# 객체지향 프로그래밍

### 목차



---

## 1. 캡슐화

▼ 1-1. 개념 및 목적

- 개념
  - 객체지향 프로그래밍에서 객체의 속성(필드)을 외부로부터 숨기고, 공개된 메서드(getter/setter)를 통해서만 접근하도록 만드는 원칙
  - 필드를 private으로 선언하고, 외부에서 직접 접근하지 못하게 제한하고, public 메서드인 getter와 setter를 제공해 값을 읽거나 수정할 수 있도록 한다. setter 내부에는 유효성 검사 로직을 넣어 잘못된 값이 들어오는 것을 막을 수도 있다.
- 목적
  1. 데이터 보호: 잘못된 값이 직접 들어가는 것을 막고, setter 내부에서 규칙을 강제함으로써 객체의 상태를 안정적으로 유지
  2. 정보 은닉: 내부 구현이 어떻게 되어 있는지는 숨겨 두고, 외부에는 단순한 사용 방법만 제공함으로써 객체 사용자가 불필요한 복잡성을 신경 쓰지 않도록 한다.
  3. 유지보수와 확장성: 내부 로직이 바뀌더라도 외부 인터페이스(getter/setter)가 같으면 사용하는 코드는 수정할 필요가 없으므로 프로그램 전체의 안정성이 높아지고 유지보수가 쉬워진다.

▼ 1-2. 샘플 코드

```
// 캡슐화(Encapsulation) 예제
class Stock {
    // 1. 필드는 외부에서 직접 접근 불가능 (private)
    private String name;
    private double price;

    // 생성자
    public Stock(String name, double price) {
        this.name = name;
        setPrice(price); // setter 사용 → 유효성 검사 포함
    }

    // 2. Getter (읽기 전용)
    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    // 3. Setter (쓰기 전용, 유효성 검사 추가)
```

```
    public void setPrice(double price) {
        if (price > 0) {
            this.price = price;
        } else {
            System.out.println("❌ 잘못된 가격: " + price);
        }
    }
}

public class EncapsulationExample {
    public static void main(String[] args) {
        // 정상적인 객체 생성
        Stock s1 = new Stock("스칼라 AI", 17000);
        System.out.println(s1.getName() + " 현재가: " + s1.getPrice());

        // setter를 통한 가격 변경 (올바른 값)
        s1.setPrice(18000);
        System.out.println("업데이트 후 가격: " + s1.getPrice());

        // setter를 통한 잘못된 값 입력 (음수)
        s1.setPrice(-5000); // → 유효성 검사에서 거부
        System.out.println("최종 가격: " + s1.getPrice());
    }
}
```

▼ 1-2-1. 코드 설명 - class Stock

```
class Stock {
    // 1. 필드는 외부에서 직접 접근 불가능 (private)
    private String name;
    private double price;

    // 생성자
    public Stock(String name, double price) {
        this.name = name;
        setPrice(price); // setter 사용 → 유효성 검사 포함
    }

    // 2. Getter (읽기 전용)
    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    // 3. Setter (쓰기 전용, 유효성 검사 추가)
    public void setPrice(double price) {
        if (price > 0) {
            this.price = price;
        } else {
            System.out.println("❌ 잘못된 가격: " + price);
        }
```

```
    }
}
```

- private String name;
  - 주식의 이름(예: "스칼라 AI")
    - private이기 때문에 클래스 외부에서는 s1.name처럼 직접 접근 불가
- private double price;
  - 주식의 가격을 저장하는 변수
  - private이기 때문에 클래스 외부에서는 직접 접근 불가
- public Stock
  - this.name = name;
    - 생성 시 입력된 이름을 객체의 name 에 저장
  - setPrice(price);
    - 가격은 바로 대입하지 않고 setPrice() 메서드를 통해 저장
- Getter
  - getName(): 주식 이름
  - getPrice(): 주식 가격
- Setter
  - if (price > 0): 유효성 검사
    - 올바른 가격(0보다 큰 수)이면 저장, 잘못된 값이면 거부하고 메시지를 출력하기.

