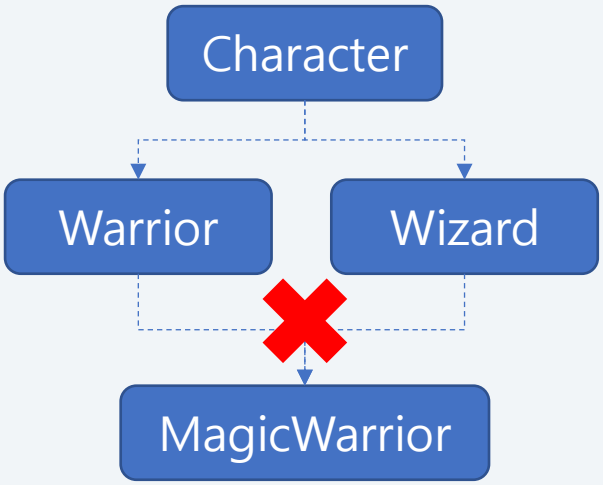
다형성과 인터페이스

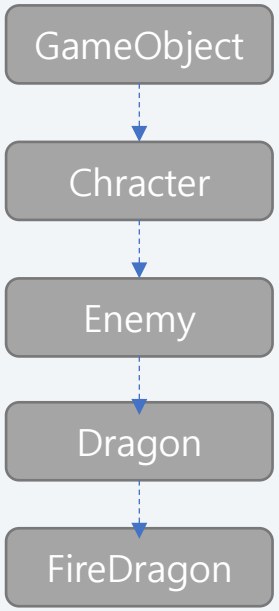
인터페이스의 개념

- **객체지향 프로그래밍의 핵심 개념**
- 클래스가 구현해야 하는 메서드와 프로퍼티의 명세 (specification)
- **코드의 유연성과 확장성을 제공하는 도구**
- '계약'과 같은 역할
  - "무엇을 해야 하는지" 정의
  - "어떻게 구현하는지"는 각 클래스에 맡김

