

6.1 객체 지향 프로그래밍

6.2 객체와 클래스

6.3 클래스 선언

6.4 객체 생성과 클래스 변수

6.5 클래스의 구성 멤버

6.6 필드 선언과 사용

6.7 생성자 선언과 호출

6.8 메소드 선언과 호출

6.9 인스턴스 멤버

6.10 정적 멤버

6.11 final 필드와 상수

6.12 패키지

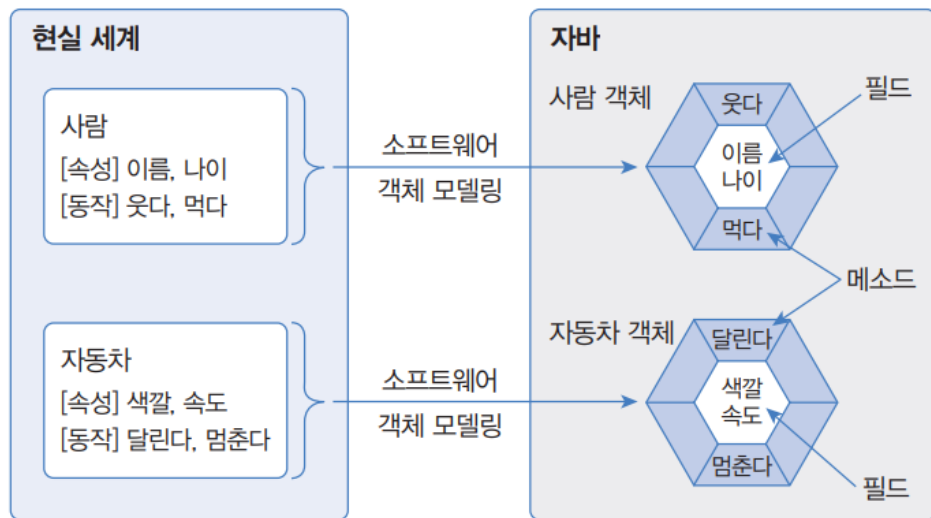
6.13 접근 제한자

6.14 Getter와 Setter

6.15 싱글톤 패턴

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

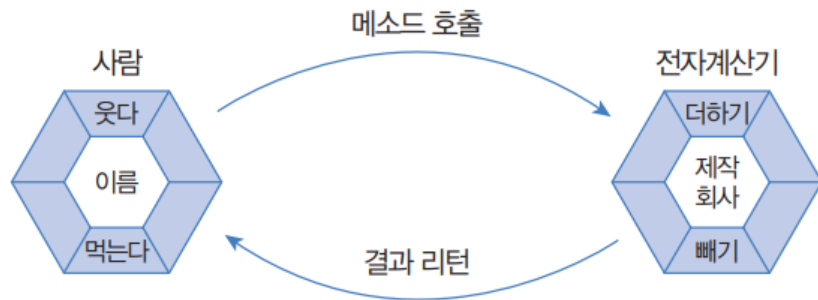


객체 지향 프로그래밍(OOP)

