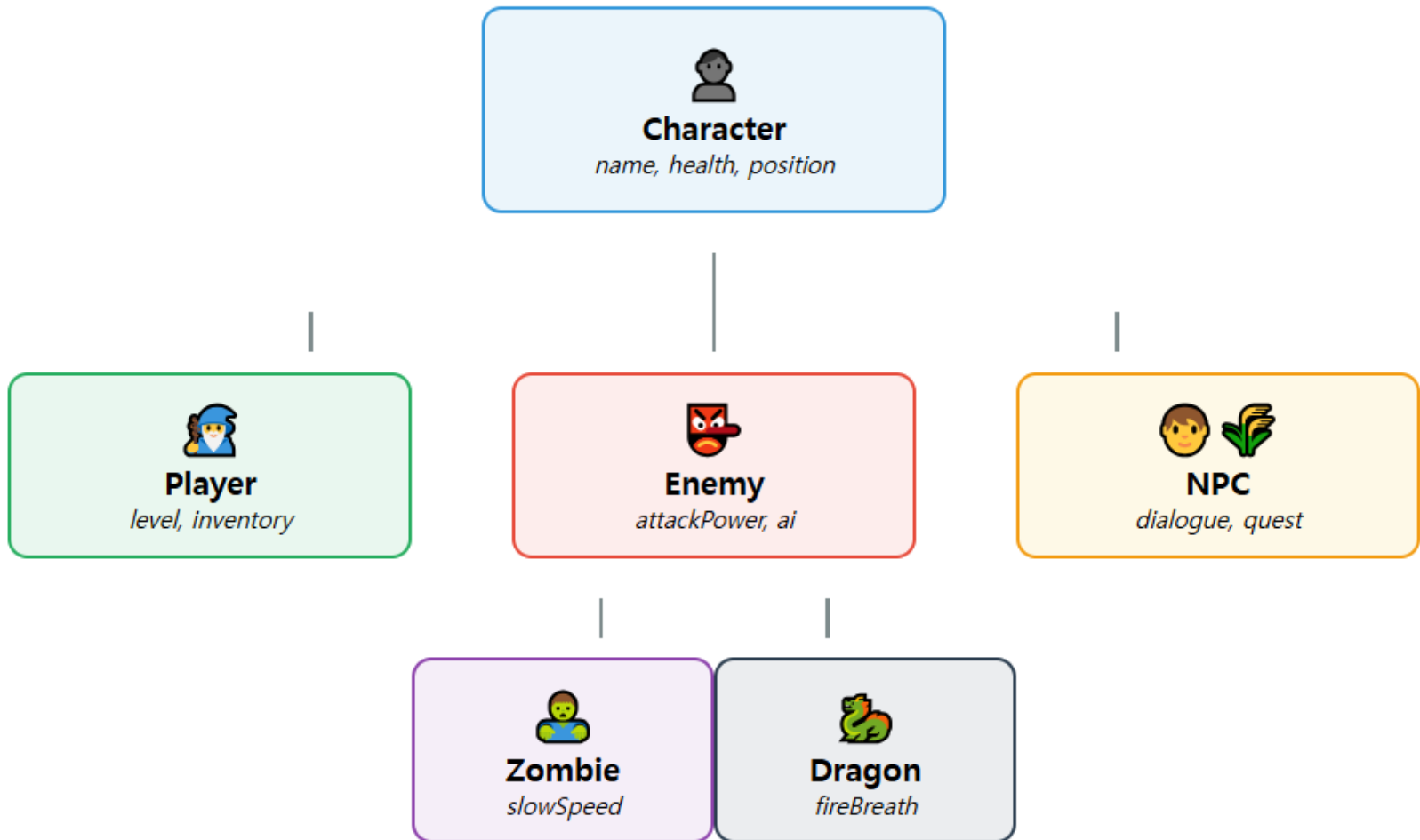
# 상속: 개념 소개

## 상속: 코드 재사용과 계층 구조화

- 상속이란?
  - 기존 클래스(부모)의 특성과 동작을 새 클래스(자식)가 물려받는 메커니즘
  - **is-a 관계**를 표현 **A는 B의 일종이다**
  - 예시: 캐릭터의 공격, 이동, 회복
- 상속의 이점
  - 코드 재사용성 향상
  - 계층적 관계 표현
  - 다형성의 기반 제공

