

# 6장 클래스

---

한림대학교 소프트웨어융합대학 양은샘.



**이것이 자바다**  
신용권의 Java 프로그래밍 정복

저자 신용권  
출판 한빛미디어 | 2015.1.5  
페이지수 1,224 | 사이즈 183\*235mm  
판매가 **서적** 27,000원  
구매이벤트 IT독자 설문이벤트 외 6건



**혼자 공부하는 자바**  
JAVA 8 & 11 지원/무료 동영상 강의 제공

저자 신용권  
출판 한빛미디어 | 2019.6.10.  
페이지수 708 | 사이즈 188\*257mm  
판매가 **서적** 21,600원

# 6장 클래스

---

- 안녕하세요? 여러분!
- 오늘은 JAVA 프로그래밍I 강좌에서 배운 클래스 단원을 복습합니다.
- 이번 장에서는 클래스에서 반드시 알아야 하는 개념인 "필드", "생성자", "메소드" 등을 알아보도록 하겠습니다.
- 그럼 학습을 시작하도록 하겠습니다.

# 학습 목차

---

1. 객체 지향 프로그래밍
2. 객체(Object)와 클래스(Class)
3. 클래스 선언
4. 객체 생성과 클래스 변수
5. 클래스의 구성 멤버
6. 필드(Field)
7. 생성자(Constructor)
8. 메소드(Method)
9. 인스턴스 멤버와 this
10. 정적 멤버와 static
11. final 필드와 상수(static final)
12. 패키지(package)
13. 접근 제한자

# 학습 목표

---

- 클래스의 구성요소 및 사용법을 안다.
- 클래스로부터 객체를 생성하고 변수로 참조할 수 있다.
- 객체의 개념과 객체의 상호작용을 안다.
- 생성자와 생성자 오버로딩을 활용할 수 있다.
- 메소드와 메소드 오버로딩을 활용할 수 있다.
- this. 와 this()의 차이점을 안다.
- static의 의미와 사용법을 안다.
- 싱글톤의 의미를 안다.
- final의 의미와 사용법을 안다.
- 상수를 만들어 사용 할 수 있다.
- 접근 제한자의 의미와 접근 범위를 안다.

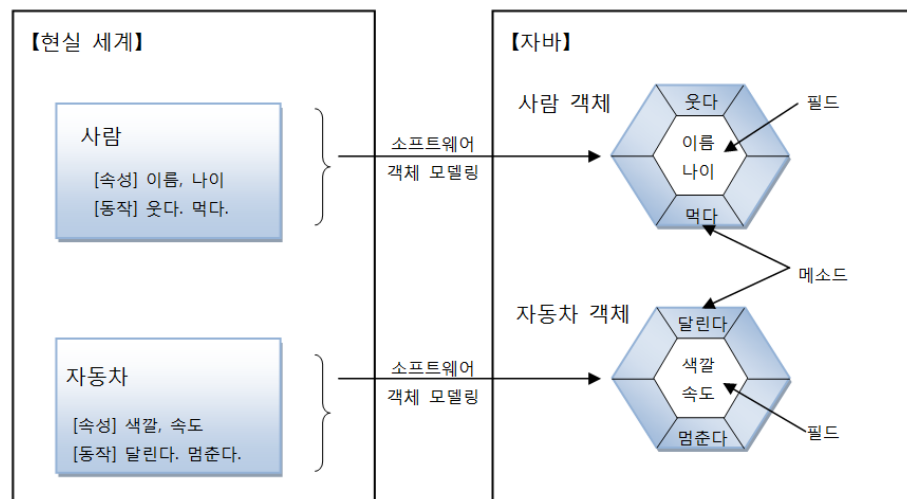
# 객체 지향 프로그래밍

## ■ 객체 지향 프로그래밍

- OOP: Object Oriented Programming
- 부품 객체를 먼저 만들고 이것들을 하나씩 조립해 완성된 프로그램을 만드는 기법

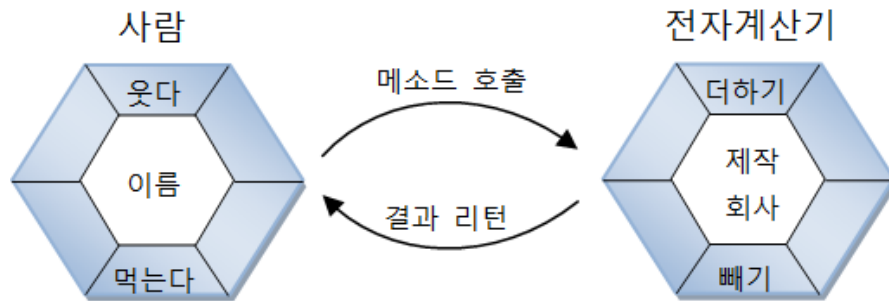
## ■ 객체(Object)란?

- 물리적으로 존재하거나 추상적으로 생각할 수 있는 것 중에서 자신의 속성을 가지며 식별 가능한 것.
- 객체 = 생성자(constructor) + 속성(필드(field)) + 동작(메소드(method))로 구성



