

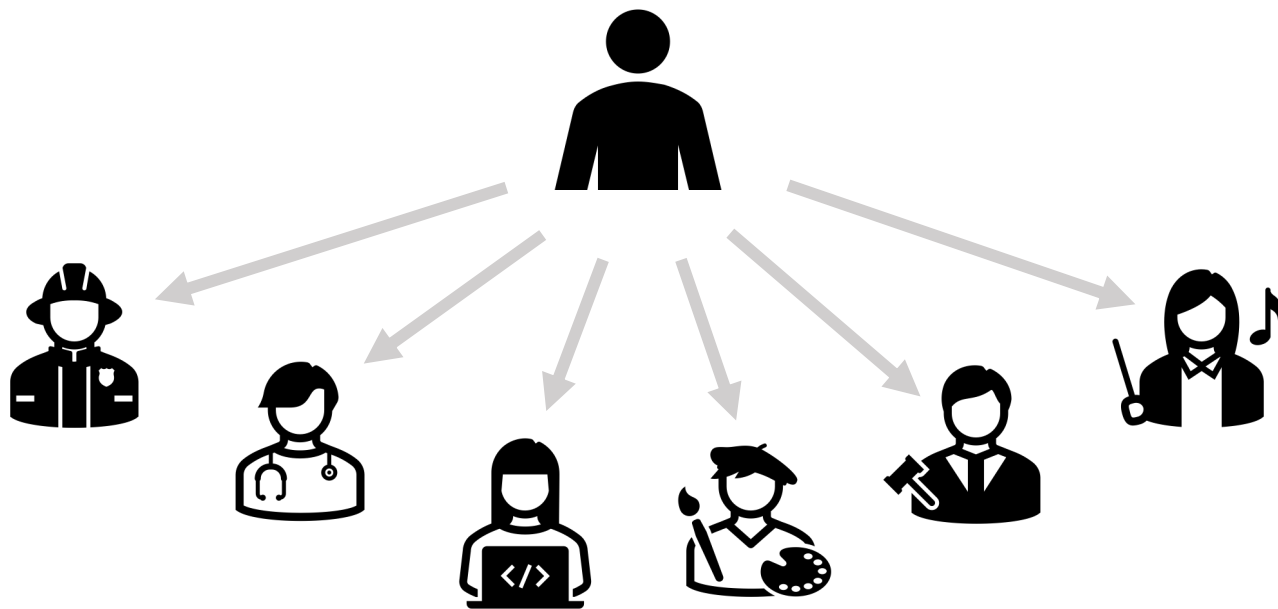


Java



객체지향 프로그래밍 (OOP)

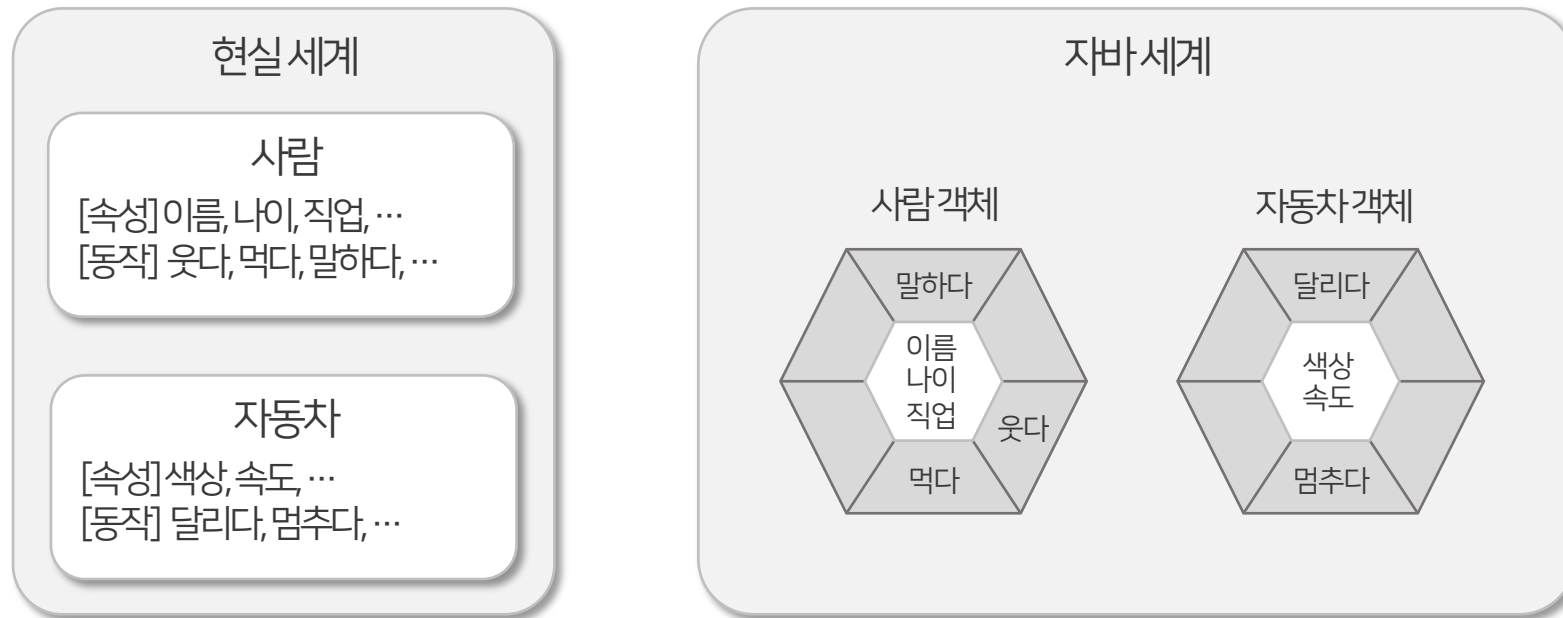
- 현실 세계에서 어떤 제품을 만들 때는 부품을 먼저 만들고, 이 부품들을 하나씩 조립해서 완제품을 만든다.
- 소프트웨어를 개발할 때에도 부품에 해당하는 객체를 먼저 만들고, 이 객체들을 하나씩 조립해서 완성된 프로그램을 만든다.
- 이러한 기법을 객체지향 프로그래밍(Object Oriented Programming)이라 한다.





객체지향 프로그래밍 (OOP)

- 객체(Object)란 물리적으로 존재하거나 개념적인 것 중에서 다른 것과 식별 가능한 것을 말한다.
 - 물리적으로 존재하는 자동차, 사람 등 / 개념적으로 존재하는 강의, 주문 등도 모두 객체가 될 수 있다.
- 객체는 속성과 동작으로 구성되며, Java에서는 이를 필드(Field)와 메서드(Method)라고 부른다.
- 현실 세계의 객체를 소프트웨어 객체로 설계하는 것을 객체 모델링(Object Modeling)이라 한다.
- 객체 모델링은 현실 세계 객체의 대표 속성과 동작을 추려내어 소프트웨어 객체의 필드와 메소드로 정의하는 과정이다.





객체지향 프로그래밍 (OOP)

- 현실 세계에서 일어나는 모든 현상이 객체와 객체 간의 상호작용으로 이뤄지듯, 객체지향 프로그램에서도 객체들은 다른 객체와 서로 상호작용하며 동작한다.
- 객체들의 상호작용 수단은 메소드(Method)이다.

- 메소드 호출은 아래와 같은 형태를 가지고 있다.

메소드명(매개값1, 매개값2, ...);

- 매개값: 메소드가 실행할 때 필요한 값
- 메소드의 리턴값은 호출한 곳에서 변수로 대입받아 사용할 수 있다.

타입 변수명 = *메소드명*(매개값1, 매개값2, ...);

- 리턴값: 메소드를 실행한 결과로 반환된 값 (호출한 곳으로 돌려주는 값)



객체지향 프로그래밍 (OOP)

- 객체는 단독으로 존재할 수 있지만 대부분은 다른 객체와 관계를 맺고 있다.
- 관계의 종류에는 집합 관계, 사용 관계, 상속 관계가 있다.

집합 관계

완성품과 부품의 관계

자동차 객체



엔진 객체



타이어 객체



핸들 객체



사용 관계

다른 객체의 필드를 읽고 변경하거나
메소드를 호출하는 관계

사람 객체



자동차 객체



상속 관계

부모에게서 필드와 메소드를
물려받는 관계

기계 객체



자동차 객체





객체지향 프로그래밍 (OOP)

- 객체지향 프로그램의 특징은 캡슐화, 상속, 다형성이다.
 - 캡슐화
객체의 필드, 메소드를 하나로 묶고, 실제 구현 내용을 외부에 감추는 것.
외부에서는 객체 내부의 구조를 알지 못하고, 객체가 노출해서 제공하는 필드와 메소드만을 이용할 수 있다.
캡슐화는 외부의 잘못된 사용으로 객체가 손상되는 것을 방지한다.
Java에서는 캡슐화를 위해 접근 제한자(Access modifier)를 사용한다.
 - 상속
부모 객체는 자식 객체에게 필드와 메소드를 물려주어 자식 객체가 이를 이용할 수 있도록 할 수 있다.
상속을 하면, 중복 코딩을 하지 않아도 되기 때문에 코드의 재사용성을 높일 수 있으며,
부모 객체에서 필드와 메소드를 수정하면, 자식 객체들은 수정된 필드와 메소드를 사용할 수 있기 때문에 유지보수 시간을 단축할 수 있다.
 - 다형성
동일한 사용 방법이나 다양한 결과가 나타나는 성질을 의미한다.
기계의 부품을 교환하면 성능이 달라지듯, 프로그램을 구성하는 객체를 바꾸면 프로그램의 실행 성능이 달라질 수 있다.
다형성을 구현하기 위해서는 자동 타입 변환과 재정의(Overriding) 기술이 필요하다.



객체지향 프로그래밍 (OOP) - 클래스

- 현실에서 자동차를 생성하려면 자동차의 설계도가 필요하듯, 객체지향 프로그래밍에도 설계도가 필요하다.
- Java에서는 클래스(Class)가 객체를 생성하기 위한 설계도 역할을 한다.
- 지금까지 생성한 클래스에는 객체를 만들지는 않았고, main() 메소드만 작성해서 실행할 목적으로 클래스를 이용했다.
- 클래스로부터 객체가 생성되는 과정을 인스턴스화(Instance)라고 하며, 생성된 객체는 인스턴스(Instance)라고 부른다.



객체지향 프로그래밍 (OOP) - 클래스

- 클래스 선언은 객체 생성을 위한 설계도를 작성하는 작업이기 때문에 어떻게 객체를 생성하고 (생성자), 객체가 가져야 할 데이터가 무엇이고 (필드), 객체의 동작은 무엇인지 (메소드)를 포함한다.
- 클래스의 선언은 소스 파일명과 동일하게 아래와 같이 작성한다.

```
public class 클래스명 {  
  
}
```

TIP

공개 클래스는 어느 위치에 있든지 패키지와 상관없이 사용할 수 있는 클래스를 의미한다.



- public class는 공개 클래스를 선언한다는 뜻으로, 하나의 소스 파일에는 소스 파일명과 동일한 하나의 클래스만 공개 클래스로 선언할 수 있다.



객체지향 프로그래밍 (OOP) - 클래스

- 하나의 소스 파일에는 여러 개의 클래스를 선언할 수 있다.
- 여러 개의 클래스 선언이 포함된 소스 파일을 컴파일 하면, 바이트코드 파일(.class)은 클래스 선언 수만큼 생겨난다.

```
public class SportsCar {  
  
}  
  
class Tire {  
  
}
```

 SportsCar.class Tire.class

- Tire 클래스도 공개 클래스로 선언하고 싶다면, Tire.java 파일을 별도로 생성해야 한다.
- 특별한 이유가 없다면, 파일 하나당 클래스 하나를 선언하는 것이 좋다.



객체지향 프로그래밍 (OOP) - 클래스

- 선언된 클래스로부터 객체를 생성하기 위해서는 객체 생성 연산자인 `new`가 필요하다.
- `new` 연산자는 객체를 생성시킨 후 객체의 주소를 반환한다. 따라서 클래스 변수에 대입할 수 있다.
또한 `new` 연산자 뒤에는 생성자 호출 코드가 있는데, 이를 바탕으로 작성해보면 아래와 같은 형태를 가진다.

클래스명 변수명 = **new** 클래스명();

```
public class Student {  
}  
  
public class StudentExample {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        System.out.println("변수 s1은 Student 객체를 참조합니다.");  
        Student s2 = new Student();  
        System.out.println("변수 s2는 또 다른 Student 객체를 참조합니다.");  
    }  
}
```

TIP

클래스에는 두 가지 용도가 있는데,

1. 직접 실행되지 않고, 다른 클래스에서 이용되는 라이브러리 클래스
2. `main()` 메소드를 가지고, 직접 실행되는 실행 클래스

여기에서 `Student`는 라이브러리 클래스이고, `StudentExample`은 실행 클래스이다.



