

2025년 상반기 K-디지털 트레이닝

# 클래스

---

[KB] IT's Your Life

# 객체지향 프로그래밍

## 객체지향 프로그래밍 개요

### 1. 부품을 조립하듯이 코딩하는 방식

- 객체지향 프로그래밍(OOP)은 프로그램을 여러 개의 객체(object)라는 단위로 나누어 개발하는 방식
- 각 객체는 데이터와 해당 데이터를 처리하는 방법(메서드)을 함께 포함

### 2. 건축과 유사한 방식

- 건축에서 건물을 설계하고 조립하는 것처럼, OOP에서도 프로그램을 설계
- 여러 객체를 조립하여 하나의 프로그램을 완성

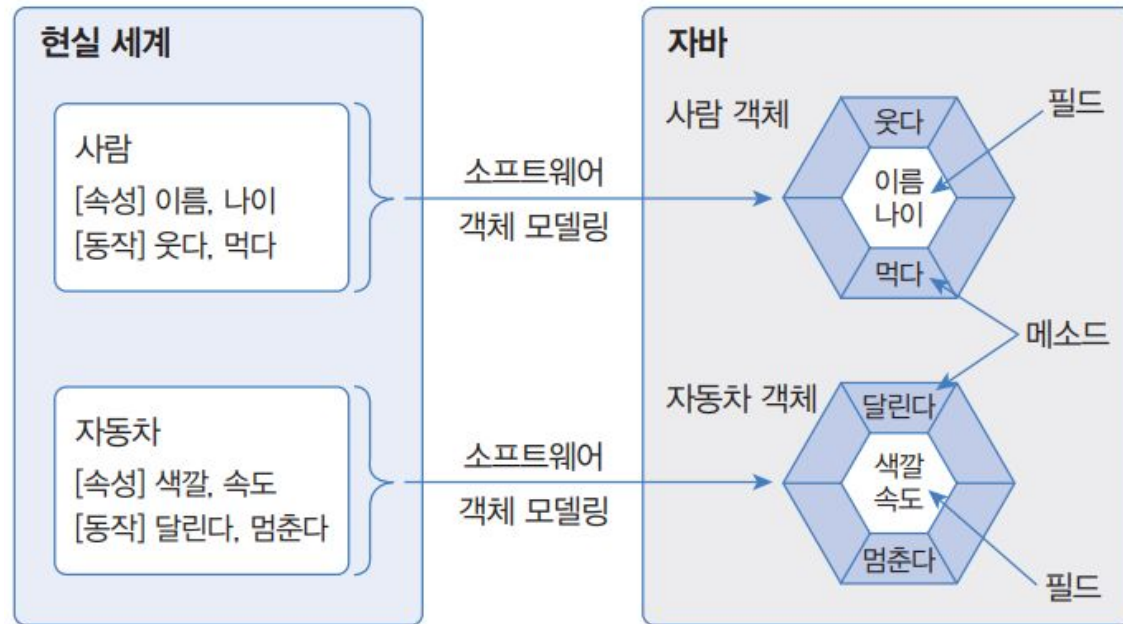
### 3. PC 조립과 유사한 방식

- 마치 컴퓨터를 여러 부품(메모리, CPU, 하드디스크 등)으로 조립하는 것처럼, OOP에서도 여러 객체를 조립하여 전체 시스템을 만든다.

# 1 객체 지향 프로그래밍

## • 객체

- 객체(object)란 물리적으로 존재하거나 개념적인 것 중에서 다른 것과 식별 가능한 것
- 객체는 속성과 동작으로 구성. 자바는 이러한 속성과 동작을 각각 필드와 메소드라고 부름



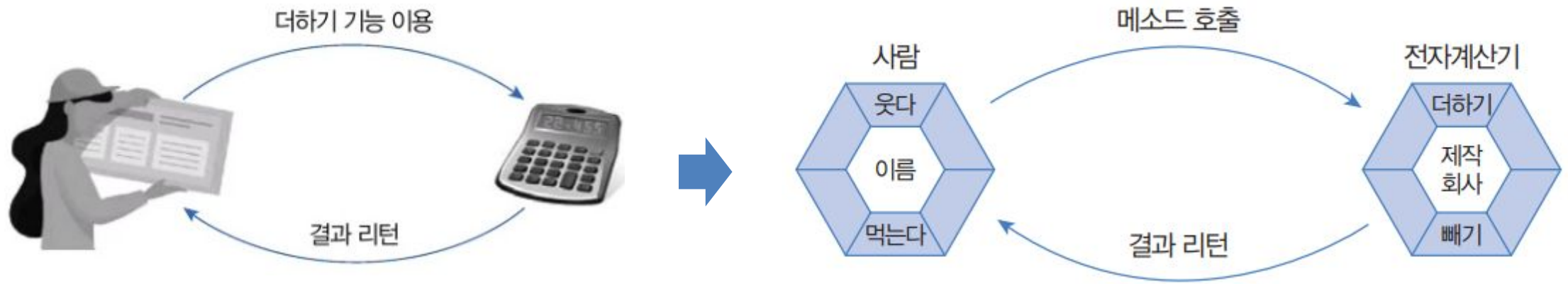
## • 객체 지향 프로그래밍 (OOP)

- 객체 객체들을 먼저 만들고, 이 객체들을 하나씩 조립해서 완성된 프로그램을 만드는 기법

# 1 객체 지향 프로그래밍

## • 객체의 상호작용

- 객체 지향 프로그램에서도 객체들은 다른 객체와 서로 상호작용하면서 동작
- 객체가 다른 객체의 기능을 이용할 때 이 메소드를 호출해 데이터를 주고받음



# 1 객체 지향 프로그래밍

## • 객체의 상호작용

- 매개값: 객체가 전달하고자 하는 데이터이며, 메소드 이름과 함께 괄호() 안에 기술
- 리턴값: 메소드의 실행의 결과이며, 호출한 곳으로 돌려주는 값

메소드(매개값1, 매개값2, ...);



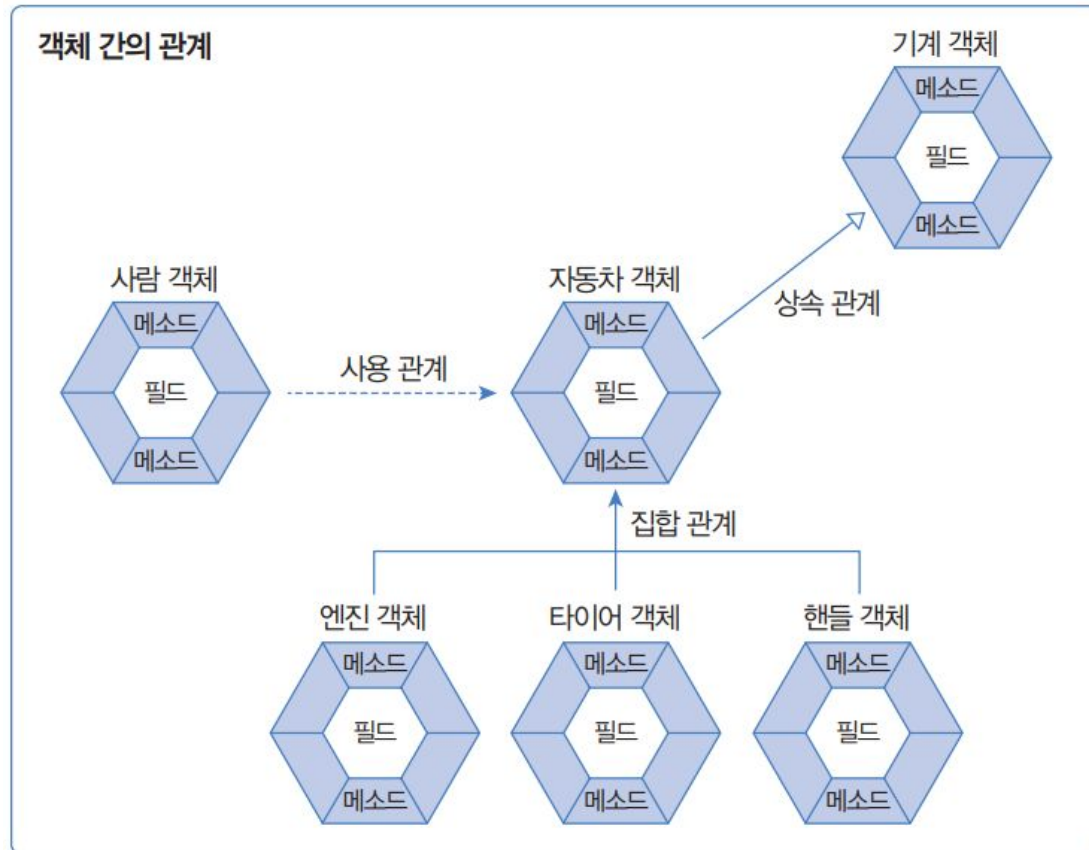
```
int result = add(10, 20);
```

리턴한 값을 int 변수에 저장

# 1 객체 지향 프로그래밍

## • 객체 간의 관계

- 집합 관계: 완제품과 부품의 관계
- 사용 관계: 다른 객체의 필드를 읽고 변경하거나 메소드를 호출하는 관계
- 상속 관계: 부모와 자식 관계. 필드, 메소드를 물려받음

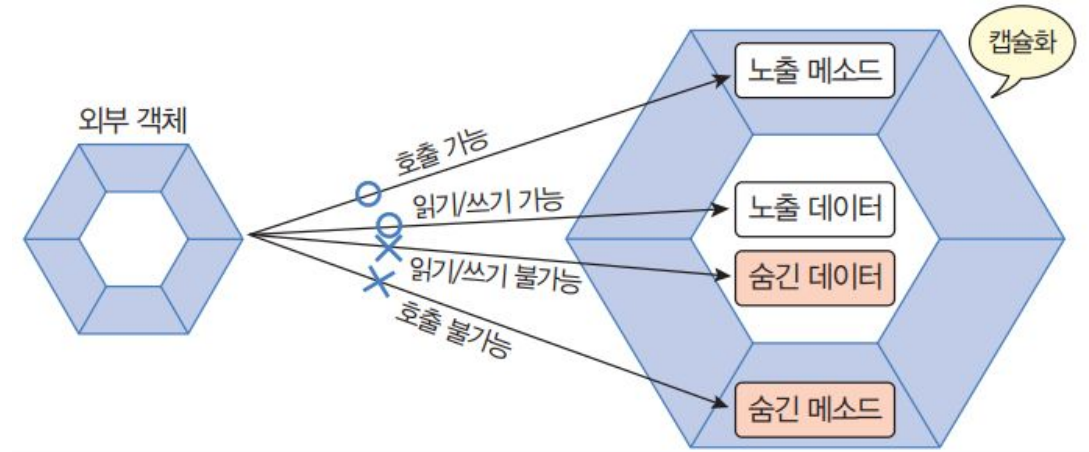


# 1 객체 지향 프로그래밍

## • 객체 지향 프로그래밍의 특징

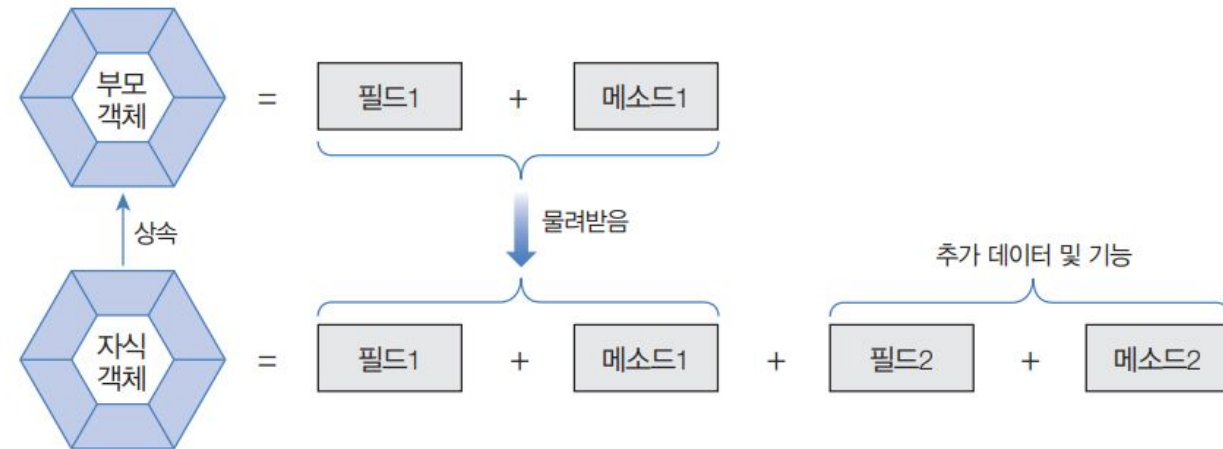
### ○ 캡슐화

- 객체의 데이터(필드), 동작(메소드)을 하나로 묶고 실제 구현 내용을 외부에 감추는 것



### ○ 상속

- 부모 객체가 자기 필드와 메소드를 자식 객체에게 물려줘 자식 객체가 사용할 수 있게 함  
→ 코드 재사용성 높이고 유지 보수 시간 최소화



# 1 객체 지향 프로그래밍

## • 객체 지향 프로그래밍의 특징

### ○ 다형성

- 사용 방법은 동일하지만 실행 결과가 다양함





## 2 객체와 클래스

### • 클래스와 인스턴스

- 객체 지향 프로그래밍에서도 객체를 생성하려면 설계도에 해당하는 클래스가 필요
- 클래스로부터 생성된 객체를 해당 클래스의 인스턴스라고 부름
- 클래스로부터 객체를 만드는 과정을 인스턴스화라고 함
- 동일한 클래스로부터 여러 개의 인스턴스를 만들 수 있음



# 클래스와 인스턴스

<https://youtu.be/QNGV5z9MR78>

**Car class(틀)**



**new**



**car1**

**object(대상)**

**== instance(실제)**

**new**



**car2**

**object**

내가 실제로 탈 대상은 class인가? object인가?