▼ 1-2-2. 코드 설명 - public class EncapsulationExample

```
public class EncapsulationExample {
    public static void main(String[] args) {
        // 정상적인 객체 생성
        Stock s1 = new Stock("스칼라 AI", 17000);
        System.out.println(s1.getName() + " 현재가: " + s1.getPrice());

        // setter를 통한 가격 변경 (올바른 값)
        s1.setPrice(18000);
        System.out.println("업데이트 후 가격: " + s1.getPrice());

        // setter를 통한 잘못된 값 입력 (음수)
        s1.setPrice(-5000); // → 유효성 검사에서 거부
        System.out.println("최종 가격: " + s1.getPrice());
    }
}
```

- public static void main(String[] args)
  - Stock 객체를 실제로 만들어서 테스트하는 클래스.
- Stock s1 = new Stock("스칼라 AI", 17000);
  - "스칼라 AI"라는 이름과 17000이라는 가격으로 객체 생성.
    - 생성자 내부에서 setPrice(17000)이 호출되므로 유효성 검사가 통과되므로 저장된다.
- System.out.println(s1.getName() + " 현재가: " + s1.getPrice());
  - getName()과 getPrice()로 값을 출력

- s1.setPrice(18000);
  - setter를 통한 가격 변경
    - setPrice(18000)은 유효성 검사를 통과하므로 price가 18000으로 업데이트된다.
- setPrice(-5000)
  - setter 내부 조건문이 거부예정.
    - "잘못된 가격: -5000" 메시지만 출력되고, price 값은 바뀌지 않고, getPrice()로 확인하면 여전히 이전 값 18000이 유지된다.

▼ 1-3. 캡슐화의 의미?

- 캡슐화의 의미
  - 중요한 데이터는 직접 노출하지 않고 private으로 은닉하며, getter/setter 같은 메서드를 통해서만 접근하도록 만들기.

▼ 1-4. this.price = price; 하지않고 setPrice(price) 한 이유?

- 이 값이 올바른지 아닌지 검사하는 로직을 넣기위해서.
- 생성자에서 this.price = price;를 바로 쓰면 잘못된 값도 그대로 들어와 버릴 수 있다. 예를 들어 new Stock("삼성", -1000) 같은 유효하지않은 객체가 생성될수있는데 setPrice(price);를 쓰면 생성되는 순간에 그 값이 유효한지 검사하고 잘못된 값은 차단할 수 있다.
- 결론
  - 생성자 안에서 직접 대입하지 않고 setter를 호출하면 내부 로직이 항상 같은 규칙을 따르게 함으로써 어디서 값을 넣든지 간에 일관성과 안전성이 유지된다.

## 2. 추상화

▼ 2-1. 개념 및 목적

- 개념
  - 객체지향 프로그래밍에서 복잡한 시스템을 단순화하기 위해 핵심적인 개념과 동작만 남기고 불필요한 세부사항을 감추는 원칙
  - 추상 클래스와 인터페이스
    - 추상 클래스: 공통된 속성과 기본 동작을 정의하면서, 일부 메서드를 추상 메서드로 남겨 자식 클래스가 반드시 구현하도록 한다.
    - 인터페이스: 특정 기능에 대한 규약을 정의하며, 이를 구현하는 클래스가 해당 메서드를 구체적으로 작성하도록 강제한다.
- 목적
  1. 복잡성 단순화: 사용자나 개발자는 내부의 복잡한 구조를 알 필요 없이, 제공되는 메서드 시그니처만 보고 객체를 사용할 수 있다.
  2. 코드의 유연성과 유지보수성 향상: 외부에서 바라보는 표면(메서드 선언)만 일정하게 유지하면 내부 구현은 자유롭게 변경하거나 최적화할 수 있다.
  3. 일관성과 확장성 확보: 추상 클래스는 공통 뼈대를 재사용하게 해주고, 인터페이스는 다양한 클래스들이 동일한 규약을 따르도록 만들어 여러 객체를 일관된 방식으로 다룰 수 있게 한다. 이를 통해 협업과 테스트가 쉬워지고, 새로운 기능 확장이 용이해진다.