객체지향 프로그래밍 (OOP) - 클래스

- 클래스 선언은 객체 생성을 위한 설계도를 작성하는 작업이기 때문에 어떻게 객체를 생성하고 (생성자), 객체가 가져야 할 데이터가 무엇이고 (필드), 객체의 동작은 무엇인지 (메소드)를 포함한다.
- 클래스의 구성 멤버
 1. 생성자: 객체 생성 시 초기화 역할 담당
 2. 필드: 객체의 데이터가 저장되는 곳
 3. 메소드: 객체의 동작으로 호출 시에 실행되는 블록

```
public class ClassName {  
    // 필드 선언  
    int fileName;  
  
    // 생성자 선언  
    ClassName() {  
  
    }  
  
    // 메소드 선언  
    int methodName() {  
  
    }  
}
```



객체지향 프로그래밍 (OOP) - 클래스

- 필드
 - 객체의 데이터를 저장하는 역할로, 선언 형태는 변수 선언과 동일하지만 쓰임새는 다르다.
- 생성자
 - new 연산자로 객체를 생성할 때, 객체의 초기화 역할을 담당한다.
 - 메소드와 유사한 선언 형태를 가지나, 반환 타입이 없다.
 - 생성자 이름은 반드시 클래스 이름과 동일해야 한다.
- 메소드
 - 객체가 수행할 동작으로, 객체 간 상호작용을 위해 호출된다.
 - 객체 내부의 함수를 메소드라 한다.

```
public class ClassName {  
    // 필드 선언  
    int fileName;  
  
    // 생성자 선언  
    ClassName() {  
  
    }  
  
    // 메소드 선언  
    int methodName() {  
  
    }  
}
```



객체지향 프로그래밍 (OOP) - 필드

- 필드는 객체의 속성 데이터를 저장하는 용도로 사용되며, 선언하는 방법은 변수 선언과 동일하다.
- 단, 반드시 클래스 블록 내부에서 선언되어야 필드 선언이 된다.
- 필드는 객체 내부에 존재하고, 객체 내부와 외부에서 모두 사용 가능하다.
- 클래스를 통해 객체가 생성될 때 생성되며, 객체가 제거될 때 삭제된다.
- 초기화를 하지 않을 경우에는 자동으로 기본값으로 초기화된다.

```
public class Car {  
    String model = "그랜저";  
    int speed;  
    boolean start;  
    Tire tire = new Tire();  
}
```

```
public class CarExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        System.out.println("모델명 :" + myCar.model);  
        System.out.println("시동 여부 :" + myCar.start);  
        System.out.println("현재 속도 :" + myCar.speed);  
    }  
}
```



객체지향 프로그래밍 (OOP) - 필드

- 필드를 사용한다는 것은 필드값을 읽고 변경하는 것을 의미한다.
- 필드는 객체의 데이터이므로, 클래스로부터 객체가 생성된 후에 필드를 사용할 수 있다.
- 필드는 객체 내부의 생성자와 메소드 내부에서 사용할 수 있으며, 객체 외부에서도 접근해서 사용할 수 있다.
- 객체 외부에서는 참조 변수와 객체 접근 연산자(.)를 이용해 필드를 읽고 변경할 수 있다.

```
public class Car {  
    String model = "그랜저";  
    int speed;  
    boolean start;  
    Tire tire = new Tire();  
}
```

```
public class CarExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        System.out.println("모델명 :" + myCar.model);  
        System.out.println("시동 여부 :" + myCar.start);  
        System.out.println("현재 속도 :" + myCar.speed);  
  
        myCar.start = true;  
        myCar.speed = 30;  
  
        System.out.println("시동 여부 :" + myCar.start);  
        System.out.println("현재 속도 :" + myCar.speed);  
    }  
}
```



객체지향 프로그래밍 (OOP) - 생성자

- 객체를 생성할 때 사용하는 new 연산자는 객체를 생성한 후 곧바로 생성자를 호출해 객체를 초기화한다.
- 객체 초기화: 필드를 초기화하거나 메소드를 호출해서 객체를 사용할 준비를 하는 것을 의미한다.

클래스명 변수명 = new 클래스명();

- 모든 클래스에는 하나 이상의 생성자가 존재한다.
- 하지만 클래스에 생성자 선언이 하나도 없어도 객체 생성은 가능하다.
- 왜냐하면 클래스에 생성자 선언이 없으면, 컴파일러는 기본 생성자를 자동으로 추가시키기 때문이다.

```
public class Car {  
    String model = "그랜저";  
    int speed;  
    boolean start;  
    Tire tire = new Tire();  
}
```

소스 파일 (Car.java)

```
public class Car {  
    String model = "그랜저";  
    int speed;  
    boolean start;  
    Tire tire = new Tire();  
  
    public Car() {}  
}
```

바이트코드 파일 (Car.class)



객체지향 프로그래밍 (OOP) - 생성자

- 그러나 개발자가 명시적으로 선언한 생성자가 있다면 컴파일러는 기본 생성자를 추가하지 않는다.

```
public class Car {
    String model = "그랜저";
    int speed;
    boolean start;
    Tire tire = new Tire();

    public Car(String m, int sp, boolean st) {
        model = m;
        speed = sp;
        start = st;
    }
}

public class CarExample {
    public static void main(String[] args) {
        Car newCar = new Car("K5", 200, true);
        System.out.println("모델명 :" + newCar.model);
        System.out.println("시동 여부 :" + newCar.start);
        System.out.println("현재 속도 :" + newCar.speed);
        // Car oldCar = new Car(); The constructor Car() is undefined
    }
}
```




객체지향 프로그래밍 (OOP) - 생성자

- 객체마다 동일한 값을 가져야 한다면, 필드 선언시 초기값을 대입하는 것이 좋고, 객체마다 다른 값을 가져야 한다면, 생성자에서 필드를 초기화하는 것이 좋다.

```
public class Korean {
    String nation = "대한민국";
    String name, ssn;

    public Korean(String n,
        String s) {
        name = n;
        ssn = s;
    }
}

public class KoreanExample {
    public static void main(String[] args) {
        Korean k1 = new Korean("최인규", "990101-1001234");
        System.out.println(k1.nation + ", " + k1.name + ", " + k1.ssn);
        Korean k2 = new Korean("김자바", "001231-4004321");
        System.out.println(k2.nation + ", " + k2.name + ", " + k2.ssn);
    }
}
```



객체지향 프로그래밍 (OOP) - 생성자

- Korean 생성자를 보면, 매개변수 이름으로 각각 n와 s를 사용했다.
- 매개변수 이름이 너무 짧으면 코드 가독성을 해칠 수 있기 때문에, 가능하면 필드명과 동일한 이름을 사용하는 것이 좋다.
- 동일한 이름을 사용하기 위해서는 필드와 매개변수를 구분해주기 위해 this 키워드를 필드명 앞에 붙여야 한다.

```
public Korean(String name, String ssn) {  
    this.name = name;  
    this.ssn = ssn;  
}
```

TIP

this는 현재 객체를 의미한다.

TIP

이클립스는 필드의 색깔을 파란색, 매개변수의 색깔을 갈색으로 보여주기 때문에 쉽게 구별할 수 있다.



객체지향 프로그래밍 (OOP) - 생성자

- 개발자가 명시적으로 생성자를 선언하는 목적은 객체를 원하는 형태로 초기화하기 위해서이다.
- 또한 여러 개의 생성자를 선언하여, 다양한 형태로 초기화를 할 수 있다.
- 다양하게 초기화를 위해서는 생성자 오버로딩(Overloading)이 필요하다.

TIP

오버로딩은 매개변수가 다른 생성자를 여러 개 선언하는 것을 의미한다.

```
public class Car {  
    String model = "그랜저";  
    String color;  
    int speed;  
    boolean start;  
  
    Car() { /* ... */}  
    Car(String model) { /* ... */}  
    Car(String model, String color) { /* ... */}  
    Car(String model, String color, int speed) { /* ... */}  
    Car(String model, String color, int speed, boolean start) { /* ... */}  
}
```



객체지향 프로그래밍 (OOP) - 생성자

- 생성자가 오버로딩되어 있을 경우에는 new 연산자로 생성자를 호출할 때 제공되는 매개값의 타입과 수에 따라 실행될 생성자가 결정된다.

```
public class Car {
    String model;
    String color;
    int speed;
    boolean start;

    Car() { /* ... */}
    Car(String model) { /* ... */}
    Car(String model, String color) { /* ... */}
    Car(String model, String color, int speed, boolean start) { /* ... */}
}

public class CarExample {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("K5");
        Car car3 = new Car("그랜저", "검정색");
        Car car4 = new Car("소나타", "흰색", 100, true);
    }
}
```



객체지향 프로그래밍 (OOP) - 생성자

- 생성자 오버로딩 시에 주의할 점은 매개변수의 타입, 개수, 순서가 다르게 선언되어야 한다는 것이다.
- 따라서 아래 코드는 오버로딩이 진행되지 않고, 컴파일 에러가 발생한다.

```
public class Car {  
    /* ... */  
    Car(String color, String model) { /* ... */}  
    Car(String model, String color) { /* ... */}  
}
```



객체지향 프로그래밍 (OOP) - 생성자

- 생성자 오버로딩이 많아질 경우, 생성자 간의 중복된 코드가 발생할 수 있다.
- 이럴 때에는 공통 코드를 한 생성자에만 집중적으로 작성하고, 나머지 생성자는 `this`를 사용하여 공통 코드를 가지고 있는 생성자를 호출하는 방법으로 개선할 수 있다.

```
public class Car {  
    String model;  
    String color;  
    int speed;  
  
    Car(String model) {  
        this.model = model;  
        this.color = "은색";  
        this.speed = 50;  
    }  
  
    Car(String model, String color) {  
        this.model = model;  
        this.color = color;  
        this.speed = 100;  
    }  
  
    Car(String model, String color, int speed) {  
        this.model = model;  
        this.color = color;  
        this.speed = speed;  
    }  
}
```

중복 코드

중복 코드

중복 코드

```
public class Car {  
    String model;  
    String color;  
    int speed;  
  
    Car(String model) {  
        this(model, "은색", 50);  
    }  
  
    Car(String model, String color) {  
        this(model, color, 100);  
    }  
  
    Car(String model, String color, int speed) {  
        this.model = model;  
        this.color = color;  
        this.speed = speed;  
    }  
}
```

공통 초기화 코드



객체지향 프로그래밍 (OOP) - 메소드

- 메소드 선언은 객체의 동작을 실행 블록을 정의하는 것을 의미하고, 메소드 호출은 실행 블록을 실제로 실행하는 것을 말한다.

- 메소드의 선언은 아래와 같다.

```
반환타입 메소드명 (타입 매개변수) {  
    실행할코드  
}
```

- 리턴 타입(Return Type)
메소드가 실행한 후 호출한 곳으로 전달하는 결과값의 데이터 타입을 의미한다.
만약 반환되는 값이 없는 경우에는 리턴 타입을 void로 작성해야 한다.
- 메소드명
메소드 이름은 소문자로 시작하는 Camel Case로 작성한다.
- 매개변수
매개변수는 메소드를 호출할 때 전달한 매개값을 받기 위해 사용된다.
전달할 매개값이 없다면 매개변수는 생략 가능하다.

```
void powerOn(boolean power) {  
    power = !power;  
}
```

```
double divide(int x, int y) {  
    double result = (double) x / y;  
    return result;  
}
```

```
String getHelloMsg() {  
    return "안녕하세요";  
}
```



객체지향 프로그래밍 (OOP) - 메소드

- 메소드를 호출한다는 것은 메소드 블록을 실행하는 것이다.
- 메서드는 객체의 동작이므로 객체가 존재하지 않으면 메소드를 호출할 수 없다. (필드와 마찬가지로)
- 클래스로부터 객체가 생성된 후에는 생성자와 또 다른 메소드 내부에서 호출할 수 있으며, 객체 외부에서도 호출할 수 있다.
- 객체 내부에서는 메소드 명으로 호출하면 되지만, 객체 외부에서는 참조 변수와 객체 접근 연산자(.)를 이용해 메소드를 호출할 수 있다.



객체지향 프로그래밍 (OOP) - 메소드

```
public class Calculator {
    boolean power;

    void powerOff() {
        System.out.println("전원을 끕니다.");
        this.power = false;
    }

    void powerOn() {
        System.out.println("전원을 켭니다.");
        this.power = true;
    }

    int plus(int x, int y) {
        return power ? x + y : null;
    }

    double divide(int x, int y) {
        return power ? (double) (x / y) :
        null;
    }
}
```

```
public class CalculatorExample {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.powerOn();

        int plusResult = calc.plus(5, 6);
        System.out.println("result1 : " +
        plusResult);

        int x = 10;
        int y = 4;
        double divideResult = calc.divide(x, y);
        System.out.println("result2 : " +
        divideResult);

        calc.powerOff();
    }
}
```



객체지향 프로그래밍 (OOP) - 메소드

- 메소드를 호출할 때는 매개변수의 개수에 맞게 매개값을 제공해야 한다.
- 하지만 매개값의 개수가 때에 따라 달라지는 경우가 있다.
이런 경우에는 가변 길이 매개변수를 사용할 수 있도록 메소드를 아래와 같이 선언해야 한다.

```
반환타입 메소드명(타입 ... 매개변수) {  
    실행할코드  
}
```

- 가변 길이 매개변수는 메소드 호출 시 매개값을 쉼표로 구분해서 개수와 상관없이 제공할 수 있게 된다.
- 매개값들은 자동으로 배열 형태로 변환되어 메소드에서 사용된다. 그렇기 때문에 매개값에 직접 배열을 제공해도 된다.

```
int sum(int ... values) {  
    int result = 0;  
    for (int i : values) {  
        result += i;  
    }  
    return result;  
}
```

```
int result = sum(1, 2, 3);
```

```
int result = sum(1, 2, 3, 4, 5);
```

```
int[] intArr = {2, 4, 6};  
int result = sum(intArr);
```

```
int result = sum(new int[] {1, 3, 5});
```



객체지향 프로그래밍 (OOP) - 메소드

- 메소드 선언에 리턴 타입이 void가 아니라면 반드시 return 문 뒤에 반환값을 지정해야 한다.
- return 문은 메소드의 실행을 강제로 종료하고 호출한 곳으로 돌아간다는 의미가 있다. 따라서 return 문 뒤의 실행문은 절대 실행되지 않는다. (Unreachable code)
- 하지만 아래와 같은 경우에는 컴파일 에러가 발생하지 않는다.

```
boolean isEmpty() {  
    if (balance == 0) {  
        System.out.println("잔액이 부족합니다.");  
        return true;  
    }  
    return false;  
}
```

TIP

if 문의 조건식에 따라 실행되는 코드가 나뉘기 때문이다.