실제로 탈 자동차는 틀이 아니라 틀로 만들어진 자동차!

object == instance

object을 instance(실제)라고 한다.

이미지 출처 <https://kids.hyundai.com/kidshyundai/AutomobileManagement/learnauto/automobileDet.kids?ctgrMgrpCd=&cotnSn=3304>

필요한 부품은 다음 방법 중 하나로 얻을 수 있음

JDK4500개가  
이미 설치됨.

mvnrepository에서  
다운로드

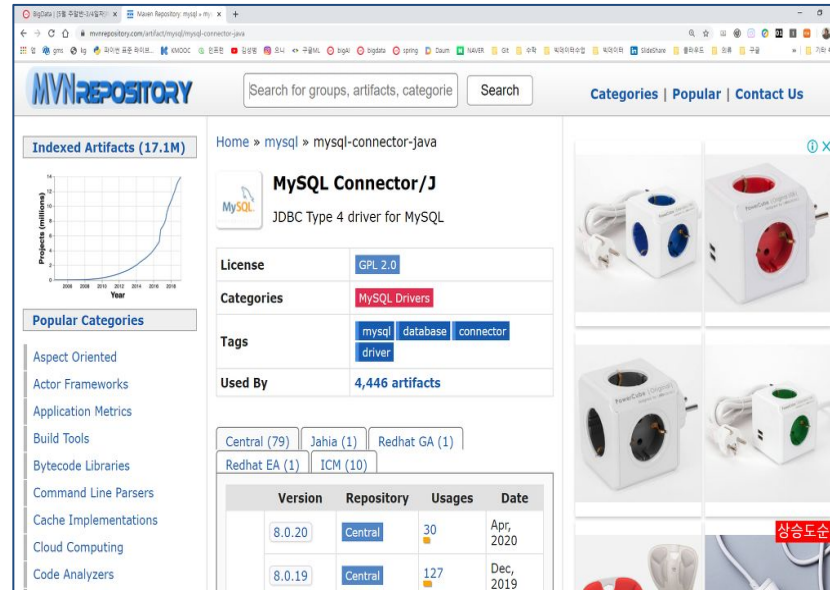
프로그래머가  
만든다.

### Java Build Path

Source Projects Libraries Order and Export

JARs and class folders on the build path:

> jsoup-1.9.2.jar - C:\Users\Administrator\Downloads  
> JRE System Library [JavaSE-1.8]



```
public class TV {
    //변수의 선언=> 타입 변수명;
    //선언의 위치=> 변수가 생존(사용)할 수 있는 범위
    //멤버변수 => 전역변수(클래스 안 전체영역에서 사용 가능)
    //      global변수, 글로벌 변수
    //글로벌변수는 자동초기화를 해준다.
    public int ch; //0으로 초기화
    public int vol;
    public boolean onOff; //false초기화
}
```

- 클래스 선언

- 객체를 생성(생성자)하고, 객체가 가져야 할 데이터(필드)가 무엇이고, 객체의 동작(메소드)은 무엇인지를 정의
- 클래스 선언은 소스 파일명과 동일하게 작성

**[클래스명.java]**

```
//클래스 선언  
public class 클래스명 {  
}
```

- 클래스명은 첫 문자를 대문자로 하고 캐멀 스타일로 작성. 숫자를 포함해도 되지만 첫 문자는 숫자가 될 수 없고, 특수 문자 중 \$, \_를 포함할 수 있음
- 공개(public) 클래스
  - 어느 위치에 있든지 패키지과 상관없이 사용할 수 있는 클래스

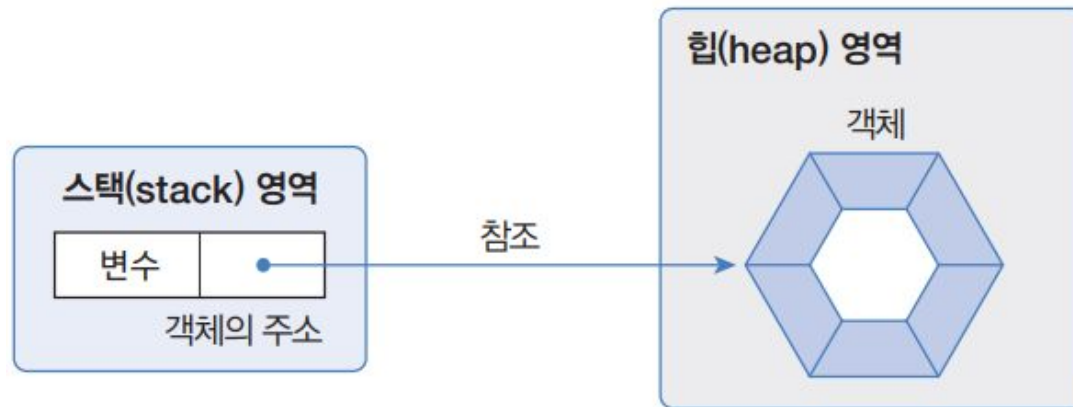
- **SportsCar.java**

```
package ch06.sec03;  
  
public class SportsCar {  
}  
  
class Tire {  
}
```

- 클래스 변수

- 클래스로부터 객체를 생성하려면 객체 생성 연산자인 **new**가 필요
- new** 연산자는 객체를 생성시키고 객체의 주소를 리턴

```
클래스 변수 = new 클래스();
```



- 라이브러리 클래스
  - 실행할 수 없으며 다른 클래스에서 이용하는 클래스
- 실행 클래스
  - main( )** 메소드를 가지고 있는 실행 가능한 클래스

### • Student.java

```
package ch06.sec04;

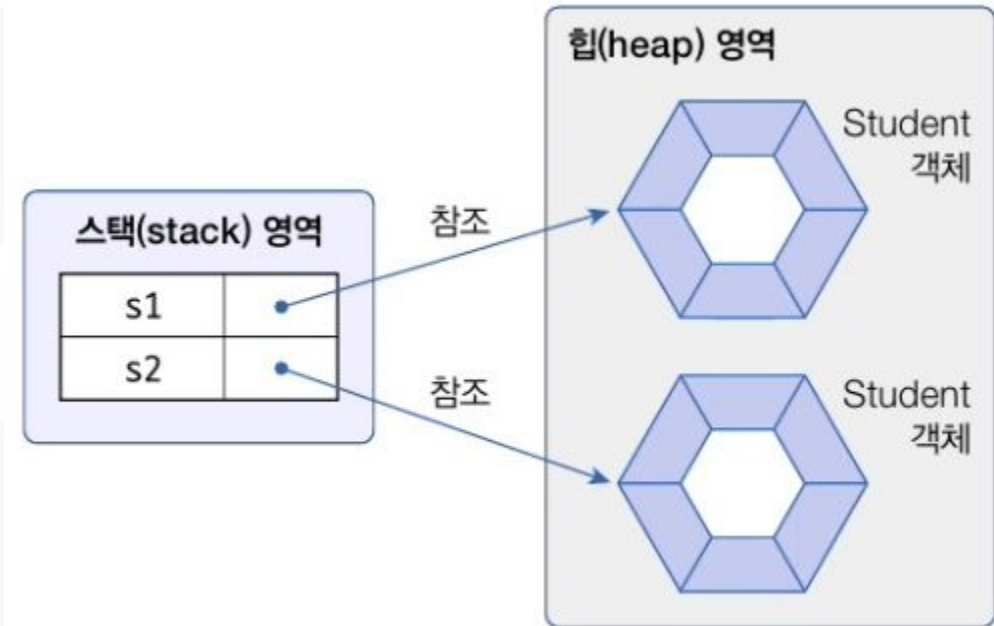
public class Student {
}
```

### • StudentExample.java

```
package ch06.sec04;

public class StudentExample {
    public static void main(String[] args) {
        Student s1 = new Student();
        System.out.println("s1 변수가 Student 객체를 참조합니다.");

        Student s2 = new Student();
        System.out.println("s2 변수가 또 다른 Student 객체를 참조합니다.");
    }
}
```



s1 변수가 Student 객체를 참조합니다.  
s2 변수가 또 다른 Student 객체를 참조합니다.

# 전화기(휴대폰)의 특징을 정리해보자.

하나의 클래스는  
변수와 메서드로 구성이  
되어있다.

	속성(attribute, property)	기능, 동작(action)
개념	전화기가 가지는 <b>기본 값 들</b> - 색, 크기, 회사명, 가격	전화기의 <b>기능들</b> - 전화하다, 문자하다, 인터넷하다.
구현	<b>변수(멤버변수, 필드)</b>	<b>메소드, 함수(멤버메서드)</b>
예	int price; String company;	public void call() { System.out.print("전 화 하 다"); }



<**기능**, 영-function(펑션)>

- 수학, it- 함수
- java- 메서드 (방식, 방법이라는 의미)
- **함수 == 메서드**



- 생성자, 필드, 메소드

- 필드

- 객체의 데이터를 저장하는 역할. 선언 형태는 변수 선언과 비슷하지만 쓰임새는 다름

- 생성자

- **new** 연산자로 객체를 생성할 때 객체의 초기화 역할.
    - 선언 형태는 메소드와 비슷하지만, 리턴 타입이 없고 이름은 클래스 이름과 동일

- 메소드

- 객체가 수행할 동작.
    - 함수로도 불림

- 필드

객체의 데이터가 저장되는 곳

- 생성자

객체 생성 시 초기화 역할 담당

- 메소드

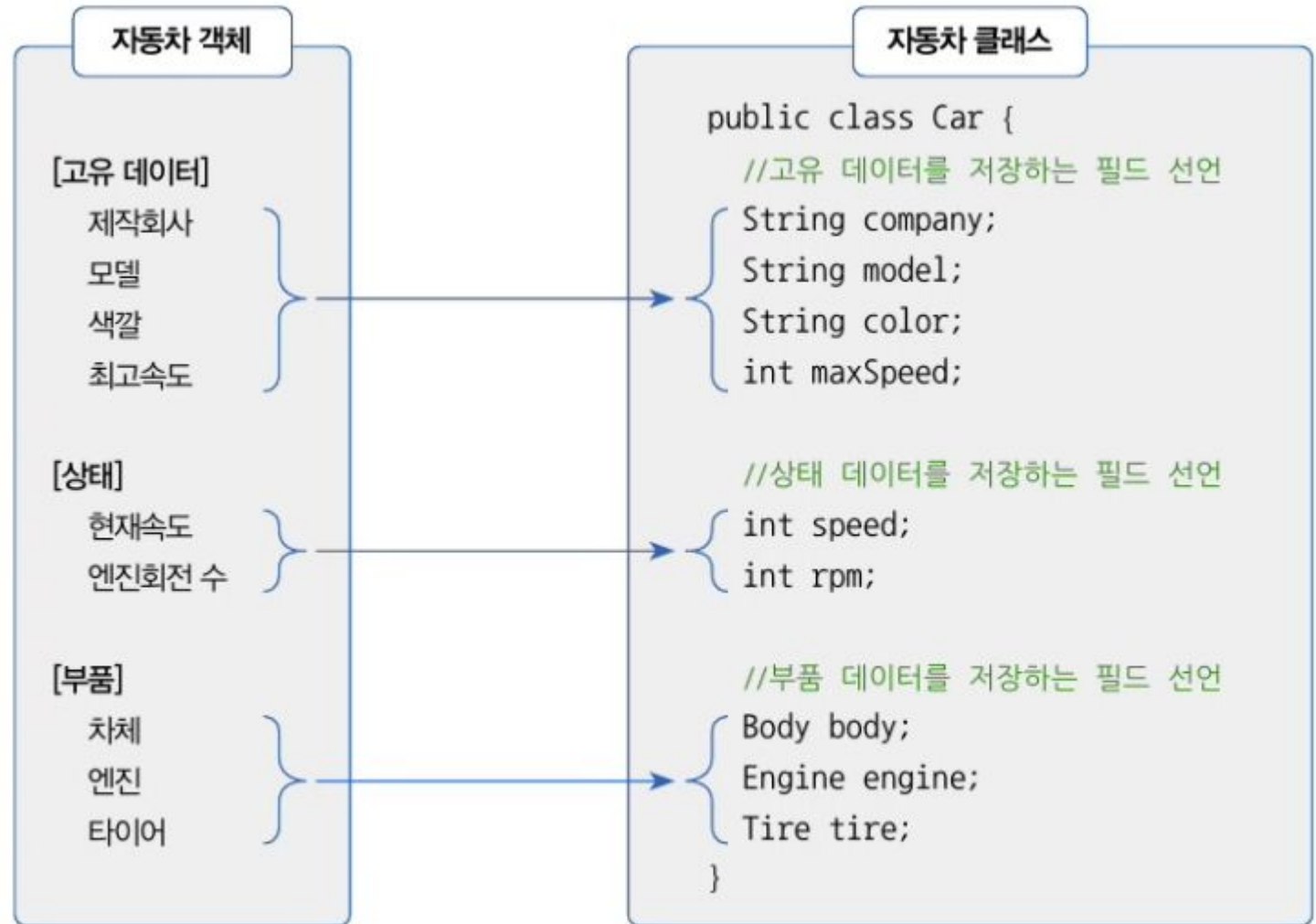
객체의 동작으로 호출 시 실행하는 블록

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

## 5 필드 선언과 사용

## • 필드 Field

- 객체의 데이터를 저장하는 역할
- 객체 데이터
  - 고유 데이터
  - 현재 상태 데이터
  - 부품 데이터 등



## 5 필드 선언과 사용

- 필드 선언

- 필드는 클래스 블록에서 선언되어야 함

```
타입 필드명 [ = 초기값] ;
```

- 타입은 필드에 저장할 데이터의 종류를 결정. 기본 타입, 참조 타입 모두 가능

```
class class Car {  
    String model = "그랜저";    //고유 데이터 필드  
    int speed = 300;            //상태 데이터 필드  
    boolean start = true;       //상태 데이터 필드  
    Tire tire = new Tire();     //부품 객체 필드  
}
```

