상속심화

일반 클래스의 한계

1. 설계 의도의 명확한 전달 부족
2. 확장 지점 제약
3. 불완전한 계약 정의
4. 일관성 보장 어려움

- 어떤 메서드가 재정의용인지, 필수 구현인지 불명확
- 자식 클래스의 구현 의무 명시 불가능
- "반드시 구현해야 함"을 강제할 방법 없음
- 다양한 파생 클래스 간의 **일관된 인터페이스 유지 힘듦**

일반 클래스 상속의 문제 예시

```csharp
public class Monster
{
    public int Health { get; protected set; }

    public Monster()
    {
        Health = 100;
        Initialize(); // 자식 클래스의 초기화를 의도
    }

    // 재정의용으로 의도되었지만 virtual이 빠짐
    protected void Initialize()
    {
        Console.WriteLine("기본 몬스터 초기화");
    }

    // 공격 메서드 - 재정의 가능
    public virtual void Attack(Character target)
    {
        // 기본 공격 구현
    }

    // 자식 클래스에서 반드시 구현해야 하지만 강제할 수 없음
    public virtual void SpecialMove()
    {
        // 기본 구현이 의미 없음
        throw new NotImplementedException("자식 클래스에서 구현해야 합니다");
    }
}
```

```csharp
// 문제가 발생할 수 있는 자식 클래스
public class Dragon : Monster
{
    private int fireBreathPower;

    // 문제 1: 부모 Initialize()가 virtual이 아니라 오버라이드 불가능
    // Dragon 생성자에서 fireBreathPower 초기화 전에 부모 생성자에서 호출됨
    protected void Initialize()
    {
        fireBreathPower = 50; // 부모의 Initialize() 대신 실행되지 않음
    }

    // 문제 2: SpecialMove() 구현 누락 가능 - 컴파일러 경고 없음
    // 런타임에 NotImplementedException 발생 가능
}
```

추상 클래스(Abstract Class)

구현의 청사진, 행동의 표준화

🎮 게임 예시: 모든 무기는 '공격'할 수 있어야 함
검은 베고, 도끼는 내려찍고, 활은 화살을 쏘고....

```
// 추상 클래스 선언
public abstract class Weapon
{
    // 일반 속성
    public string Name { get; protected set; }
    public int Damage { get; protected set; }

    // 일반 메서드 (모든 무기에 공통적인 구현)
    public virtual void Equip()
    {
        Console.WriteLine($"{Name}을(를) 장착했습니다.");
    }

    // 추상 메서드 (각 무기마다 다른 구현)
    public abstract void Attack();
}
```

추상 클래스(Abstract Class)

Weapon 추상클래스

```
// 추상 클래스 선언
public abstract class Weapon
{
    // 일반 속성
    public string Name { get; protected set; }
    public int Damage { get; protected set; }

    // 일반 메서드 (모든 무기에 공통적인 구현)
    public virtual void Equip()
    {
        Console.WriteLine($"{Name}을(를) 장착했습니다.");
    }

    // 추상 메서드 (각 무기마다 다른 구현)
    public abstract void Attack();
}
```

Weapon 추상클래스를 상속한 Sword

```
public class Sword : Weapon
{
    public Sword(string name, int damage)
    {
        Name = name;
        Damage = damage;
    }

    // 추상 메서드 구현 - 필수!
    public override void Attack()
    {
        Console.WriteLine($"{Name}으로 {Damage}의 데미지로 베었습니다!");
    }
}
```

추상 클래스의 필요성

- 미완성 설계임을 명시
- 필수 구현 메서드를 abstract로 강제
- 공통 기능 구현 제공 가능
- 확장 가능한 템플릿 제공

```
public abstract class Monster
{
    public int Health { get; protected set; }

    public Monster()
    {
        Health = 100;
        Initialize(); // 템플릿 메서드 패턴
    }

    // 자식 클래스에서 반드시 구현해야 함
    protected abstract void Initialize();

    // 공통 구현 제공
    public virtual void TakeDamage(int amount)
    {
        Health -= amount;
        if (Health <= 0) Die();
    }

    // 필수 구현 메서드
    public abstract void SpecialMove();

    // 후크 메서드 - 기본 구현 있음
    protected virtual void Die()
    {
        Console.WriteLine("몬스터가 쓰러졌습니다");
    }
}
```

Virtual 메서드 vs Absctract 메서드 ?