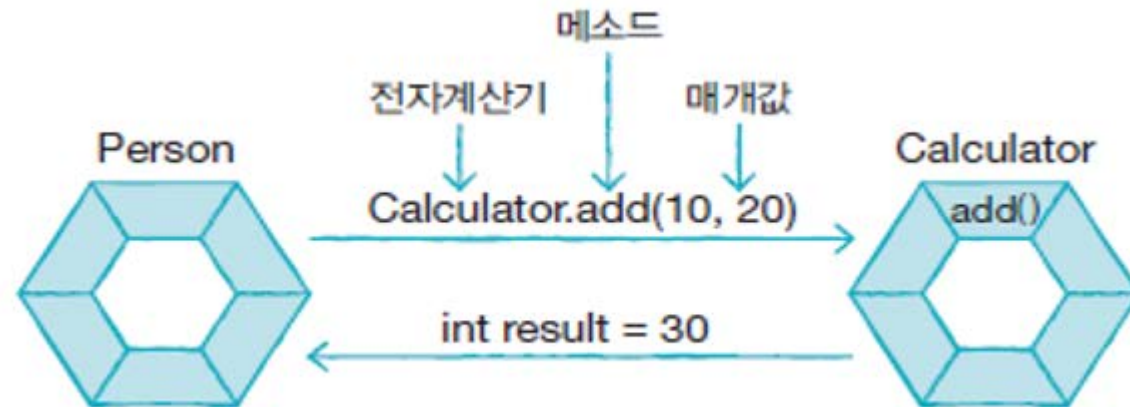
# 객체의 상호 작용

- 메소드를 통해 객체들이 상호 작용
- 메소드 호출 : 객체가 다른 객체의 기능을 이용하는 것



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

리턴한 값을 int 변수에 저장



# 객체간의 관계

## ■ 객체 간의 관계

- 집합 관계

- 완성품과 부품의 관계

- 사용 관계

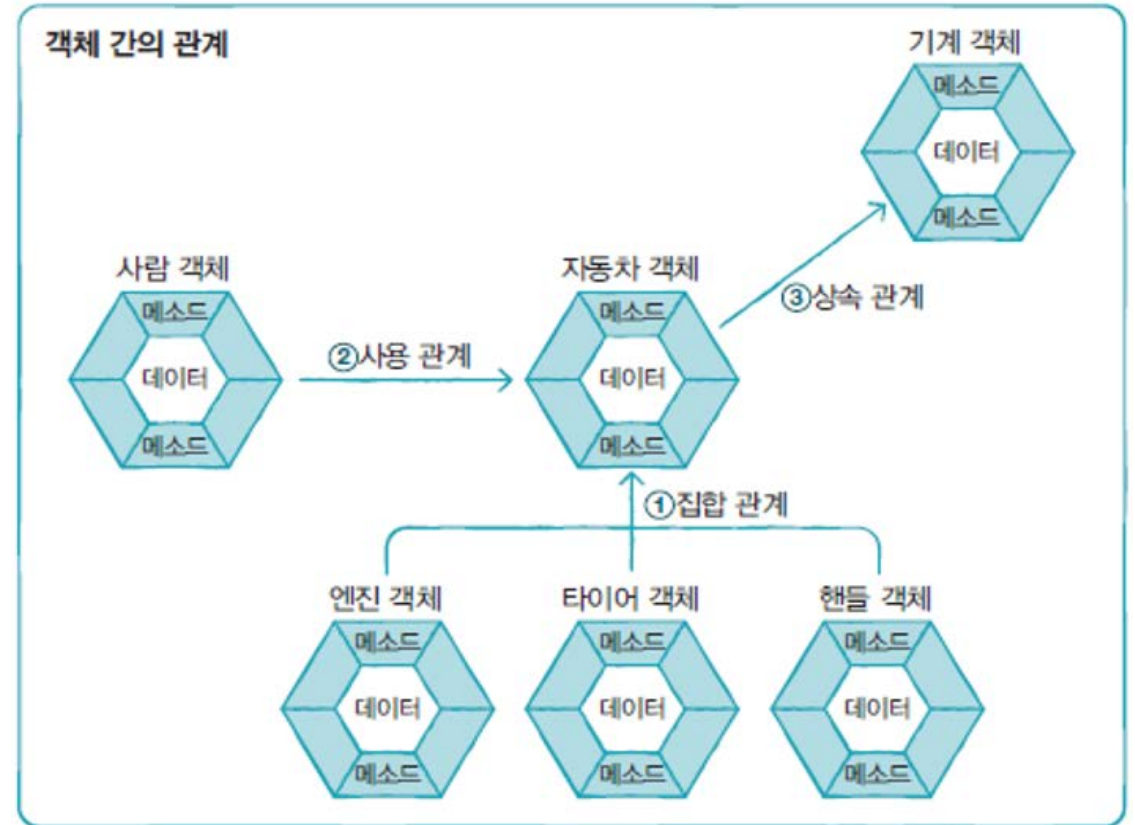
- 객체가 다른 객체를 사용하는 관계

- 상속 관계

- 상위(부모) 객체를 기반으로 하위(자식) 객체를 생성

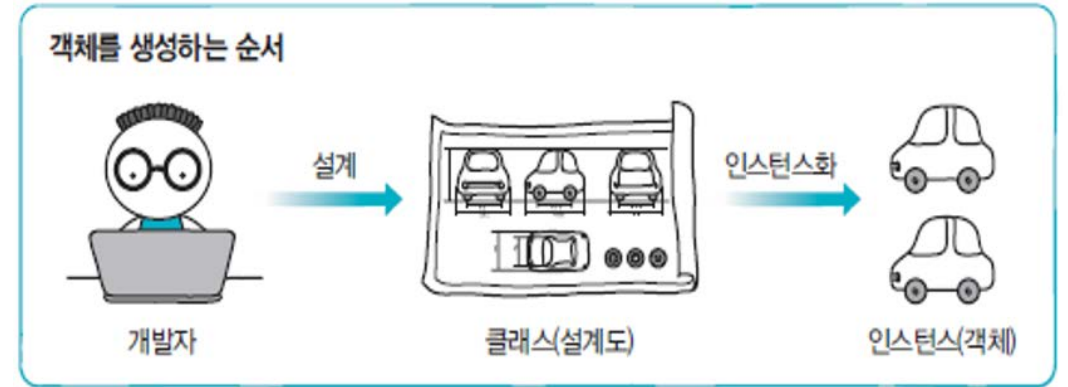
## ■ 객체 지향 프로그램

- 집합/사용 관계에 있는 객체를 하나씩 설계한 후 조립하여 프로그램 개발



# 객체(Object)와 클래스(Class)

- 클래스(Class)
  - 자바의 설계도
  - 객체를 생성하기 위한 필드, 생성자, 메소드 정의
- 객체(Object)
  - 인스턴스(instance) : 클래스로부터 만들어진 객체
  - 하나의 클래스로부터 여러 개의 인스턴스를 만들 수 있음
- 객체지향 프로그래밍 단계
  - 클래스 설계 -> 설계된 클래스로 사용할 객체 생성 -> 객체 이용





# 클래스의 이름

## ■ 자바 식별자 작성 규칙

- 한글 이름도 가능하나, 영어 이름으로 작성
- 알파벳 대소문자는 서로 다른 문자로 인식
- 첫 글자와 연결된 다른 단어의 첫 글자는 대문자로 작성하는 것이 관례

번호	작성 규칙	예
1	하나 이상의 문자로 이루어져야 한다.	Car, SportsCar
2	첫 번째 글자는 숫자가 올 수 없다.	Car, 3Car(x)
3	'\$', '_' 외의 특수 문자는 사용할 수 없다.	\$Car, _Car, @Car(x), #Car(x)
4	자바 키워드는 사용할 수 없다.	int(x), for(x)

```
public class 클래스이름 {  
  
}
```

# 클래스 선언과 컴파일

- 소스 파일 생성 : 클래스이름.java (대소문자 주의)