## 5 필드 선언과 사용

- 필드 선언

- 초기값을 제공하지 않을 경우 필드는 객체 생성 시 자동으로 기본값으로 초기화

분류		데이터 타입	기본값
기본 타입	정수 타입	byte	0
		char	\u0000 (빈 공백)
		short	0
		int	0
		long	0L
	실수 타입	float	0.0F
		double	0.0
	논리 타입	boolean	false
참조 타입		배열	null
		클래스(String 포함)	null
		인터페이스	null

```
class class Car {
    String model;    //null
    int speed;       //0
    boolean start;   //false
    Tire tire;       //null
}
```

- **Car.java**

```
package ch06.sec06.exam01;
```

```
public class Car {  
    //필드 선언  
    String model;  
    boolean start;  
    int speed;  
}
```

- **CarExample.java**

```
package ch06.sec06.exam01;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

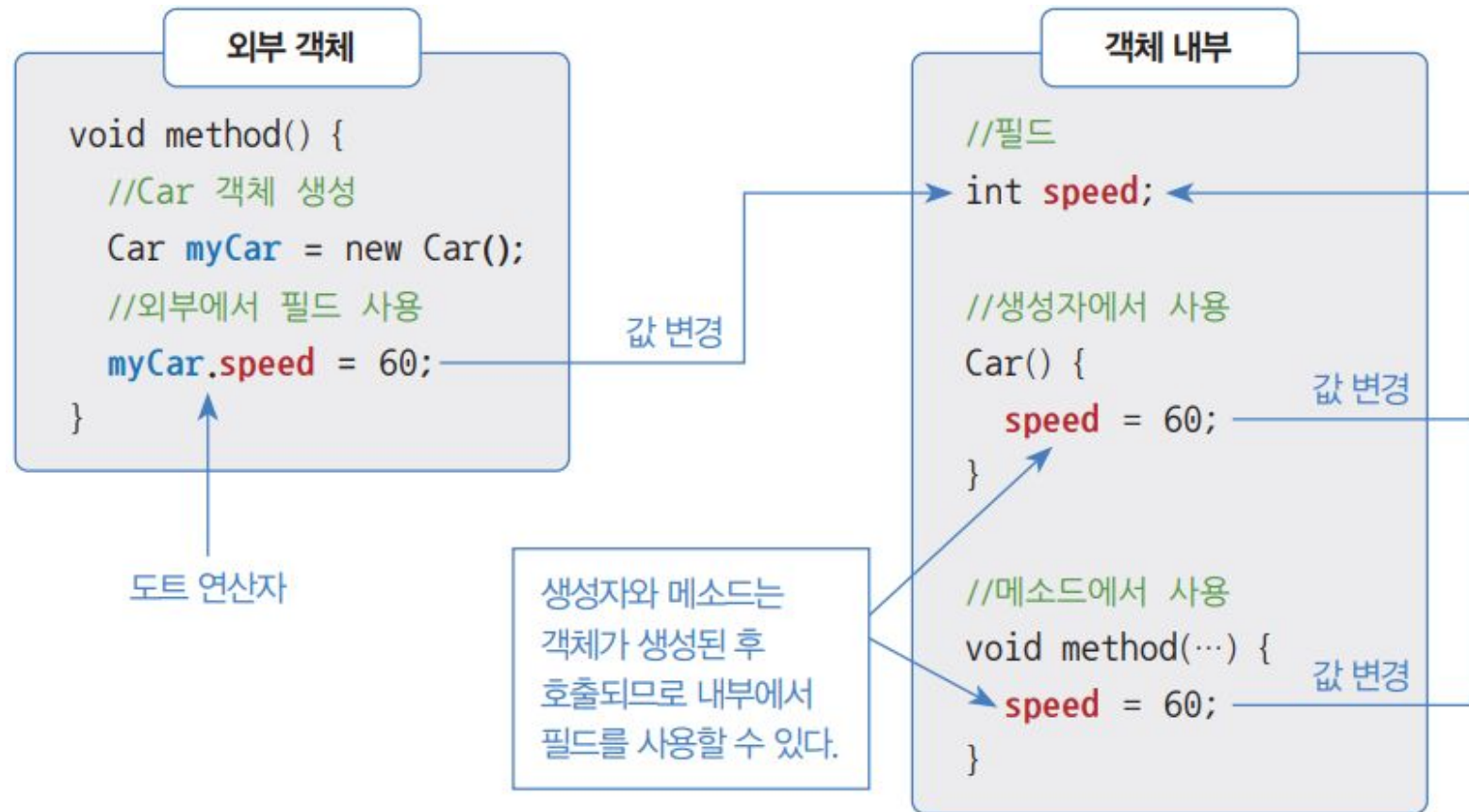
        //Car 객체의 필드값 읽기
        System.out.println("모델명: " + myCar.model);
        System.out.println("시동여부: " + myCar.start);
        System.out.println("현재속도: " + myCar.speed);
    }
}
```

```
모델명: null
시동여부: false
현재속도: 0
```

## 6 필드 선언과 사용

### • 필드 사용

- 필드값을 읽고 변경하는 것. 클래스로부터 객체가 생성된 후에 필드를 사용할 수 있음
- 필드는 객체 내부의 생성자와 메소드 내부에서 사용할 수 있고,
- 객체 외부에서도 접근해서 사용할 수 있음



- **Car.java**

```
package ch06.sec06.exam02;

public class Car {
    //필드 선언
    String company = "현대자동차";
    String model = "그랜저";
    String color = "검정";
    int maxSpeed = 350;
    int speed;
}
```



- **CarExample.java**

```
package ch06.sec06.exam02;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //Car 객체의 필드값 읽기
        System.out.println("제작회사: " + myCar.company);
        System.out.println("모델명: " + myCar.model);
        System.out.println("색깔: " + myCar.color);
        System.out.println("최고속도: " + myCar.maxSpeed);
        System.out.println("현재속도: " + myCar.speed);

        //Car 객체의 필드값 변경
        myCar.speed = 60;
        System.out.println("수정된 속도: " + myCar.speed);
    }
}
```

```
제작회사: 현대자동차
모델명: 그랜저
색깔: 검정
최고속도: 350
현재속도: 0
수정된 속도: 60
```

## 7 생성자 선언과 호출

- **new 연산자**

- 객체를 생성한 후 연이어 생성자(constructor)를 호출해서 객체를 초기화 함
- 객체 초기화
  - 필드 초기화를 하거나 메서드를 호출해서 객체를 사용할 준비하는 것

```
클래스 변수 = new 클래스();  
                생성자 호출
```

- 생성자가 성공적으로 실행이 끝나면 **new** 연산자는 객체의 주소를 반환
- 반환된 주소는 클래스 변수에 대입

## 7 생성자 선언과 호출

### • 기본 생성자

- 모든 클래스는 생성자가 존재하며, 하나 이상 가질 수 있음
- 클래스에 생성자 선언이 없으면 컴파일러는 기본 생성자를 바이트코드 파일에 자동으로 추가
  - 명시적으로 선언한 생성자가 있다면 기본 생성자를 추가하지 않음

```
[public] 클래스() { }
```

소스 파일(Car.java)

```
public class Car {  
  
}
```

바이트코드 파일(Car.class)

```
public class Car {  
    public Car() { }    //자동 추가  
}
```

→  
컴파일

```
Car myCar = new Car();  
           기본 생성자 호출
```

## 7 생성자 선언과 호출

### • 생성자 선언

- 객체를 다양하게 초기화하기 위해 생성자를 직접 선언할 수 있음

```
클래스(매개변수, ... ) {  
    //객체의 초기화 코드  
}
```

} 생성자 블록

- 생성자는 메소드와 비슷한 모양을 가지고 있으나, 리턴 타입이 없고 클래스 이름과 동일
- 매개변수의 타입은 매개값의 종류에 맞게 작성

```
public class Car {  
    //생성자 선언  
    Car(String model, String color, int maxSpeed) { ... }  
}  
  
Car myCar = new Car("그랜저", "검정", 300);
```

## 7 생성자 선언과 호출

### • Car.java

```
package ch06.sec07.exam01;

public class Car {
    //생성자 선언
    Car(String model, String color, int maxSpeed) {
}
}
```

### • CarExample.java

```
package ch06.sec07.exam01;

public class CarExample {
    public static void main(String[] args) {

        Car myCar = new Car("그랜저", "검정", 250);
        //Car myCar = new Car(); //기본 생성자 호출 못함
    }
}
```

## 7 생성자 선언과 호출

### • 필드 초기화

- 객체마다 동일한 값을 가지고 있다면 필드 선언 시 초기값을 대입하는 것이 좋음
- 객체마다 다른 값을 가져야 한다면 생성자에서 필드를 초기화하는 것이 좋음

```
public class Korean {  
    //필드 선언  
    String nation = "대한민국";  
    String name; ← 초기화  
    String ssn; ← 초기화  
  
    //생성자 선언  
    public Korean(String n, String s) {  
        name = n; ← 초기화  
        ssn = s; ← 초기화  
    }  
}
```

매개값으로 받은 이름과 주민등록번호를  
필드 초기값으로 사용

```
Korean k1 = new Korean("박자바", "011225-1234567");  
Korean k2 = new Korean("김자바", "930525-0654321");
```

- **Korean.java**

```
package ch06.sec07.exam02;

public class Korean {
    //필드 선언
    String nation = "대한민국";
    String name;
    String ssn;

    //생성자 선언
    public Korean(String n, String s) {
        name = n;
        ssn = s;
    }
}
```

- **KoreanExample.java**

```
package ch06.sec07.exam02;
```

```
public class KoreanExample {  
    public static void main(String[] args) {  
        //Korean 객체 생성  
        Korean k1 = new Korean("박자바", "011225-1234567");  
        //Korean 객체 데이터 읽기  
        System.out.println("k1.nation : " + k1.nation);  
        System.out.println("k1.name : " + k1.name);  
        System.out.println("k1.ssn : " + k1.ssn);  
        System.out.println();  
  
        //또 다른 Korean 객체 생성  
        Korean k2 = new Korean("김자바", "930525-0654321");  
        //또 다른 Korean 객체 데이터 읽기  
        System.out.println("k2.nation : " + k2.nation);  
        System.out.println("k2.name : " + k2.name);  
        System.out.println("k2.ssn : " + k2.ssn);  
    }  
}
```

```
k1.nation : 대한민국  
k1.name : 박자바  
k1.ssn : 011225-1234567  
  
k2.nation : 대한민국  
k2.name : 김자바  
k2.ssn : 930525-0654321
```



## 7 생성자 선언과 호출

- 생성자에서의 변수명
  - 지역변수와 필드 변수명이 동일한 경우 지역변수로 해석
  - 서로 다른 이름을 부여하는 경우 변수명 관리 힘들어짐
  - **this** 키워드
    - 현재 인스턴스에 대한 참조 변수
    - 자동으로 생성됨
    - **this.변수명**으로 필드 변수에 접근 가능

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

- **Korean.java**

```
package ch06.sec07.exam03;

public class Korean {
    // 필드 선언
    String nation = "대한민국";
    String name;
    String ssn;

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

## 7 생성자 선언과 호출

### • 생성자 오버로딩

- 매개변수를 달리하는 생성자를 여러 개 선언하는 것

```
public class Car {  
    Car() { ... }  
    Car(String model) { ... }  
    Car(String model, String color) { ... }  
    Car(String model, String color, int maxSpeed) { ... }  
}
```

#### [생성자 오버로딩]

매개변수의 타입, 개수, 순서가  
다르게 여러 개의 생성자 선언

- 매개변수의 타입, 개수, 순서가 똑같은 경우 매개변수 이름만 바꾸는 것은 생성자 오버로딩이 아님

```
Car(String model, String color) { ... }  
Car(String color, String model) { ... }    //오버로딩이 아님, 컴파일 에러 발생
```

## 7 생성자 선언과 호출

- 생성자 오버로딩

- 생성자가 오버로딩되어 있을 경우, **new** 연산자로 생성자를 호출할 때 제공되는 매개값의 타입과 수에 따라 실행될 생성자가 결정

```
Car car1 = new Car();           → Car() {...}
Car car2 = new Car("그랜저");    → Car(String model) {...}
Car car3 = new Car("그랜저", "흰색"); → Car(String model, String color) {...}
Car car4 = new Car("그랜저", "흰색", 300); → Car(String model, String color, int
                                         maxSpeed) {...}
```

- Car.java

```
package ch06.sec07.exam04;

public class Car {
    //필드 선언
    String company = "현대자동차";
    String model;
    String color;
    int maxSpeed;

    //생성자 선언
    Car() {}

    Car(String model) {
        this.model = model;
    }

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

    Car(String model, String color, int maxSpeed) {
        this.model = model;
        this.color = color;
        this.maxSpeed = maxSpeed;
    }
}
```

- CarExample.java

```
package ch06.sec07.exam04;
public class CarExample {
    public static void main(String[] args) {
        Car car1 = new Car();
        System.out.println("car1.company : " + car1.company);
        System.out.println();

        Car car2 = new Car("자가용");
        System.out.println("car2.company : " + car2.company);
        System.out.println("car2.model : " + car2.model);
        System.out.println();

        Car car3 = new Car("자가용", "빨강");
        System.out.println("car3.company : " + car3.company);
        System.out.println("car3.model : " + car3.model);
        System.out.println("car3.color : " + car3.color);
        System.out.println();

