# 6. 상속과 다형성

## `extends` 키워드

### ✅ 1. `extends`란?

> 자바에서 **클래스 간의 상속(Inheritance)** 관계를 정의할 때 사용하는 키워드.

- **자식 클래스가 부모 클래스의 속성과 메서드를 상속받을 수 있도록 함**
- 클래스 정의 시 사용:

```
class 자식클래스 extends 부모클래스
```

---

### ✅ 2. 기본 문법

```
class Animal {
    void eat() {
        System.out.println("먹는다");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("멍멍 짖는다");
    }
}
```

#### 사용 예:

```
Dog d = new Dog();
d.eat();  // 부모의 메서드 호출 가능
d.bark(); // 자식의 메서드
```

---

### ✅ 3. 상속의 효과

- 부모 클래스의 **멤버 변수**와 **메서드**를 **자식 클래스가 그대로 사용 가능**
- 코드의 **재사용성 증가**
- **계층 구조 설계**에 적합

---

## ✅ 4. 단일 상속 (Java는 다중 상속 불가)

Java는 **클래스에 대해 단일 상속만 허용**한다.

```
class A {}
class B {}

class C extends A, B {}  // ❌ 오류! 다중 상속 불가
```

> 대신 다중 상속이 필요한 경우 **인터페이스**를 사용한다.

---

## ✅ 5. 생성자와 상속 (`super()` 호출)

- 자식 클래스가 생성될 때 **부모 클래스의 생성자가 먼저 호출됨**
- 자식 생성자에서 `super()`를 명시적으로 호출하거나,
  명시하지 않으면 **자동으로 기본 생성자(super())가 호출**됨

```
class Animal {
    Animal() {
        System.out.println("Animal 생성자");
    }
}

class Dog extends Animal {
    Dog() {
        super();  // 생략 가능
        System.out.println("Dog 생성자");
    }
}
```

---

## ✅ 6. 메서드 오버라이딩과 extends

- 자식 클래스는 부모의 메서드를 **오버라이딩(재정의)** 할 수 있음
- 런타임 시점에는 **실제 객체의 메서드가 호출됨 (다형성)**

```
class Animal {
    void speak() {
        System.out.println("...");
    }
}

class Dog extends Animal {
    @Override
    void speak() {
        System.out.println("멍멍");
    }
}
```

---

## ✅ 7. 접근 제어자의 영향

| 접근자 | 자식 클래스에서 접근 가능 여부 |
|---|---|
| `public` | O |
| `protected` | O |
| `default` (패키지 동일 시) | O |
| `private` | ❌ (직접 접근 불가) |

> `private` 필드는 직접 상속받지 않지만, **getter/setter를 통해 간접 사용** 가능

---

## ✅ 8. 상속 구조 예시

```
class Person {
    String name;
    void introduce() {
        System.out.println("이름은 " + name);
    }
}

class Student extends Person {
    int grade;
    void study() {
        System.out.println("공부 중");
    }
}
```

```
Student s = new Student();
s.name = "홍길동";    // 부모 필드 사용
s.introduce();       // 부모 메서드 사용
s.study();           // 자식 메서드 사용
```

---

## ✅ 9. `extends`와 관련된 키워드 요약

| 키워드 | 설명 |
|---|---|
| `extends` | 클래스 상속 시 사용 |
| `super` | 부모 클래스의 생성자, 필드, 메서드에 접근 |
| `@Override` | 오버라이딩 표시 |
| `instanceof` | 상속 구조에서 객체 타입 비교 |

---

## ✅ 10. 핵심 요약

| 항목 | 내용 |
|---|---|
| 기능 | 자식 클래스가 부모의 속성과 기능을 상속 |
| 이점 | 코드 재사용, 계층적 설계 |
| 제약 | 단일 상속만 허용 (인터페이스는 다중 구현 가능) |
| 관련 개념 | super, 오버라이딩, 다형성 |