▼ 2-2. 샘플 코드

```
// 추상 클래스: 공통 자산
abstract class Asset {
    protected String name;
    protected double price;
```

```
    public Asset(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // 자식 클래스들이 반드시 구현해야 하는 추상 메서드
    public abstract void printInfo();
}

// 인터페이스
interface Valuable {
    void printInfo(); // 반드시 구현해야 함

    // default 메서드 (인터페이스도 구현 제공 가능)
    default void updatePrice(double price) {
        System.out.println("가격을 " + price + "원으로 업데이트했습니다.");
    }
}

// 일반주 클래스: 추상 클래스 상속 + 인터페이스 구현
class Stock extends Asset implements Valuable {
    public Stock(String name, double price) {
        super(name, price);
    }

    @Override
    public void printInfo() {
        System.out.println("[일반주] 종목: " + name + " / 현재가: " + price + "원");
    }
}

// 우선주 클래스: 추상 클래스 상속 + 인터페이스 구현
class PreferredStock extends Asset implements Valuable {
    private double dividendRate;

    public PreferredStock(String name, double price, double dividendRate) {
        super(name, price);
        this.dividendRate = dividendRate;
    }

    @Override
    public void printInfo() {
        System.out.println("[우선주] 종목: " + name +
                           " / 현재가: " + price + "원" +
                           " / 배당률: " + dividendRate + "%");
    }
}

// 실행 클래스
public class AbstractionExample {
    public static void main(String[] args) {
        // 추상 클래스는 직접 인스턴스화 불가 → 자식 클래스를 통해 사용
        Asset samsung = new Stock("삼성전자", 72000);
        Asset lgPref = new PreferredStock("LG전자우", 83000, 4.5);
```

```
        // 다형성: 같은 printInfo() 호출이지만 실제 객체에 따라 다르게 동작
        samsung.printInfo();
        lgPref.printInfo();

        // 인터페이스 default 메서드 사용
        Valuable v = new Stock("카카오", 57000);
        v.updatePrice(58000);
    }
}
```

▼ 2-2-1. 코드 설명 - 추상 클래스 Asset

```
abstract class Asset {
    protected String name;
    protected double price;

    public Asset(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // 자식 클래스들이 반드시 구현해야 하는 추상 메서드
    public abstract void printInfo();
}
```

- abstract class Asset
  - 추상 클래스 정의
- protected String name, protected double price
  - name, price 필드
  - protected 접근제어자 사용해서 같은 패키지 내부와 자식 클래스에서만 접근 가능하게한다.
- public Asset(String name, double price) {this.name = name; this.price = price;}
  - name과 price를 초기화
- public abstract void printInfo();
  - printInfo()는 추상 메서드로 선언되어 있고 구현은 없다.
    - sset을 상속받는 자식 클래스들은 반드시 printInfo()를 구현해야한다 즉 Asset은 "공통 자산"이라는 추상적인 개념만 정의하고 구체적인 세부 내용은 자식 클래스에서 맡기는 구조.

▼ 2-2-2. 코드 설명 - 인터페이스 Valuable

```
interface Valuable {
    void printInfo(); // 반드시 구현해야 함

    // default 메서드 (인터페이스도 구현 제공 가능)
    default void updatePrice(double price) {
        System.out.println("가격을 " + price + "원으로 업데이트했습니다.");
    }
}
```

- interface Valuable
  - 객체가 가져야 할 행동 규약
- void printInfo();

    - 선언만 되어 있고 구현은 없음. 인터페이스를 구현하는 클래스는 반드시 이 메서드를 작성해야한다.
- default void updatePrice(double price) {System.out.println("가격을 " + price + "원으로 업데이트했습니다.");}
    - 기본 구현: "가격을 업데이트했습니다"라는 메시지를 출력하기.