- 소스 작성

```
public class 클래스이름 {  
  
}
```

컴파일

javac.exe

클래스이름.class

- 소스 파일당 하나의 클래스를 선언하는 것이 관례
  - 두 개 이상의 클래스도 선언 가능
- 소스 파일 이름과 동일한 클래스만 public으로 선언 가능
- 선언한 개수만큼 바이트 코드 파일 생성

```
Car.java  
public class Car {  
}  
  
class Tire {  
}
```

컴파일  
javac.exe

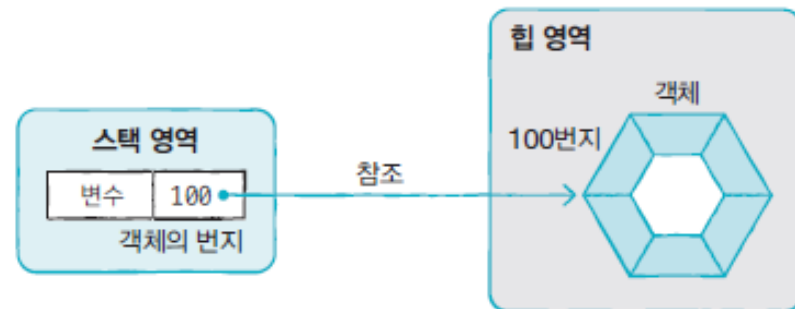
Car.class  
Tire.class

# 객체 생성

- new 연산자
  - 객체 생성 역할
  - new 클래스(); //생성자를 호출하는 코드
  - new 연산자는 힙 메모리 영역에 객체 생성 후, 객체의 생성 번지를 리턴

```
클래스 변수;  
변수 = new 클래스();
```

```
클래스 변수 = new 클래스();
```

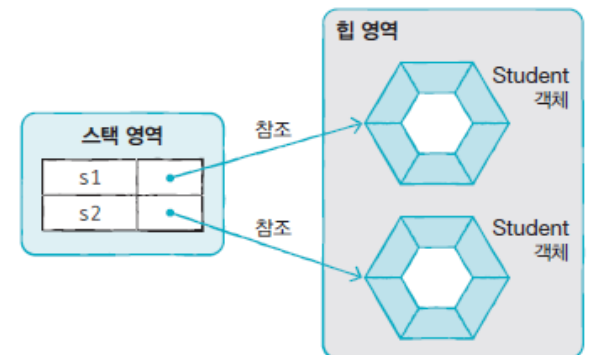


# 클래스의 두 용도

- 라이브러리(API : Application Program Interface) 클래스
  - 다른 클래스에서 이용할 목적으로 만든 클래스
  - 객체 생성 및 메소드 제공 역할 : Student.java
- 실행 클래스
  - main() 메소드를 가지고 있는 클래스로, 실행할 목적으로 만든 클래스 : StudentExample.java

```
public class Student {  
}
```

```
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.");  
    }  
}
```



1개의 애플리케이션 = (1개의 실행클래스) + (n개의 라이브러리 클래스)

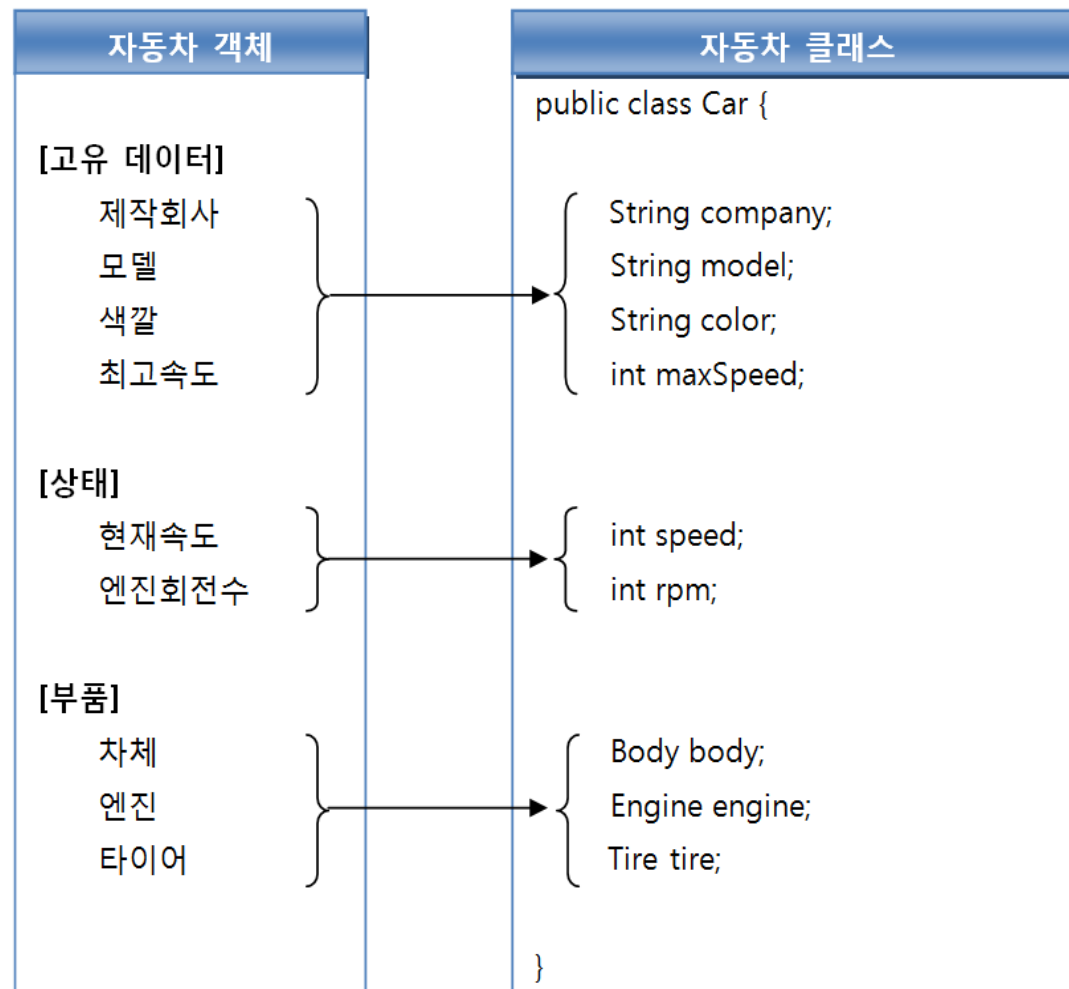
# 클래스 구성 멤버

- **필드(Field)** ————— 객체의 데이터가 저장되는 곳
- **생성자(Constructor)** ————— 객체 생성시 초기화 역할 담당
- **메소드(Method)** ————— 객체의 동작에 해당하는 실행 블록