# 메서드 오버라이딩

## ✅ 1. 오버라이딩이란?

> **부모 클래스에서 정의한 메서드를**
> **자식 클래스에서 재정의하여 사용하는 것**

- 메서드 이름, 매개변수, 반환형이 **동일해야 함**
- 자식 클래스에서 **기능을 수정하거나 확장**하고자 할 때 사용

---

## ✅ 2. 문법 구조

```
class Parent {
    void greet() {
        System.out.println("안녕하세요");
    }
}

class Child extends Parent {
    @Override
    void greet() {
        System.out.println("안녕!");
    }
}
```

```
Parent p = new Child();
p.greet();  // 출력: 안녕!
```

> 오버라이딩된 메서드가 **실제 객체 타입 기준으로 실행됨** (다형성의 핵심)

---

## ✅ 3. 오버라이딩 조건

| 조건 | 설명 |
|---|---|
| ✅ 메서드 이름 동일 | `greet()` 등 |

| 조건 | 설명 |
| --- | --- |
| ✅ 매개변수 동일 | `(int x)` vs `(int x)` |
| ✅ 반환형 동일 | `void`, `int`, `String`, 등 |
| ✅ 접근 제어자는 부모보다 넓거나 동일 | `protected → public` 가능, `public → private` ❌ |
| ✅ 예외는 부모보다 좁거나 없음 | `throws` 범위 줄이기만 가능 |

---

## ✅ 4. `@Override` 어노테이션

> **오버라이딩임을 명확하게 컴파일러에게 알리는 표시**

```
@Override
public void greet() {
    System.out.println("Hello");
}
```

### 이점

- 오타/오버라이딩 조건 미충족 시 **컴파일 에러 발생**
- 의도된 오버라이딩인지 명확하게 표시 가능

---

## ✅ 5. 오버라이딩 vs 오버로딩

| 항목 | 오버라이딩 | 오버로딩 |
| --- | --- | --- |
| 정의 | 부모의 메서드를 자식에서 재정의 | 같은 클래스에서 같은 이름의 메서드 여러 개 정의 |
| 조건 | 시그니처 완전 동일 | 매개변수 다르면 가능 |
| 목적 | 다형성, 확장 | 편리한 호출 |
| 키워드 | `@Override` | 사용 안 함 |

---

## ✅ 6. 다형성과의 관계

> 오버라이딩은 **동적 바인딩(dynamic binding)**을 가능하게 하여
> **하나의 타입으로 다양한 실행 결과**를 만들어낸다.

```
class Animal {
    void sound() {
        System.out.println("...");
    }
}

class Dog extends Animal {
```

```
    @Override
    void sound() {
        System.out.println("멍멍");
    }
}

class Cat extends Animal {
    @Override
    void sound() {
        System.out.println("야옹");
    }
}
```

```
Animal a = new Dog(); a.sound();  // 멍멍
a = new Cat();        a.sound();  // 야옹
```

---

## ✅ 7. `super` 키워드로 부모 메서드 호출

> 오버라이딩한 메서드 안에서 부모의 메서드를 호출하고 싶을 때

```
class Bird extends Animal {
    @Override
    void sound() {
        super.sound();  // Animal의 sound() 호출
        System.out.println("짹짹");
    }
}
```

---

## ✅ 8. 오버라이딩 주의사항

- `private` 메서드는 상속되지 않음 → 오버라이딩 불가
- `static` 메서드는 클래스 소속 → 오버라이딩이 아니라 **숨김(hiding)**
- 생성자는 오버라이딩할 수 없음 (오버로딩만 가능)

---

## ✅ 9. 예제 문제

```
class Vehicle {
    void move() {
        System.out.println("움직인다");
    }
}

class Airplane extends Vehicle {
    @Override
    void move() {
        System.out.println("하늘을 난다");
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Vehicle v = new Airplane();
        v.move();  // 어떤 출력이 나올까?
    }
}
```

> 정답: `하늘을 난다`
> 이유: `v`는 `Vehicle` 타입이지만 **객체는** `Airplane`,
> 즉 **오버라이딩된 메서드가 실행됨 (동적 바인딩)**

---

## ✅ 10. 요약 정리

| 항목 | 설명 |
|---|---|
| 개념 | 부모 메서드를 자식 클래스에서 재정의 |
| 핵심 조건 | 메서드 이름, 매개변수, 반환형 일치 |
| 사용 목적 | 다형성 구현, 기능 수정/확장 |
| 어노테이션 | `@Override` 권장 |
| 관련 개념 | 동적 바인딩, super 호출, 접근 제어 |