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

객체의 상호작용

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

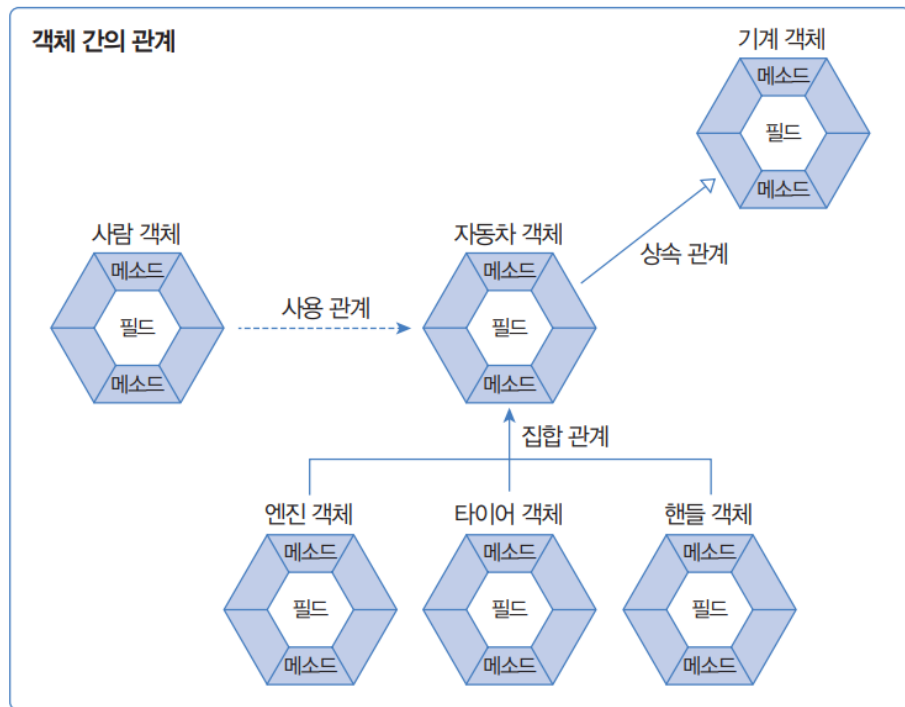


- 매개값: 객체가 전달하고자 하는 데이터이며, 메소드 이름과 함께 괄호() 안에 기술
- 리턴값: 메소드의 실행의 결과이며, 호출한 곳으로 돌려주는 값

```
메소드(매개값1, 매개값2, ...);
```

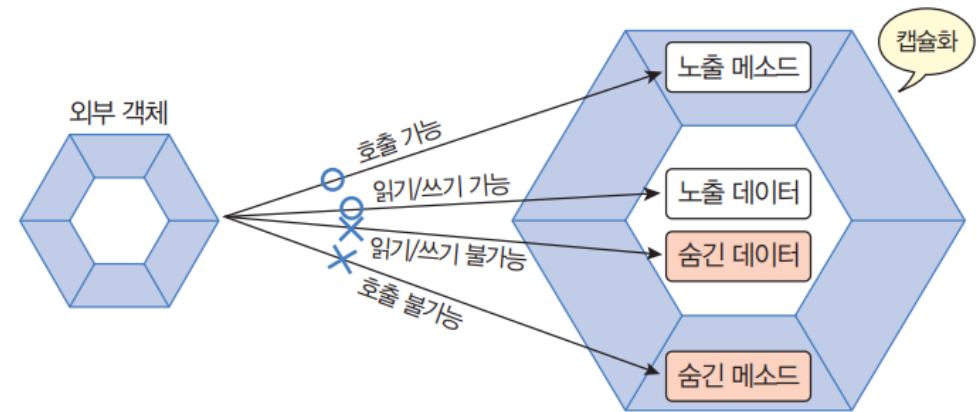
객체 간의 관계

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

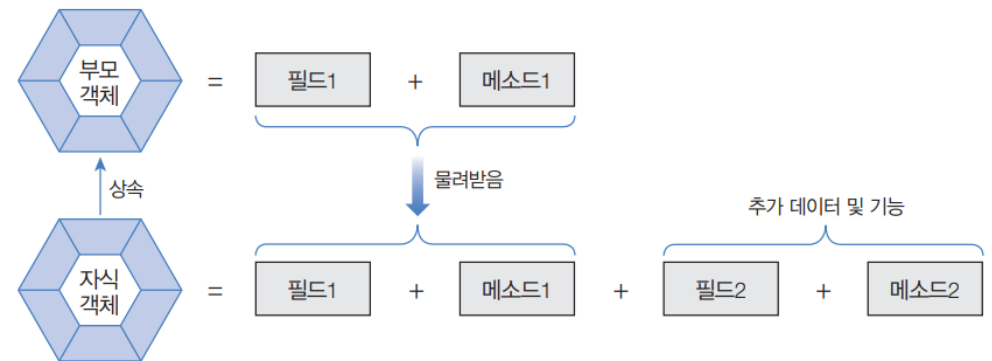


객체 지향 프로그래밍의 특징

- 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



- 다형성: 사용 방법은 동일하지만 실행 결과가 다양함

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



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]

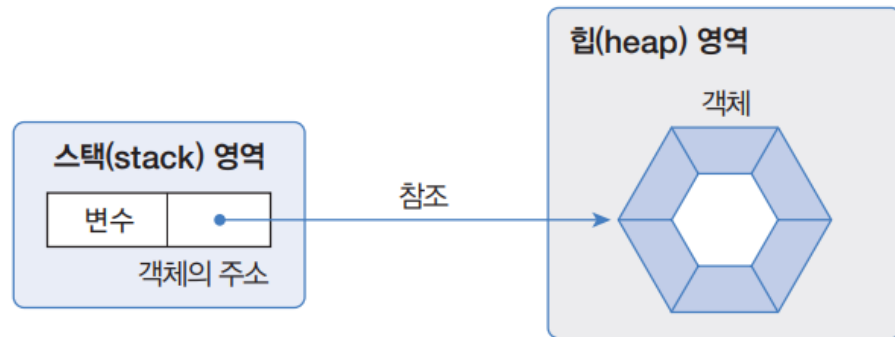
```
//클래스 선언  
public class 클래스명 {  
}
```