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

# 필드(field)

- 필드(field)
  - 객체의 고유 데이터
  - 객체가 가져야 할 부품 값 또는 객체
  - 객체의 현재 상태 데이터 등을 저장
- 필드 선언
  - 클래스의 중괄호 블록 어디서든 존재 가능
  - 생성자와 메소드의 중괄호 블록 내부에는 선언 불가
  - 변수와 선언 형태 유사하나 변수 아님에 주의



# 필드(field) 초기 값

- 초기값은 주어질 수도, 생략할 수도 있음
- 초기값이 지정되지 않은 필드
  - 객체 생성시 자동으로 기본값으로 초기화

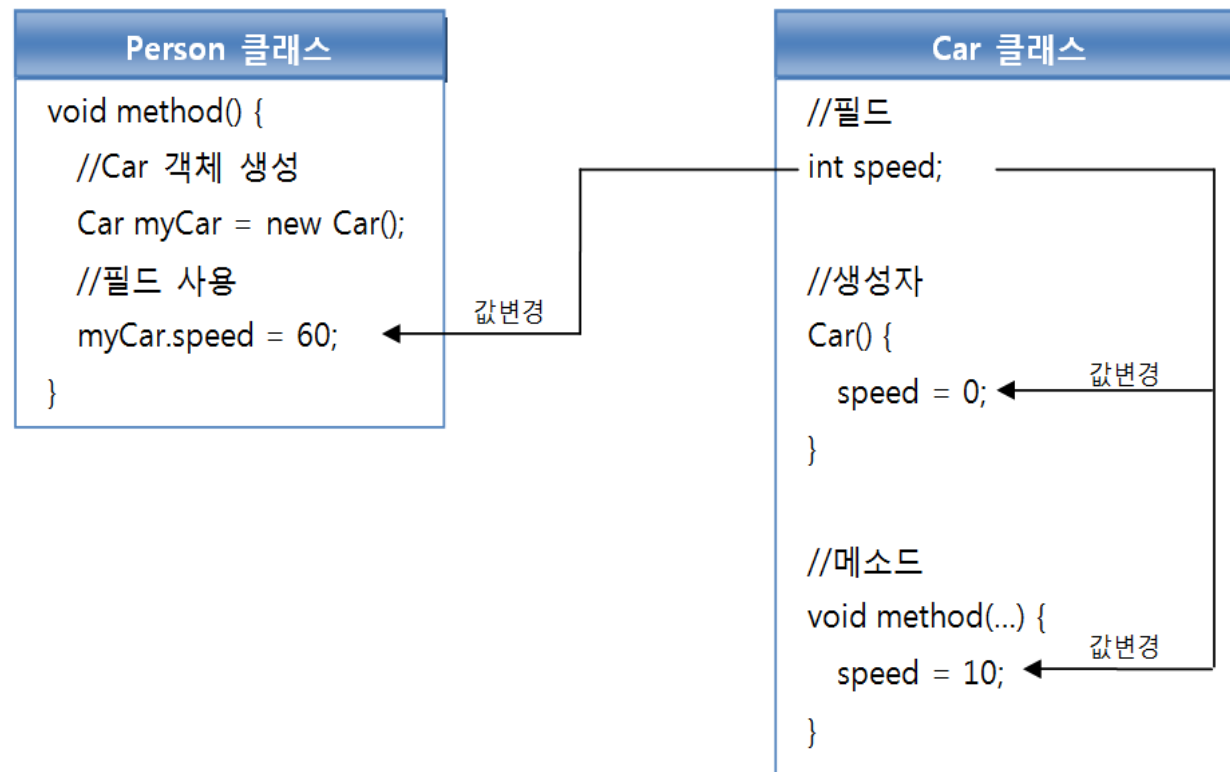
타입 필드 [= 초기값];

```
String company = "현대자동차";  
String model = "그랜저";  
int maxSpeed = 300;  
int productionYear;  
int currentSpeed;  
boolean engineStart;
```

분류		데이터 타입	초기값
기본 타입	정수 타입	byte	0
		char	������ (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

# 필드 사용

- 필드 값을 읽고, 변경하는 작업
  - 객체 내부: "필드이름" 으로 바로 접근
  - 객체 외부: "객체변수.필드이름" 으로 접근



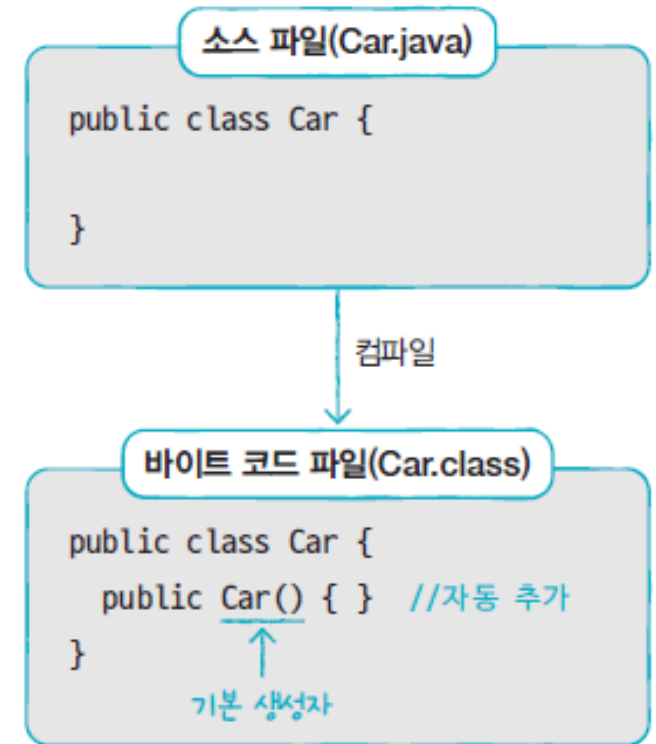


# 생성자(constructor)

- new 연산자에 의해 호출되어 객체의 초기화 담당
- 기본 생성자(Default Constructor)
  - 모든 클래스는 생성자가 반드시 존재하며 하나 이상 가질 수 있음
  - 생성자 선언을 생략하면 컴파일러는 다음과 같은 기본 생성자 추가

```
Car myCar = new Car();
```