        Car car4 = new Car("택시", "검정", 200);
        System.out.println("car4.company : " + car4.company);
        System.out.println("car4.model : " + car4.model);
        System.out.println("car4.color : " + car4.color);
        System.out.println("car4.maxSpeed : " + car4.maxSpeed);
    }
}
```

car1.company : 현대자동차

car2.company : 현대자동차  
car2.model : 자가용

car3.company : 현대자동차  
car3.model : 자가용  
car3.color : 빨강

car4.company : 현대자동차  
car4.model : 택시  
car4.color : 검정  
car4.maxSpeed : 200

## 7 생성자 선언과 호출

- 다른 생성자 호출

- 생성자 오버로딩이 많아질 경우 생성자 간의 중복된 코드가 발생할 수 있음

```
Car(String model) {  
    this.model = model;  
    this.color = "은색";  
    this.maxSpeed = 250;  
}  
  
Car(String model, String color) {  
    this.model = model;  
    this.color = color;  
    this.maxSpeed = 250;  
}  
  
Car(String model, String color, int maxSpeed) {  
    this.model = model;  
    this.color = color;  
    this.maxSpeed = maxSpeed;  
}
```

중복 코드

중복 코드

중복 코드

## 7 생성자 선언과 호출

### • 다른 생성자 호출

- 이 경우 공통 코드를 한 생성자에만 집중적으로 작성
- 나머지 생성자는 `this (...)`를 사용해 공통 코드를 가진 생성자를 호출

```
Car(String model) {  
    this(model, "은색", 250);  
}  
  
Car(String model, String color) {  
    this(model, color, 250);  
}  
  
Car(String model, String color, int maxSpeed) {  
    this.model = model;  
    this.color = color;  
    this.maxSpeed = maxSpeed;  
}
```

호출

호출

공통 초기화 코드



## 7 생성자 선언과 호출

- 다른 생성자 호출

```
Car(String model) {  
    this(model, "은색", 250);  
    //추가적인 실행문  
}  
  
Car(String model, String color, int maxSpeed) {  
    this.model = model;  
    this.color = color;  
    this.maxSpeed = maxSpeed;  
}
```

The diagram illustrates a recursive call to the constructor `Car(String model, String color, int maxSpeed)` from within the `Car(String model)` constructor. A blue line originates from the `this(model, "은색", 250);` statement in the first constructor, moves right, then down, then left, ending with an arrow pointing to the opening curly brace of the second constructor. A second blue line originates from the `//추가적인 실행문` comment in the first constructor, moves left, then down, then left, also ending with an arrow pointing to the opening curly brace of the second constructor.

- **Car.java**

```
package ch06.sec07.exam05;
```

```
public class Car {
```

```
    // 필드 선언
```

```
    String company = "현대자동차";
```

```
    String model;
```

```
    String color;
```

```
    int maxSpeed;
```

```
    Car(String model) {
```

```
        //20라인 생성자 호출
```

```
        this(model, "은색", 250);
```

```
    }
```

```
    Car(String model, String color) {
```

```
        //20라인 생성자 호출
```

```
        this(model, color, 250);
```

```
    }
```

```
    Car(String model, String color, int maxSpeed) {
```

```
        this.model = model;
```

```
        this.color = color;
```

```
        this.maxSpeed = maxSpeed;
```

```
    }
```

```
}
```

- **CarExample.java**

```
package ch06.sec07.exam05;

public class CarExample {
    public static void main(String[] args) {
        Car car1 = new Car("자가용");
        System.out.println("car1.company : " + car1.company);
        System.out.println("car1.model : " + car1.model);
        System.out.println();

        Car car2 = new Car("자가용", "빨강");
        System.out.println("car2.company : " + car2.company);
        System.out.println("car2.model : " + car2.model);
        System.out.println("car2.color : " + car2.color);
        System.out.println();

        Car car3 = new Car("택시", "검정", 200);
        System.out.println("car3.company : " + car3.company);
        System.out.println("car3.model : " + car3.model);
        System.out.println("car3.color : " + car3.color);
        System.out.println("car3.maxSpeed : " + car3.maxSpeed);
    }
}
```

```
car1.company : 현대자동차
car1.model : 자가용
```

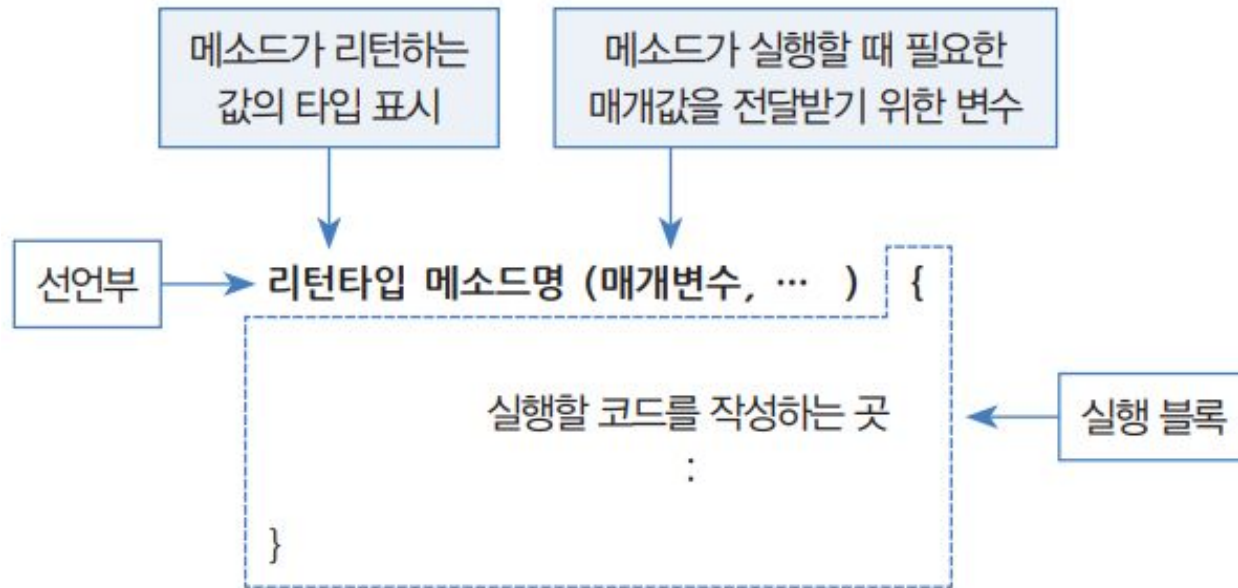
```
car2.company : 현대자동차
car2.model : 자가용
car2.color : 빨강
```

```
car3.company : 현대자동차
car3.model : 택시
car3.color : 검정
car3.maxSpeed : 200
```

## 8 메소드 선언과 호출

### • 메소드 선언

- 객체의 동작을 실행 블록으로 정의하는 것.



- 메소드 선언

- 리턴 타입: 메소드 실행 후 호출한 곳으로 전달하는 결과값의 타입

```
void powerOn() { ... }           //리턴값의 없는 메소드 선언  
double divide(int x, int y) { ... } //double 타입 값을 리턴하는 메소드 선언
```

- 메소드명: 메소드명은 첫 문자를 소문자로 시작하고, 캐멀 스타일로 작성

```
void run() { ... }  
void setSpeed(int speed) { ... }  
String getName() { ... }
```

- 매개변수: 메소드를 호출할 때 전달한 매개값을 받기 위해 사용

```
double divide(int x, int y) { ... }
```

- 실행 블록: 메소드 호출 시 실행되는 부분

# 입력값/반환값 확인

```

계산기3.java  내가계3.java  계산기.java  계산기2.java
1 package 메서드연습;
2
3 public class 계산기 {
4     public void 더하기(int x, int y) { //더하기(2,3)
5         System.out.println("더하기 기능 처리 방법 내용....");
6         System.out.println(x + y);
7     }
8     public void 곱하기(int x, int y, int z) {
9         System.out.println("곱하기 기능 처리 방법 내용....");
10        System.out.println(x * y * z);
11    }
12    public int 빼기(int x, int y) { //매개변수(parameter)
13        System.out.println("빼기 기능 처리 방법 내용");
14        return x - y;
15    }
16 }

```

```

계산기3.java  내가계3.java  계산기.java  계산기2.java  내가계.java
1 package 메서드연습;
2
3 public class 내가계 {
4
5     public static void main(String[] args) {
6         계산기 cal = new 계산기();
7         cal.더하기(2, 3);
8         cal.곱하기(30, 20, 10); //void
9         int result = cal.빼기(500, 200); //300
10        System.out.println(result - 100);
11    }
12 }

```

## • Calculator.java

```
package ch06.sec08.exam01;

public class Calculator {
    //리턴값이 없는 메소드 선언
    void powerOn() {
        System.out.println("전원을 켭니다.");
    }

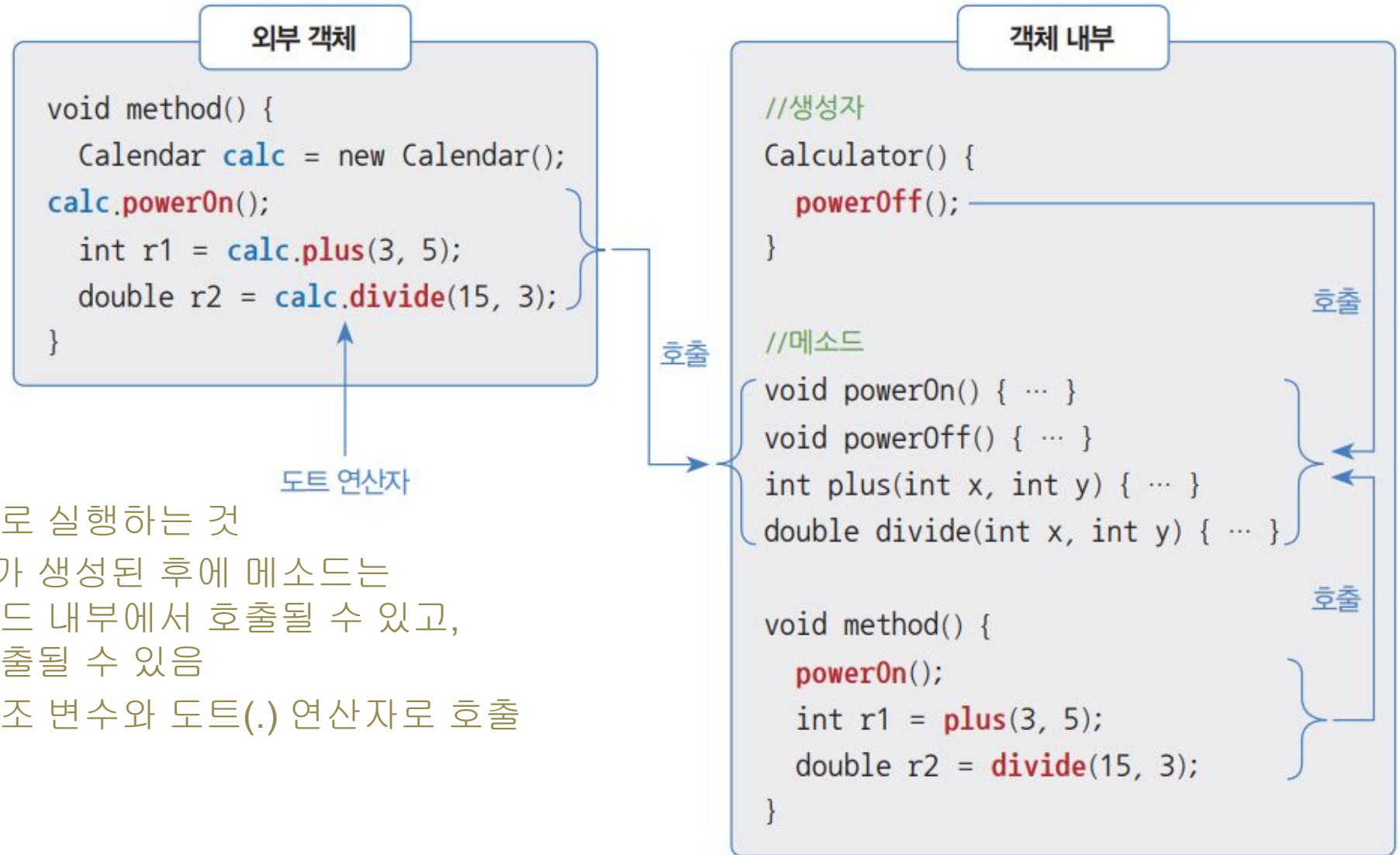
    //리턴값이 없는 메소드 선언
    void powerOff() {
        System.out.println("전원을 끕니다.");
    }

    //호출 시 두 정수 값을 전달받고,
    //호출한 곳으로 결과값 int를 리턴하는 메소드 선언
    int plus(int x, int y) {
        int result = x + y;
        return result; //리턴값 지정;
    }

    //호출 시 두 정수 값을 전달받고,
    //호출한 곳으로 결과값 double을 리턴하는 메소드선언
    double divide(int x, int y) {
        double result = (double)x / (double)y;
        return result; //리턴값 지정;
    }
}
```

## 8 메소드 선언과 호출

## • 메소드 호출



- 메소드 블록을 실제로 실행하는 것
- 클래스로부터 객체가 생성된 후에 메소드는 생성자와 다른 메소드 내부에서 호출될 수 있고, 객체 외부에서도 호출될 수 있음
- 외부 객체에서는 참조 변수와 도트(.) 연산자로 호출



## • CalculatorExample.java