▼ 2-2-3. 코드 설명 - Stock 클래스

```
class Stock extends Asset implements Valuable {
    public Stock(String name, double price) {
        super(name, price);
    }

    @Override
    public void printInfo() {
        System.out.println("[일반주] 종목: " + name + " / 현재가: " + price + "원");
    }
}
```

- class Stock extends Asset implements Valuable
    - 추상 클래스 Asset을 상속하고 인터페이스 Valuable을 구현한다.
- public Stock(String name, double price) {super(name, price);}
    - 생성자가 부모 클래스 Asset의 생성자를 호출해 name, price를 초기화한다.
- @Override public void printInfo()
    - printInfo() 메서드를 오버라이딩하여 일반주 종목 정보를 출력.

▼ 2-2-4. 코드 설명 - PreferredStock 클래스

```
class PreferredStock extends Asset implements Valuable {
    private double dividendRate;

    public PreferredStock(String name, double price, double dividendRate) {
        super(name, price);
        this.dividendRate = dividendRate;
    }

    @Override
    public void printInfo() {
        System.out.println("[우선주] 종목: " + name +
                           " / 현재가: " + price + "원" +
                           " / 배당률: " + dividendRate + "%");
    }
}
```

- PreferredStock extends Asset implements Valuable
    - `Asset`을 상속, `Valuable`을 구현.
- private double dividendRate;
    - 추가로 `dividendRate`(배당률)라는 필드를 가짐.
- public PreferredStock(String name, double price, double dividendRate) {super(name, price); this.dividendRate = dividendRate; }
    - 생성자를 통해 `name`, `price`, `dividendRate`를 초기화.
- @Override public void printInfo()

- `printInfo()`를 오버라이딩하여 우선주의 정보(배당률 포함)를 출력하기.

▼ 2-3. Asset, Valuable의 Stock으로의 흐름과 Asset, Valuable의 PreferredStock으로의 흐름?

- 추상 클래스 Asset
  - "모든 자산이라면 name과 price를 가져야 하며, 자신을 소개하는 방법인 printInfo() 메서드를 반드시 가져야 한다"라는 기본 골격을 생성하고 printInfo()를 선언만 해둔다.
- 인터페이스 Valuable
  - "가치 있는 자산이라면 반드시 printInfo()를 구현해야 한다"라는 규약을 정의하고, 추가로 updatePrice(double price)라는 기본 기능을 메뉴얼에 적어둔다.
- Stock 클래스
  1. Asset을 상속받아서 name과 price 필드를 물려받음
  2. printInfo()를 구현하면서 "나는 일반주이고, 종목명은 name, 현재가는 price원이다"라는 구체적인 출력 내용을 정의
  3. 동시에 Valuable 인터페이스를 구현
  4. 규약을 확인해보니 printInfo()는 이미 Asset에서 추상 메서드로 선언되어 있었고, Stock이 그것을 구체적으로 작성했으므로 인터페이스 규약을 만족
  5. Valuable 인터페이스를 구현했으므로 printInfo()와 updatePrice(double price) 메서드를 사용할 수 있음
  6. 결국 Asset에서 내려온 골격(name, price, printInfo())과 Valuable에서 정한 규칙 및 기능(printInfo(), updatePrice(double price))이 Stock 클래스 안에서 결합됨
- PreferredStock 클래스
  1. Asset을 상속받아 기본 필드인 name과 price를 물려받고, printInfo를 구현
  2. 일반주와는 다르게 배당률이라는 고유한 특징이 있으므로 새로운 필드 dividendRate를 추가.
  3. printInfo에서는 이름, 가격과 함께 배당률도 출력.
- 결론
  - Asset이 제공하는 공통 골격(name, price, printInfo())과 Valuable이 정한 규약(printInfo()) 및 기능(updatePrice(double price))이 Stock과 PreferredStock에 각각 결합되어 Stock은 일반주로서 name과 price를 출력하고 PreferredStock은 여기에 dividendRate를 더해 고유 특성을 반영한다.