↑  
기본 생성자



# 생성자 선언

---

- 디폴트 생성자 대신 개발자가 직접 선언
- 개발자 선언한 생성자 존재 시 컴파일러는 기본 생성자를 추가하지 않음
- 클래스에 생성자가 명시적으로 선언되었을 경우 반드시 선언된 생성자를 호출하여 객체 생성
- 매개 변수 선언은 생략할 수도 있고 여러 개 선언할 수도 있음

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

```
Car myCar = new Car("그랜저", "검정", 300);
```

# 생성자의 필드 초기화

- 매개 변수 이름은 필드 이름과 유사하거나 동일한 것을 사용하기를 권장
- 필드와 매개 변수 이름 완전히 동일할 경우 this.필드로 표현

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

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

```
public Korean(String name, String ssn) {  
    this.name = name;  
        ↑      ↑  
    필드 매개 변수  
    this.ssn = ssn;  
        ↑      ↑  
    필드 매개 변수  
}
```

# 생성자 오버로딩(overloading)

- 매개변수의 타입, 개수, 순서가 다른 생성자를 여러 개 선언
  - 매개 변수의 타입, 개수, 선언된 순서 같은 경우, 매개 변수 이름만 바꾸는 것은 생성자 오버로딩 아님
- 외부에서 제공되는 다양한 데이터를 사용하여 객체화하기 위해 생성자 오버로딩 사용

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

```
Car car1 = new Car();  
Car car2 = new Car("그랜저");  
Car car3 = new Car("그랜저", "흰색");  
Car car4 = new Car("그랜저", "흰색", 300);
```

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

# 다른 생성자 호출 this()

- 필드의 초기화 내용을 한 생성자에만 집중해서 작성
- 나머지 생성자는 초기화 내용 가진 생성자인 this()를 호출
- 생성자 오버로딩 증가 시 중복 코드 발생 문제 해결
- this()는 생성자 첫 줄에서만 허용

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

} 중복 코드

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

} 중복 코드

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

} 중복 코드

```
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;  
}
```

} 공통 실행 코드



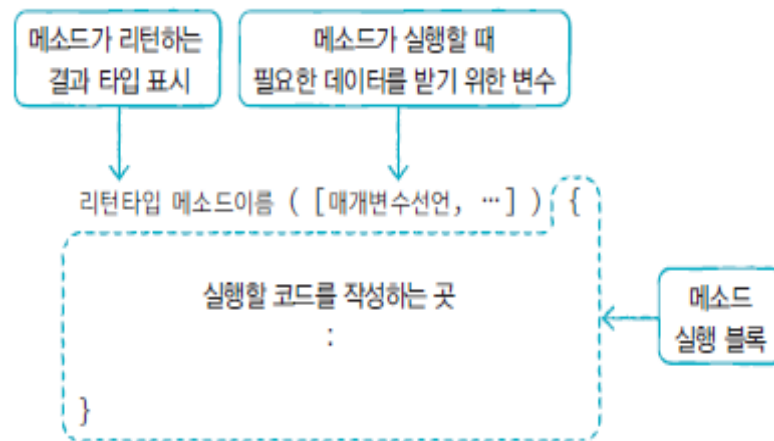
# 메소드(method)

## ■ 메소드란?

- 객체의 동작(기능)
- 메소드를 호출하면 중괄호 { } 블록에 있는 모든 코드들이 일괄 실행 됨

## ■ 메소드 선언

- 리턴 타입 : 메소드가 리턴하는 결과의 타입 표시
- 메소드 이름 : 메소드의 기능 드러나도록 식별자 규칙에 맞는 이름 권장
- 매개 변수 선언 : 메소드 실행할 때 필요한 데이터를 받기 위한 변수 선언
- 메소드 실행 블록 : 실행할 코드 작성



# 메소드 리턴 타입

## ■ 리턴 타입

- 메소드를 실행한 후 결과 값의 타입
- 리턴 값 없을 수도 있음
- 리턴 값 있는 경우 리턴 타입이 선언부에 명시

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

- 리턴 값 존재 여부에 따라 메소드 호출 방법 다름

```
powerOn();  
double result = divide( 10, 20 );
```

## ■ 메소드 이름

- 숫자로 시작할 수 없음.
- \$와 \_ 제외한 특수문자 사용 불가
- 메소드 이름은 관례적으로 소문자로 작성
- 서로 다른 단어가 혼합된 이름 : 뒤이어 오는 단어의 첫 글자를 대문자로 작성

```
int result = divide( 10, 20 ); //컴파일 에러
```

```
void run() { ... }  
void startEngine() { ... }  
String getName() { ... }  
int[] getScores() { ... }
```

# 메소드 매개변수

---

- 매개 변수 선언

- 메소드 실행에 필요한 데이터를 외부에서 받아 저장할 목적으로 사용

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

```
double result = divide( 10, 20 );
```

```
byte b1 = 10;
```

```
byte b2 = 20;
```

```
double result = divide( b1, b2 );
```

- 잘못된 매개 값을 사용하여 컴파일 에러가 발생하는 경우

```
double result = divide( 10.5, 20.0 );
```



# 메소드 매개 변수의 개수를 모를 경우

---

- 매개 변수를 배열 타입으로 선언

```
int sum1(int[] values) { }
```

```
int[] values = { 1, 2, 3 };
```

```
int result = sum1(values);
```

```
int result = sum1(new int[] { 1, 2, 3, 4, 5 });
```

# 리턴(return)

- 메소드 선언에 리턴 타입 있는 메소드는 return을 사용하여 리턴 값 지정

```
boolean isLeftGas() {  
    if(gas==0) {  
        System.out.println("gas 가 없습니다.");  
        return false;  
    }  
    System.out.println("gas 가 있습니다.");  
    return true;  
}
```