객체지향 프로그래밍 (OOP) - 메소드

- 생성자처럼 메소드도 오버로딩이 가능하다.
- 즉, 같은 클래스 내부에 메소드의 이름은 같지만 매개변수의 타입, 개수, 순서가 다른 메소드를 선언하는 것을 의미한다.
- 메소드 오버로딩의 가장 큰 목적은 다양한 매개값을 처리하기 위함이다.
- 2개의 int 타입 변수의 합계를 구하는 sum() 메소드가 있다면, double 타입 매개값은 처리할 수 없을 것이다. 만약 double 타입 변수도 처리하고 싶을 경우에, 오버로딩을 하면 된다.

```
int sum(int ... values) {  
    int result = 0;  
    for (int i : values) {  
        result += i;  
    }  
    return result;  
}
```

```
double sum(double ... values) {  
    double result = 0;  
    for (double i : values) {  
        result += i;  
    }  
    return result;  
}
```

- 메소드 오버로딩의 가장 대표적인 예는 콘솔에 출력하는 System.out.println() 메소드이다.

- println() : void - PrintStream
- println(boolean x) : void - PrintStream
- println(char x) : void - PrintStream
- println(char[] x) : void - PrintStream
- println(double x) : void - PrintStream
- println(float x) : void - PrintStream
- println(int x) : void - PrintStream
- println(long x) : void - PrintStream
- println(Object x) : void - PrintStream
- println(String x) : void - PrintStream



객체지향 프로그래밍 (OOP) - 인스턴스 멤버 & 정적 멤버

- 필드와 메소드는 선언 방법에 따라 인스턴스 멤버와 정적 멤버로 분류할 수 있다.
- 인스턴스 멤버로 선언되면, 객체가 생성되어야 사용할 수 있고, 정적 멤버로 선언되면 객체 생성 없이도 사용할 수 있다.

구분	설명
인스턴스 멤버 (Instance Member)	객체에 소속된 멤버 (객체를 생성해야만 사용할 수 있는 멤버)
정적 멤버 (Static Member)	클래스에 고정된 멤버 (객체 없이도 사용할 수 있는 멤버)



객체지향 프로그래밍 (OOP) - 인스턴스 멤버

- 인스턴스 멤버는 객체에 소속된 멤버를 의미하며, 지금까지 선언한 필드와 메소드는 모두 인스턴스 멤버였다.
- gas 필드와 setSpeed() 메소드는 인스턴스 멤버이기 때문에 외부 클래스에서 사용하기 위해서는 Car 객체를 먼저 생성하고 참조 변수로 접근해서 사용해야 한다.
- gas 필드는 객체마다 별도로 존재하고 있으며, setSpeed() 메소드는 각 객체마다 존재하는 것이 아니라 메소드 영역에 저장되고 공유된다.
- 메소드가 객체마다 공유된다면 중복 저장으로 인해 메모리 효율이 떨어진다.
따라서 메소드 코드는 메소드 영역에서 공유해서 사용하고, 객체 없이는 사용을 못하도록 제한을 걸어둔 것으로 볼 수 있다.

```
public class Car {  
    int gas, speed;  
    void setSpeed(int speed) {  
        this.speed = speed;  
    }  
}
```

```
public class CarExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.gas = 10;  
        myCar.setSpeed(60);  
  
        Car yourCar = new Car();  
        yourCar.gas = 20;  
        yourCar.setSpeed(80);  
    }  
}
```



객체지향 프로그래밍 (OOP) - 인스턴스 멤버

- 객체 내부에서는 인스턴스 멤버에 접근하기 위해서는 this 키워드를 사용한다.
- 우리 스스로를 '나'라고 표현하듯, 객체는 자신을 'this'라고 한다.

```
public class Car {
    String model;
    int gas, speed;
    Car(String model) {
        this.model = model;
    }
    void setSpeed(int speed) {
        this.speed = speed;
    }
    void run() {
        // this 생략 가능
        // this.setSpeed(100);
        // System.out.println(this.model + " 이/가 달립니다. (시속: " + this.speed + "km/h)");
        setSpeed(100);
        System.out.println(model + " 이/가 달립니다. (시속: " + speed + "km/h)");
    }
}

public class CarExample {
    public static void main(String[] args) {
        Car myCar = new Car("소나타");
        myCar.run();
    }
}
```



객체지향 프로그래밍 (OOP) - 정적 멤버

- Java는 클래스 로더(Class Loader)를 이용해서 클래스를 메모리 영역에 저장하고 사용한다.
- 정적 멤버란 메모리 영역의 클래스에 고정적으로 위치하는 멤버를 말한다.
그렇게 함으로써 정적 멤버는 객체를 생성할 필요 없이 클래스를 통해 바로 사용이 가능하다.
- 필드와 메소드 앞에 static 키워드를 추가하면, 모두 정적 멤버가 될 수 있다.
- 정적 멤버는 객체 생성 없이 클래스 이름과 객체 접근 연산자(.)로 접근 가능하다.

TIP

정적 멤버는 객체 참조 변수를 통해서 접근도 가능하지만 클래스 이름으로 접근하는 것이 정석이다.

```
public class Korean {
    String name;
    static String nation = "KOREA";
    Korean(String name) {
        this.name = name;
    }
    void sayName() {
        System.out.println("제 이름은 " + name + "입니다.");
    }
    static void sayHello() {
        System.out.println("안녕하세요");
    }
}
```

```
public class KoreanExample {
    public static void main(String[] args)
    {
        Korean.sayHello();
        System.out.println(Korean.nation);
        Korean sunsin = new Korean("이순신");
        System.out.println(sunsin.nation);
        sunsin.sayHello();
        sunsin.sayName();
    }
}
```




객체지향 프로그래밍 (OOP) - 정적 멤버

- 정적 필드는 객체 생성 없이도 사용할 수 있기 때문에 생성자에서 초기화 작업을 하지 않는다. (생성자는 객체 생성 후 실행된다.)
- 객체마다 가지고 있을 필요성이 없는 공용적인 필드는 객체마다 따로 가지고 있을 필요가 없기 때문에 정적 필드로 선언하는 것이 좋다.
- 인스턴스 필드를 이용하지 않는 메소드는 정적 필드로 선언하는 것이 좋다.
- 정적 필드에 초기값을 주는 작업이 복잡하다면, 정적 블록을 이용해야 한다.
- 정적 블록은 클래스가 메모리로 로딩될 때 자동으로 실행된다.

```
public class SmartPhone {  
    static String company = "Apple";  
    static String model = "iPhone 16";  
    static String info;  
    static {  
        System.out.println("정적 블록을 실행합니다.");  
        info = company + "-" + model;  
    }  
}
```

```
public class SmartPhoneExample {  
    public static void main(String[] args) {  
        System.out.println(SmartPhone.info);  
    }  
}
```



객체지향 프로그래밍 (OOP) - 정적 멤버

- 정적 메소드와 정적 블록은 객체가 없어도 실행된다는 특징이 있기 때문에 내부에서 인스턴스 멤버를 사용할 수 없고, this 키워드도 사용할 수 없다.

```
public class ClassName {
    int instanceField;
    void instanceMethod() { }

    static int staticField;
    static void staticMethod() { }
    static void staticMethod2() {
        // instanceField = 10; // Cannot make a static reference to the non-static field
        // instanceMethod(); // Cannot make a static reference to the non-static method
        // this.staticField = 10; // Cannot use this in a static context
        // this.staticMethod(); // Cannot use this in a static context
        staticField = 10;
        staticMethod();
    }

    static {
        // instanceField = 10; // Cannot make a static reference to the non-static field
        // instanceMethod(); // Cannot make a static reference to the non-static method
        // this.staticField = 10; // Cannot use this in a static context
        // this.staticMethod(); // Cannot use this in a static context
        staticField = 10;
        staticMethod();
    }
}
```



객체지향 프로그래밍 (OOP) - 정적 멤버

- 만약 정적 메소드와 정적 블록 내부에서 인스턴스 멤버를 사용하고 싶다면, 내부에서 객체를 생성하고 참조 변수로 접근해야 한다.

```
public class ClassName {  
    int instanceField;  
    void instanceMethod() { }  
  
    static int staticField;  
    static void staticMethod() { }  
    static void staticMethod2() {  
        ClassName instance = new ClassName();  
        instance.instanceField = 10;  
        instance.instanceMethod();  
        staticField = 10;  
        staticMethod();  
    }  
  
    static {  
        ClassName instance = new ClassName();  
        instance.instanceField = 10;  
        instance.instanceMethod();  
        staticField = 10;  
        staticMethod();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 정적 멤버

- main() 메소드도 정적 메소드이기 때문에 객체를 생성하지 않으면 인스턴스 멤버를 main() 메소드에서 사용할 수 없다.

```
public class StaticExample {  
    int instanceField;  
    void instanceMethod() { }  
  
    public static void main(String[] args) {  
        // instanceField = 10; // Cannot make a static reference to the non-static field  
        // instanceMethod(); // Cannot make a static reference to the non-static method  
  
        StaticExample instance = new StaticExample();  
        instance.instanceField = 10;  
        instance.instanceMethod();  
    }  
}
```



객체지향 프로그래밍 (OOP) - final 필드와 상수

- 인스턴스 필드와 정적 필드의 값은 언제든지 변경될 수 있다. 그러나 경우에 따라서는 값을 변경하는 것을 막고 읽기만 허용해야 할 때가 있다.
- 이러한 경우에는 final 필드와 상수를 선언해서 사용해야 한다.
- final 필드에 초기값을 줄 수 있는 방법에는 두 가지가 있다.

TIP

고정된 값이라면 필드 선언 시에 주는 것이 가장 간단하다.

객체 생성 시 외부 값에 의해 final 필드를 초기화하는 경우에는 생성자에서 해야 한다.

1. 필드 선언 시에 초기값 대입

```
public class ClassName {  
    final int final_field_1 = 10;  
}
```

2. 생성자에서 초기값 대입

```
public class ClassName {  
    final int final_field_2;  
    ClassName() {  
        FINAL_FIELD_2 = 10;  
    }  
}
```



객체지향 프로그래밍 (OOP) - final 필드와 상수

```
public class Korean {  
    final String nation = "대한민국";  
    final String ssn;  
    String name;  
  
    Korean(String name, String ssn) {  
        this.ssn = ssn;  
        this.name = name;  
    }  
}
```

```
public class KoreanExample {  
  
    public static void main(String[] args) {  
        Korean student = new Korean("홍길동", "990101-1234567");  
  
        System.out.println(student.nation);  
        System.out.println(student.ssn);  
        System.out.println(student.name);  
        // The final field cannot be assigned  
        // student.nation = "USA";  
        // student.ssn = "980101-1234567";  
        student.name = "김길동";  
    }  
}
```



객체지향 프로그래밍 (OOP) - final 필드와 상수

- 앞의 Korean 클래스의 nation 필드처럼 객체마다 따로 저장할 필요가 없고, 여러 개의 값을 가질 필요가 없는 경우에는 static이면서 final인 특성을 부여하여 상수로 선언할 수 있다.
- 상수명은 모두 대문자로 작성하는 것이 관례이며, 단어가 혼합된 경우에는 언더스코어(_)로 단어를 연결해 사용한다.

```
static final double PI = 3.14159;  
static final double MAX_SPEED = 200;
```



객체지향 프로그래밍 (OOP) - final 필드와 상수

```
public class Earth {  
    static final double EARTH_RADIUS = 6400;  
    static final double EARTH_SURFACE_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;  
}
```

```
public class EarthExample {  
    public static void main(String[] args) {  
        System.out.println("지구의 반지름: " + Earth.EARTH_RADIUS + "km");  
        System.out.println("지구의 표면적: " + Earth.EARTH_SURFACE_AREA + "km^2");  
    }  
}
```




객체지향 프로그래밍 (OOP) - 패키지

- 지금까지 예제들을 패키지 안에 생성해서 관리했다.
- Java의 패키지는 단순히 디렉토리(폴더)만을 의미하는 것이 아니라, 클래스의 일부분이 되어 클래스를 식별하는 용도로 사용된다.
- 따라서 클래스의 전체 이름에 패키지도 포함된다.
- 패키지는 상위 패키지와 하위 패키지를 점(.)으로 구분하며, 주로 개발 회사 도메인 이름의 역순으로 만드는 것이 관례이다.
 - 예를 들어 회사의 도메인이 mycompany.com이라면, com.mycompany로 패키지를 만든다.
- 패키지는 클래스를 컴파일하는 과정에서 클래스의 패키지 선언을 보고 자동으로 디렉토리를 생성시킨다.
- 패키지 이름은 모두 소문자로 작성하고, 회사 도메인 이름의 역순을 적은 후, 마지막에는 프로젝트의 이름을 붙여주는 것이 일반적이다.

```
package com.mycompany.projectname;
```



객체지향 프로그래밍 (OOP) - 패키지

- 같은 패키지에 있는 클래스는 아무런 조건없이 사용할 수 있지만, 다른 패키지에 있는 클래스를 사용하기 위해서는 import 문을 이용해야 한다.
- import 문을 이용해 어떤 패키지의 클래스인지 명시해야 한다.

```
package com.mycompany;
```

```
import com.hankook.Tire;
```

```
public class Car {  
    String brand, model;  
    int speed;  
    final int max_speed;  
    Tire tire = new Tire();
```

```
    Car(String brand, String model, int speed, int max_speed) {  
        this.brand = brand;  
        this.model = model;  
        this.speed = speed;  
        this.max_speed = max_speed;  
    }  
}
```

TIP

이클립스 자동 IMPORT 단축키
CTRL + SHIFT + O
CMD + SHIFT + O



객체지향 프로그래밍 (OOP) - 패키지

- 만약 동일한 패키지에 포함된 다수의 클래스를 사용해야 한다면, 클래스 이름을 생략하고 *을 사용할 수 있다.