## 3. 다형성

▼ 3-1. 개념 및 목적

- 개념
  - 객체지향 프로그래밍에서 하나의 타입으로 여러 형태의 동작을 표현 즉 같은 이름의 메서드를 호출하더라도 객체의 실제 타입에 따라 실행되는 동작이 달라지는 특성
  - 이를 가능하게 하는 조건은 상속과 메서드 오버라이딩으로 구현되고 보통 업캐스팅과 함께 활용된다.
    - 부모 클래스 타입의 참조 변수를 통해 메서드를 호출하면, 실행 시점에는 실제 객체 타입에 맞는 오버라이딩된 메서드가 실행된다.
- 목적
  1. 코드의 유연성 확보: 하나의 부모 타입으로 여러 자식 객체를 다룰 수 있기 때문에, 코드 구조를 단순하게 유지하면서 다양한 객체를 일관된 방식으로 처리할 수 있어서 새로운 자식 클래스가 추가되더라도 기존 코드를 거의 수정하지 않고 확장이 가능하다.

▼ 3-2. 샘플 코드

```
// 부모 클래스
class Stock {
    protected String name;
```

```
    protected double price;

    public Stock(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // 부모 메서드
    public void printInfo() {
        System.out.println("[일반주] 종목: " + name + ", 가격: " + price + "원");
    }
}

// 자식 클래스
class PreferredStock extends Stock {
    double dividendRate;

    public PreferredStock(String name, double price, double dividendRate) {
        super(name, price);
        this.dividendRate = dividendRate;
    }

    // 부모 메서드를 오버라이딩
    @Override
    public void printInfo() {
        System.out.println("[우선주] 종목: " + name + ", 가격: " + price + "원, 배당률: " + dividendRate + "%");
    }

    // 자식 클래스만 가진 메서드
    public void showDividend() {
        System.out.println("배당률은 " + dividendRate + "% 입니다.");
    }
}

// 실행 클래스
public class PolymorphismExample {
    public static void main(String[] args) {
        // 업캐스팅 (자식 → 부모 타입)
        Stock stock = new PreferredStock("스칼라 AI", 17500, 5.0);

        // 부모 타입으로 참조했지만, 실제 실행은 자식 클래스의 메서드가 호출됨
        stock.printInfo();

        // 다운캐스팅 (부모 타입 → 자식 타입)
        if (stock instanceof PreferredStock) {
            PreferredStock ps = (PreferredStock) stock;
            ps.showDividend(); // 자식 클래스 고유 메서드 사용 가능
        }
    }
}
```

▼ 3-2-1. 코드 설명 - Stock

```
class Stock {
    protected String name;
```

```
    protected double price;

    public Stock(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // 부모 메서드
    public void printInfo() {
        System.out.println("[일반주] 종목: " + name + ", 가격: " + price + "원");
    }
}
```

- class Stock
  - 주식 개념 부모 클래스
- protected String name; protected double price;
  - 주식의 이름과 가격을 저장하는 필드(멤버 변수)
  - protected
    - 같은 패키지 내부나 상속받은 자식 클래스에서 접근 가능하다. 외부에서는 직접 접근 불가하다.
- `public Stock(String name, double price)`
  - 생성자(Constructor)
  - `name`과 `price`를 받아 초기화
- `public void printInfo()`
  - 주식 정보를 출력
  - System.out.println("[일반주] 종목: " + name + ", 가격: " + price + "원");
    - “일반주”라고 표시하고 종목명과 가격을 보여준다
  - 자식 클래스에서 오버라이딩 대상인 메서드