- return의 리턴 값은 리턴 타입이거나 리턴 타입으로 변환될 수 있어야 함


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

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

# 메소드 실행을 강제 종료

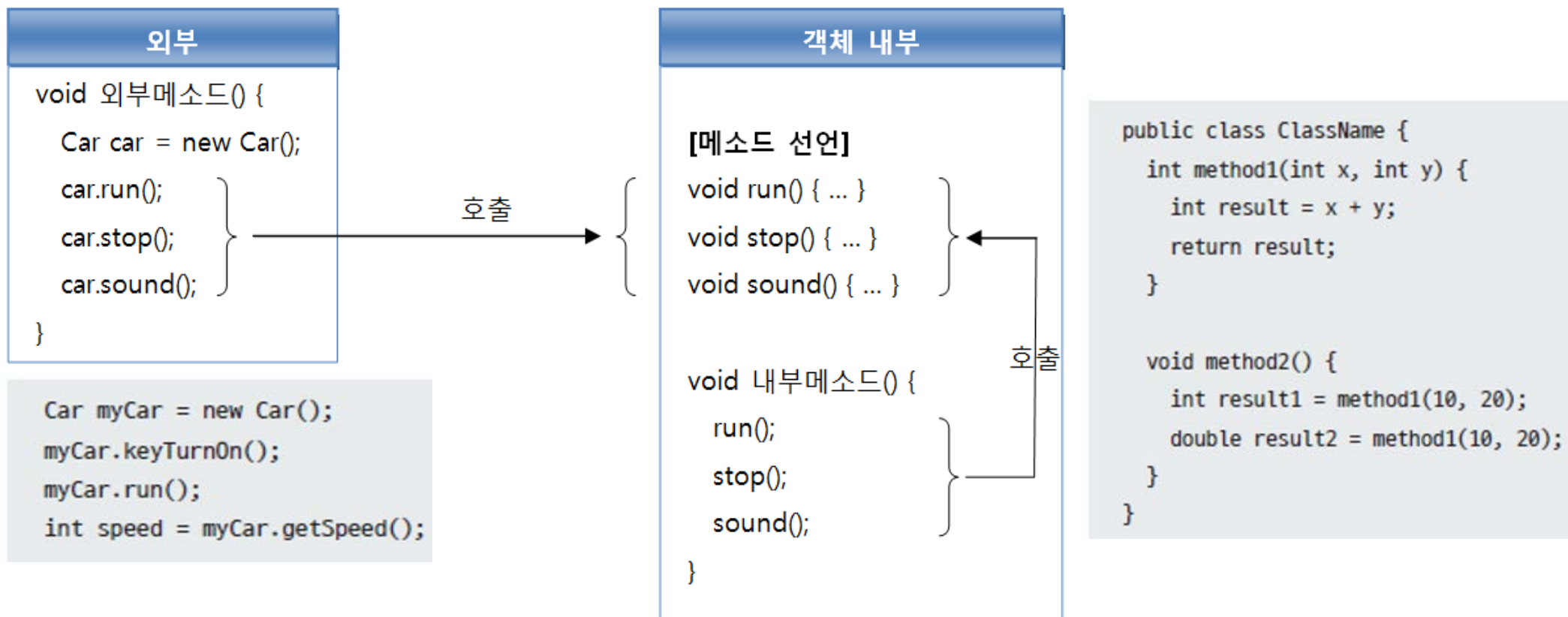
- void로 선언된 메소드에서 return을 사용하여 메소드 실행을 강제로 종료

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



# 메소드 호출

- 클래스 내/외부의 호출에 의해 메소드 실행
  - 클래스 내부 : 메소드 이름으로 호출
  - 클래스 외부 : 객체 생성 후, 참조 변수를 이용해 호출



# Getter와 Setter

- 클래스를 선언할 때 필드는 일반적으로 private 접근 제한
  - 외부에서 객체에 마음대로 접근할 경우 객체의 무결성이 깨질 수 있기 때문

- Setter

- 외부의 값을 받아 필드의 값을 변경하는 것이 목적
- 매개 값을 검증하여 유효한 값만 필드로 저장할 수 있음
- setFieldName(타입 변수) 메소드 사용

- Getter

- 외부로 private 필드 값을 전달하는 것이 목적
- 필드 값을 가공해서 외부로 전달할 수도 있음
- getFieldName() 또는 isFieldName() 메소드
- 필드 타입이 boolean일 경우 isFieldName()

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

```
    if(speed < 0) {  
        this.speed = 0;  
        return;
```

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

```
    } else {  
        this.speed = speed;  
    }  
}
```

```
double getSpeed() {
```

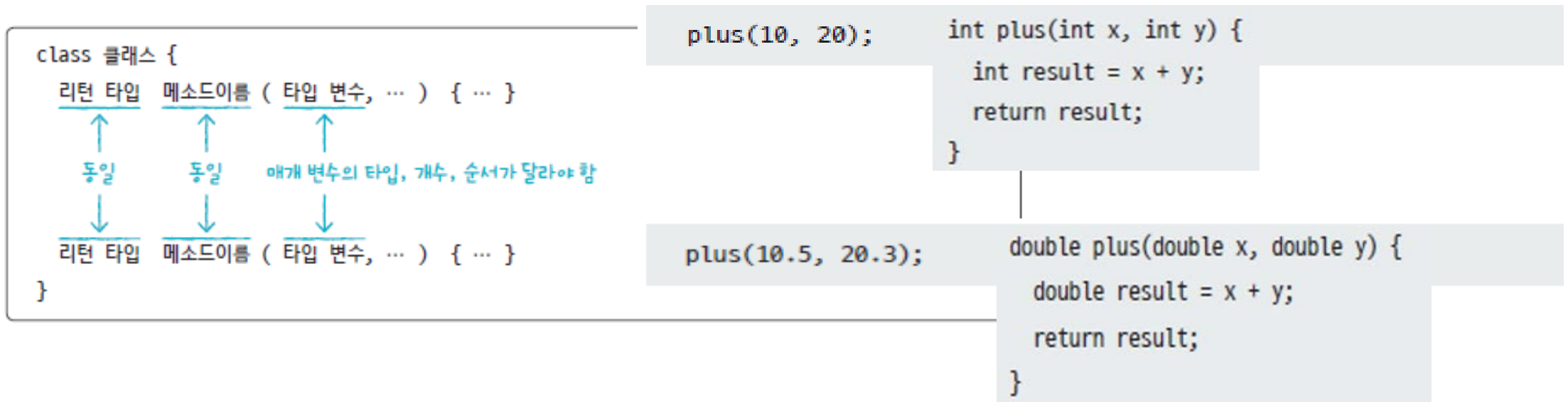
```
    double km = speed*1.6;  
    return km;
```

← 필드값인 마일을 km 단위로 환산 후 외부로 리턴

```
}
```

# 메소드 오버로딩(Overloading)

- 클래스 내에 같은 이름의 메소드를 여러 개 선언하는 것
- 하나의 메소드 이름으로 다양한 매개 값을 받기 위해 메소드 오버로딩 사용
  - 오버로딩의 조건 : 매개 변수의 타입, 개수, 순서 중 하나라도 달라야 함



- 메소드 오버로딩이 아닌 경우

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

# 인스턴스 멤버와 this

---

## ■ 인스턴스 멤버란?

- 객체(인스턴스) 마다 가지고 있는 필드와 메소드 (인스턴스 필드, 인스턴스 메소드라고 부름)
- 인스턴스 멤버는 객체에 소속된 멤버이기 때문에 객체 없이는 사용불가