# 상속: 개념 소개

🎮 게임 캐릭터 상속 계층도



# 상속의 기본 구조

```
// 부모 클래스 정의
public class Character
{
    public string Name { get; set; }
    public int Health { get; protected set; }

    public Character(string name, int health)
    {
        Name = name;
        Health = health;
    }

    public virtual void TakeDamage(int damage)
    {
        Health -= damage;
        if (Health < 0) Health = 0;
    }
}
```

```
// 자식 클래스 정의
public class Enemy : Character // ':' 기호로 상속 표현
{
    public int AttackPower { get; protected set; }

    // base 키워드로 부모 생성자 호출
    public Enemy(string name, int health, int attackPower)
        : base(name, health)
    {
        AttackPower = attackPower;
    }

    public void Attack(Character target)
    {
        target.TakeDamage(AttackPower);
    }
}
```

# 메서드 재정의(Override)

## 상속: 코드 재사용과 계층 구조화

- 메서드 재정의란?
  - 부모 클래스의 메서드를 자식 클래스에서 새롭게 구현
  - 같은 메서드 이름과 시그니처를 유지하면서 다른 동작 구현
  - 다형성의 핵심 메커니즘
- 필요한 키워드
  - 부모 클래스: virtual (가상 메서드 선언)
  - 자식 클래스: override (메서드 재정의)
  - base 키워드: 부모 메서드 호출

# 메서드 재정의(Override)

```
// 부모 클래스
public class Character
{
    // 가상 메서드
    public virtual void TakeDamage(int damage)
    {
        Health -= damage;
        if (Health < 0) Health = 0;
        Console.WriteLine($"{Name}이(가) {damage}의 데미지를 받았습니다.");
    }
}
```

# 메서드 재정의(Override)

```
// 자식 클래스
public class Tank : Character
{
    public int Defense { get; private set; }

    // 메서드 재정의
    public override void TakeDamage(int damage)
    {
        // 방어력 적용 (데미지 감소)
        int reducedDamage = Math.Max(1, damage - Defense);

        // 부모 메서드 호출
        base.TakeDamage(reducedDamage);

        Console.WriteLine($"방어력으로 인해 데미지가 {damage - reducedDamage} 감소했습니다.");
    }
}
```

# 메서드 재정의(Override)

## 메서드 재정의 동작 방식

### Character 클래스 (부모)

#### virtual 메서드

```
public virtual void TakeDamage(int damage)
{
    Health -= damage;
    if (Health < 0) Health = 0;
}
```



override



base.TakeDamage()



추가 동작

### Tank 클래스

#### override 메서드

```
// 방어력 적용
reducedDamage = damage -
Defense;
base.TakeDamage(reducedDamage);
```

### Wizard 클래스

#### override 메서드

```
// 마법 방어막 확인
if (shieldActive)
    damage /= 2;
base.TakeDamage(damage);
```

### Zombie 클래스

#### override 메서드

```
base.TakeDamage(damage);
// 체력 회복
if (Health > 0)
    Health += 1;
```



# 상속과 생성자

## 상속: 코드 재사용과 계층 구조화

- **생성자 호출 순서**
  - 자식 객체 생성 시, 부모 생성자가 먼저 호출됨
  - 기본 생성자가 없는 부모 클래스를 상속할 때는 명시적 호출 필요
- **base 키워드**
  - 부모 클래스의 생성자를 호출하는 데 사용
  - 부모의 초기화 로직을 재사용

# 상속과 생성자

```
public class Weapon
{
    public string Name { get; private set; }
    public int Damage { get; private set; }

    // 매개변수가 있는 생성자
    public Weapon(string name, int damage)
    {
        Name = name;
        Damage = damage;
        Console.WriteLine($"무기 '{name}' 생성됨");
    }
}
```

# 상속과 생성자

```
public class MagicWeapon : Weapon
{
    public int ManaCost { get; private set; }

    // base 키워드로 부모 생성자 호출
    public MagicWeapon(string name, int damage, int manaCost)
        : base(name, damage) // Weapon(name, damage) 호출
    {
        ManaCost = manaCost;
        Console.WriteLine($"마법 무기 '{name}' 생성됨, 마나 소모: {manaCost}");
    }
}

// 사용 예:
MagicWeapon staff = new MagicWeapon("불꽃 지팡이", 25, 10);
// 출력:
// 무기 '불꽃 지팡이' 생성됨
// 마법 무기 '불꽃 지팡이' 생성됨, 마나 소모: 10
```