▼ 3-2-2. 코드 설명 - PreferredStock

```
class PreferredStock extends Stock {
    double dividendRate;

    public PreferredStock(String name, double price, double dividendRate) {
        super(name, price);
        this.dividendRate = dividendRate;
    }

    // 부모 메서드를 오버라이딩
    @Override
    public void printInfo() {
        System.out.println("[우선주] 종목: " + name + ", 가격: " + price + "원, 배당률: " + dividendRate + "%");
    }

    // 자식 클래스만 가진 메서드
    public void showDividend() {
        System.out.println("배당률은 " + dividendRate + "% 입니다.");
    }
}
```

- class PreferredStock extends Stock
  - Stock을 상속받은 자식 클래스.
  - 상속을 통해 name과 price를 물려받았는데 배당률(dividendRate)이라는 속성을 추가하여 "우선주"를 구체화함.
  - super(name, price);
    - 부모 클래스의 생성자를 호출
  - @Override public void printInfo()
    - 부모 클래스 printInfo()를 오버라이딩
    - 실행 시점에는 동적 바인딩에 의해, 객체의 실제 타입이 PreferredStock이면 이 메서드가 실행된다.
- public void showDividend()
  - 자식 클래스에만 있는 메서드. 배당률을 따로 출력하는 기능.
  - 부모 타입 변수로는 접근할 수 없고, 자식 타입으로 다운캐스팅해야 호출할 수 있다.

▼ 3-3. "@Override public void printInfo()를 오버라이딩하면 실행 시점에 객체의 실제 타입에 맞는 메서드가 호출된다"의 의미?

- @Override public void printInfo()를 오버라이딩?
  - 부모 Stock에는 printInfo()가 있는데 자식 PreferredStock이 똑같은 메서드 시그니처(메서드 이름, 매개변수 목록, 반환형이 동일)로 다시 정의하면 그게 오버라이딩.
- printInfo() 실행 시점에 객체의 실제 타입에 맞는 메서드가 호출된다?
  - printInfo()같은 인스턴스 메서드는 2단계로 처리되는데
    1. 메서드 호출
       - 컴파일러는? 변수의 선언 타입을 보고 "이 메서드를 불러도 되는지" 확인한다.
       - Stock s = new PreferredStock(...) 일때 s.printInfo(); 하면 s가 Stock 타입이니까, Stock 클래스에 printInfo()가 있는지만 확인한다.
    2. 실제 구현
       - JVM은? 실제 객체가 누구인지 확인하는데
       - 지금 s가 참조하는 건 Stock이 아니라 PreferredStock 객체니까 "PreferredStock에 printInfo()가 오버라이딩돼 있네? 그럼 이걸 실행해야겠다." 하고 결정한다.
- 결론
  - printInfo() 호출하면 컴파일러는 변수선언을 보고 s가 Stock 타입이고 Stock 안에 printInfo() 있으니까 호출 승인하고, 어떤 버전의 printInfo()가 실행될지는 아직 정해지지 않았고, JVM이 객체를 확인했을때 Stock객체라면 부모 클래스 버전 `printInfo`()이 실행, PreferredStock이라면 그 클래스에서 정의된 `printInfo`()를 실행한다.