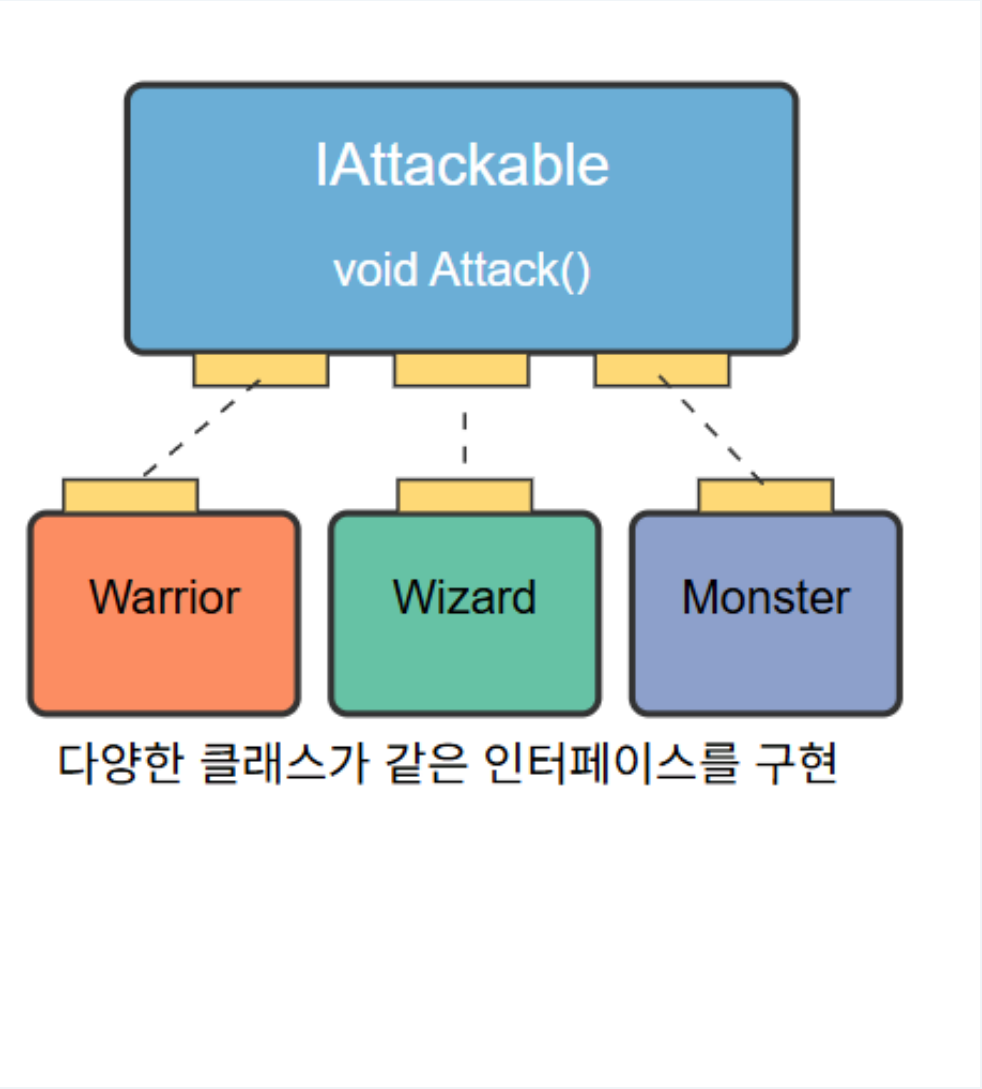
다형성 심화: 상속의 한계

- **단일 상속의 제약**
  - 한 클래스는 하나의 클래스만 직접 상속 가능
  - 복합적 기능 조합이 어려움
- **깊은 상속 계층의 문제**
  - 코드 추적 및 유지보수 어려움
  - 상위 클래스 변경 시 모든 하위 클래스에 영향
- **"is-a" 관계가 아닌 경우**
  - 모든 관계가 "A는 B이다"로 표현되지 않음
  - "할 수 있는 능력"에 초점이 필요한 경우





인터페이스의 핵심 특징


IAttackable
void Attack()
Warrior
Wizard
Monster
다양한 클래스가 같은 인터페이스를 구현

인터페이스의 핵심 특징

구현 없는 명세

- 메서드와 프로퍼티의 시그니처만 정의
- 실제 구현은 클래스에서 담당

다중 구현 가능

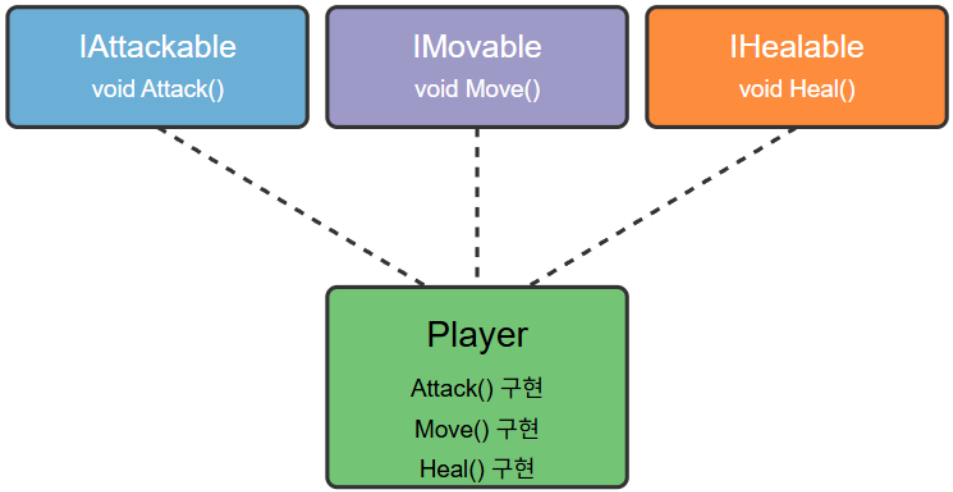
- 한 클래스가 여러 인터페이스 동시 구현 가능
- 상속의 단일 제약 보완

"can-do" 관계 표현

- 상속: "A는 B이다(is-a)" 관계
- 인터페이스: "A는 B를 할 수 있다(can-do)" 관계

계약으로서의 역할

- 인터페이스 구현 시 모든 멤버 구현 의무
- 클래스 간 일관된 동작 보장



하나의 클래스가 여러 인터페이스 구현

C#에서의 인터페이스 문법

인터페이스 선언 및 구현

```
public interface IDamageable
{
    void TakeDamage(int amount);
    bool IsDestroyed { get; }
}

public class Player : Character, IDamageable
{
    private int health = 100;

    public bool IsDestroyed => health <= 0;

    public void TakeDamage(int amount)
    {
        health -= amount;
        if (health < 0) health = 0;
    }
}
```