```
package ch06.sec08.exam01;

public class CalculatorExample {
    public static void main(String[] args) {
        Calculator myCalc = new Calculator();          //Calculator 객체 생성

        myCalc.powerOn();                                //리턴값이 없는 powerOn() 메소드 호출

        //plus 메소드 호출 시 5와 6을 매개값으로 제공하고,
        //덧셈 결과를 리턴 받아 result1 변수에 대입
        int result1 = myCalc.plus(5, 6);
        System.out.println("result1: " + result1);

        int x = 10;
        int y = 4;
        //divide() 메소드 호출 시 변수 x와 y의 값을 매개값으로 제공하고,
        //나눗셈 결과를 리턴 받아 result2 변수에 대입
        double result2 = myCalc.divide(x, y);
        System.out.println("result2: " + result2);

        //리턴값이 없는 powerOff() 메소드 호출
        myCalc.powerOff();
    }
}
```

전원을 켭니다.  
result1: 11  
result2: 2.5  
전원을 끕니다.

- 가변길이 매개변수

- 메소드가 가변길이 매개변수를 가지고 있다면 매개변수의 개수와 상관없이 매개값을 줄 수 있음

```
int sum(int ... values) {  
}
```

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

- 메소드 호출 시 매개값을 쉼표로 구분해서 개수와 상관없이 제공할 수 있음
- 매개값들은 자동으로 배열 항목으로 변환되어 메소드에서 사용됨

```
int[] values = { 1, 2, 3 };  
int result = sum(values);
```

```
int result = sum(new int[] { 1, 2, 3 });
```

- **Computer.java**

```
package ch06.sec08.exam02;

public class Computer {
    //가변길이 매개변수를 갖는 메소드 선언
    int sum(int ... values) {
        //sum 변수 선언
        int sum = 0;

        //values는 배열 타입의 변수처럼 사용
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }

        //합산 결과를 리턴
        return sum;
    }
}
```

- ComputerExample.java

```
package ch06.sec08.exam02;

public class ComputerExample {
    public static void main(String[] args) {
        Computer myCom = new Computer();           //Computer 객체 생성

        //sum() 메소드 호출 시 매개값 1, 2, 3을 제공, 합산 결과를 리턴 받아 result1 변수에 대입
        int result1 = myCom.sum(1, 2, 3);
        System.out.println("result1: " + result1);

        //sum() 메소드 호출 시 매개값 1, 2, 3, 4, 5를 제공, 합산 결과를 리턴 받아 result2 변수에 대입
        int result2 = myCom.sum(1, 2, 3, 4, 5);
        System.out.println("result2: " + result2);

        //sum() 메소드 호출 시 배열을 제공, 합산 결과를 리턴 받아 result3 변수에 대입
        int[] values = { 1, 2, 3, 4, 5 };
        int result3 = myCom.sum(values);
        System.out.println("result3: " + result3);

        //sum() 메소드 호출 시 배열을 제공하고
        //합산 결과를 리턴 받아 result4 변수에 대입
        int result4 = myCom.sum(new int[] { 1, 2, 3, 4, 5 });
        System.out.println("result4: " + result4);
    }
}
```

```
result1: 6
result2: 15
result3: 15
result4: 15
```

## 8 메소드 선언과 호출

- **return 문**

- 메소드의 실행을 강제 종료하고 호출한 곳으로 돌아간다는 의미
- 메소드 선언에 리턴 타입이 있을 경우에는 **return** 문 뒤에 리턴값을 추가로 지정해야 함

```
return [리턴값];
```

- **return** 문 이후에 실행문을 작성하면 'Unreachable code'라는 컴파일 에러가 발생

```
int plus(int x, int y) {  
    int result = x + y;  
    return result;  
    System.out.println(result); //Unreachable code  
}
```

- Car.java

```
package ch06.sec08.exam03;

public class Car {
    //필드 선언
    int gas;

    //리턴값이 없는 메소드로 매개값을 받아서 gas 필드값을 변경
    void setGas(int gas) {
        this.gas = gas;
    }

    //리턴값이 boolean인 메소드로 gas 필드값이 0이면 false를, 0이 아니면 true를 리턴
    boolean isLeftGas() {
        if (gas == 0) {
            System.out.println("gas가 없습니다.");
            return false; // false를 리턴하고 메소드 종료
        }
        System.out.println("gas가 있습니다.");
        return true; // true를 리턴하고 메소드 종료
    }
}
```

- Car.java

//리턴값이 없는 메소드로 gas 필드값이 0이면 return 문으로 메소드를 종료

```
void run() {  
    while (true) {  
        if (gas > 0) {  
            System.out.println("달립니다.(gas잔량:" + gas + ")");  
            gas -= 1;  
        } else {  
            System.out.println("멈춥니다.(gas잔량:" + gas + ")");  
            return; // 메소드 종료  
        }  
    }  
}
```

- **CarExample.java**

```
package ch06.sec08.exam03;

public class CarExample {
    public static void main(String[] args) {
        //Car 객체 생성
        Car myCar = new Car();

        //리턴값이 없는 setGas() 메소드 호출
        myCar.setGas(5);

        //isLeftGas() 메소드를 호출해서 받은 리턴값이 true일 경우 if 블록 실행
        if(myCar.isLeftGas()) {
            System.out.println("출발합니다.");

            //리턴값이 없는 run() 메소드 호출
            myCar.run();
        }

        System.out.println("gas를 주입하세요.");
    }
}
```

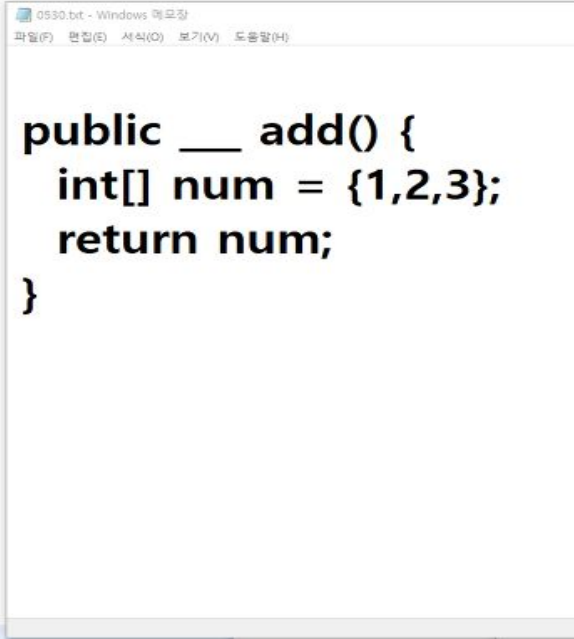
```
gas가 있습니다.
출발합니다.
달립니다.(gas잔량:5)
달립니다.(gas잔량:4)
달립니다.(gas잔량:3)
달립니다.(gas잔량:2)
달립니다.(gas잔량:1)
멈춥니다.(gas잔량:0)
gas를 주입하세요.
```



# 반환 명시

- \_\_\_\_에 들어갈 적절한 단어를 넣어 계산기2를 완성해보자.

```
public class 계산기2 {  
    public ____ add(int x, int y) {  
        return x + y;  
    }  
  
    public ____ add(int x, double y) {  
        return x + y;  
    }  
  
    public ____ add(double x, double y) {  
        return x + y;  
    }  
  
    public ____ add(String x, int y) {  
        return x + y;  
    }  
}
```



A screenshot of a Windows Notepad window titled "0530.txt - Windows 메모장". The window contains the following Java code snippet:

```
public ____ add() {  
    int[] num = {1,2,3};  
    return num;  
}
```

## 8 메소드 선언과 호출

## • 메소드 오버로딩

- 메소드 이름은 같되 매개변수의 타입, 개수, 순서가 다른 메소드를 여러 개 선언하는 것

```
class 클래스 {
  리턴타입 메소드이름 ( 타입 변수, ... ) { ... }
  ↑         ↑         ↑
  무관      동일      타입, 개수, 순서가 달라야 함
  ↓         ↓         ↓
  리턴타입 메소드이름 ( 타입 변수, ... ) { ... }
}
```

```
void println() { .. }
void println(double x) { .. }
void println(int x) { .. }
void println(String x) { .. }
```

```
int plus(int x, int y) {
  int result = x + y;
  return result;
}
```

```
double plus(double x, double y) {
  double result = x + y;
  return result;
}
```

- **Calculator.java**

```
package ch06.sec08.exam04;

public class Calculator {
    //정사각형의 넓이
    double areaRectangle(double width) {
        return width * width;
    }

    //직사각형의 넓이
    double areaRectangle(double width, double height) {
        return width * height;
    }
}
```

- **CalculatorExample.java**

```
package ch06.sec08.exam04;
```

```
public class CalculatorExample {  
    public static void main(String[] args) {
```

```
        //객체 생성
```

```
        Calculator myCalcu = new Calculator();
```

```
        //정사각형의 넓이 구하기
```

```
        double result1 = myCalcu.areaRectangle(10);
```

double areaRectangle(double width) 호출

```
        //직사각형의 넓이 구하기
```

```
        double result2 = myCalcu.areaRectangle(10, 20);
```

double areaRectangle(double width, double height) 호출

```
        System.out.println("정사각형 넓이=" + result1);
```

```
        System.out.println("직사각형 넓이=" + result2);
```

```
    }
```

```
}
```

정사각형 넓이=100.0

직사각형 넓이=200.0

- 선언 방법에 따른 멤버의 구분

구분	설명
인스턴스(instance) 멤버	객체에 소속된 멤버 (객체를 생성해야만 사용할 수 있는 멤버)
정적(static) 멤버	클래스에 고정된 멤버 (객체 없이도 사용할 수 있는 멤버)

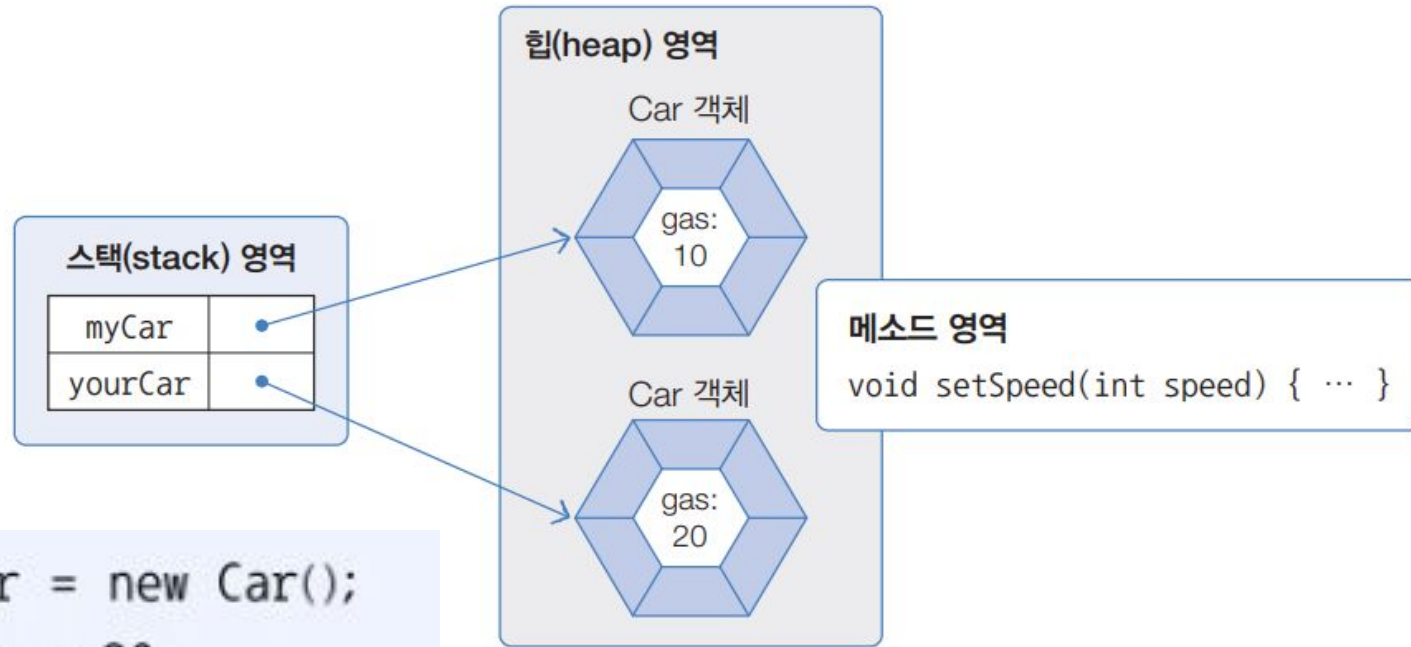
- 인스턴스 멤버 선언 및 사용

- 인스턴스 멤버: 필드와 메소드 등 객체에 소속된 멤버

```
public class Car {  
    //인스턴스 필드 선언  
    int gas;  
  
    //인스턴스 메소드 선언  
    void setSpeed(int speed) { ... }  
}
```

```
Car myCar = new Car();  
myCar.gas = 10;  
myCar.setSpeed(60);
```

```
Car yourCar = new Car();  
yourCar.gas = 20;  
yourCar.setSpeed(80);
```



- **this** 키워드

- 객체 내부에서는 인스턴스 멤버에 접근하기 위해 **this**를 사용. 객체는 자신을 '**this**'라고 지칭
- 생성자와 메소드의 매개변수명이 인스턴스 멤버인 필드명과 동일한 경우, 인스턴스 필드임을 강조하고자 할 때 **this**를 주로 사용

- **Car.java**

```
package ch06.sec09;

public class Car {
    //필드 선언
    String model;
    int speed;

    //생성자 선언
    Car(String model) {
        this.model = model; //매개변수를 필드에 대입 (this 생략 불가)
    }

    //메소드 선언
    void setSpeed(int speed) {
        this.speed = speed; //매개변수를 필드에 대입 (this 생략 불가)
    }

    void run() {
        this.setSpeed(100);
        System.out.println(this.model + "가 달립니다.(시속:" + this.speed + "km/h)");
    }
}
```



- **CarExample.java**

```
package ch06.sec09;