# 타입 캐스팅 (업캐스팅, 다운캐스팅)

## ✅ 1. 타입 캐스팅이란?

> 한 객체를 **다른 타입으로 변환해서 사용하는 것**
> → 주로 **상속 관계에서 부모-자식 타입 간의 변환**이 일어남

### 🔸 목적

- 다형성을 활용한 일반화
- 특정 기능을 다시 사용하기 위한 구체화

---

## ✅ 2. 업캐스팅 (Upcasting)

> 자식 클래스 객체를 **부모 클래스 타입으로 참조**하는 것
> → **자동 형변환(암시적 변환)**

### 🔹 문법

```
부모타입 변수 = new 자식타입();
```

### 🔹 예제

```
class Animal {
    void speak() {
        System.out.println("동물이 말함");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("멍멍");
    }
}

Animal a = new Dog();  // 업캐스팅
a.speak();             // ✅ 사용 가능 (부모 메서드)
```

### ✅ 특징

- **자동으로 변환됨**
- **부모 타입의 멤버만 사용 가능**
- **다형성 구현**에 필수적

---

## ✅ 3. 다운캐스팅 (Downcasting)

> 부모 타입의 객체를 **자식 타입으로 변환**하는 것
> → **명시적 형변환이 필요함**

### 🔹 문법

```
자식타입 변수 = (자식타입) 부모변수;
```

### 🔹 예제

```
Animal a = new Dog();      // 업캐스팅
Dog d = (Dog) a;           // 다운캐스팅
d.bark();                  // ✅ 자식 메서드 사용 가능
```

> 객체는 실제로 `Dog`이기 때문에 문제 없음

---

## ✅ 4. 잘못된 다운캐스팅 → `ClassCastException`

```
Animal a = new Animal();  // 실제 Animal 객체
Dog d = (Dog) a;          // ❌ 런타임 오류
```

### 결과

```
Exception in thread "main" java.lang.ClassCastException: Animal cannot be cast to Dog
```

### 해결 방법: `instanceof` 사용

```
if (a instanceof Dog) {
    Dog d = (Dog) a;
    d.bark();
}
```

---

## ✅ 5. 업/다운캐스팅 비교

| 항목 | 업캐스팅 (Upcasting) | 다운캐스팅 (Downcasting) |
|---|---|---|
| 변환 방향 | 자식 → 부모 | 부모 → 자식 |
| 문법 | 자동 | 명시적 `(타입)` 필요 |
| 가능 여부 | 항상 가능 | **객체가 진짜 자식일 때만 가능** |
| 사용 목적 | 일반화, 다형성 활용 | 자식 기능 접근 |
| 예외 발생 | ❌ 없음 | ✅ `ClassCastException` 가능성 |

---

## ✅ 6. 실전 응용 예제

```
class Animal {
    void speak() {
        System.out.println("...");
    }
}

class Cat extends Animal {
    void meow() {
        System.out.println("야옹");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Cat();  // 업캐스팅
        a.speak();             // Animal의 메서드 사용
```

```
17
18          if (a instanceof Cat) {
19              Cat c = (Cat) a;    // 다운캐스팅
20              c.meow();           // Cat 고유 메서드 사용
21          }
22      }
23  }
```

## ✅ 7. 업/다운캐스팅을 활용한 다형성 패턴

```
1   Animal[] animals = new Animal[3];
2   animals[0] = new Dog();
3   animals[1] = new Cat();
4   animals[2] = new Dog();
5
6   for (Animal a : animals) {
7       a.speak();
8
9       if (a instanceof Dog) {
10          ((Dog) a).bark();
11      }
12  }
```

> 배열에는 `Animal` 타입으로 넣지만, 실제 타입에 따라 `다운캐스팅`해서 확장 기능 사용

## ✅ 요약 정리

| 개념 | 설명 | 문법 |
|---|---|---|
| 업캐스팅 | 자식 → 부모, 자동 변환 | `Animal a = new Dog();` |
| 다운캐스팅 | 부모 → 자식, 명시적 변환 | `Dog d = (Dog) a;` |
| 예외 방지 | 타입 검사 필요 | `instanceof` 활용 |