## ■ this

- 객체(인스턴스) 자신의 참조(번지)를 가지고 있는 키워드
- 객체 내부에서 인스턴스 멤버임을 명확히 하기 위해 this. 사용
- 매개변수와 필드명이 동일할 때 인스턴스 필드임을 명확히 하기 위해 사용

```
Car(String model) {  
    this.model = model;  
}  
  
void setModel(String model) {  
    this.model = model;  
}
```

# 정적 멤버와 static

- 정적 (static) 멤버

- 클래스에 고정된 멤버로서 객체를 생성하지 않고 사용할 수 있는 필드와 메소드

- 정적 멤버 선언

- 필드 또는 메소드를 선언할 때 static 키워드를 붙임

```
public class 클래스 {  
    //정적 필드  
    static 타입 필드 [= 초기값];  
  
    //정적 메소드  
    static 리턴 타입 메소드( 매개변수선언, ... ) { ... }  
}
```

```
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);
```



# 인스턴스 멤버 vs 정적 멤버

## ■ 필드

- 객체마다 가지고 있어야 할 데이터 : 인스턴스 필드
- 공용 데이터 : 정적 필드

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

## ■ 메소드

- 인스턴스 필드로 작업해야 할 메소드 : 인스턴스 메소드
- 인스턴스 필드로 작업하지 않는 메소드 : 정적 메소드

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

# 정적 메소드 선언 시 주의할 점

- 정적 메소드 선언 시
  - 내부에 인스턴스 필드 및 메소드 사용 불가
  - 자신 참조인 this 키워드 사용 불가
- 객체가 없어도 실행 가능
- main() 메소드는 static 메소드이므로 동일 규칙 적용  
public static void main(String args[]);

- 정적 메소드에서 인스턴스 멤버 사용하려는 경우  
객체 우선 생성 후 참조 변수로 접근

```
static void Method3() {  
    ClassName obj = new ClassName();  
    obj.field1 = 10;  
    obj.method1();  
}
```

```
public class ClassName {  
    //인스턴스 필드와 메소드  
    int field1;  
    void method1() { ... }  
    //정적 필드와 메소드  
    static int field2;  
    static void method2() { ... }  
  
    //정적 메소드  
    static void Method3 {  
        this.field1 = 10; // (x)  
        this.method1();  // (x)  
        field2 = 10;      // (o)  
        method2();        // (o)  
    }  
}
```

← 컴파일 에러

# 싱글톤 (singleton)

- 전체 프로그램에서 단 하나의 객체만 만들도록 보장하는 코딩 기법
- 싱글톤 작성 방법
  - 외부에서 new 연산자로 생성자를 호출할 수 없도록 private 접근 제한자를 생성자 앞에 붙임
  - 자신의 타입으로 정적 필드 선언 후, 자신의 객체를 생성해 초기화, 외부에서 필드 값 변경 할 수 없도록 private 접근 제한자 붙임.
  - 외부에서 호출 가능한 정적 메소드 getInstance() 선언, 정적 필드에서 참조하고 있는 자신의 객체 리턴

```
public class 클래스 {  
    //정적 필드  
    private static 클래스 singleton = new 클래스();  
  
    //생성자  
    private 클래스() {}  
  
    //정적 메소드  
    static 클래스 getInstance() {  
        return singleton;  
    }  
}
```

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

```
/*  
Singleton obj1 = new Singleton(); //컴파일 에러  
Singleton obj2 = new Singleton(); //컴파일 에러  
*/  
  
Singleton obj1 = Singleton.getInstance();  
Singleton obj2 = Singleton.getInstance();  
  
if(obj1 == obj2) {  
    System.out.println("같은 Singleton 객체 입니다.");  
} else {  
    System.out.println("다른 Singleton 객체 입니다.");  
}
```

# final 필드와 상수(static final)

- final 필드
  - 최종적인 값을 갖고 있는 필드 = 값을 변경할 수 없는 필드
- final 필드의 초기값 지정 방법
  - 필드 선언 시 초기화
  - 객체 생성 시 생성자에서 초기화

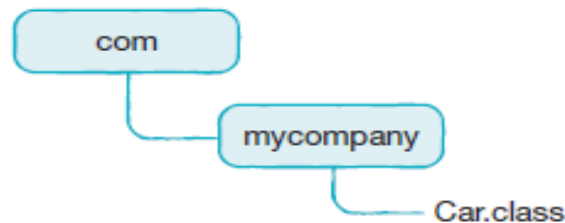
```
public class Person {  
    final String nation = "Korea";  
    final String ssn;  
    String name;  
  
    public Person(String ssn, String name) {  
        this.ssn = ssn;  
        this.name = name;  
    }  
}
```

```
static final double PI = 3.14159;  
static final double EARTH_RADIUS = 6400;  
static final double EARTH_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
```

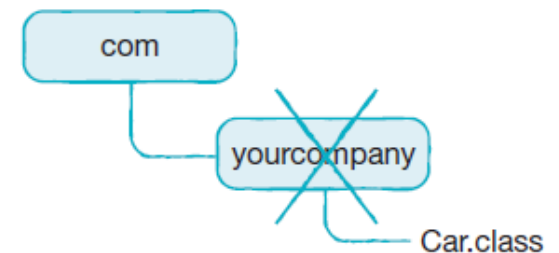
# 패키지(package)

- 클래스 작성 시 해당 클래스가 어떤 패키지에 속할 것인지를 선언
  - 클래스 파일은(~.class) 선언된 패키지와 동일한 폴더 안에서만 동작,
  - 다른 폴더 안에 넣으면 동작하지 않음
- 패키지 이름 규칙
  - java로 시작하는 패키지는 자바 표준 API 에서만 사용하므로 사용 불가
  - 숫자로 시작 불가
  - \_ 및 \$ 제외한 특수문자 사용 불가
  - 모두 소문자로 작성하는 것이 관례

```
package 상위패키지.하위패키지;  
  
public class ClassName { ... }
```



```
package com.mycompany;  
  
public class Car { ... }
```



# import

- 사용하고자 하는 클래스 또는 인터페이스가 다른 패키지에 소속된 경우
- 해당 패키지 클래스 또는 인터페이스를 가져와 사용할 것임을 컴파일러에 통지

```
import 상위패키지.하위패키지.클래스이름;  
import 상위패키지.하위패키지.*;
```