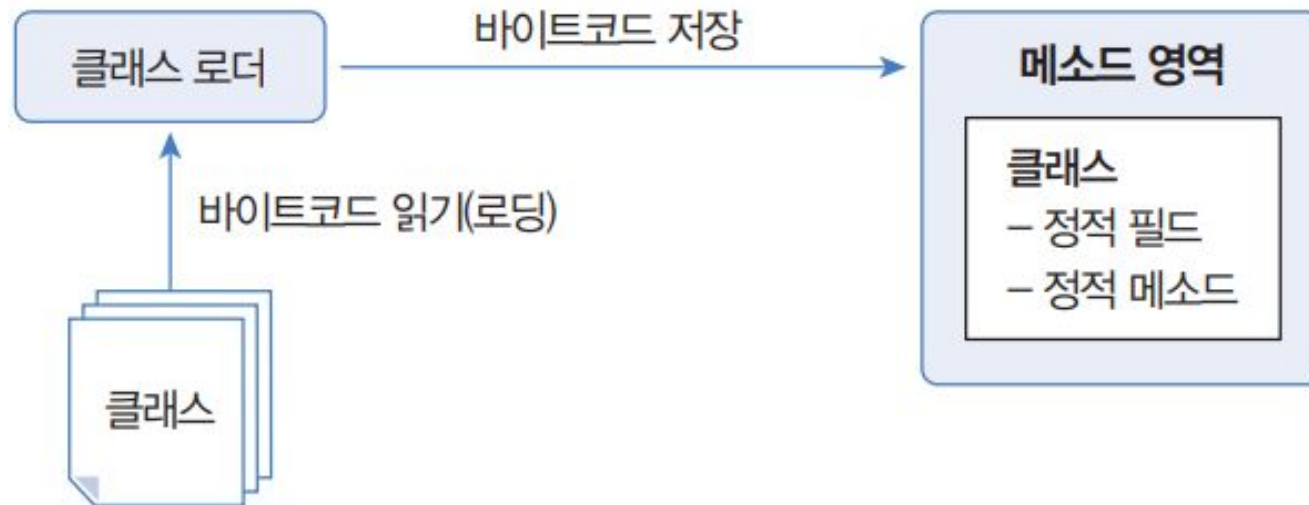
public class CarExample {
    public static void main(String[] args) {
        Car myCar = new Car("포르쉐");
        Car yourCar = new Car("벤츠");

        myCar.run();
        yourCar.run();
    }
}
```

포르쉐가 달립니다.(시속:100km/h)  
벤츠가 달립니다.(시속:100km/h)

- 정적 멤버 선언

- 정적 멤버: 메소드 영역의 클래스에 고정적으로 위치하는 멤버



- 정적 멤버 선언

- **static** 키워드를 추가해 정적 필드와 정적 메소드로 선언

```
public class 클래스 {
    //정적 필드 선언
    static 타입 필드 [= 초기값];

    //정적 메소드
    static 리턴타입 메소드( 매개변수, ... ) { ... }
}
```

```
public class Calculator {
    String color;           //계산기별로 색깔이 다를 수 있다.
    static double pi = 3.14159; //계산기에서 사용하는 파이( $\pi$ ) 값은 동일하다.
}
```

```
public class Calculator {
    String color;           //인스턴스 필드
    void setColor(String color) { this.color = color; } //인스턴스 메소드
    static int plus(int x, int y) { return x + y; } //정적 메소드
    static int minus(int x, int y) { return x - y; } //정적 메소드
}
```

- 정적 멤버 사용

- 클래스가 메모리로 로딩되면 정적 멤버를 바로 사용할 수 있음
- 클래스 이름과 함께 도트(.) 연산자로 접근

```
public class Calculator {  
    static double pi = 3.14159;  
    static int plus(int x, int y) { ... }  
    static int minus(int x, int y) { ... }  
}
```

```
double result1 = 10 * 10 * Calculator.pi;  
int result2 = Calculator.plus(10, 5);  
int result3 = Calculator.minus(10, 5);
```

- 정적 필드와 정적 메소드는 객체 참조 변수로도 접근

- **Calculator.java**

```
package ch06.sec10.exam01;

public class Calculator {
    static double pi = 3.14159;

    static int plus(int x, int y) {
        return x + y;
    }

    static int minus(int x, int y) {
        return x - y;
    }
}
```

- **CalculatorExample.java**

```
package ch06.sec10.exam01;

public class CalculatorExample {
    public static void main(String[] args) {
        double result1 = 10 * 10 * Calculator.pi;
        int result2 = Calculator.plus(10, 5);
        int result3 = Calculator.minus(10, 5);

        System.out.println("result1 : " + result1);
        System.out.println("result2 : " + result2);
        System.out.println("result3 : " + result3);
    }
}
```

```
result1 : 314.159
result2 : 15
result3 : 5
```

- 정적 블록

- 정적 필드를 선언할 때 복잡한 초기화 작업이 필요하다면 정적 블록을 이용

```
static {  
    ...  
}
```

- 정적 블록은 클래스가 메모리로 로딩될 때 자동으로 실행
- 정적 블록이 클래스 내부에 여러 개가 선언되어 있을 경우에는 선언된 순서대로 실행
- 정적 필드는 객체 생성 없이도 사용할 수 있기 때문에 생성자에서 초기화 작업을 하지 않음

- **Television.java**

```
package ch06.sec10.exam02;

public class Television {
    static String company = "MyCompany";
    static String model = "LCD";
    static String info;

    static {
        info = company + "-" + model;
    }
}
```



- **TelevisionExample.java**

```
package ch06.sec10.exam02;  
  
public class TelevisionExample {  
    public static void main(String[] args) {  
        System.out.println(Television.info);  
    }  
}
```

MyCompany-LCD

- 인스턴스 멤버 사용 불가

- 정적 메소드와 정적 블록은 내부에 인스턴스 필드나 인스턴스 메소드를 사용할 수 없으며 **this**도 사용할 수 없음
- 정적 메소드와 정적 블록에서 인스턴스 멤버를 사용하고 싶다면 객체를 먼저 생성하고 참조 변수로 접근

```
static void Method3() {  
    //객체 생성  
    ClassName obj = new ClassName();  
    //인스턴스 멤버 사용  
    obj.field1 = 10;  
    obj.method1();  
}
```

- **Car.java**

```
package ch06.sec10.exam03;

public class Car {
    //인스턴스 필드 선언
    int speed;

    //인스턴스 메소드 선언
    void run() {
        System.out.println(speed + "으로 달립니다.");
    }

    static void simulate() {
        //객체 생성
        Car myCar = new Car();
        //인스턴스 멤버 사용
        myCar.speed = 200;
        myCar.run();
    }
}
```

200으로 달립니다.

60으로 달립니다.

simulate();

- **final 필드 선언**

- final 필드는 초기값이 저장되면 최종적인 값이 되어서 프로그램 실행 도중에 수정할 수 없게 됨

```
final 타입 필드 [=초기값];
```

- final 필드 초기화
  - 필드 선언 시에 초기값을 대입
  - 생성자에서 초기값을 대입

- **Korean.java**

```
package ch06.sec11.exam01;

public class Korean {
    //인스턴스 final 필드 선언
    final String nation = "대한민국";
    final String ssn;

    //인스턴스 필드 선언
    String name;

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

- **Korean.java**

```
package ch06.sec11.exam01;

public class KoreanExample {
    public static void main(String[] args) {
        //객체 생성 시 주민등록번호와 이름 전달
        Korean k1 = new Korean("123456-1234567", "감자바");

        //필드값 읽기
        System.out.println(k1.nation);
        System.out.println(k1.ssn);
        System.out.println(k1.name);

        //Final 필드는 값을 변경할 수 없음
        //k1.nation = "USA";
        //k1.ssn = "123-12-1234";

        //비 final 필드는 값 변경 가능
        k1.name = "김자바";
    }
}
```

```
대한민국
123456-1234567
감자바
```

- 상수 선언

- 상수: 불변의 값을 저장하는 필드
- 상수는 객체마다 저장할 필요가 없고, 여러 개의 값을 가져도 안 되기 때문에 **static**이면서 **final**

```
static final 타입 상수 [= 초기값];
```

```
static final 타입 상수;  
static {  
    상수 = 초기값;  
}
```

- 상수명 표기 관례
  - 대문자로 구성되는 스네이크 표기법(\_연결)

```
static final double PI = 3.14159;  
static final double EARTH_SURFACE_AREA = 5.147185403641517E8;
```

- Earth.java

```
package ch06.sec11.exam02;

public class Earth {
    //상수 선언 및 초기화
    static final double EARTH_RADIUS = 6400;

    //상수 선언
    static final double EARTH_SURFACE_AREA;

    //정적 블록에서 상수 초기화
    static {
        EARTH_SURFACE_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
    }
}
```



- **EarthExample.java**

```
package ch06.sec11.exam02;

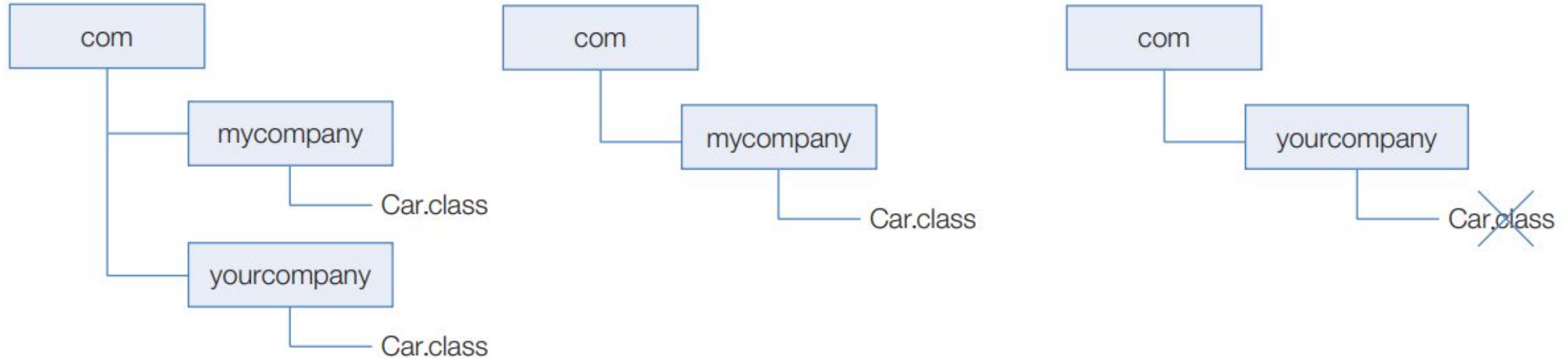
public class EarthExample {
    public static void main(String[] args) {
        //상수 읽기
        System.out.println("지구의 반지름: " + Earth.EARTH_RADIUS + "km");
        System.out.println("지구의 표면적: " + Earth.EARTH_SURFACE_AREA + "km^2");
    }
}
```

지구의 반지름: 6400.0km

지구의 표면적: 5.147185403641517E8km^2

## • 자바의 패키지

- 클래스의 일부분이며, 클래스를 식별하는 용도
- 패키지는 주로 개발 회사의 도메인 이름의 역순으로 만듦
- 상위 패키지와 하위 패키지를 도트(.)로 구분
- 패키지에 속한 바이트코드 파일(~.class)은 따로 떼어내어 다른 디렉토리로 이동할 수 없음



- 패키지 선언

- 패키지 선언은 `package` 키워드와 함께 패키지 이름을 기술한 것. 항상 소스 파일 최상단에 위치

```
package 상위패키지.하위패키지;
```

```
public class 클래스명 { ... }
```

- 패키지 이름은 모두 소문자로 작성. 패키지 이름이 서로 중복되지 않도록 회사 도메인 이름의 역순으로 작성하고, 마지막에는 프로젝트 이름을 붙여줌

```
com.samsung.projectname
```

```
com.lg.projectname
```

```
org.apache.projectname
```


## • import문

- 다른 패키지에 있는 클래스를 사용하려면 **import** 문으로 어떤 패키지의 클래스를 사용하는지 명시

```
package com.mycompany;

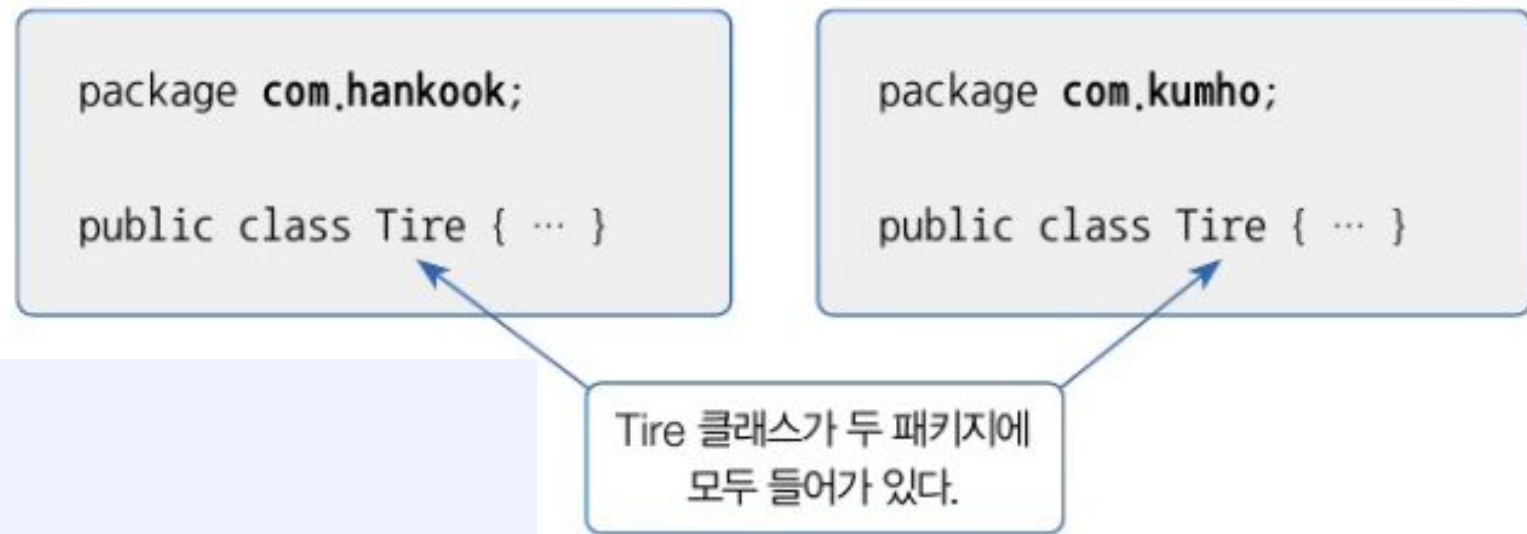
import com.hankook.Tire;

public class Car {
    //필드 선언 시 com.hankook.Tire 클래스를 사용
    Tire tire = new Tire();
}
```