```
package com.mycompany;
```

```
import com.hankook.*; // 해당 패키지 내 클래스 모두 사용 가능
```

```
public class Car {  
    String brand, model;  
    int speed;  
    final int max_speed;  
    Tire tire = new Tire();  
  
    Car(String brand, String model, int speed, int max_speed) {  
        this.brand = brand;  
        this.model = model;  
        this.speed = speed;  
        this.max_speed = max_speed;  
    }  
}
```



객체지향 프로그래밍 (OOP) - 패키지

- *은 하위 패키지를 포함하지는 않기 때문에 com.hankook.project라는 패키지 내부에 있는 클래스를 사용할 경우에는 import문을 추가 작성해야 한다.

```
package com.mycompany;
```

```
import com.hankook.*; // 해당 패키지 내 클래스 모두 사용 가능
```

```
import com.hankook.project.*; // 해당 패키지 내 클래스 모두 사용 가능
```

```
public class Car {  
    String brand, model;  
    int speed;  
    final int max_speed;  
    Tire tire = new Tire();  
  
    Car(String brand, String model, int speed, int max_speed) {  
        this.brand = brand;  
        this.model = model;  
        this.speed = speed;  
        this.max_speed = max_speed;  
    }  
}
```



객체지향 프로그래밍 (OOP) - 패키지

- 만약 `com.hankook.Tire`와 `com.kumho.Tire`를 모두 import하고 사용할 경우에는 클래스의 전체 이름을 사용해서 어떤 클래스를 사용하는지 명시해야 한다.
- 전체 이름을 사용할 경우에는 import 문을 작성할 필요가 없다.

```
package com.mycompany;
```

```
public class Car {  
    String brand, model;  
    int speed;  
    final int max_speed;  
    com.kumho.Tire tire = new com.kumho.Tire();  
  
    Car(String brand, String model, int speed, int max_speed) {  
        this.brand = brand;  
        this.model = model;  
        this.speed = speed;  
        this.max_speed = max_speed;  
    }  
}
```



객체지향 프로그래밍 (OOP) - 접근 제한자

- 중요한 필드와 메소드가 외부로 노출되지 않도록 하면 객체의 무결성(결점이 없는 성질)을 유지하고자 한다면, 객체의 필드를 외부에서 변경하는 것을 막고, 메소드를 외부에서 호출하는 것을 막을 필요가 있다.
- Java는 이러한 기능을 구현하기 위해 접근 제한자(Access Modifier)를 사용한다.
- 접근 제한자는 크게 3가지로 구분할 수 있다.

TIP

(default)는 접근 제한자가 붙지 않은 상태를 의미한다.

접근 제한자	제한대상	제한범위
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능
(default)	클래스, 필드, 생성자, 메소드	같은 패키지에서만 사용 가능
private	필드, 생성자, 메소드	객체 내부에서만 사용 가능



객체지향 프로그래밍 (OOP) - 접근 제한자

- 클래스는 public과 default 접근제한을 가질 수 있다.

```
class A {                                public class B {  
  
}
```

- 클래스 A는 같은 패키지에서만 사용 가능하며, 클래스 B는 다른 패키지에서도 사용할 수 있다.

```
package c;  
  
import a.*;  
import b.*;  
  
public class C {  
    // A a; The type A is not visible  
    B b;  
}
```



객체지향 프로그래밍 (OOP) - 접근 제한자

- 생성자는 public, default, protected, private 접근 제한을 가질 수 있다.

```
package a;

public class ClassName {
    byte a;
    short b;
    int c;

    public ClassName(byte a) { }

    ClassName(short b) { }

    private ClassName(int c) { }

    ClassName a1 = new ClassName(a);
    ClassName a2 = new ClassName(b);
    ClassName ac = new ClassName(c);
}
```

```
package a;

public class A {
    byte a;
    short b;
    int c;
    ClassName a1 = new ClassName(a);
    ClassName a2 = new ClassName(b);
    // ClassName a3 = new ClassName(c);
}

package b;

import a.ClassName;

public class B {
    byte a;
    short b;
    int c;
    ClassName a1 = new ClassName(a);
    // ClassName a2 = new ClassName(b);
    // ClassName a3 = new ClassName(c);
}
```

TIP

protected는 상속에서 알아본다.



객체지향 프로그래밍 (OOP) - 접근 제한자

- 필드와 메소드는 public, default, protected, private 접근 제한을 가질 수 있다.

```
package a;

public class ClassName {
    public int field1;
    int field2;
    private int field3;

    public ClassName() {
        field1 = 1;
        field2 = 1;
        field3 = 1;
        method1();
        method2();
        method3();
    }

    public void method1() {}
    void method2() {}
    private void method3() {}
}
```

```
package a;

public class A {
    public void methodA() {
        ClassName cn = new ClassName();
        cn.field1 = 1;
        cn.field2 = 1;
        // cn.field3 = 1;
        cn.method1();
        cn.method2();
        cn.method3();
    }
}
```

TIP

protected는 상속에서 알아본다.

```
package b;

import a.ClassName;

public class B {
    public void methodB() {
        ClassName cn = new ClassName();
        cn.field1 = 1;
        // cn.field2 = 1;
        // cn.field3 = 1;
        cn.method1();
        // cn.method2();
        // cn.method3();
    }
}
```



객체지향 프로그래밍 (OOP) - Getter, Setter

- 객체의 필드를 외부에서 마음대로 읽고 변경하게 되면, 객체의 무결성이 깨질 수 있다.
- 예를 들어, 자동차의 속력은 음수가 될 수 없는데, 외부에서 음수로 변경하면 객체의 무결성에 위배된다.

```
Car myCar = new Car();  
myCar.speed = -100;
```

- 이러한 문제점으로 인해, 객체지향 프로그래밍에서는 외부에서 직접적인 필드의 접근을 차단한다. 그 대신, 메소드를 통해 필드에 접근할 수 있도록 한다.
- 메소드를 통한 필드 접근을 하면, 메소드가 데이터 값을 검증해서 유효한 값만 필드 값으로 저장할 수 있게 만들 수 있다.
- 이러한 메소드를 Setter라고 한다.

```
private int speed;  
  
public void setSpeed(int speed) {  
    this.speed = speed > 0 ? speed : 0;  
}
```



객체지향 프로그래밍 (OOP) - Getter, Setter

- 외부에서 객체의 필드를 읽을 때에도 메소드가 필요한 경우가 있다.
- 필드 값이 객체 외부에서 사용하기에 적절하지 않은 경우, 메소드를 이용해 적절한 형태로 변환해서 반환할 수 있기 때문이다.
- 이러한 메소드를 Getter라고 한다.

```
private int speed;
```

```
public void setSpeed(int speed) {  
    this.speed = speed > 0 ? speed : 0;  
}
```

```
public int getSpeed() {  
    return speed;  
}
```



객체지향 프로그래밍 (OOP) - Getter, Setter

- Getter와 Setter의 기본 작성법은 아래와 같다.

```
public class ClassName {  
    private int fieldName1;  
    private boolean fieldName2;  
  
    public int getFieldName1() {  
        return fieldName1;  
    }  
  
    public void setFieldName1(int fieldName1) {  
        this.fieldName1 = fieldName1;  
    }  
  
    public boolean isFieldName2() {  
        return fieldName2;  
    }  
  
    public void setFieldName2(boolean fieldName2) {  
        this.fieldName2 = fieldName2;  
    }  
}
```

TIP

이클립스는 선언된 필드에 대해 자동으로 Getter, Setter를 생성시킬 수 있다.

[Source] - [Generate Getters and Setters]

TIP

필드 타입이 boolean인 경우,
Getter는 get으로 시작하지 않고 is로 시작하는 것이 관례이다.



객체지향 프로그래밍 (OOP) - 싱글톤 패턴

- 클래스의 객체를 단 하나만 생성하고, 해당 객체를 어디서든지 접근할 수 있도록 하는 디자인 패턴이 싱글톤 패턴이다.
- 주로 애플리케이션 전체에서 하나의 객체만 존재해야 하는 경우 사용한다.
- 싱글톤 패턴의 핵심은 생성자를 private으로 접근을 제한하여 외부에서 new 연산자로 객체를 생성할 수 없도록 하는 것이다.
- 외부에서 생성자 호출이 불가능해지고, 정적 메소드를 통해 간접적으로 객체를 얻는 것만이 가능해진다.
- 싱글톤 패턴을 사용하는 이유
 1. 공통된 자원을 관리하거나 설정 값을 유지할 수 있다.
 2. 불필요한 객체 생성을 방지하여 메모리를 절약할 수 있다.



객체지향 프로그래밍 (OOP) - 싱글톤 패턴

```
package com.singleton;

public class Singleton {
    private static Singleton instance = new Singleton();

    // private 생성자
    private Singleton() {}

    // 인스턴스를 얻는 정적 메소드
    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

```
package com.example;

import com.singleton.Singleton;

public class SingletonExample {

    public static void main(String[] args) {
        // Singleton instance = new Singleton();
        Singleton instance1 = Singleton.getInstance();
        Singleton instance2 = Singleton.getInstance();
        System.out.println(instance1 == instance2);
    }
}
```



객체지향 프로그래밍 (OOP) [실습]

- 현실 세계의 회원을 Member 클래스로 모델링하려 한다.
- 회원의 데이터로는 이름, 아이디, 비밀번호, 나이가 있다.
- 이 데이터들을 가지는 Member 클래스를 선언해보자.

데이터 이름	필드 이름	타입
이름	name	문자열
아이디	id	문자열
패스워드	password	문자열
나이	age	정수

```
package com.oop.practice;
```

```
public class Member {
```

```
}
```



객체지향 프로그래밍 (OOP) [실습]

- 앞에서 생성한 Member 클래스에 생성자를 추가하고자 한다.
- name 필드와 id 필드를 외부에서 받은 값으로 초기화할 수 있도록 생성자를 선언해보자.

```
package com.oop.practice;
```

```
public class Member {  
    String name, id, password;  
    int age;  
}
```




객체지향 프로그래밍 (OOP) [실습]

- 아래 코드와 내용을 파악하고, MemberService 클래스를 생성해보자.
 - login() 메소드는 매개값 id가 "hong", 매개값 password가 "12345"일 경우에만 true로 리턴
 - logout() 메소드는 id + "님이 로그아웃 되었습니다."가 콘솔에 출력

```
package com.oop.practice;

public class MemberExample {
    public static void main(String[] args) {
        MemberService ms = new MemberService();
        boolean result = ms.login("hong", "12345");
        if (result) {
            System.out.println("로그인 되었습니다.");
            ms.logout("hong");
        } else {
            System.out.println("id 또는 password가 올바르지 않습니다.");
        }
    }
}
```



객체지향 프로그래밍 (OOP) [실습]

- Printer 클래스의 println() 메소드는 매개값을 콘솔에 출력한다.
- 매개값으로는 int, Boolean, double, String 값을 줄 수 있다.
- 아래의 조건과 예제 코드를 보고 Printer 클래스에서 println() 메소드를 선언해보자.

```
package com.oop.practice;

public class PrinterExample {
    public static void main(String[] args) {
        Printer p = new Printer();
        p.println(10);
        p.println(true);
        p.println(5.7);
        p.println("홍길동");
    }
}
```



객체지향 프로그래밍 (OOP) [실습]

- Printer 객체를 생성하지 않고도 아래와 같이 `println()` 메소드를 호출할 수 있도록 `Printer` 클래스를 수정해보자

```
package com.oop.practice;

public class PrinterExample {
    public static void main(String[] args) {
        Printer.println(10);
        Printer.println(true);
        Printer.println(5.7);
        Printer.println("홍길동");
    }
}
```



객체지향 프로그래밍 (OOP) [실습]

- 은행 계좌 객체인 Account 객체는 잔고(balance) 필드를 가지고 있다.
- balance 필드는 음수값이 될 수 없고, 최대 1,000,000원까지만 저장할 수 있다. $[0 \leq \text{balance} \leq 1,000,000]$
- 외부에서 balance 필드를 마음대로 변경할 수 없고, 범위내의 값만 가질 수 있도록 Account 클래스를 작성해보자.