인터페이스와 다형성의 결합

클래스 계층과 무관한 다형성

- 서로 다른 클래스 계층의 객체들이 동일한 방식으로 상호작용
- 인터페이스를 통한 참조로 다양한 객체 조작 가능

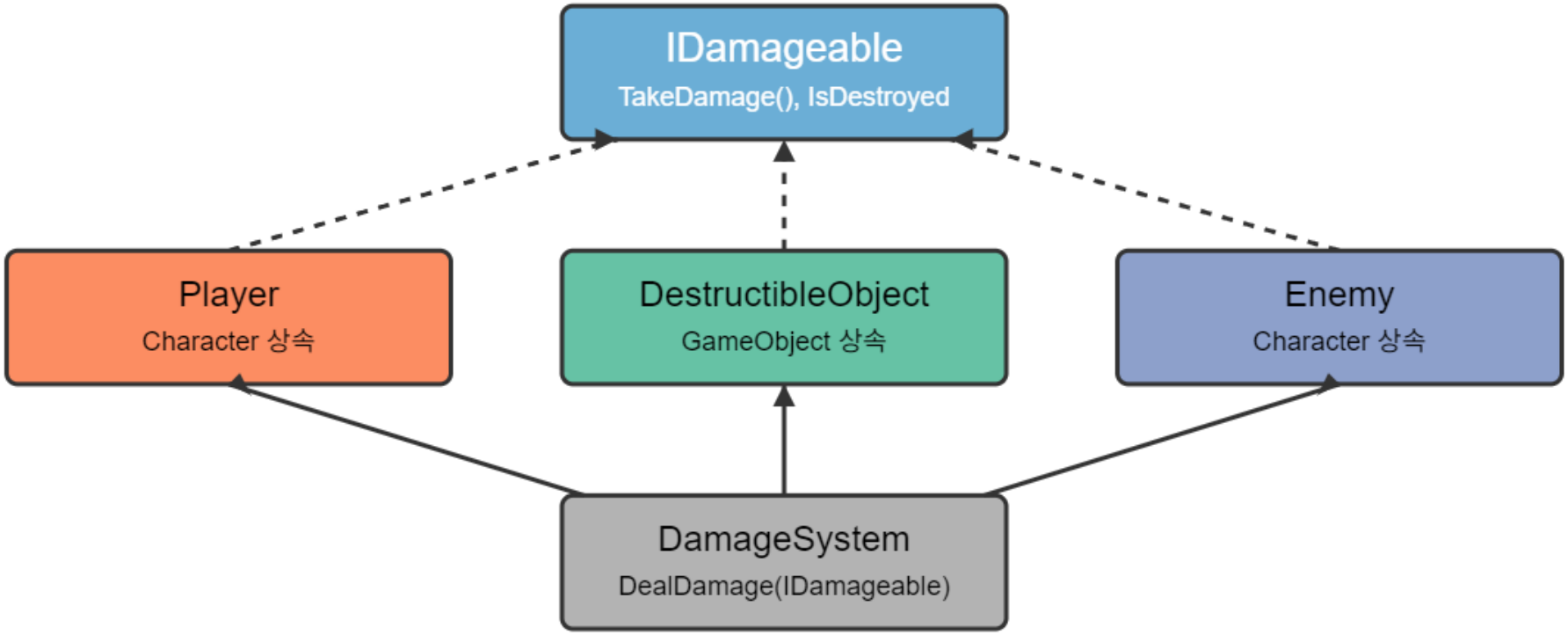
"is-a"와 "can-do" 관계의 조합

- 상속으로 정체성 표현
- 인터페이스로 능력 표현

```
// 모든 공격 가능한 대상을 일관되게 처리
void DealDamage(IDamageable target, int amount)
{
    target.TakeDamage(amount);

    if(target.IsDestroyed)
        HandleDestruction(target);
}
```