- 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 클래스: 어느 위치에 있든지 패키지와 상관없이 사용할 수 있는 클래스

클래스 변수

- 클래스로부터 객체를 생성하려면 객체 생성 연산자인 new가 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

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() { ... }  
}
```

필드 선언

- 필드는 클래스 블록에서 선언되어야 함

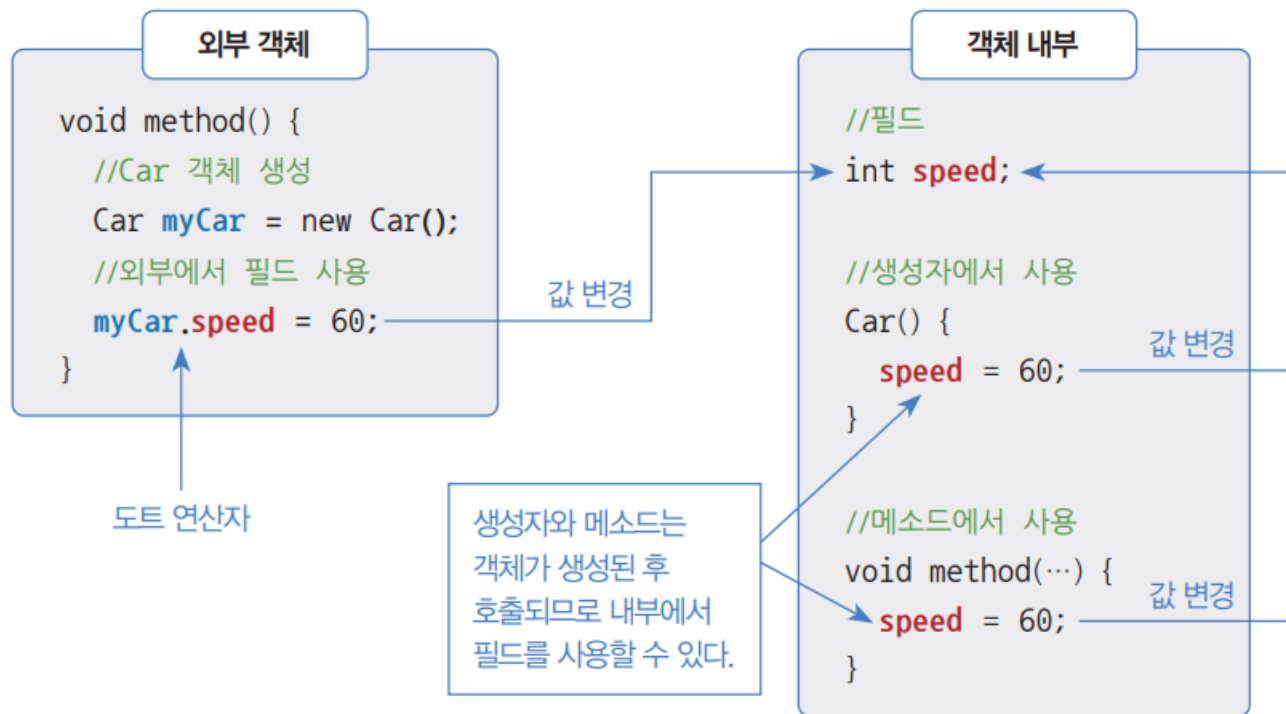
```
타입 필드명 [ = 초기값 ] ;
```

- 타입은 필드에 저장할 데이터의 종류를 결정. 기본 타입, 참조 타입 모두 가능
- 초기값을 제공하지 않을 경우 필드는 객체 생성 시 자동으로 기본값으로 초기화

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

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



기본 생성자

- 클래스에 생성자 선언이 없으면 컴파일러는 기본 생성자를 바이트코드 파일에 자동으로 추가

```
[public] 클래스() { }
```

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

필드 초기화

- 객체마다 동일한 값을 갖고 있다면 필드 선언 시 초기값을 대입하는 것이 좋고, 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; ← 매개값으로 받은 이름과 주민등록번호를  
        ssn = s; ← 필드 초기값으로 사용  
    }  
}
```