▼ cf - 동적 바인딩

실행할 때 객체의 실제 타입을 보고 그에 맞는 메서드를 선택하는게 동적 바인딩. (s라는 변수가 Stock 타입으로 선언되어 있어도, new PreferredStock(...)로 만든 객체를 가리키고 있다면 자식 쪽에 오버라이딩된 메서드가 실행됨)

## 4. 상속

▼ 4-1. 개념 및 목적

- 개념
  - 기존(부모) 클래스가 가진 속성과 메서드를 새로운(자식) 클래스가 계승하여 활용할 수 있도록 하는 개념
  - 자식 클래스는 부모 클래스가 정의한 필드와 메서드를 직접 사용할 수 있다.
    - 필요에 따라 새로운 속성과 기능을 추가하거나, 부모 메서드를 오버라이딩(Overriding)하여 구체적인 동작을 재정의할 수 있고
    - 이를 통해 자식 클래스는 부모 클래스가 제공하는 공통 기능을 기반으로 기본 구조와 일관성을 유지하면서도, 고유한 특성과 요구 사항을 반영하여 더욱 구체적이고 특화된 클래스로 확장될 수 있다.

- 목적
  1. 코드 재사용성: 부모 클래스에 정의된 공통 속성과 기능을 여러 자식 클래스에서 공유할 수 있어, 중복 코드를 줄이고 전체 코드 구조를 간결하게 만든다.
  2. 유지보수성과 확장성: 공통 로직은 부모 클래스에만 수정하면 되고, 자식 클래스는 필요에 따라 기능을 덧붙이거나 오버라이딩을 통해 동작을 변경할 수 있어 유지보수가 쉽고 새로운 기능 추가도 용이하다.
  3. 다형성 기반 마련: 부모 타입으로 자식 객체를 다룰 수 있고, 실행 시점에는 실제 객체의 타입에 맞는 동작이 수행되므로 유연한 구조를 만들 수 있다.

▼ 4-2. 샘플 코드

```
// 부모 클래스
class Stock {
    protected String name;
    protected double price;

    public Stock(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // 부모 메서드
    public void printInfo() {
        System.out.println("[일반주] 종목: " + name + ", 가격: " + price + "원");
    }
}

// 자식 클래스
class PreferredStock extends Stock {
    double dividendRate;

    public PreferredStock(String name, double price, double dividendRate) {
        super(name, price);
        this.dividendRate = dividendRate;
    }

    // 부모 메서드를 오버라이딩
    @Override
    public void printInfo() {
        System.out.println("[우선주] 종목: " + name + ", 가격: " + price + "원, 배당률: " + dividendRate + "%");
    }

    // 자식 클래스만 가진 메서드
    public void showDividend() {
        System.out.println("배당률은 " + dividendRate + "% 입니다.");
    }
}

// 실행 클래스
public class PolymorphismExample {
    public static void main(String[] args) {
        // 업캐스팅 (자식 → 부모 타입)
        Stock stock = new PreferredStock("스칼라 AI", 17500, 5.0);

        // 부모 타입으로 참조했지만, 실제 실행은 자식 클래스의 메서드가 호출됨
```

```
        stock.printInfo();

        // 다운캐스팅 (부모 타입 → 자식 타입)
        if (stock instanceof PreferredStock) {
            PreferredStock ps = (PreferredStock) stock;
            ps.showDividend(); // 자식 클래스 고유 메서드 사용 가능
        }
    }
}
```

▼ 4-2-1. 코드 설명 - 부모 클래스 Stock

```
class Stock {
    protected String name;
    protected double price;

    public Stock(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // 부모 메서드
    public void printInfo() {
        System.out.println("[일반주] 종목: " + name + ", 가격: " + price + "원");
    }
}
```

- class Stock
  - 주식 개념 부모 클래스
- protected String name; protected double price;
  - 주식의 이름과 가격을 저장하는 필드(멤버 변수)
  - protected
    - 같은 패키지 내부나 상속받은 자식 클래스에서 접근 가능하다. 외부에서는 직접 접근 불가하다.
- `public Stock(String name, double price)`
  - 생성자(Constructor)
  - `name` 과 `price` 를 받아 초기화
- `public void printInfo()`
  - 주식 정보를 출력
  - System.out.println("[일반주] 종목: " + name + ", 가격: " + price + "원");
    - “일반주”라고 표시하고 종목명과 가격을 보여준다
  - 자식 클래스에서 오버라이딩 대상인 메서드