1

### 객체 생성 시작

```
MagicWeapon staff = new MagicWeapon("불꽃 지팡이", 25, 10);
```

2

### 메모리 할당

*Weapon* 부분과 *MagicWeapon* 부분을 위한 메모리 공간 할당

3

### 부모 클래스 생성자 호출

```
Weapon(string name, int damage)
```

*Name, Damage* 초기화

출력: 무기 '불꽃 지팡이' 생성됨

4

### 자식 클래스 생성자 실행

```
MagicWeapon(string name, int damage, int manaCost)
```

*ManaCost* 초기화

출력: 마법 무기 '불꽃 지팡이' 생성됨, 마나 소모: 10

5

### 객체 생성 완료

*staff* 변수가 완전히 초기화된 *MagicWeapon* 객체를 참조

# 상속과 접근 제어

- 접근 제어자

- private: 해당 클래스 내에서만 접근 가능
- protected: 해당 클래스와 파생 클래스에서 접근 가능
- public: 어디서나 접근 가능

- 설계 관점에서의 접근 제어

- 너무 많은 public 멤버: 캡슐화 약화
- 너무 많은 private 멤버: 상속의 이점 감소
- protected: 상속과 캡슐화의 균형

# 상속과 접근 제어

```
public class Character
{
    public string Name { get; set; }           // 누구나 접근 가능
    private int health;                       // Character 클래스만 접근 가능
    protected int maxHealth;                 // Character와 파생 클래스만 접근 가능

    public Character(string name, int maxHp)
    {
        Name = name;
        maxHealth = maxHp;
        health = maxHp;
    }

    // 공개 인터페이스
    public bool IsAlive => health > 0;

    public virtual void TakeDamage(int damage)
    {
        health -= damage;
        if (health < 0) health = 0;
    }

    // 내부 구현
    protected void RestoreFullHealth()
    {
        health = maxHealth;
    }
}
```

# 상속과 접근 제어

```
public class Hero : Character
{
    private int lives;

    public Hero(string name, int maxHp, int startLives) : base(name, maxHp)
    {
        lives = startLives;
    }

    public override void TakeDamage(int damage)
    {
        base.TakeDamage(damage);

        // 사망 시 생명 하나 사용하고 체력 회복
        if (!IsAlive && lives > 0)
        {
            lives--;
            RestoreFullHealth(); // protected 메서드 접근 가능
            // health = maxHealth; // private 필드이므로 직접 접근 불가
        }
    }
}
```

## 접근 제어자와 접근 범위

### Character 클래스 (부모)

#### public

```
string Name { get; set; }  
bool IsAlive { get; }  
void TakeDamage(int)
```

#### protected

```
int maxHealth;  
void RestoreFullHealth()
```

#### private

```
int health;
```

### Hero 클래스 (자식)

#### 상속된 public

```
string Name { get; set; }  
bool IsAlive { get; }  
void TakeDamage(int)
```

#### 상속된 protected

```
int maxHealth;  
void RestoreFullHealth()
```

#### 접근 불가 (private)

```
int health; ❌
```

#### Hero 클래스 자체 멤버

```
private int lives;  
public override void TakeDamage(int)
```



# 상속의 한계와 대안

- 상속의 한계
  - 깊은 상속 계층의 복잡성
  - 단일 상속만 가능 (다중 상속 불가)
  - 강한 결합도: 부모 클래스 변경이 모든 자식에 영향
- 컴포지션(구성)
  - **has-a** 관계 **A는 B를 가지고 있다**
  - 객체가 다른 객체를 포함하는 방식
  - 더 유연하고 느슨한 결합 제공
- 상속 vs 컴포지션: 언제 무엇을 사용할까?
  - 상속: 명확한 "is-a" 관계, 공통 기능이 많을 때
  - 컴포지션: 유연성이 필요하거나 동작이 자주 변경될 때

# 상속의 한계와 대안

// 상속 접근법

```
public class Sword : Weapon
{
    public void Slash() { /* ... */ }
}

public class Wizard : Character
{
    public void CastSpell() { /* ... */ }
}
```

// 컴포지션 접근법

```
public class Player
{
    private Weapon equippedWeapon; // 포함 관계
    private Inventory inventory;    // 포함 관계

    public void EquipWeapon(Weapon weapon)
    {
        equippedWeapon = weapon;
    }

    public void Attack()
    {
        if (equippedWeapon != null)
            equippedWeapon.Use();
    }
}
```