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, 선언된 순서가 똑같은 경우 매개변수 이름만 바꾸는 것은 생성자 오버로딩이 아님
- 생성자가 오버로딩되어 있을 경우, new 연산자로 생성자를 호출할 때 제공되는 매개값의 타입과 수에 따라 실행될 생성자가 결정

다른 생성자 호출

- 생성자 오버로딩이 많아질 경우 생성자 간의 중복된 코드가 발생할 수 있음
- 이 경우 공통 코드를 한 생성자에만 집중적으로 작성하고, 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;  
}
```

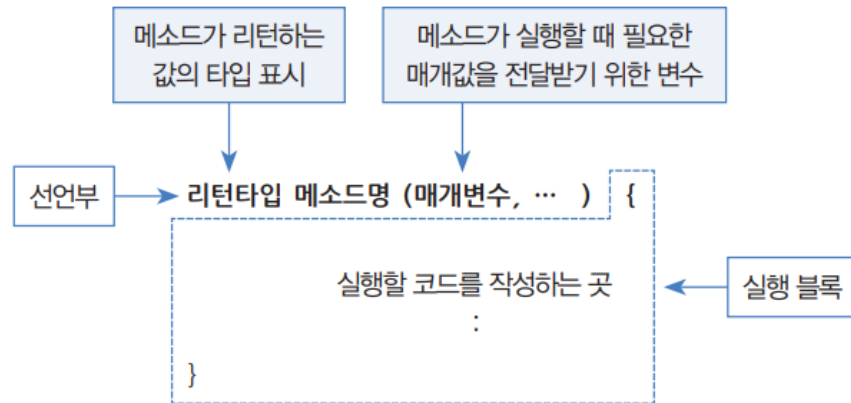
호출

호출

공통 초기화 코드

메소드 선언

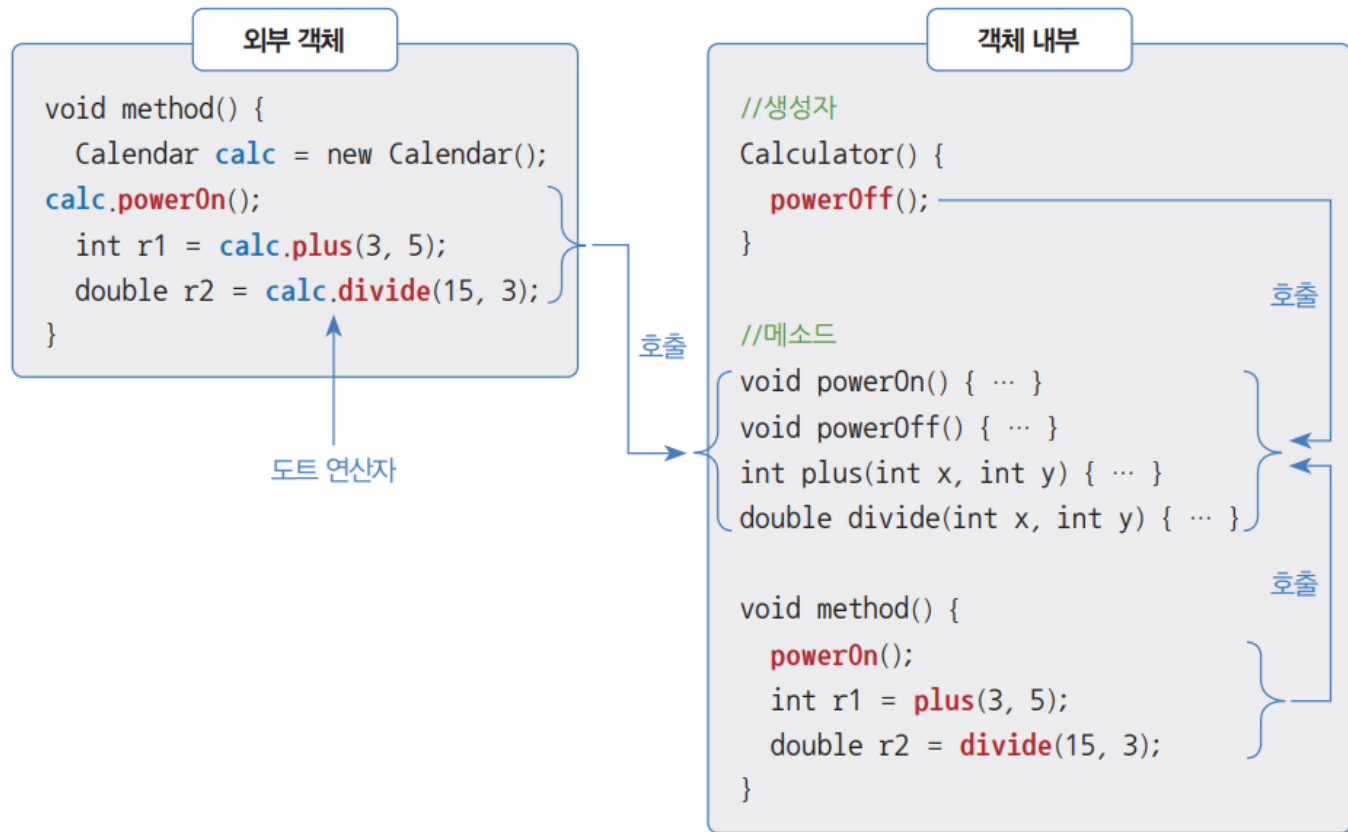
- 객체의 동작을 실행 블록으로 정의하는 것.



- 리턴 타입: 메소드 실행 후 호출한 곳으로 전달하는 결과값의 타입
- 메소드명: 메소드명은 첫 문자를 소문자로 시작하고, 캐멀 스타일로 작성
- 매개변수: 메소드를 호출할 때 전달한 매개값을 받기 위해 사용
- 실행 블록: 메소드 호출 시 실행되는 부분

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



가변길이 매개변수

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

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

- 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 });
```