1. Getter와 Setter를 이용
2. 0과 1,000,000은 MIN_BALANCE와 MAX_BALANCE 상수를 선언해서 이용
3. 만약 balance를 범위밖의 값으로 설정하게 되면, 기존 balance 값을 유지

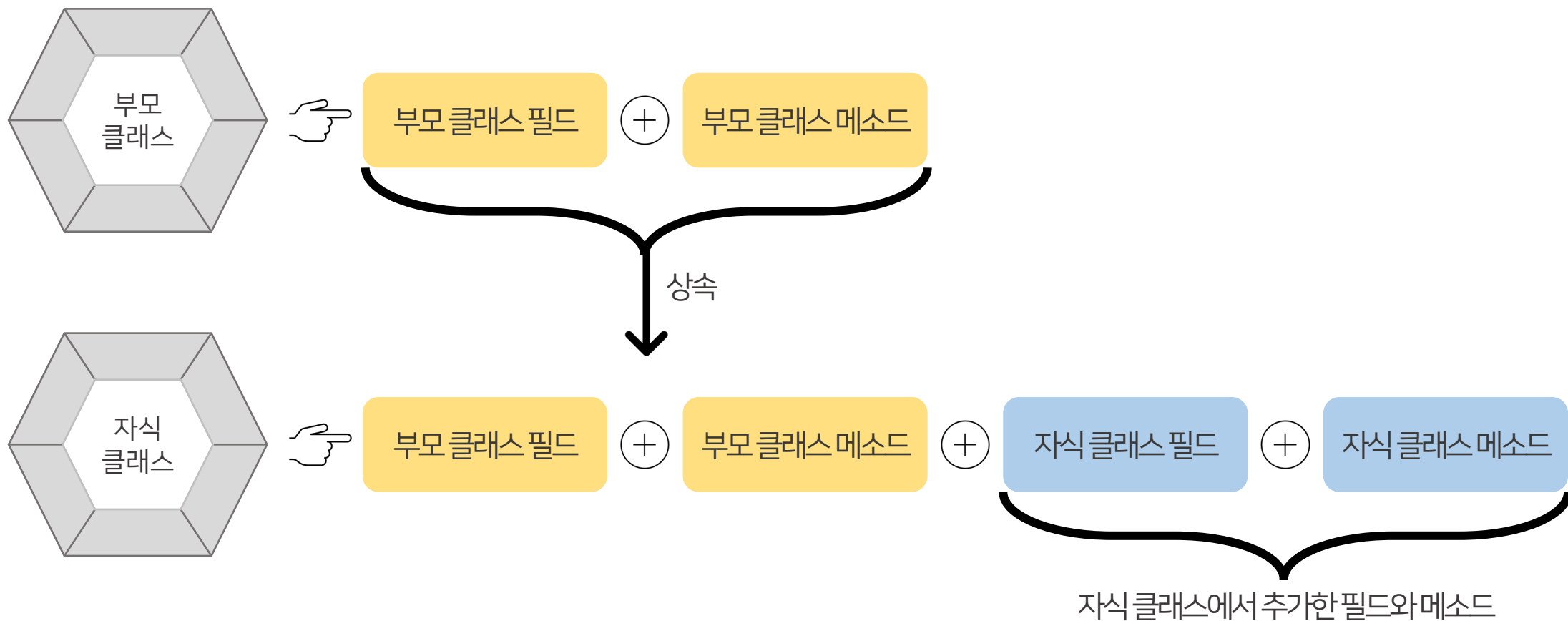
```
package com.oop.practice;
```

```
public class Bank {  
    public static void main(String[] args) {  
        Account acc = new Account();  
  
        acc.setBalance(10000);  
        System.out.println("현재 잔고: " + acc.getBalance()); // 현재 잔고: 10000  
  
        acc.setBalance(-100);  
        System.out.println("현재 잔고: " + acc.getBalance()); // 현재 잔고: 10000  
    }  
}
```



객체지향 프로그래밍 (OOP) - 상속

- 상속(Inheritance)은 부모가 자식에게 물려주는 행위로, 객체지향 프로그램에서 부모 클래스의 필드와 메소드를 자식 클래스에게 물려주는 것을 말한다.





객체지향 프로그래밍 (OOP) - 상속

- 상속을 하게 되면, 이미 잘 개발되어 있는 클래스를 재사용해서 새로운 클래스를 만들기 때문에 중복된 코드를 줄여준다.
- 또한 부모 클래스를 수정하면, 상속을 받은 모든 자식 클래스에도 반영이 되기 때문에 클래스의 수정도 최소화할 수 있다.
- 현실에서 상속은 부모가 자식을 선택하지만, 프로그램에서는 자식이 부모를 선택해 상속을 받는다.
- 자식 클래스를 선언할 때 extends 키워드 뒤에 부모 클래스를 기술한다.

```
public class Child extends Parents { }
```

- 다른 언어와는 달리, Java는 다중 상속을 허용하지 않는다. 따라서 하나의 부모 클래스만을 상속할 수 있다.



객체지향 프로그래밍 (OOP) - 상속

```
package com.oop.inheritance;
```

```
public class Parents {  
    String eyeColor = "갈색";  
  
    void walk() {  
        System.out.println("뒤뚱뒤뚱");  
    }  
}
```

```
package com.oop.inheritance;
```

```
public class Child extends Parents {  
    String job = "개발자";  
  
    void hello() {  
        System.out.println("안녕하세요");  
    }  
}
```

```
package com.oop.inheritance;
```

```
public class InheritanceExample {
```

```
    public static void main(String[] args) {  
        Parents parents = new Parents();  
        System.out.println(parents.eyeColor);  
        parents.walk();  
  
        Child child = new Child();  
        System.out.println(child.eyeColor);  
        System.out.println(child.job);  
        child.walk();  
        child.hello();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 상속

```
package com.oop.inheritance2;

public class Phone {
    public String model, color;

    public void bell() {
        System.out.println("벨이 울립니다.");
    }

    public void sendVoice(String message) {
        System.out.println("본인: " + message);
    }

    public void receiveVoice(String message) {
        System.out.println("상대방: " + message);
    }

    public void hangUp() {
        System.out.println("전화를 끊습니다.");
    }
}
```

```
package com.oop.inheritance2;

public class SmartPhone extends Phone {
    public boolean wifi;

    public SmartPhone(String model, String color) {
        this.model = model;
        this.color = color;
    }

    public void setWifi(boolean wifi) {
        if (wifi) {
            System.out.println("와이파이를 켭니다.");
        } else {
            System.out.println("와이파이를 끕니다.");
        }
        this.wifi = wifi;
    }

    public void internet() {
        System.out.println("인터넷에 연결합니다.");
    }
}
```




객체지향 프로그래밍 (OOP) - 상속

```
package com.oop.inheritance2;
```

```
public class SmartPhoneExample {  
    public static void main(String[] args) {  
        SmartPhone myPhone = new SmartPhone("갤럭시", "흰색");  
        System.out.println("모델:" + myPhone.model + ", 색상:" + myPhone.color);  
        System.out.println("와이파이 상태: " + myPhone.wifi);  
  
        myPhone.bell();  
        myPhone.sendVoice("여보세요");  
        myPhone.receiveVoice("안녕하세요. 저는 홍길동인데요.");  
        myPhone.sendVoice("네, 반갑습니다.");  
        myPhone.hangUp();  
  
        myPhone.setWifi(!myPhone.wifi);  
        myPhone.internet();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 상속

- 부모가 없는 자식이 있을 수 없다. 따라서 자식 객체를 생성하면 부모 객체가 먼저 생성되고 나서 자식 객체가 생성된다.
- 앞선 코드는 SmartPhone 객체만 생성되는 것처럼 보이지만, 사실은 부모인 Phone 객체가 먼저 생성되고 그 다음에 SmartPhone 객체가 생성된 것이다.
- 모든 객체는 생성자를 호출해야만 생성되는데, Phone 객체가 어떻게 생성된 것일까?
 - 바로 자식 생성자에 이 비밀이 숨어있다.
 - 자식 생성자 내부에는 `super()`가 숨겨져 있고, `super()`가 동작하면서 부모 생성자가 호출이 된다.
 - 현재 부모 생성자는 작성하지 않았기 때문에 컴파일 과정에서 기본 생성자가 자동으로 추가되어 있다.

```
public class SmartPhone extends Phone {  
    ...  
    public SmartPhone(String model, String color) {  
        super();  
        this.model = model;  
        this.color = color;  
    }  
    ...  
}
```

TIP

`super()`는 컴파일 과정에서 자동으로 추가된다.



객체지향 프로그래밍 (OOP) - 상속

- 앞서 작성한 Phone 클래스와 SmartPhone 클래스를 수정해보고 다시 실행해보자.

```
public class Phone {  
    ...  
    public Phone() {  
        System.out.println("Phone 생성자가 호출됩니다.");  
    }  
    ...  
}  
  
public class SmartPhone {  
    ...  
    public SmartPhone(String model, String color) {  
        System.out.println("SmartPhone 생성자가 호출됩니다.");  
        this.model = model;  
        this.color = color;  
    }  
    ...  
}
```



객체지향 프로그래밍 (OOP) - 상속

- 만약 부모 클래스에서 매개변수를 갖는 생성자가 있고, 기본 생성자를 생성하지 않았다면 `super(매개값, ...)` 형태로 코드를 직접 넣어야 한다.
- 해당 코드는 매개값의 타입과 개수가 일치하는 부모 생성자를 찾아 호출하게 된다.

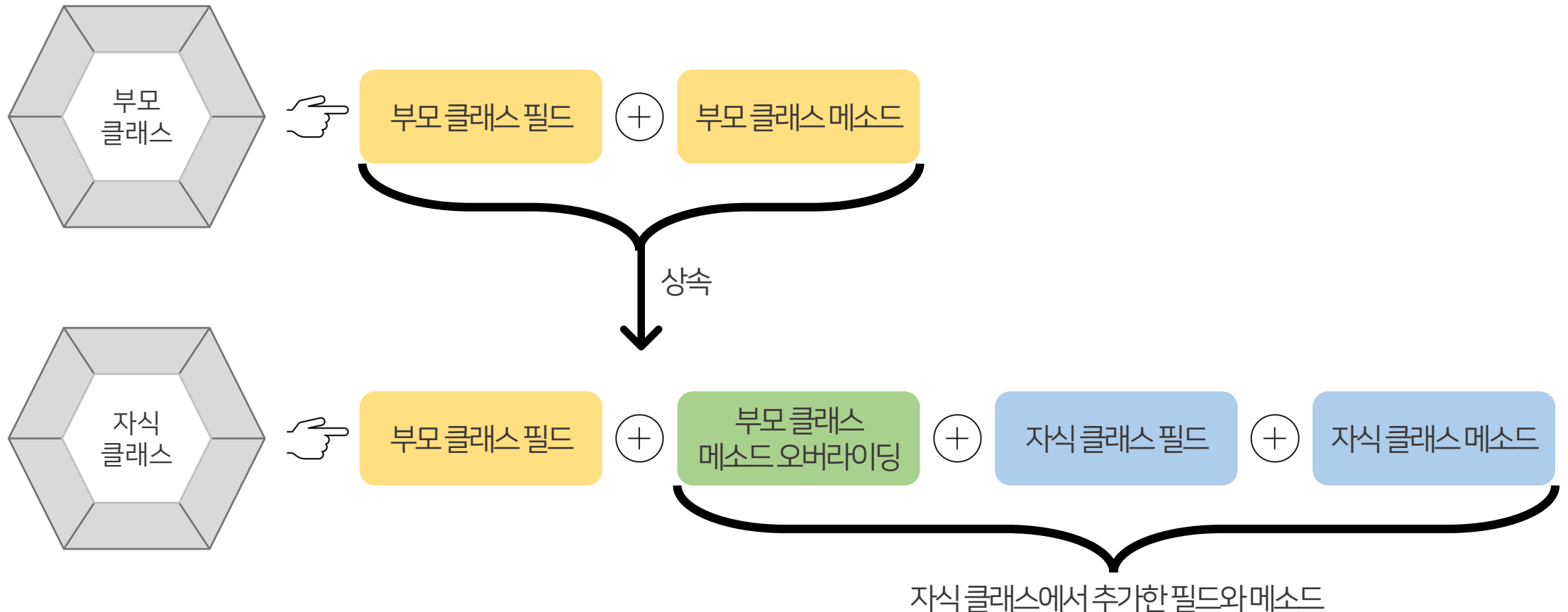
- 다시 Phone, SmartPhone 클래스를 수정해보자.

```
public class Phone {  
    ...  
    public Phone(String model, String color) {  
        System.out.println("Phone 생성자가 호출됩니다.");  
        this.model = model;  
        this.color = color;  
    }  
    ...  
}  
  
public class SmartPhone {  
    ...  
    public SmartPhone(String model, String color) {  
        super(model, color);  
        System.out.println("SmartPhone 생성자가 호출됩니다.");  
    }  
    ...  
}
```



객체지향 프로그래밍 (OOP) - 오버라이딩

- 부모 클래스의 모든 메소드가 자식 클래스에 알맞게 설계되어 있다면 가장 이상적이겠지만, 어떤 메소드는 자식 클래스가 사용하기에 적합하지 않을 수 있다.
- 이러한 메소드는 자식 클래스에서 재정의해서 사용해야 한다. 이것을 메소드 오버라이딩(Method Overriding)이라 한다.





객체지향 프로그래밍 (OOP) - 오버라이딩

- 상속된 메소드를 자식 클래스에서 재정의(오버라이딩)하게 되면, 해당 부모 메소드는 숨겨지고 자식 메소드가 우선적으로 사용된다.

```
public class Parents {  
    void method1() {...}  
    void method2() {...}  
}
```

```
public class Child extends Parents {  
    void method2() {...}  
    void method3() {...}  
}
```

```
public class InheritanceExample {  
  
    public static void main(String[] args) {  
        Child child = new Child();  
  
        child.method1();  
  
        child.method2();  
  
        child.method3();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 오버라이딩

- 메소드 오버라이딩을 할 때에는 몇 가지 규칙을 지켜야 한다.

- 부모 메소드의 선언부(리턴 타입, 메소드 이름, 매개변수)와 동일해야 한다.
- 접근 제한을 더 강하게 오버라이딩하는 것은 불가능하다.
- 새로운 예외를 throws 할 수 없다.

TIP

public > protected > (default) > private 순으로 접근 제한이 강해진다.

TIP

예외처리는 나중에 학습한다.



객체지향 프로그래밍 (OOP) - 오버라이딩

```
package com.oop.inheritance3;
```

```
public class Calculator {  
    public double getCircleArea(double r) {  
        System.out.println("Calculator 객체에서 원의 넓이를 구한다.");  
        return 3.14 * r * r;  
    }  
}
```

```
package com.oop.inheritance3;
```

```
public class Computer extends Calculator {  
    @Override  
    public double getCircleArea(double r) {  
        System.out.println("Computer 객체에서 원의 넓이를 구한다.");  
        return Math.PI * r * r;  
    }  
}
```

TIP

@Override 어노테이션
컴파일 시 정확히 오버라이딩되었는지 체크해준다. (생략 가능)

TIP

이클립스는 오버라이딩 메소드 자동 생성을 지원한다.
자식 클래스에서 Source - Override/Implement Methods 선택



객체지향 프로그래밍 (OOP) - 오버라이딩

```
package com.oop.inheritance3;
```

```
public class ComputerExample {
```

```
    public static void main(String[] args) {  
        int r = 10;
```

```
        Calculator calc = new Calculator();  
        System.out.println(calc.getCircleArea(r));
```

```
        Computer com = new Computer();  
        System.out.println(com.getCircleArea(r));
```

```
    }
```

```
}
```




객체지향 프로그래밍 (OOP) - 오버라이딩

- 메소드 오버라이딩을 사용하면, 부모 메소드 대신 자식 메소드가 사용된다.
- 따라서 부모 메소드에서 약간의 코드를 추가하여 재정의하려는 경우, 부모 메소드와 중복된 내용을 자식 메소드에 작성해야 한다.
- 이러한 문제는 자식 메소드 내에서 부모 메소드를 호출하는 것으로 해결할 수 있다.
 - 자식 메소드 내에서 `super` 키워드와 객체 접근 연산자(`.`)를 활용하면 숨겨진 부모 메소드를 호출할 수 있다.