- **import** 문은 패키지 선언과 클래스 선언 사이에 작성.
- **import** 키워드 뒤에는 사용하고자 하는 클래스의 전체 이름을 기술  
→ 일반적으로 **IDE**의 자동 완성 기능으로 자동 추가 이용

- 다른 패키지에 있는 동일 이름의 클래스를 사용하기
  - 이름 충돌 때문에 `import` 문 사용 불가
  - 전체 클래스 경로명을 사용



```
package com.hyundai;

import com.hankook.*;
import com.kumho.*;

public class Car {
    //필드 선언
    Tire tire = new Tire();
}
```

컴파일 에러

```
com.hankook.Tire tire = new com.hankook.Tire();
```

- **hankook.Tire.java**

```
package ch06.sec12.hankook;
```

```
public class Tire {  
}
```

- **hankook.SnowTire.java**

```
package ch06.sec12.hankook;
```

```
public class SnowTire {  
}
```

- **kumho.Tire.java**

```
package ch06.sec12.kumho;
```

```
public class Tire {  
}
```

- **kumho.AllSeasonTire.java**

```
package ch06.sec12.kumho;
```

```
public class AllSeasonTire {  
}
```

- **hyundai.Car.java**

```
package ch06.sec12.hyundai;
```

```
//import 문으로 다른 패키지 클래스 사용을 명시
```

```
import ch06.sec12.hankook.SnowTire;
```

```
import ch06.sec12.kumho.AllSeasonTire;
```

```
public class Car {
```

```
    //부품 필드 선언
```

```
    ch06.sec12.hankook.Tire tire1 = new ch06.sec12.hankook.Tire();
```

```
    ch06.sec12.kumho.Tire tire2 = new ch06.sec12.kumho.Tire();
```

```
    SnowTire tire3 = new SnowTire();
```

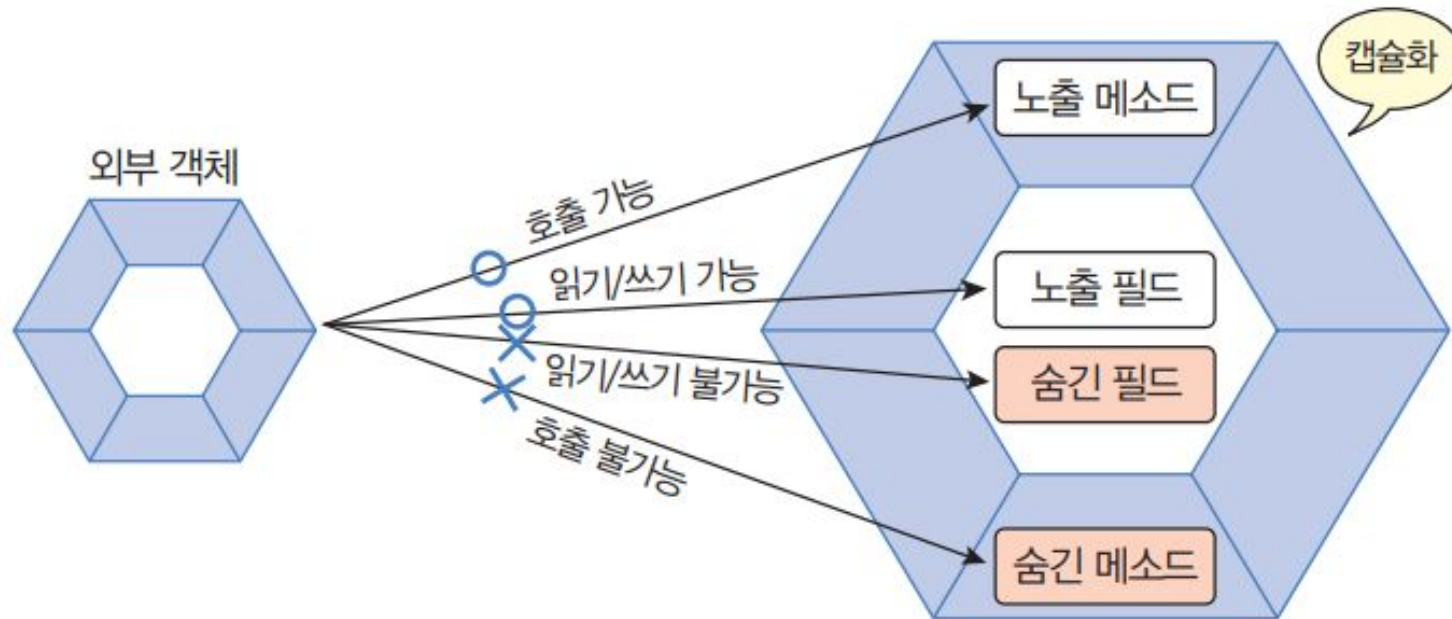
```
    AllSeasonTire tire4 = new AllSeasonTire();
```

```
}
```



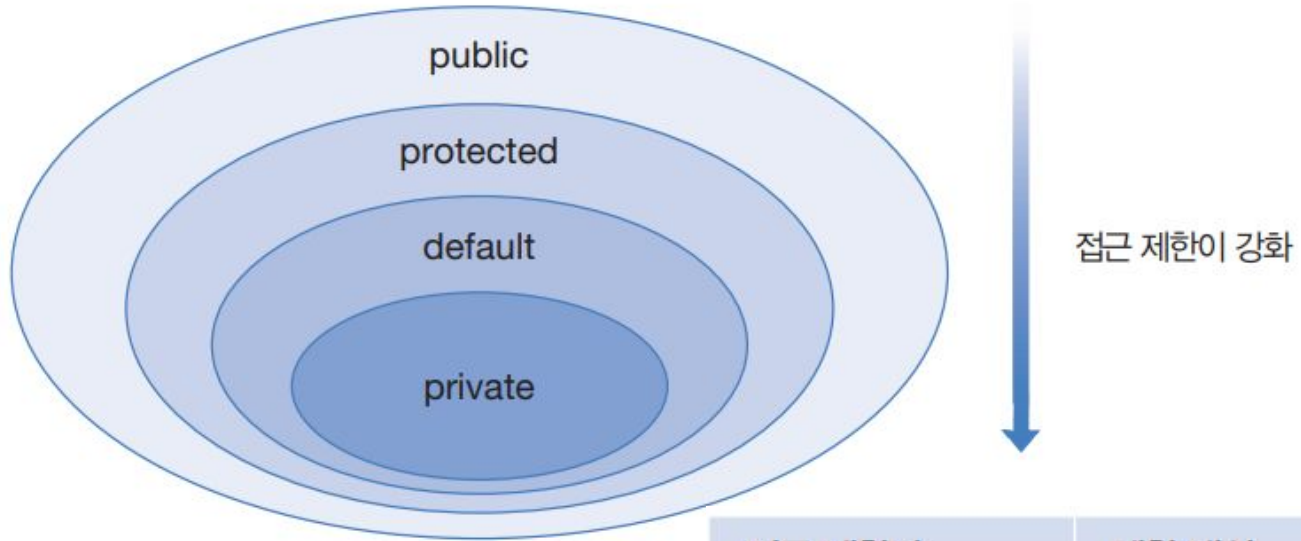
## • 접근 제한자

- 중요한 필드와 메소드가 외부로 노출되지 않도록 해 객체의 무결성을 유지하기 위해서 접근 제한자 사용



## • 접근 제한자

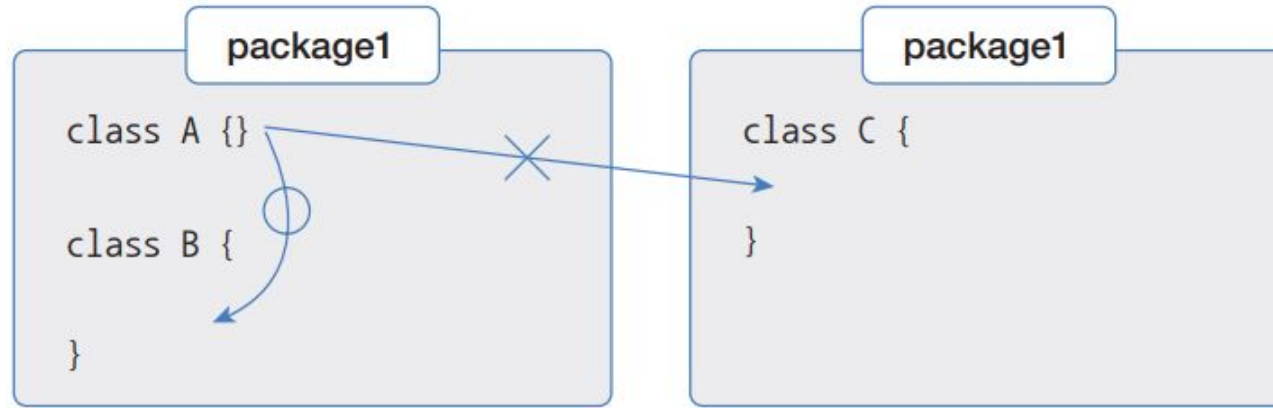
- 접근 제한자는 public, protected, private의 세 가지 종류



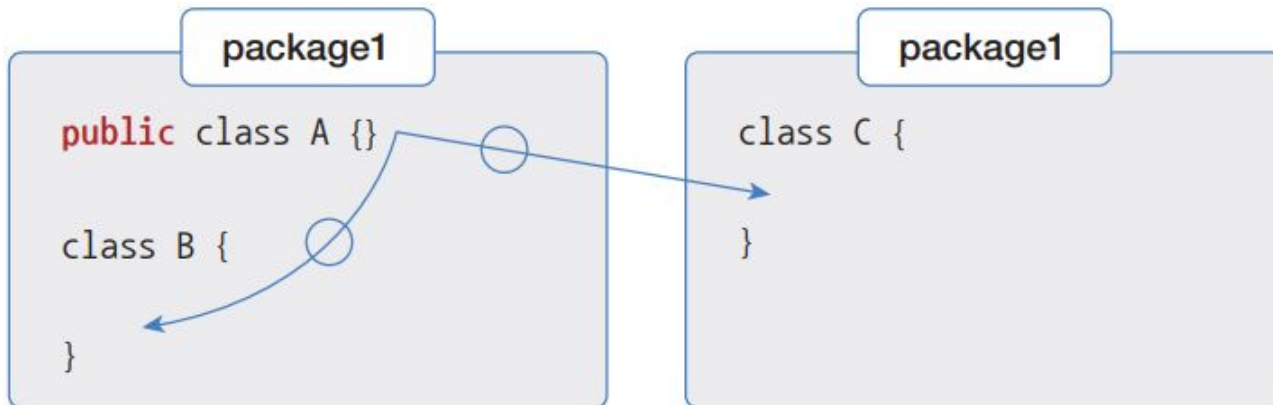
접근 제한자	제한 대상	제한 범위
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능 (7장 상속에서 자세히 설명)
(default)	클래스, 필드, 생성자, 메소드	같은 패키지
private	필드, 생성자, 메소드	객체 내부

- 클래스의 접근 제한

- 클래스를 선언할 때 **public** 접근 제한자를 생략하면 클래스는 다른 패키지에서 사용할 수 없음



- 클래스를 선언할 때 **public** 접근 제한자를 붙이면 클래스는 같은 패키지뿐만 아니라 다른 패키지에서도 사용할 수 있음



- **package1.A.java**

```
package ch06.sec13.exam01.package1;
```

```
class A {      // 디폴트 접근 제한  
}
```

- **package1.B.java**

```
package ch06.sec13.exam01.package1; // 동일 패키지
```

```
public class B {  
    //필드 선언  
    A a; //o      □ A 클래스 접근 가능(필드로 선언할 수 있음)  
}
```

- **package2.C.java**

```
package ch06.sec13.exam01.package2;           // 패키지 다름

import ch06.sec13.exam01.package1.*;

public class C {
    //필드 선언
    //A a; //x   □ A 클래스 접근 불가(컴파일 에러), 디폴트 접근 제한은 같은 패키지만 가능
    B b; //o
}
```

- 생성자의 접근 제한

- 생성자는 `public`, `default`, `private` 접근 제한을 가질 수 있음

```
public class ClassName {
    //생성자 선언
    [ public | private ] ClassName(...) { ... }
}
```

접근 제한자	생성자	설명
public	클래스(...)	모든 패키지에서 생성자를 호출할 수 있다. = 모든 패키지에서 객체를 생성할 수 있다.
	클래스(...)	같은 패키지에서만 생성자를 호출할 수 있다. = 같은 패키지에서만 객체를 생성할 수 있다.
private	클래스(...)	클래스 내부에서만 생성자를 호출할 수 있다. = 클래스 내부에서만 객체를 생성할 수 있다.