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

메소드 오버로딩

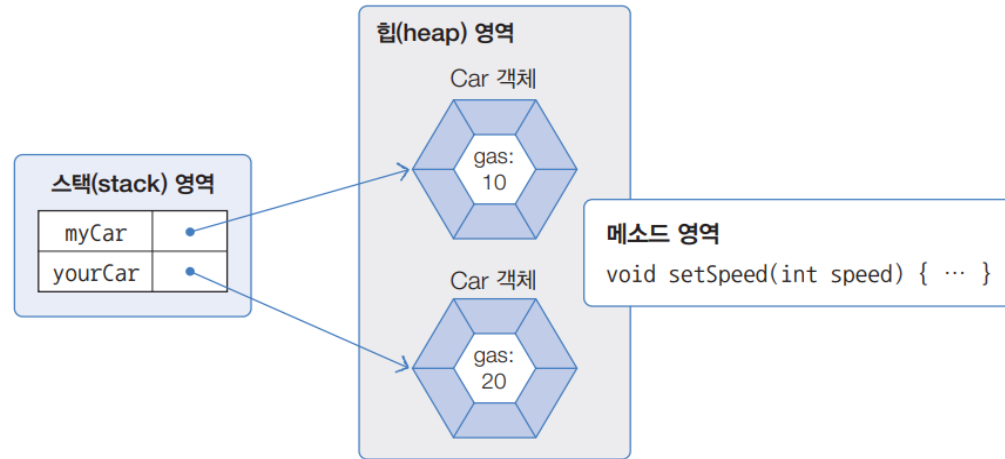
- 메소드 이름은 같되 매개변수의 타입, 개수, 순서가 다른 메소드를 여러 개 선언하는 것

```
class 클래스 {  
  리턴타입 메소드이름 ( 타입 변수, ... ) { ... }  
  리턴타입 메소드이름 ( 타입 변수, ... ) { ... }  
}
```