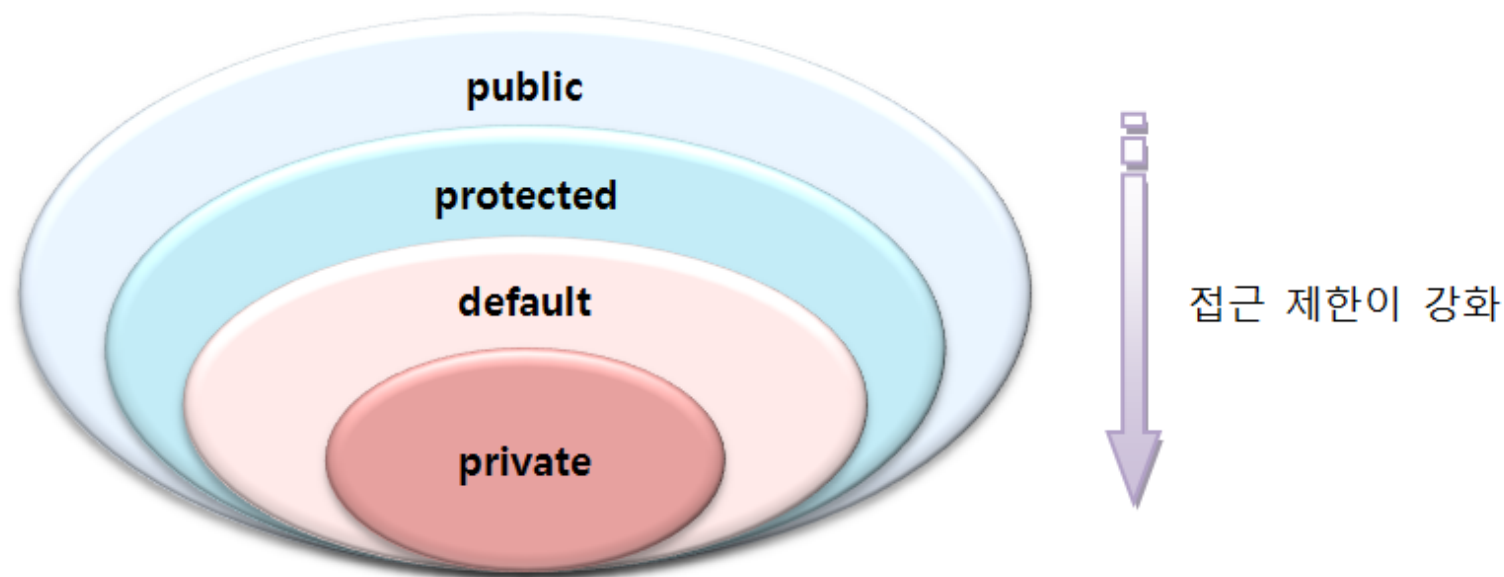
- 패키지 선언과 클래스 선언 사이에 작성
- 하위 패키지는 별도로 import 해야함

```
import com.hankook.*;  
import com.hankook.project.*;
```

```
package com.mycompany;  
  
import com.hankook.Tire;  
[ 또는 import com.hankook.*; ]  
  
public class Car {  
    Tire tire = new Tire();  
}
```

- 다른 패키지에 동일한 이름의 클래스가 있을 경우
  - import와 상관없이 클래스 전체 이름을 기술

# 접근 제한자(Access Modifier)

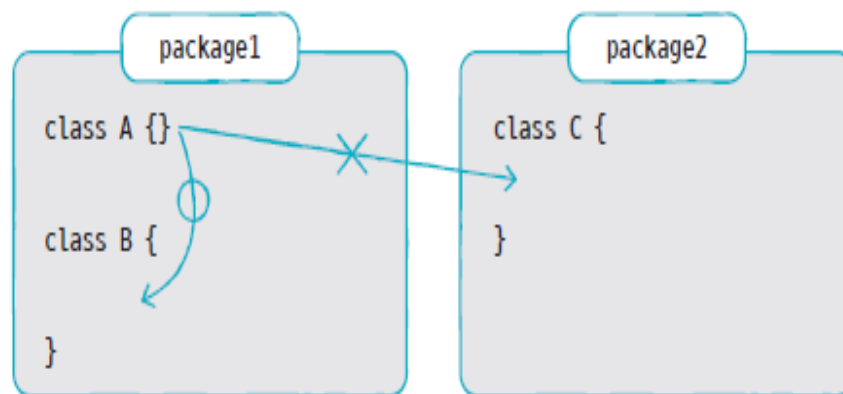


접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스

# 클래스 접근 제한

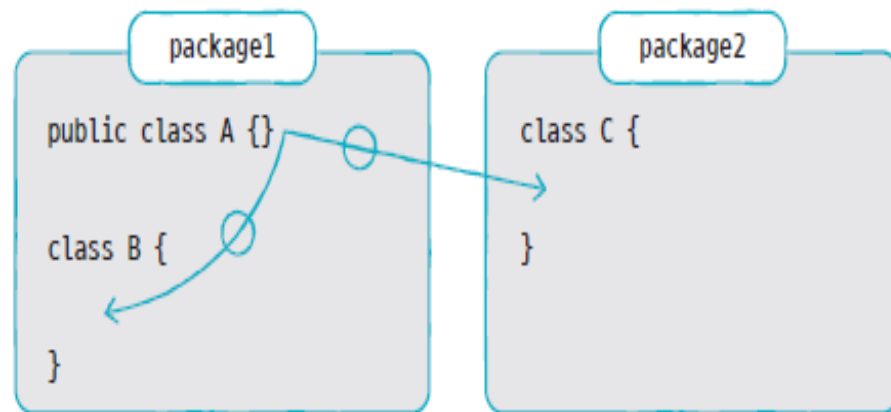
## ■ default

- 클래스 선언할 때 public 생략한 경우
- 다른 패키지에서는 사용 불가



## ■ public

- 다른 개발자가 사용할 수 있도록 라이브러리 클래스로 만들 때 유용





# 개념 확인 학습 & 적용 확인 학습 & 응용 프로그래밍

---

- 첨부된 실습 자료를 참조하시기 바랍니다.
  - java\_06장\_클래스\_ex.pdf

# Q & A

---

- “클래스”에 대한 학습이 모두 끝났습니다.
- 모든 내용을 이해 하셨나요?
- 아직 이해가 안되는 내용이 있다면 다시 한번 복습하시기 바랍니다.
- 질문은 한림 SmartLEAD 쪽지 또는 e-mail 또는 전화상담을 이용하시기 바랍니다.



- 과제(기간 내 제출)와 퀴즈(기간내 시도 횟수 1회)가 제출되었습니다.
- 다음 시간에는 “상속”을 공부하도록 하겠습니다.
- 수고하셨습니다.^^