```
public class Parents {  
    void method() {...}  
}
```

```
public class Child extends Parents {  
    @Override  
    void method() {  
        super.method();  
        ...  
    }  
}
```





객체지향 프로그래밍 (OOP) - 오버라이딩

- 메소드 오버라이딩을 사용하면, 부모 메소드 대신 자식 메소드가 사용된다.
- 따라서 부모 메소드에서 약간의 코드를 추가하여 재정의하려는 경우, 부모 메소드와 중복된 내용을 자식 메소드에 작성해야 한다.
- 이러한 문제는 자식 메소드 내에서 부모 메소드를 호출하는 것으로 해결할 수 있다.
 - 자식 메소드 내에서 `super` 키워드와 객체 접근 연산자(`.`)를 활용하면 숨겨진 부모 메소드를 호출할 수 있다.

```
package com.oop.inheritance3;

public class Airplane {
    public void land() {
        System.out.println("착륙합니다.");
    }
    public void fly() {
        System.out.println("일반 비행합니다.");
    }
    public void takeOff() {
        System.out.println("이륙합니다.");
    }
}
```

```
package com.oop.inheritance3;

public class SupersonicAirplane extends Airplane {
    public static final int NORMAL = 1;
    public static final int SUPERSONIC = 2;
    public int flyMode = NORMAL;

    @Override
    public void fly() {
        if(flyMode == SUPERSONIC) {
            System.out.println();
        } else {
            super.fly();
        }
    }
}
```



객체지향 프로그래밍 (OOP) - final

```
package com.oop.inheritance3;
```

```
public class FlyExample {  
    public static void main(String[] args) {  
        SupersonicAirplane sa = new SupersonicAirplane();  
        sa.takeOff();  
        sa.fly();  
        sa.flyMode = SupersonicAirplane.SUPERSONIC;  
        sa.fly();  
        sa.land();  
    }  
}
```



객체지향 프로그래밍 (OOP) - final

- 필드 선언시에 final을 붙이면 초기값 설정 후 값을 변경할 수 없는 상수가 되었다.
- 클래스와 메소드에 final을 붙이면 어떻게 될 지 알아보자.
- final 클래스
 - 클래스에 final을 붙이면 상속할 수 없는 클래스가 된다.
 - 즉, 부모 클래스가 될 수 없다.
- final 메소드
 - 메소드에 final을 붙이면 오버라이딩할 수 없는 메소드가 된다.
 - 즉, 자식 클래스에서 물려받은 그대로 쓸 수 밖에 없는 메소드가 된다.



객체지향 프로그래밍 (OOP) - protected

- 접근제한자를 통해 클래스에 접근할 수 있는 범위를 결정하거나 객체 외부에서 필드, 생성자, 메소드의 접근 여부를 결정했다.
- protected 접근 제한자는 필드, 생성자, 메소드에서 쓰이는 접근 제한자로 상속과 관련이 있다.
- public은 패키지 관계없이 어디서든 사용 가능하고, (default)는 다른 패키지에서의 접근을 제한한다.
- protected는 그 중간에 해당하며, 같은 패키지이거나 다른 패키지라면 자식 클래스만 접근할 수 있도록 허용한다.

접근 제한자	제한 대상	제한 범위
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능
(default)	클래스, 필드, 생성자, 메소드	같은 패키지에서만 사용 가능
private	필드, 생성자, 메소드	객체 내부에서만 사용 가능



객체지향 프로그래밍 (OOP) - protected

```
package a;
```

```
public class ClassName {  
    protected int field = 1;  
  
    protected ClassName() { }  
  
    protected void method() {}  
}
```

```
package b;  
import a.ClassName;
```

```
public class B {  
    // ClassName cn = new ClassName();  
    // int value = cn.field;  
    // cn.method();  
}
```

```
package a;
```

```
public class A {  
    public void methodA() {  
        ClassName cn = new ClassName();  
        int value = cn.field;  
        cn.method();  
    }  
}
```

```
package c;  
import a.ClassName;
```

```
public class C extends ClassName {  
    public C() {  
        super();  
    }  
    public void methodC() {  
        super.method();  
        this.field = super.field;  
    }  
}
```



객체지향 프로그래밍 (OOP) - 타입 변환

- 이미 기본타입의 변환(Promotion, Casting)에 대해서 학습한 적이 있다.
- 클래스도 타입 변환이 있는데, 클래스의 타입 변환은 상속 관계에 있는 클래스 사이에서 발생한다.

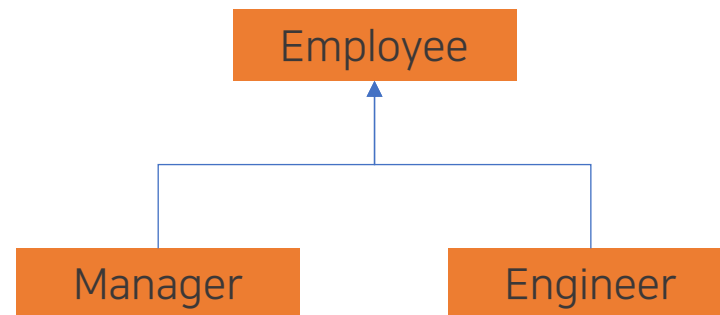
```
int intValue = 10;  
long longValue = intValue;
```

```
Manager mng = new Manger();  
Employee emp = mng;
```

```
Employee emp = new Employee();  
Manager mng = new Manager();  
Engineer eng = new Engineer();
```



```
Employee emp = new Employee();  
Employee mng = new Manager();  
Employee eng = new Engineer();
```





객체지향 프로그래밍 (OOP) - 타입 변환

- 자동 타입 변환(Promotion)

```
Parents p = new Child();
```

- 자식은 부모의 특징과 기능을 상속받기 때문에 부모와 동일하게 취급될 수 있다.
- 부모 타입으로 변환한 경우에는 부모 클래스에 선언된 필드와 메소드만 접근할 수 있게 된다.
- 그러나 자식 클래스에서 오버라이딩된 메소드가 있다면, 오버라이딩된 메소드가 호출된다.
- 즉, 인스턴스 변수 p는 아래의 필드와 메소드를 갖는다.
 - 부모 클래스의 필드
 - 부모 클래스의 메소드, 자식 클래스가 오버라이딩한 메소드



객체지향 프로그래밍 (OOP) - 타입 변환

```
package com.oop.inheritance;
```

```
public class Child extends Parents {  
    String job = "개발자";  
    void hello() {  
        System.out.println("안녕하세요");  
    }  
    @Override  
    void walk() {  
        System.out.println("뚜벅뚜벅");  
    }  
}
```

```
package com.oop.inheritance;
```

```
public class Parents {  
    String eyeColor = "갈색";  
    void walk() {  
        System.out.println("뒤뚱뒤뚱");  
    }  
    void eat() {  
        System.out.println("냠냠");  
    }  
}
```

```
package com.oop.inheritance;
```

```
public class InheritanceExample {  
    public static void main(String[] args) {  
        Parents p1 = new Parents();  
        System.out.println(p1.eyeColor);  
        p1.eat();  
        p1.walk();  
  
        System.out.println();  
  
        Parents p2 = new Child();  
        System.out.println(p2.eyeColor);  
        p2.eat();  
        p2.walk();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 타입 변환

- 강제타입 변환(Casting)

```
Parents p = new Child();  
Child c = (Child) p;
```

- 자식타입은 부모타입으로 자동 변환되지만, 반대로 부모타입은 자식 타입으로 자동 변환되지 않는다.
- 부모타입 객체를 자식 타입으로 무조건 강제 변환할 수는 없다.
- 자식 객체가 부모타입으로 자동 변환된 경우에만 강제타입 변환을 사용할 수 있다.
- 자식 타입에 선언된 필드와 메소드를 사용해야 한다면, 강제타입 변환을 해서 이용해야 한다.



객체지향 프로그래밍 (OOP) - 타입 변환

```
package com.oop.inheritance;
```

```
public class InheritanceExample {  
    public static void main(String[] args) {  
        Parents p1 = new Parents();  
        System.out.println(p1.eyeColor);  
        p1.eat();  
        p1.walk();  
  
        System.out.println();  
  
        Parents p2 = new Child();  
        System.out.println(p2.eyeColor);  
        p2.eat();  
        p2.walk();  
  
        Child c = (Child) p2;  
        System.out.println(c.job);  
        c.hello();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 다형성

- 다형성은 상속(타입 변환)을 전제로, 하나의 객체 타입이 여러 타입을 참조할 수 있는 능력을 의미한다.
- 동일한 사용방법이지만 다양한 실행 결과를 나타내게 하는 성질을 가지고 있다.
- 배열에 다형성 적용하기
 - 일반적으로 배열은 같은 타입만을 저장할 수 있지만, 다형성을 이용하면 다른 타입도 저장할 수 있게 된다.

```
Employee[] emp = { new Employee("홍길동"), new Manager("이순신"), new Engineer("유관순") };
```



객체지향 프로그래밍 (OOP) - 다형성

- 메서드 매개변수에 다형성 적용하기
 - 메서드의 매개변수에 다형성을 적용시키면, 메서드를 호출할 때 다른 타입도 저장할 수 있어서 오버로딩을 사용하지 않아도 된다.

```
package com.oop.polymorphism;

public class Tire {
    public void roll() {
        System.out.println("굴러갑니다.");
    }
}
```

```
package com.oop.polymorphism;

public class HankookTire extends Tire {
    @Override
    public void roll() {
        System.out.println("한국타이어가 굴러갑니다.");
    }
}
```

```
package com.oop.polymorphism;

public class KumhoTire extends Tire {
    @Override
    public void roll() {
        System.out.println("금호타이어가 굴러갑니다.");
    }
}
```



객체지향 프로그래밍 (OOP) - 다형성

```
package com.oop.polymorphism;
```

```
public class Car {  
    private Tire tire;  
    public Tire getTire() {  
        return tire;  
    }  
  
    public void setTire(Tire tire) {  
        this.tire = tire;  
    }  
  
    public void drive() {  
        tire.roll();  
    }  
}
```

```
package com.oop.polymorphism;
```

```
public class CarExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
  
        myCar.setTire(new Tire());  
        myCar.drive();  
  
        System.out.println();  
  
        myCar.setTire(new HankookTire());  
        myCar.drive();  
  
        myCar.setTire(new KumhoTire());  
        myCar.drive();  
    }  
}
```



객체지향 프로그래밍 (OOP) - instanceof

- 변수가 참조하는 객체의 타입을 확인하고자 할 때, instanceof 연산자를 이용할 수 있다.
- instanceof 연산자의 좌측에는 객체, 우측에는 타입을 기술하면 해당 객체와 타입이 일치하는지 여부를 boolean으로 반환한다.

boolean result = instance **instanceof** Type;

```
public class HankookTire extends Tire {
    String HankookStyle = "한국";
    @Override
    public void roll() {
        System.out.println("한국타이어가 굴러갑니다.");
    }
}
```

```
public class Car {
    private Tire tire;
    public Tire getTire() {
        return tire;
    }

    public void setTire(Tire tire) {
        this.tire = tire;
    }

    public void drive() {
        if(tire instanceof HankookTire) {
            HankookTire hkTire = (HankookTire) tire;
            System.out.println(hkTire.HankookStyle);
        }
        tire.roll();
    }
}
```




객체지향 프로그래밍 (OOP) - instanceof

- Java 12부터는 instanceof 연산의 결과가 true일 경우, 우측 타입의 변수를 사용할 수 있어서 강제 타입 변환을 할 필요가 없다.

```
public void drive() {  
    if(tire instanceof HankookTire) {  
        HankookTire hkTire = (HankookTire) tire;  
        System.out.println(hkTire.HankookStyle);  
    }  
    tire.roll();  
}
```



```
public void drive() {  
    if(tire instanceof HankookTire hkTire) {  
        System.out.println(hkTire.HankookStyle);  
    }  
    tire.roll();  
}
```



객체지향 프로그래밍 (OOP) - Object 클래스

- Object 클래스는 Java에서 최상위 클래스이다.
- 즉, Java의 모든 클래스는 Object의 자식이거나 자손 클래스가 된다.
- 클래스를 생성할 때 명시적으로 extend를 작성하지 않은 경우, 자동으로 Object를 상속받게 된다.
- Object 클래스는 다양한 메소드들로 구성되어 있고, 해당 메소드들은 모든 클래스에서 이용할 수 있다.
 - equals() 메소드, hashCode() 메소드, toString() 메소드 등...



객체지향 프로그래밍 (OOP) - 추상 클래스

- 사전적 의미의 추상(Abstract)은 실체 간의 공통되는 특성을 추출한 것을 말한다.
- 객체를 생성할 수 있는 클래스를 실제 클래스라고 한다면, 실제 클래스들의 공통적인 필드나 메소드를 추출해서 선언한 클래스를 추상 클래스라 한다.
- 추상 클래스는 실제 클래스의 부모 역할을 한다.

추상 클래스

Animal



실체 클래스

Pig



Monkey



Panda



Rabbit





객체지향 프로그래밍 (OOP) - 추상 클래스

- 클래스 선언에 `abstract` 키워드를 붙이면 추상 클래스가 된다.