- **package1.A.java**

```
package ch06.sec13.exam02.package1;
```

```
public class A {
```

```
    //필드 선언
```

```
    A a1 = new A(true);
```

```
    A a2 = new A(1);
```

```
    A a3 = new A("문자열");
```

```
    //public 접근 제한 생성자 선언
```

```
    public A(boolean b) {  
}
```

```
    //default 접근 제한 생성자 선언
```

```
    A(int b) {  
}
```

```
    //private 접근 제한 생성자 선언
```

```
    private A(String s) {  
}
```

```
}
```

- **package1.B.java**

```
package ch06.sec13.exam02.package1;    // 패키지 동일

public class B {
    // 필드 선언
    A a1 = new A(true); //o
    A a2 = new A(1);      //o
    //A a3 = new A("문자열"); //x
}
```

□ **private** 생성자는 외부에서 접근 불가(컴파일 에러)



- **package2.C.java**

```
package ch06.sec13.exam02.package2;

import ch06.sec13.exam02.package1.*;

public class C {
    //필드 선언
    A a1 = new A(true); //o
    //A a2 = new A(1);      //x
    //A a3 = new A("문자열"); //x
}
```

- default 생성자 접근 불가
- private 생성자 접근 불가

## • 필드와 메소드의 접근 제한

- 필드와 메소드는 `public`, `default`, `private` 접근 제한을 가질 수 있음

```
//필드 선언
[ public | private ] 타입 필드;

//메소드 선언
[ public | private ] 리턴타입 메소드(...) { ... }
```

접근 제한자	생성자	설명
public	필드 메소드(...)	모든 패키지에서 필드를 읽고 변경할 수 있다. 모든 패키지에서 메소드를 호출할 수 있다.
	필드 메소드(...)	같은 패키지에서만 필드를 읽고 변경할 수 있다. 같은 패키지에서만 메소드를 호출할 수 있다.
private	필드 메소드(...)	클래스 내부에서만 필드를 읽고 변경할 수 있다. 클래스 내부에서만 메소드를 호출할 수 있다.

- package1.A.java

```
package ch06.sec13.exam03.package1;
```

```
public class A {
```

```
    //public 접근 제한을 갖는 필드 선언
```

```
    public int field1;
```

```
    //default 접근 제한을 갖는 필드 선언
```

```
    int field2;
```

```
    //private 접근 제한을 갖는 필드 선언
```

```
    private int field3;
```

```
    //생성자 선언
```

```
    public A() {
```

```
        field1 = 1;           //o
```

```
        field2 = 1;           //o
```

```
        field3 = 1;           //o
```

```
        method1();            //o
```

```
        method2();            //o
```

```
        method3();            //o
```

```
    }
```

```
    //public 접근 제한을 갖는 메소드 선언
```

```
    public void method1() {
```

```
    }
```

```
    //default 접근 제한을 갖는 메소드 선언
```

```
    void method2() {
```

```
    }
```

```
    //private 접근 제한을 갖는 메소드 선언
```

```
    private void method3() {
```

```
    }
```

```
}
```

- **package1.B.java**

```
package ch06.sec13.exam03.package1;    // 패키지가 동일

public class B {
    public void method() {
        //객체 생성
        A a = new A();

        //필드값 변경
        a.field1 = 1;        // o
        a.field2 = 1;        // o □ 패키지가 같으면 같은 패키지에서는 가능
        //a.field3 = 1;    // x □ private 필드 접근 불가(컴파일 에러)

        //메소드 호출
        a.method1();        // o
        a.method2();        // o □ 패키지가 같으면 같은 패키지에서는 가능
        //a.method3();    // x □ private 필드 접근 불가(컴파일 에러)
    }
}
```

- package2.C.java

```
package ch06.sec13.exam03.package2;    // 패키지가 다름

import ch06.sec13.exam03.package1.*;

public class C {
    public C() {
        //객체 생성
        A a = new A();

        //필드값 변경
        a.field1 = 1;           // (o)
        //a.field2 = 1; // (x) → 패키지가 다르면 default 접근 불가(컴파일 에러)
        //a.field3 = 1; // (x) → private 접근 불가(컴파일 에러)

        //메소드 호출
        a.method1();           // (o)
        //a.method2(); // (x) → 패키지가 다르면 default 접근 불가(컴파일 에러)
        //a.method3(); // (x) → private 접근 불가(컴파일 에러)
    }
}
```

- 객체의 필드(데이터) 은닉

- 외부에서 직접 접근하는 경우 잘못된 데이터 입력 가능  
→ 객체의 무결성(결점이 없는 성질)이 깨짐
  - private 또는 디폴트 접근 제한자로 보호
- Setter 메서드
  - 데이터를 검증해서 유효한 값만 필드에 저장하는 메소드

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

```
private double speed;
```

```
public void setSpeed(double speed) {
```

```
    if(speed < 0) {
```

```
        this.speed = 0;
```

```
        return;
```

```
    } else {
```

```
        this.speed = speed;
```

```
    }
```

```
}
```

매개값이 음수일 경우 speed 필드에  
0으로 저장하고, 메소드 실행 종료

- 객체의 필드(데이터) 은닉

- Getter 메서드

- 필드값이 객체 외부에서 사용하기에 부적절한 경우, 적절한 값으로 변환해서 리턴할 수 있는 메소드

```
private double speed;    //speed의 단위는 마일
```

```
public double getSpeed() {  
    double km = speed*1.6;  
    return km;  
}
```

필드값인 마일을 km 단위로 환산 후  
외부로 리턴

## 14 Getter와 Setter

- 객체의 필드(데이터) 은닉
  - 필드에 대한 접근은 Getter, Setter 메서드로 접근

```
private 타입 fieldName;
```

필드 접근 제한자: private

```
//Getter
```

```
public 타입 getFieldName() {  
    return fieldName;  
}
```

접근 제한자: public

리턴 타입: 필드타입

메소드 이름: get + 필드이름(첫 글자 대문자)

리턴값: 필드값

```
//Setter
```

```
public void setFieldName(타입 fieldName) {  
    this.fieldName = fieldName;  
}
```

접근 제한자: public

리턴 타입: void

메소드 이름: set + 필드이름(첫 글자 대문자)

매개변수 타입: 필드타입



## 14 Getter와 Setter

- 객체의 필드(데이터) 은닉
  - boolean 타입에 대한 Getter 메서드는 isXxxx()가 관례

```
private boolean stop;
```

필드 접근 제한자: private

```
//Getter
```

```
public boolean isStop() {  
    return stop;  
}
```

접근 제한자: public

리턴 타입: 필드타입

메소드 이름: is + 필드이름(첫 글자 대문자)

리턴값: 필드값

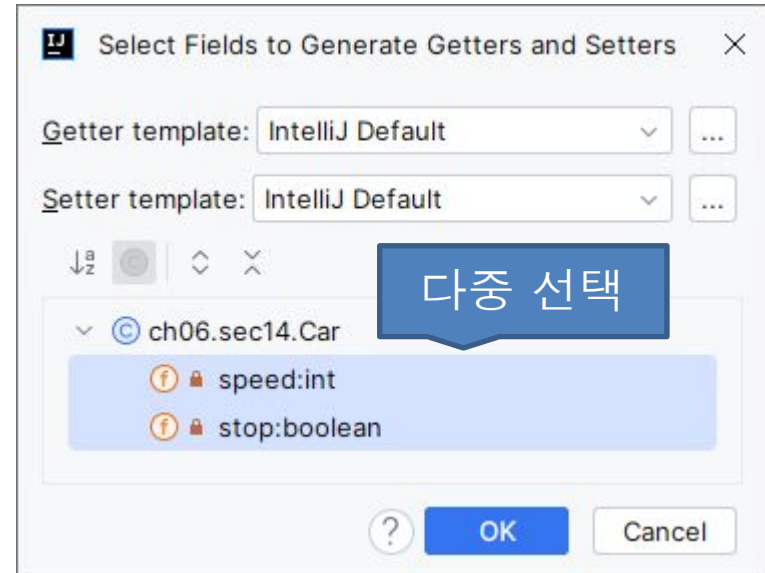
- **Getter, Setter 자동 생성**
  - IDE에서 자동 생성 기능 제공

```
public class Car {  
    // 필드 선언  
    no usages  
    private int speed;  
    no usages  
    private boolean stop;
```

Alt + Ins

**Generate**

Constructor  
Getter  
Setter  
**Getter and Setter**  
equals() and hashCode()  
toString()  
Override Methods... Ctrl+O  
Test...  
Copyright



- **Car.java**

```
package ch06.sec14;

public class Car {
    //필드 선언
    private int speed;
    private boolean stop;

    //speed 필드의 Getter/Setter 선언
    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        if(speed < 0) {
            this.speed = 0;
            return;
        } else {
            this.speed = speed;
        }
    }

    //stop 필드의 Getter/Setter 선언
    public boolean isStop() {
        return stop;
    }

    public void setStop(boolean stop) {
        this.stop = stop;
        if(stop == true) this.speed = 0;
    }
}
```

- CarExample.java

```
package ch06.sec14;

public class CarExample {
    public static void main(String[] args) {
        //객체 생성
        Car myCar = new Car();

        //잘못된 속도 변경
        myCar.setSpeed(-50);
        System.out.println("현재 속도: " + myCar.getSpeed());

        //올바른 속도 변경
        myCar.setSpeed(60);
        System.out.println("현재 속도: " + myCar.getSpeed());

        //멈춤
        if(!myCar.isStop()) {
            myCar.setStop(true);
        }
        System.out.println("현재 속도: " + myCar.getSpeed());
    }
}
```

```
현재 속도: 0
현재 속도: 60
현재 속도: 0
```

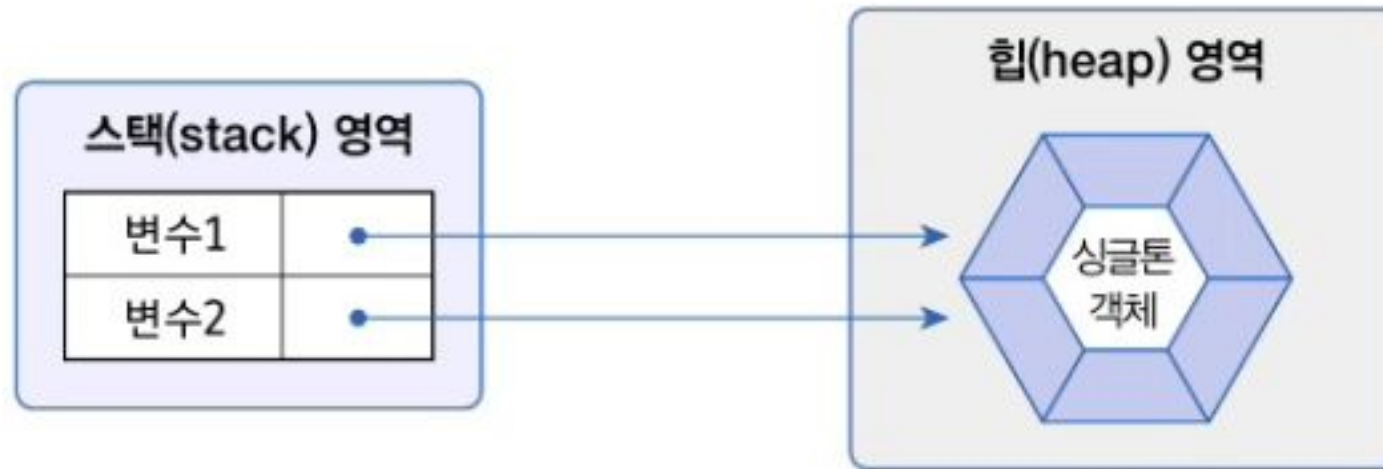
- 싱글톤 Singleton 패턴

- 생성자를 **private** 접근 제한해서 외부에서 **new** 연산자로 생성자를 호출할 수 없도록 막아서 외부에서 마음대로 객체를 생성하지 못하게 함
- 대신 싱글톤 패턴이 제공하는 **정적 메소드를 통해 간접적으로 객체를 얻을 수 있음**

```
public class 클래스 {  
    //private 접근 권한을 갖는 정적 필드 선언과 초기화  
    private static 클래스 singleton = new 클래스(); ..... ①  
  
    //private 접근 권한을 갖는 생성자 선언  
    private 클래스() {}  
  
    //public 접근 권한을 갖는 정적 메소드 선언  
    public static 클래스 getInstance() { ..... ②  
        return singleton;  
    }  
}
```

- 싱글톤 Singleton 패턴

```
클래스 변수1 = 클래스.getInstance();  
클래스 변수2 = 클래스.getInstance();
```



- Singleton.java

```
package ch06.sec15;

public class Singleton {
    //private 접근 권한을 갖는 정적 필드 선언과 초기화
    private static Singleton singleton = new Singleton();

    //private 접근 권한을 갖는 생성자 선언
    private Singleton() {
    }

    //public 접근 권한을 갖는 정적 메소드 선언
    public static Singleton getInstance() {
        return singleton;
    }
}
```

## ● SingletonExample.java

```
package ch06.sec15;

public class SingletonExample {
    public static void main(String[] args) {
        /*
        Singleton obj1 = new Singleton(); //컴파일 에러
        Singleton obj2 = new Singleton(); //컴파일 에러
        */

        //정적 메소드를 호출해서 싱글톤 객체 얻음
        Singleton obj1 = Singleton.getInstance();
        Singleton obj2 = Singleton.getInstance();

        //동일한 객체를 참조하는지 확인
        if(obj1 == obj2) {
            System.out.println("같은 Singleton 객체입니다.");
        } else {
            System.out.println("다른 Singleton 객체입니다.");
        }
    }
}
```

같은 Singleton 객체입니다.