# `instanceof` 연산자

## ✅ 1. `instanceof`란?

> 객체가 **특정 클래스 또는 그 자식 클래스/인터페이스의 인스턴스인지**를 검사하는 연산자

- `true` **또는** `false`**를 반환**하는 **논리 연산자**
- 주로 **다운캐스팅 전에 타입을 확인**할 때 사용

## ✅ 2. 기본 문법

```
객체 instanceof 타입
```

**예:**

```
if (obj instanceof String) {
    // obj는 String 타입이거나 String을 상속한 타입
}
```

---

## ✅ 3. 예제: 안전한 다운캐스팅

```
class Animal {}
class Dog extends Animal {
    void bark() { System.out.println("멍멍"); }
}

Animal a = new Dog();  // 업캐스팅

if (a instanceof Dog) {
    Dog d = (Dog) a;  // 다운캐스팅 전 검사
    d.bark();         // 안전하게 실행
}
```

> `instanceof`를 사용하지 않으면 `ClassCastException`이 발생할 수도 있음

---

## ✅ 4. 상속 구조에서 동작

```
class A {}
class B extends A {}
class C extends B {}

C c = new C();
System.out.println(c instanceof C); // true
System.out.println(c instanceof B); // true
System.out.println(c instanceof A); // true
System.out.println(c instanceof Object); // true
```

> 자바에서는 모든 클래스가 **Object의 자식**이므로 최상위까지 `true` 반환됨

---

## ✅ 5. null과 instanceof

```
String s = null;
System.out.println(s instanceof String);  // false
```

```
null은 어떤 타입의 인스턴스도 아니므로 항상 false
```

## ✅ 6. instanceof + 패턴 매칭 (Java 16+)

> **형변환까지 자동으로 처리**하는 문법

```java
Object obj = "hello";

if (obj instanceof String s) {
    System.out.println(s.toUpperCase());  // s는 이미 String으로 형변환됨
}
```

> Java 16 이상에서는 이 **패턴 매칭 문법**을 적극 활용 가능

## ✅ 7. instanceof와 equals 오버라이드 주의

클래스 간 비교 시 `equals()` 메서드를 오버라이드할 경우,
`instanceof` 검사로 **타입 안전성** 확보 가능

```java
@Override
public boolean equals(Object obj) {
    if (!(obj instanceof MyClass)) return false;
    MyClass other = (MyClass) obj;
    // ...
}
```

## ✅ 8. instanceof 사용 주의점

| 주의사항 | 설명 |
|---|---|
| `instanceof`는 타입에 의존적 | 너무 많이 사용하면 다형성을 약화시킴 |
| 다운캐스팅 없이 동작 가능 | `instanceof`로 검사만 하고 형변환 안 하면 의미 없음 |
| 인터페이스에도 사용 가능 | 객체가 인터페이스를 구현했는지 확인 가능 |
| null 체크 포함 안 됨 | null은 항상 false 반환 |

## ✅ 9. instanceof vs getClass()

| 항목 | instanceof | getClass() |
|---|---|---|
| 상속 구조 검사 | O (부모 포함) | ❌ (정확히 일치해야 함) |
| null 안전성 | O (null이면 false) | ❌ (nullPointerException 발생) |

| 항목 | instanceof | getClass() |
| --- | --- | --- |
| 용도 | 다형성 판단 | 정확한 클래스 타입 비교 |

```
Object obj = new String("hi");
System.out.println(obj instanceof String);       // true
System.out.println(obj.getClass() == String.class); // true
```

---

## ✅ 10. 요약 정리

| 항목 | 설명 |
| --- | --- |
| 역할 | 객체가 특정 타입인지 검사 |
| 반환값 | boolean (`true` or `false`) |
| 사용 목적 | 안전한 다운캐스팅, 타입 비교, equals 구현 |
| 패턴 매칭 | Java 16+부터 형변환까지 자동 처리 |
| null 검사 | `null instanceof` → 항상 false |