▼ 4-2-2. 코드 설명 - 자식 클래스 PreferredStock

```
class PreferredStock extends Stock {
    double dividendRate;

    public PreferredStock(String name, double price, double dividendRate) {
        super(name, price);
        this.dividendRate = dividendRate;
```

```
    }

    // 부모 메서드를 오버라이딩
    @Override
    public void printInfo() {
        System.out.println("[우선주] 종목: " + name + ", 가격: " + price + "원, 배당률: " + dividendRate + "%");
    }

    // 자식 클래스만 가진 메서드
    public void showDividend() {
        System.out.println("배당률은 " + dividendRate + "% 입니다.");
    }
}
```

- class PreferredStock extends Stock
  - Stock을 상속받은 자식 클래스.
  - 상속을 통해 name과 price를 물려받았는데 배당률(dividendRate)이라는 속성을 추가하여 “우선주”를 구체화함.
  - super(name, price);
    - 부모 클래스의 생성자를 호출
  - @Override public void printInfo()
    - 부모 클래스 printInfo()를 오버라이딩
    - 실행 시점에는 동적 바인딩에 의해, 객체의 실제 타입이 PreferredStock이면 이 메서드가 실행된다.
- public void showDividend()
  - 자식 클래스에만 있는 메서드. 배당률을 따로 출력하는 기능.
  - 부모 타입 변수로는 접근할 수 없고, 자식 타입으로 다운캐스팅해야 호출할 수 있다.

▼ 4-3. 업캐스팅과 다운캐스팅

- 업캐스팅
  - 자식 객체를 부모 타입 변수에 담는것
  - PreferredStock이 Stock을 상속받는 상황에서 Stock stock = new PreferredStock(...);처럼 쓰면 실제 객체는 PreferredStock이지만 참조 변수의 타입을 Stock으로 지정했기 때문에 컴파일러는 이 객체를 부모 클래스 객체 형식으로 인지한다.
  - 실행 시점에 stock.printInfo()를 호출하면 실제 객체가 PreferredStock이므로 자식이 오버라이딩한 메서드가 실행된다. 핵심은 부모객체처럼 인지되면서도 실제동작은 자식클래스의 성질이 반영된다.
- 업캐스팅 하는이유?
  - 여러 종류의 자식 클래스를 일괄적으로 묶어서 처리할수있기때문에 코드가 단순해진다.
- 다운캐스팅
  - 부모 타입 변수에 들어 있는 객체를 다시 자식 타입 변수로 변환하는것.
  - Stock stock이라는 부모 타입 참조가 있지만, 실제 객체가 PreferredStock이라면 (PreferredStock) stock으로 형변환을 거치면 자식 타입 변수로 다룰수있다 즉 자식만이 가진 고유한 메서드 showDividend() 를 호출할수있다.
- 결론
  - 상속 구조에서는 같은 객체를 필요에 따라 업캐스팅 ↔ 다운캐스팅으로 부모 클래스 ↔ 자식 클래스로 바꿔 다루면서 공통성과 특수성을 효율적으로 반영할수있다.

## 5. 공통 특성: 인터페이스와 구현의 분리

- 캡슐화, 추상화, 다형성, 상속은 결국 인터페이스와 구현이 분리되는걸 활용하는게 포인트인것같은데 인터페이스와 구현의 분리가 각각 어떻게 활용되었는가?

- 캡슐화 - 데이터(구현)를 숨기고 메서드(인터페이스)만 공개
- 추상화 - “무엇을 할 수 있는가”(인터페이스)와 “어떻게 할 것인가”(구현)를 분리
- 다형성 - 부모의 틀(인터페이스)은 유지하면서, 자식에서 구체 구현을 다양하게 정의
- 상속 - 호출하는 쪽은 부모 타입(인터페이스)만 보고, 실행되는 쪽은 실제 객체의 구현을 따른다.