```
package com.oop.abstract1;
```

```
public abstract class Animal { }
```

- 추상 클래스는 직접 객체로 생성할 수 없다.

```
Animal animal = new Animal();
```

```
public class Pig extends Animal { }
```

```
public class Monkey extends Animal { }
```

```
public class Panda extends Animal { }
```

```
public class Rabbit extends Animal { }
```



객체지향 프로그래밍 (OOP) - 추상 메소드

- 자식 클래스들이 가지고 있는 공통 메소드를 뽑아내어 추상 클래스로 만들때, 실행내용이 자식마다 달라지는 경우가 있을 수 있다.
- 이런 경우를 해결하기 위해, 추상 클래스에는 추상 메소드를 선언할 수 있도록 되어있다.
- 추상 메소드는 `abstract` 키워드가 붙고, 메소드 실행내용({})이 없는 것이 특징이다.

abstract 리턴타입 메소드명();

- 추상 메소드는 공통 메소드라는 것만 정의할 뿐, 실행내용을 가지지 않는다.

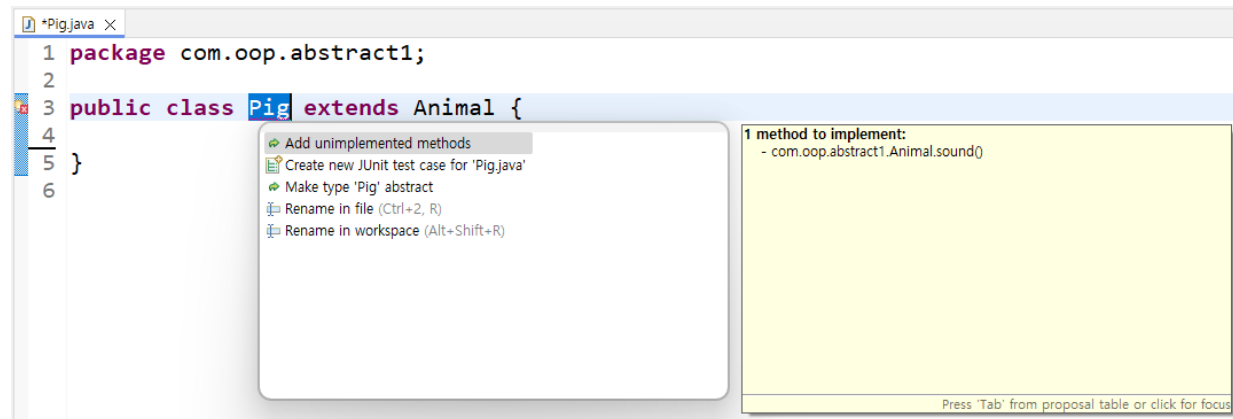
```
package com.oop.abstract1;
```

```
public abstract class Animal {  
    abstract void sound();  
}
```



객체지향 프로그래밍 (OOP) - 추상 메소드

- 실행 내용이 있어야 메소드를 호출할 수 있기 때문에, 자식 클래스에서 메소드를 구현해주어야 한다. 즉, 오버라이딩이 필수이다.



```
package com.oop.abstract1;
```

```
public class Pig extends Animal {  
    @Override  
    void sound() {  
        System.out.println("꿀꿀");  
    }  
}
```

- Monkey, Panda, Rabbit 클래스도 만들어보자.



객체지향 프로그래밍 (OOP) - 추상 메소드

```
public class AnimalExample {
    public static void main(String[] args) {
        Pig p = new Pig();
        Monkey m = new Monkey();
        Panda pd = new Panda();
        Rabbit r = new Rabbit();

        p.sound();
        m.sound();
        pd.sound();
        r.sound();

        System.out.println();
        Animal[] aniArr = {p, m, pd, r};
        for (Animal ani : aniArr) {
            ani.sound();
        }

        System.out.println();
        animalSound(p);
        animalSound(m);
        animalSound(pd);
        animalSound(r);
    }
    public static void animalSound(Animal animal) {
        animal.sound();
    }
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

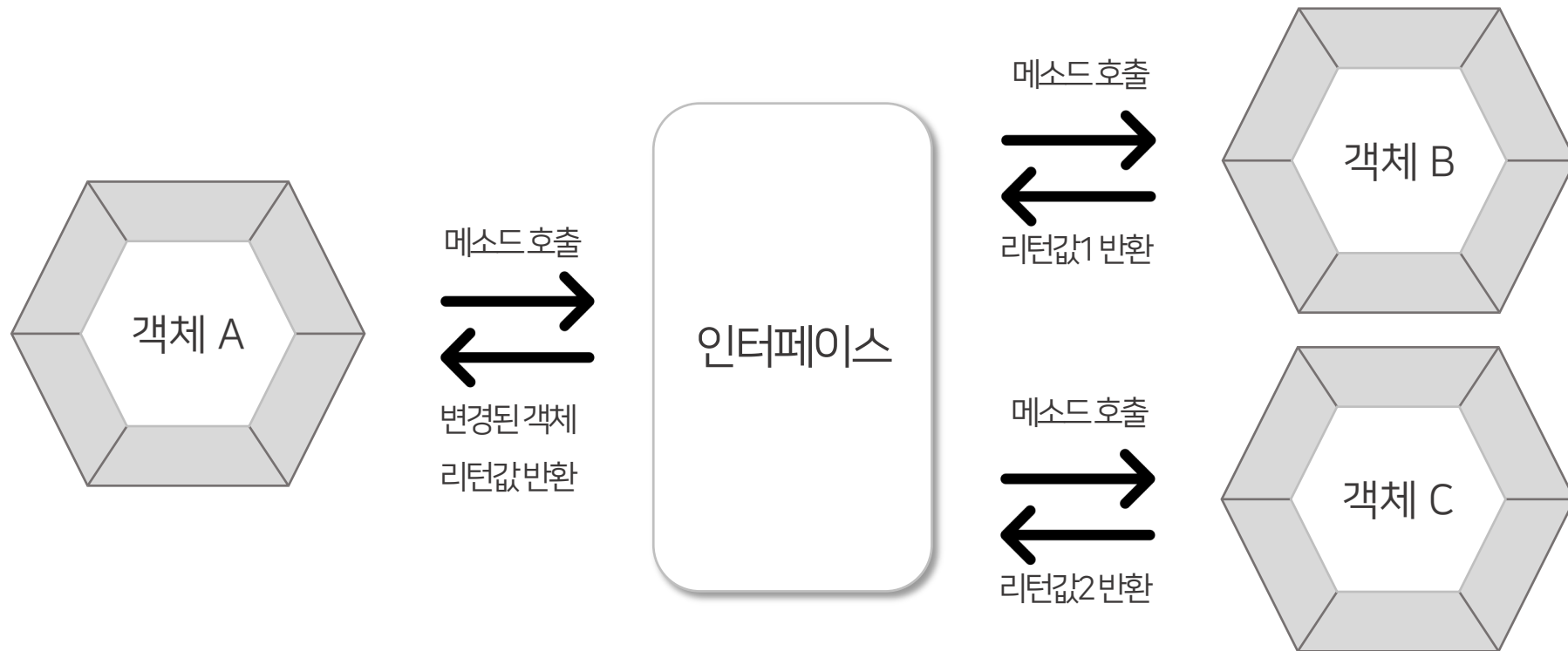
- 인터페이스(interface)는 두 장치를 연결하는 접속기라는 뜻을 가지고 Java에서 인터페이스는 두 객체를 연결하는 역할을 한다.
- 객체 A가 인터페이스의 메소드를 호출하면, 인터페이스는 객체 B의 메소드를 호출하고 그 결과를 받아 객체 A로 전달해준다.
- 객체 B로 직접 호출하는 것이 더욱 간단한데, 왜 인터페이스를 거치도록 해야하는가?





객체지향 프로그래밍 (OOP) - 인터페이스

- 만약 객체 B가 객체 C로 바뀐다고 가정해보자.
- 객체 A에서 직접 호출하는 경우에는 객체 A의 소스 코드에서 객체 B를 객체 C로 변경해줘야 한다.
- 하지만 인터페이스를 통해 호출하면 객체 C로 바뀌더라도 객체 A의 소스 코드를 변경할 필요가 없다.





객체지향 프로그래밍 (OOP) - 인터페이스

- 인터페이스는 다형성 구현에 주로 사용되는 기술이다.
- 상속을 통해 다형성을 구현할 수 있지만, 인터페이스를 통해 구현하는 경우가 더 많다.
- 인터페이스는 .java 파일로 작성되고, .class 파일로 컴파일되기 때문에 물리적 형태는 클래스와 동일하다.

```
interface 인터페이스명 { }  
public interface 인터페이스명 { }
```

- 중괄호 안에는 인터페이스가 가지는 멤버들을 선언할 수 있다.

```
package com.oop.interface1;  
  
public interface RemoteControl {  
    // public 추상 메소드  
    public void turnOn();  
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

- 이제 객체 A는 인터페이스가 가지고 있는 추상 메소드를 호출할 수 있다.
- 그렇다면 객체 B는 인터페이스에 선언되어 있는 추상 메서드를 재정의해서 구현을 해두어야 할 것이다.
- 이때, 객체 B와 같은 구현 객체는 인터페이스를 구현하고 있음을 명시해야 한다.

```
class B implements RemoteControl {  
  
}
```





객체지향 프로그래밍 (OOP) - 인터페이스

```
public class TV implements RemoteControl {  
  
}
```





객체지향 프로그래밍 (OOP) - 인터페이스

```
1 package com.oop.interface1;
2
3 public class TV implements RemoteControl {
4
5 }
6
```

- Add unimplemented methods
- Create new JUnit test case for 'TV.java'
- Make type 'TV' abstract
- Rename in file (Ctrl+2, R)
- Rename in workspace (Alt+Shift+R)

1 method to implement:

- com.oop.interface1.RemoteControl.turnOn()

Press 'Tab' from proposal table or click for focus

```
package com.oop.interface1;

public class TV implements RemoteControl {
    @Override
    public void turnOn() {
        System.out.println("TV를 켭니다.");
    }
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

```
package com.oop.interface1;

public class remoteControlExample {
    public static void main(String[] args) {
        TV tv = new TV();
        tv.turnOn();
    }
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

- 인터페이스도 참조 타입에 속하므로 타입이 될 수 있다.
- 인터페이스를 통해 구현 객체를 사용하려면 인터페이스 변수에 구현 객체를 대입해야 한다.

```
package com.oop.interface1;
```

```
public class remoteControlExample {  
    public static void main(String[] args) {  
        RemoteControl rc = new TV();  
        rc.turnOn();  
  
        rc = new Audio();  
        rc.turnOn();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

- 인터페이스는 `public static final` 특성을 갖는 불변의 상수 필드만을 가질 수 있다.
- 인터페이스에서 선언된 필드는 모두 `public static final` 특성을 갖기 때문에 이를 생략하더라도 자동적으로 컴파일 과정에서 붙게 된다.

```
int MAX_VOLUME = 10;  
public static final int MIN_VOLUME = 0;
```

- 상수는 구현 객체와 관련이 없는 인터페이스 소속 멤버이므로, 바로 접근해서 값을 얻을 수 있다.

```
System.out.println("최고 볼륨: " + RemoteControl.MAX_VOLUME);  
System.out.println("최저 볼륨: " + RemoteControl.MIN_VOLUME);
```




객체지향 프로그래밍 (OOP) - 인터페이스

- 인터페이스는 추상 메서드를 멤버로 가질 수 있다.
- 추상 메서드는 중괄호가 없는 메서드이기 때문에 인터페이스에서 선언된 추상 메서드는 `public abstract`를 생략하더라도 자동으로 컴파일 과정에서 붙게 된다.

```
// public 추상 메서드
public void turnOn();
public abstract void turnOff();
void setVolume(int volume);
```

- 구현 클래스인 TV와 Audio는 인터페이스에 선언된 모든 추상 메서드를 재정의해서, 실행 코드를 가지고 있어야 한다. 추상 메서드를 오버라이딩해서 작성해보자.
 - "TV를 켭니다", "TV를 끕니다"와 같은 메시지를 출력
 - 매개변수 volume이 최대값보다 크다면 최대값으로 설정, 최소값보다 작다면 최소값으로 설정, 범위 내에 있다면 매개변수 값으로 설정 후 현재 볼륨 출력 (`private int volume` 필드 추가)
- 인터페이스의 추상 메서드는 `public` 접근 제한을 갖기 때문에 재정의 시에 `public`이 추가되어 있어야 한다.



객체지향 프로그래밍 (OOP) - 인터페이스

- 인터페이스에는 완전한 실행 코드를 가진 디폴트 메소드를 선언할 수 있다. (Java 8 이상)

```
// default 메소드
default void setMute(boolean mute) {
    if (mute) {
        System.out.println("음소거 되었습니다.");
        setVolume(MIN_VOLUME);
    } else {
        System.out.println("음소거 해제되었습니다.");
    }
}
```

- default 키워드를 반드시 명시해야 인터페이스 내부에서 구현되는 메소드를 만들 수 있다.
- 구현 클래스에서는 디폴트 메소드도 재정의해서 자신에 맞게 오버라이딩 할 수 있다. 오버라이딩을 할 경우에는 public을 반드시 붙여야 하고, default는 생략해야 한다.