## 상속 vs 컴포지션

### 상속 (is-a 관계)

#### Character

*health, name, position*  
*Move(), TakeDamage()*



#### Wizard

*mana, spells*  
*CastSpell()*

#### Warrior

*strength, rage*  
*SwingSword()*

#### 특징:

- "마법사는 캐릭터이다"
- 모든 상태와 동작 상속
- 강한 결합, 제한된 유연성

### 컴포지션 (has-a 관계)

#### Player

*name, position*  
*Move(), Attack()*



**Weapon**  
Use()

**Inventory**  
AddItem()

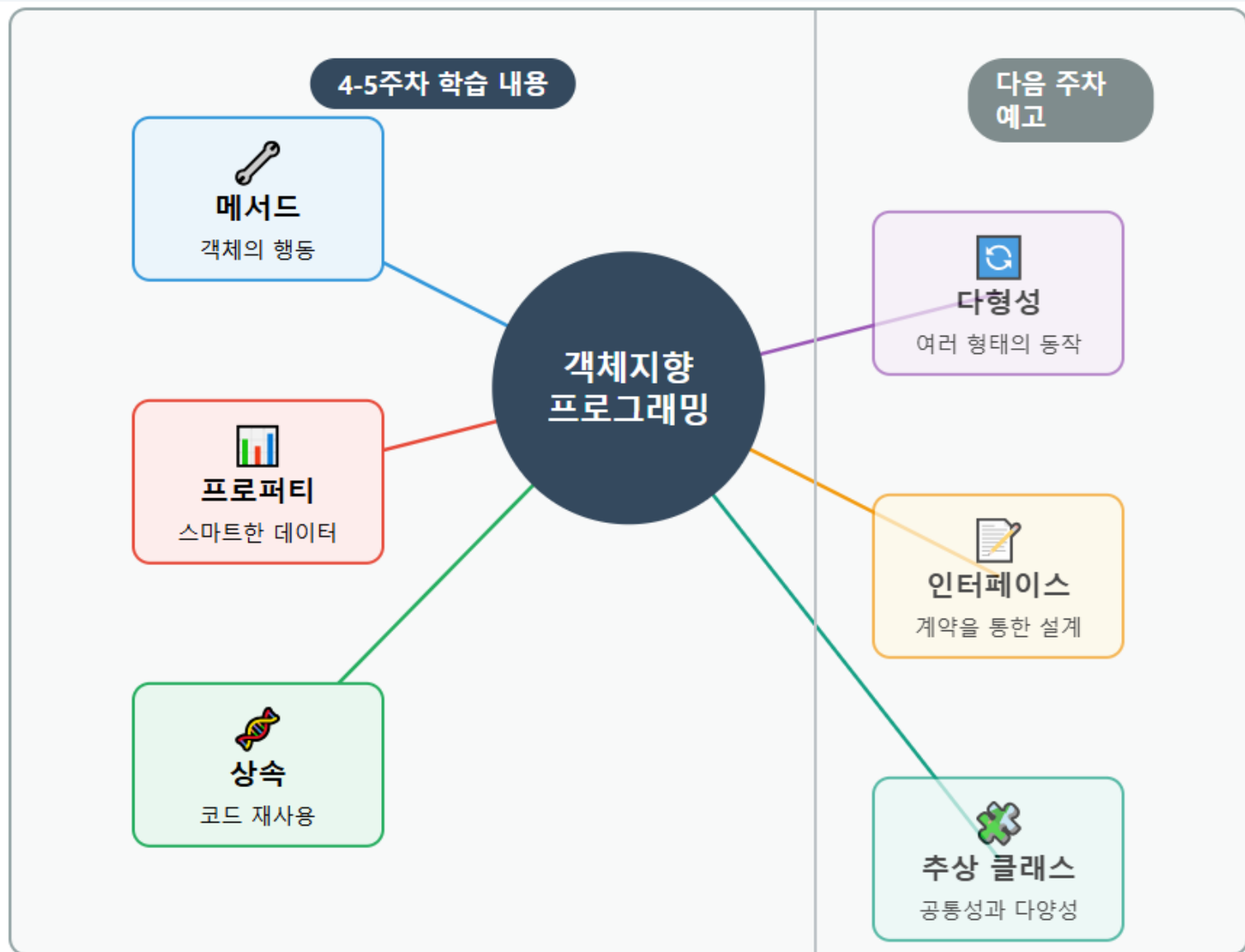
**Stats**  
LevelUp()

#### 특징:

- "플레이어는 무기를 가지고 있다"
- 런타임에 동작 변경 가능
- 느슨한 결합, 높은 유연성

좋은 객체지향 설계는 상속과 컴포지션을 적절히 조합하는 것.  
각 접근법의 장단점을 이해하고, 상황에 맞게 선택하는 능력이 중요.

# 요약 및 다음 주차 예고



**END.**