인터페이스와 다형성의 결합



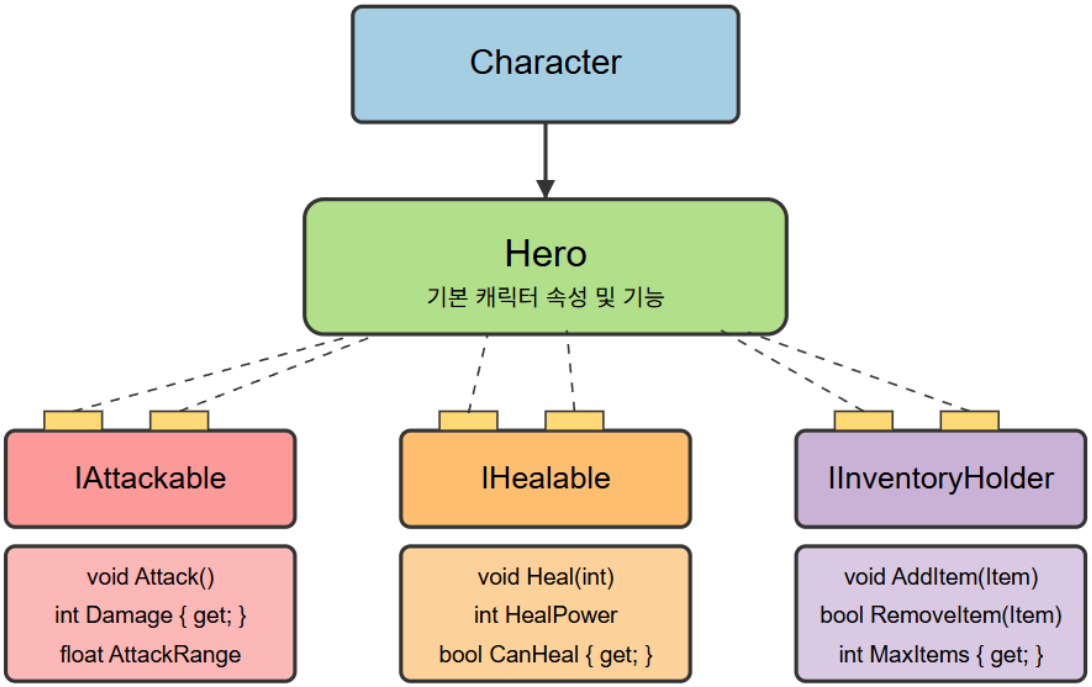
인터페이스를 통한 다형성

서로 다른 클래스 계층의 객체들을 동일한 인터페이스로 처리

레고 블록처럼 조합하는 인터페이스

다중 인터페이스 구현

- 한 클래스가 여러 인터페이스를 동시에 구현
- 객체에 다양한 "능력" 부여 가능



레고 블록처럼 조합하는 인터페이스

다양한 "능력 모듈"을 하나의 클래스에 조립

레고 블록처럼 조합하는 인터페이스

인터페이스 계층 설계

- 작고 집중된 인터페이스 설계가 효과적
  - 단일 책임 원칙(SRP) 적용
  - 인터페이스 분리 원칙(ISP) 적용

```
public class Hero : Character, IAttackable,
                    IHealable, IInventoryHolder
{
    // 공격 기능 구현
    public void Attack(IDamageable target) { ... }

    // 치유 기능 구현
    public void Heal(int amount) { ... }

    // 인벤토리 기능 구현
    public void AddItem(Item item) { ... }
    public bool RemoveItem(Item item) { ... }
}
```

인터페이스 설계 원칙

인터페이스 분리 원칙(ISP)