가상(Virtual) 메서드

```
public class Character
{
  // 가상 메서드 - 기본 구현 제공
  public virtual void Attack()
  {
    // 기본 공격 구현
    Console.WriteLine("기본 공격!");
  }
}
```

특징:

- ✓ 기본 구현 제공 (메서드 본문이 있음)
- ⟳ 자식 클래스에서 재정의(Override) 선택적
- ✓ 일반 클래스와 추상 클래스 모두에서 사용 가능
- ✓ 자식 클래스에서 base.Method()로 부모 구현 호출 가능

> **사용 시나리오:** 대부분의 자식 클래스가 공통된 동작을 하면서, 일부 자식만 다른 동작을 할 때

추상(Abstract) 메서드

```
public abstract class Weapon
{
  // 추상 메서드 - 구현부 없음
  public abstract void Attack();
  // 세미콜론으로 끝남, 구현부 없음
}
```

특징:

- ✗ 구현부 없음 (메서드 시그니처만 선언)
- ✓ 자식 클래스에서 반드시 구현해야 함 (강제성)
- ✗ 추상 클래스 내에서만 선언 가능
- ✗ 부모 구현이 없으므로 base.Method() 호출 불가

> **사용 시나리오:** 동작의 방식이 자식 클래스마다 완전히 다를 때, 인터페이스를 강제할 때

🎮 게임 시나리오:
가상 메서드: 모든 캐릭터는 기본적으로 주먹으로 공격할 수 있지만, 전사는 칼로, 마법사는 마법으로 공격
추상 메서드: 모든 무기는 공격 방식이 완전히 다름! 검은 베고, 도끼는 내려찍고, 활은 화살을 쏨

다양한 파생 클래스의 일관된 인터페이스 필요성

- **(예시)다양한 몬스터 구현의 문제**
  - **게임에서 다양한 몬스터 타입 구현 필요**
    - 근접 공격 몬스터 (좀비, 오크 등)
    - 원거리 공격 몬스터 (해골 궁수, 마법사 등)
    - 특수 능력 몬스터 (보스, 엘리트 등)

->

각 몬스터마다 다른 동작 방식

```
public class Zombie : Monster
{
    public void MeleeAttack(Character target) { ... }
    public void Bite(Character target) { ... }
}

public class SkeletonArcher : Monster
{
    public void RangedAttack(Character target) { ... }
    public void Dodge() { ... }
}
```

다양한 파생 클래스의 일관된 인터페이스 필요성

```
if (monster is Zombie zombie)
    zombie.MeleeAttack(player);
else if (monster is SkeletonArcher archer)
    archer.RangedAttack(player);
else if (monster is Boss boss)
    boss.SpecialAttack(player);
```

일관성 없이 설계된 클래스를 사용하는 코드는 이렇게 복잡한 조건문이 필요해짐

"is-a" 관계의 확장된 의미

전통적인 "is-a" 관계

- 상속을 통한 "is-a" 관계 표현
  - "자식 클래스는 부모 클래스의 일종이다"
  - 분류학적 관계를 코드로 표현

```
// 고양이는 동물이다
public class Cat : Animal { }

// 원은 도형이다
public class Circle : Shape { }
```

리스코프 치환 원칙(LSP)과의 연관성

- 부모 클래스 타입의 객체는 자식 클래스 객체로 대체 가능해야 함
- 자식 클래스는 부모 클래스의 모든 행동을 유지해야 함

확장된 "is-a" 관계의 핵심 질문

1. **행동 일관성**: 자식이 부모의 행동 계약을 지키는가?
2. **대체 가능성**: 부모 타입 대신 자식 타입을 사용할 수 있는가?
3. **관계의 안정성**: 이 관계가 시간이 지나도 유효한가?
4. **캡슐화 보존**: 상속이 부모 클래스의 캡슐화를 해치지 않는가?

END.