---

## ✅ 결론

- `instanceof`는 **다운캐스팅의 안전 장치**
- **다형성 기반 로직에서 필수**
- Java 16 이상이면 **패턴 매칭 문법**을 적극 활용하자

# `final` 키워드

## ✅ 1. `final`이란?

> Java에서 `final`은 **"한 번 정해지면 더는 바꿀 수 없다"**는 의미를 가진 제한자(modifier)

- 변수: **값 재할당 금지**
- 메서드: **오버라이딩 금지**
- 클래스: **상속 금지**

---

## ✅ 2. `final` 변수

### 🔹 2-1. 지역 변수

```
final int x = 10;
x = 20; // ❌ 컴파일 에러
```

> 한 번 초기화되면 값 변경 불가

### 🔹 2-2. 멤버 변수 (필드)

```
class Circle {
    final double PI = 3.14;  // 상수처럼 사용
}
```

> 클래스 내에서 상수로 사용하는 경우, 보통 `static final`로 선언

```
public static final int MAX_VALUE = 100; // 관례: 대문자
```

---

### 🔹 2-3. 참조형 변수

```
final StringBuilder sb = new StringBuilder("Hello");
sb.append(" World"); // ✅ 내부 변경은 가능
sb = new StringBuilder(); // ❌ 재할당은 불가능
```

> 참조형에서 **주소값 변경은 금지**, **내부 내용은 변경 가능**

---

## ✅ 3. `final` 메서드

> 해당 메서드는 **하위 클래스에서 오버라이딩할 수 없음**

```
class Animal {
    final void sleep() {
        System.out.println("자는 중...");
    }
}

class Dog extends Animal {
    // void sleep() ❌ 오버라이딩 불가
}
```

> 주로 **핵심 동작 보호** 또는 **보안상 확장 금지** 목적으로 사용

---

## ✅ 4. `final` 클래스

> 해당 클래스를 **상속받을 수 없음**

```
final class Constants {
    // 상수 모음
}

class MyConstants extends Constants { // ❌ 컴파일 에러
}
```

> 대표적인 예: `java.lang.String`, `java.lang.Math`, `java.lang.System`

```
public final class String {
    // 내부 구현...
}
```

---

## ✅ 5. `final` 매개변수

> 메서드 인자 값을 **메서드 내부에서 변경 못하게 막음**

```
void print(final int x) {
    x = 5;  // ❌ 컴파일 에러
}
```

> **람다식이나 익명 내부 클래스에서도 필수로 `final` 또는 effectively final**이어야 함

---

## ✅ 6. `final` + `static` 조합

| 조합 | 용도 |
| --- | --- |
| `static final` | 상수 선언 (`public static final int MAX = 100;`) |
| `final static` | 순서만 바뀐 것, 동일한 의미 |

> 상수 정의 시 `static final`과 대문자+밑줄이 관례

---

## ✅ 7. `final` vs `const`

> Java에는 C/C++의 `const`가 없고, 대신 `final`이 존재함
> (`const` 키워드는 예약어지만 사용되지 않음)

---

## ✅ 8. `final` vs `abstract`

| 항목 | `final` | `abstract` |
| --- | --- | --- |
| 의미 | 확장을 막는다 | 구현을 강제한다 |
| 클래스 | 상속 금지 | 반드시 상속 필요 |
| 메서드 | 오버라이딩 금지 | 반드시 오버라이딩 필요 |
| 목적 | 불변/보호 | 설계 확장성 유도 |

---

## ✅ 9. 주요 사용 예

- 상수 정의: `public static final int PORT = 8080;`
- 상속 금지 클래스 설계: `final class SecurityManager`
- 설계 보호: `final void criticalMethod()`
- 람다 표현식 사용: `Runnable r = () -> System.out.println(x); // x는 final`

---

## ✅ 10. 요약 정리

| 위치 | 역할 |
| --- | --- |
| 변수 | 값 재할당 금지 |
| 참조형 변수 | 참조 주소 변경 금지 (내용 변경은 가능) |
| 메서드 | 자식 클래스에서 오버라이딩 금지 |
| 클래스 | 상속 금지 |
| 매개변수 | 메서드 내부에서 값 변경 불가 |
| 조합 사용 | `static final` → 상수 |