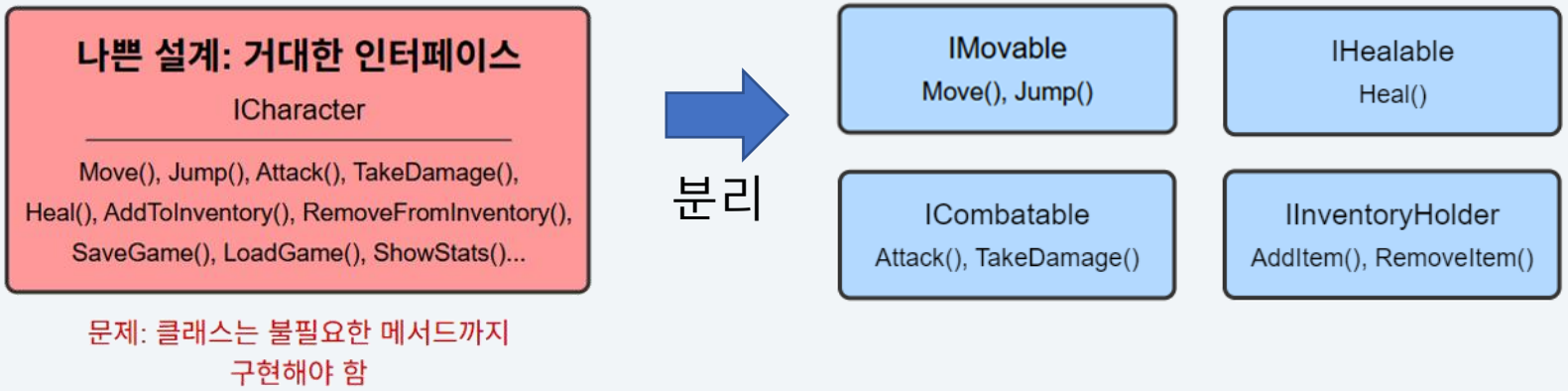
- "클라이언트는 자신이 사용하지 않는 메서드에 의존해서는 안 된다"
- 하나의 큰 인터페이스보다 여러 개의 작은 인터페이스로 분리

인터페이스 특성

- 명확한 목적과 책임을 가진 인터페이스 설계
- 간결한 이름과 직관적인 메서드 시그니처 사용

인터페이스 계층 구조

- 상위 인터페이스와 하위 인터페이스의 계층 설계
- 유사한 작업을 수행하는 인터페이스들의 공통 기능 추출



인터페이스 vs 추상 클래스

인터페이스

```
public interface IWeapon
{
  void Attack();
  int Damage { get; }
}
```

특징:

- 구현부 없는 메서드/프로퍼티만
- 다중 구현 가능
- "can-do" 관계 ("할 수 있다")
- 상태(필드) 포함 불가
- 생성자 불가

추상 클래스

```
public abstract class Weapon
{
  protected int damage;
  public abstract void Attack();
  public virtual int GetDamage()
  { return damage; }
}
```

특징:

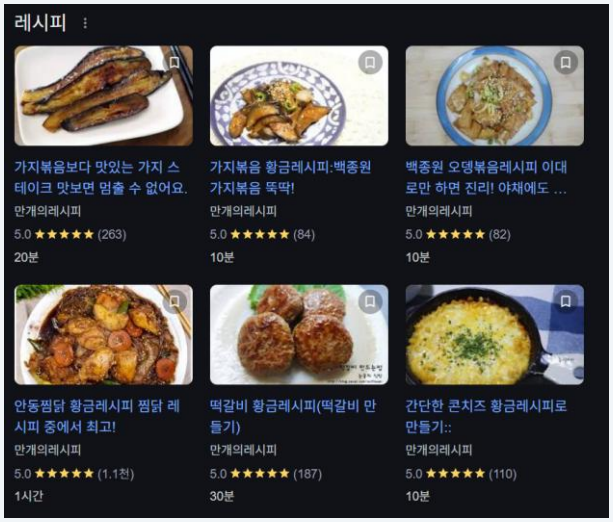
- 추상 메서드와 구현된 메서드 혼합
- 단일 상속만 가능
- "is-a" 관계 ("~이다")
- 상태(필드)와 생성자 포함 가능

인터페이스 vs 추상 클래스

선택 기준

- 여러 클래스에 공통 기능 구현이 필요하면 → 추상 클래스
- 관련 없는 클래스들에 공통 동작이 필요하면 → 인터페이스

반조리 식품과 같은 추상 클래스



모든 재료와 조리과정은 클래스 스스로 준비해야하는 레시피 같은 인터페이스

인터페이스 총정리

인터페이스의 핵심 개념

- "무엇을 할 수 있는지" 정의하는 계약
- 다형성을 통한 유연한 설계 가능
- 클래스 계층과 독립적인 기능 구현

인터페이스 설계 원칙

- 작고 집중된 인터페이스 (ISP)
- 명확한 목적과 책임을 가진 인터페이스
- 인터페이스 기반 상호작용으로 느슨한 결합 달성

END.