객체지향 프로그래밍 (OOP) - 인터페이스

- Audio 구현 클래스에서 setMute() 메소드를 아래와 같이 오버라이딩 해보자.

```
private int memoryVolume;

@Override
public void setMute(boolean mute) {
    if (mute) {
        System.out.println("음소거 되었습니다.");
        this.memoryVolume = this.volume;
        setVolume(MIN_VOLUME);
    } else {
        System.out.println("음소거 해제되었습니다.");
        setVolume(memoryVolume);
    }
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

```
package com.oop.interface1;
```

```
public class remoteControlExample {  
    public static void main(String[] args) {  
        RemoteControl rc = new TV();  
  
        rc.turnOn();  
        rc.setVolume(5);  
        rc.setMute(true);  
        rc.setMute(false);  
        System.out.println();  
  
        rc = new Audio();  
        rc.turnOn();  
        rc.setVolume(20);  
        rc.setMute(true);  
        rc.setMute(false);  
    }  
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

- 인터페이스에는 구현 객체없이 호출되는 정적 메소드 선언도 가능하다.
- 정적 메소드는 public 또는 private으로 제한할 수 있으며, public의 경우에는 접근 제한자 생략이 가능하다.

```
// 정적 메소드 (public 생략 가능)
public static void changeBattery() {
    System.out.println("건전지를 교환합니다.");
}
```

```
package com.oop.interface1;
```

```
public class remoteControlExample {
    public static void main(String[] args) {
        RemoteControl rc = new TV();
        ...
        RemoteControl.changeBattery();
    }
}
```



객체지향 프로그래밍 (OOP) - 인터페이스

- 지금까지 알아본 인터페이스의 상수 필드, 추상 메소드, 디폴트 메소드, 정적 메소드는 모두 public 접근 제한을 갖는다.
- public을 생략하더라도 컴파일 과정에서 public 접근 제한자가 붙어 항상 외부에서의 접근이 가능해진다.
- 이제 인터페이스 외부에서는 접근할 수 없고, 인스턴스 내부에서만 접근할 수 있는 private 메소드를 알아보자.
 - private 메소드 : 디폴트 메소드에서만 호출이 가능
 - private 정적 메소드 : 디폴트 메소드와 정적 메소드 안에서 호출 가능



객체지향 프로그래밍 (OOP) - 인터페이스

```
public interface Service {
    default void defaultMethod1() {
        System.out.println("디폴트 메소드1");
        staticCommon();
        defaultCommon();
    }
    default void defaultMethod2() {
        System.out.println("디폴트 메소드2");
        staticCommon();
        defaultCommon();
    }
    private void defaultCommon() {
        System.out.println("디폴트 메소드 공통 사용");
    }
    static void staticMethod1() {
        System.out.println("정적 메소드1");
        staticCommon();
        // defaultCommon(); private은 디폴트 메소드에서만 가능
    }
    static void staticMethod2() {
        System.out.println("정적 메소드2");
        staticCommon();
        // defaultCommon(); private은 디폴트 메소드에서만 가능
    }
    private static void staticCommon() {
        System.out.println("공통 사용");
    }
}
```

```
public class ServiceImpl implements Service {
}

public class ServiceExample {
    public static void main(String[] args) {
        Service service = new ServiceImpl();

        // 디폴트 메소드 호출
        service.defaultMethod1();
        System.out.println();
        service.defaultMethod2();
        System.out.println();

        // 정적 메소드 호출
        Service.staticMethod1();
        System.out.println();
        Service.staticMethod2();
        System.out.println();
    }
}
```



객체지향 프로그래밍 (OOP) - 다중 인터페이스 구현

- 구현 객체는 하나 이상의 인터페이스를 implements 할 수 있다. implements 뒤에 쉼표(,)를 이용해 구분 작성하면 된다.

```
public class 구현클래스 implements 인터페이스A, 인터페이스B {  
    // 모든 추상 메소드 오버라이딩  
}
```

- 다중 인터페이스를 구현하는 객체이기 때문에 두 인터페이스 타입의 변수에 각각 대입될 수 있다.

```
인터페이스A 변수A = new 구현클래스();  
인터페이스B 변수B = new 구현클래스();
```

- 구현 객체가 어떤 인터페이스 변수에 대입되느냐에 따라 변수를 통해 호출할 수 있는 추상 메소드가 결정된다.



객체지향 프로그래밍 (OOP) - 다중 인터페이스 구현

```
package com.oop.interface3;
```

```
public interface PhoneService {  
    void turnOn();  
    void turnOff();  
    void call(String number);  
}
```

```
public interface SmartService {  
    void openApp(String appName);  
}
```

```
public class SmartPhone implements PhoneService,  
SmartService {  
    @Override  
    public void openApp(String appName) {  
        System.out.println(appName + "을 열었습니다.");  
    }  
    @Override  
    public void turnOn() {  
        System.out.println("스마트폰을 켭니다.");  
    }  
    @Override  
    public void turnOff() {  
        System.out.println("스마트폰을 끕니다.");  
    }  
    @Override  
    public void call(String number) {  
        System.out.println(number + "에 전화합니다.");  
    }  
}
```



객체지향 프로그래밍 (OOP) - 다중 인터페이스 구현

```
public class SmartPhoneExample {  
    public static void main(String[] args) {  
        PhoneService ps = new SmartPhone();  
        ps.turnOn();  
        ps.call("엄마");  
        ps.turnOff();  
  
        System.out.println();  
  
        SmartService ss = new SmartPhone();  
        ss.openApp("카카오톡");  
  
        System.out.println();  
  
        SmartPhone sp = new SmartPhone();  
        sp.turnOn();  
        sp.call("아빠");  
        sp.openApp("유튜브");  
        sp.turnOff();  
    }  
}
```



객체지향 프로그래밍 (OOP) - 인터페이스 상속

- 인터페이스는 다른 인터페이스를 상속할 수 있다. 게다가 클래스와 달리 다중 상속을 허용한다.

public interface 인터페이스 **extends** 부모인터페이스1, 부모인터페이스2 { }

- 해당 인터페이스의 구현 클래스는 인터페이스가진 추상 메소드 뿐만 아니라 부모 인터페이스가 가지고 있는 추상 메소드까지 모두 오버라이딩 해야 한다.
- 그리고 구현 객체는 자식과 부모 인터페이스 변수에 모두 대입될 수 있다.

```
인터페이스 변수A = new 구현클래스();  
부모인터페이스1 변수B = new 구현클래스();  
부모인터페이스2 변수C = new 구현클래스();
```

- 구현 객체가 자식 인터페이스 변수에 대입되면 자식과 부모 인터페이스의 추상 메소드를 모두 호출할 수 있으나, 부모 인터페이스 변수에 대입되면 부모 인터페이스에 선언된 추상 메소드만 호출 가능하다.



객체지향 프로그래밍 (OOP) - 인터페이스 상속

```
package com.oop.interface4;

public interface Mother {
    void methodM();
}

public interface Father {
    void methodF();
}

public interface Child extends Mother, Father {
    void methodC();
}
```

```
public class InterfaceImpl implements Child {
    @Override
    public void methodM() {
        System.out.println("엄마 메소드 실행");
    }

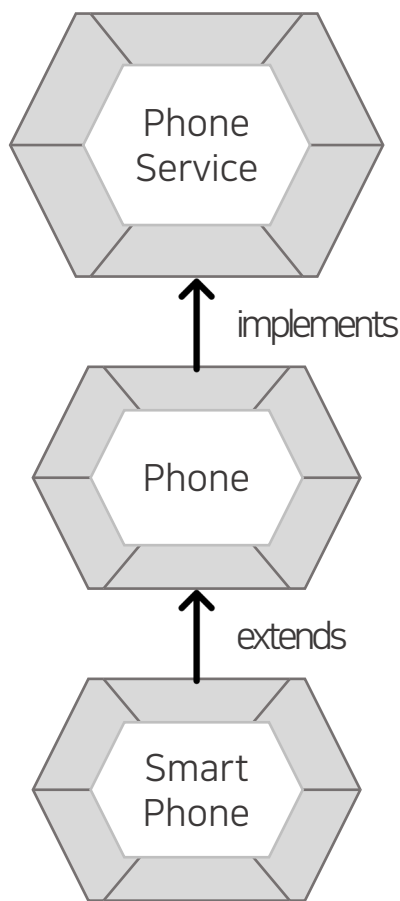
    @Override
    public void methodF() {
        System.out.println("아빠 메소드 실행");
    }

    @Override
    public void methodC() {
        System.out.println("자식 메소드 실행");
    }
}
```



객체지향 프로그래밍 (OOP) - 인터페이스 타입 변환

- 인터페이스의 타입 변환은 인터페이스와 구현 클래스 간에 발생한다.
- 부모 클래스가 인터페이스를 구현하고 있다면 자식 클래스도 해당 인터페이스 타입으로 자동 타입 변환될 수 있다.

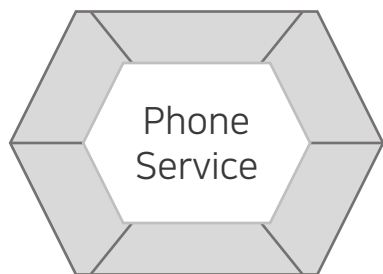


```
package com.oop.interface3;
```

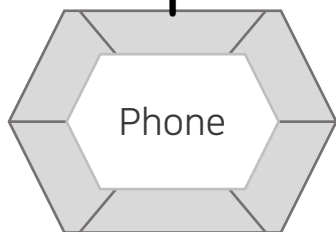
```
public class Phone implements PhoneService {  
    @Override  
    public void turnOn() {  
        System.out.println("전화기를 켭니다.");  
    }  
    @Override  
    public void turnOff() {  
        System.out.println("전화기를 끕니다.");  
    }  
    @Override  
    public void call(String number) {  
        System.out.println(number + "에 전화합니다.");  
    }  
}
```



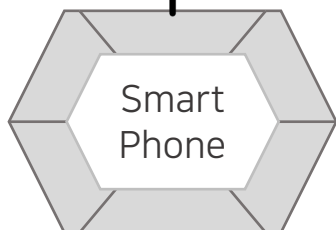
객체지향 프로그래밍 (OOP) - 인터페이스 타입 변환



인터페이스 PhoneService



구현 클래스 Phone (Phone implements PhoneService)



자식 클래스 SmartPhone (SmartPhone extends Phone)

```
PhoneService ps = new Phone();  
PhoneService ps = new SmartPhone();
```

TIP

인터페이스 PhoneService를 직, 간접적으로 구현하고 있는 클래스 Phone, SmartPhone은 모두 PhoneService로 자동 타입 변환될 수 있다.



객체지향 프로그래밍 (OOP) - 인터페이스 타입 변환

- 인터페이스 타입을 구현 클래스 타입으로 강제로 타입 변환을 시킬 수 있다.
- 인터페이스 타입으로 변환된 구현 객체는 인터페이스에 선언된 메소드만 사용 가능하기 때문에, 만약 구현 클래스의 메소드를 호출하고 싶다면 구현 클래스 타입으로 강제 타입 변환을 시켜주어야 한다.

```
package com.oop.interface3;

public class SmartPhoneExample {
    public static void main(String[] args) {
        PhoneService ps = new SmartPhone();
        // ps.openapp(); 호출 불가

        SmartPhone sp = (SmartPhone) ps; // 강제 변환
        sp.openApp("토스증권");
    }
}
```



객체지향 프로그래밍 (OOP) - 인터페이스 다형성

- 상속과 마찬가지로 인터페이스도 다형성을 구현하는 주된 기술로 사용된다.
- 현업에서는 상속보다 인터페이스를 통해 다형성 구현을 더 많이 한다.

```
package com.oop.interface5;
```

```
public interface Tire {  
    void roll();  
}
```

```
public class HankookTire implements Tire {  
    @Override  
    public void roll() {  
        System.out.println("한국 타이어가 굴러갑니다.");  
    }  
}
```

```
public class KumhoTire implements Tire {  
    @Override  
    public void roll() {  
        System.out.println("금호 타이어가 굴러갑니다.");  
    }  
}
```

```
public class Car {  
    Tire frontTire = new HankookTire();  
    Tire rearTire = new HankookTire();  
    void run() {  
        frontTire.roll();  
        rearTire.roll();  
    }  
}
```

```
public class CarExample {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
        myCar.run();  
        myCar.frontTire = new KumhoTire();  
        myCar.run();  
    }  
}
```




객체지향 프로그래밍 (OOP) - 인터페이스 다형성

- 매개변수 타입을 인터페이스로 선언하여, 메소드 호출 시 다양한 구현 객체를 대입할 수 있다.

```
package com.oop.interface6;
```

```
public interface Vehicle {  
    void run();  
}
```

```
public class Taxi implements Vehicle {  
    @Override  
    public void run() {  
        System.out.println("택시가 달립니다.");  
    }  
}
```

```
public class Bus implements Vehicle {  
    @Override  
    public void run() {  
        System.out.println("버스가 달립니다.");  
    }  
    public void checkFare() {  
        System.out.println("요금을 확인합니다.");  
    }  
}
```

```
public class Driver {  
    void drive(Vehicle vehicle) {  
        vehicle.run();  
    }  
}
```

```
public class DriverExample {  
    public static void main(String[] args) {  
        Driver d = new Driver();  
        d.drive(new Bus());  
        d.drive(new Taxi());  
    }  
}
```



객체지향 프로그래밍 (OOP) – instanceof

- 인터페이스에서도 instanceof 연산자를 사용 가능하다.

```
package com.oop.interface6;

public class InstanceofExample {
    public static void main(String[] args) {
        ride(new Taxi());
        System.out.println();
        ride(new Bus());
    }

    public static void ride(Vehicle v) {
        // if (v instanceof Bus) {
        //     Bus bus = (Bus) v;
        //     bus.checkFare();
        // }
        if (v instanceof Bus b) {
            b.checkFare();
        }
        v.run();
    }
}
```

TIP

주석은 아래 조건문과 동일한 동작입니다.