무관 동일 타입, 개수, 순서가 달라야 함

인스턴스 멤버 선언 및 사용

- 인스턴스 멤버: 필드와 메소드 등 객체에 소속된 멤버

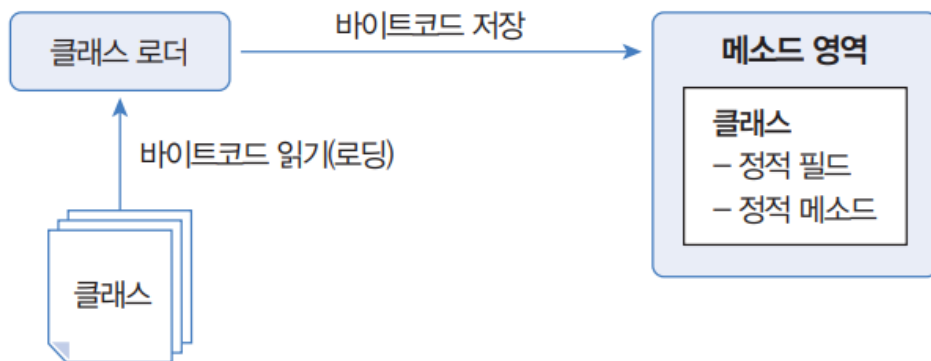


this 키워드

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

정적 멤버 선언

- 정적 멤버: 메소드 영역의 클래스에 고정적으로 위치하는 멤버



- static 키워드를 추가해 정적 필드와 정적 메소드로 선언

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

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

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

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();  
}
```


final 필드 선언

- final 필드는 초기값이 저장되면 최종적인 값이 되어서 프로그램 실행 도중에 수정할 수 없게 됨
- final 필드에 초기값을 주려면 필드 선언 시에 초기값을 대입하거나 생성자에서 초기값을 대입

```
final 타입 필드 [=초기값];
```

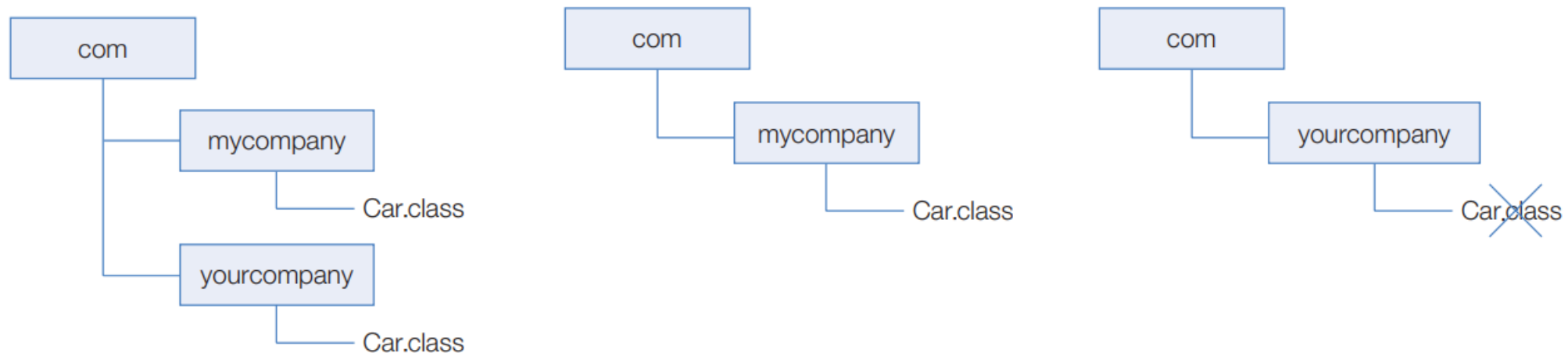
상수 선언

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

```
static final 타입 상수 [= 초기값];
```

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



자바프로그래밍

06 클래스

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
```

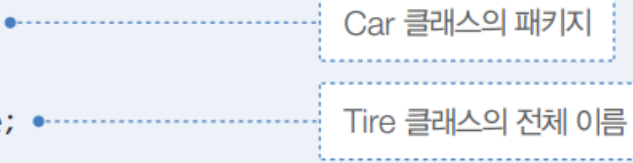
import문

- 다른 패키지에 있는 클래스를 사용하려면 import 문을 이용해서 어떤 패키지의 클래스를 사용하는지 명시

```
package com.mycompany;

import com.hankook.Tire;

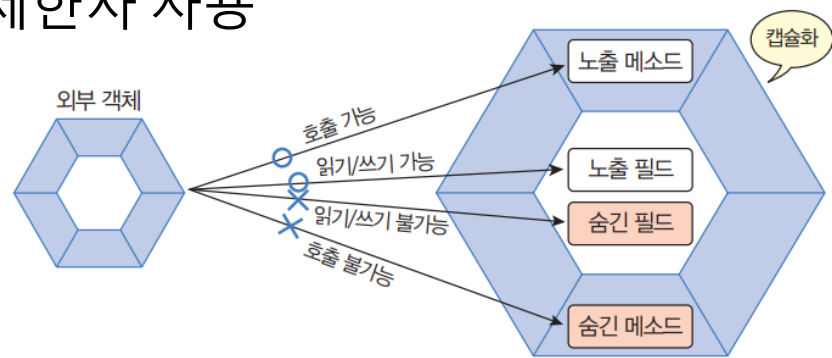
public class Car {
    //필드 선언 시 com.hankook.Tire 클래스를 사용
    Tire tire = new Tire();
}
```



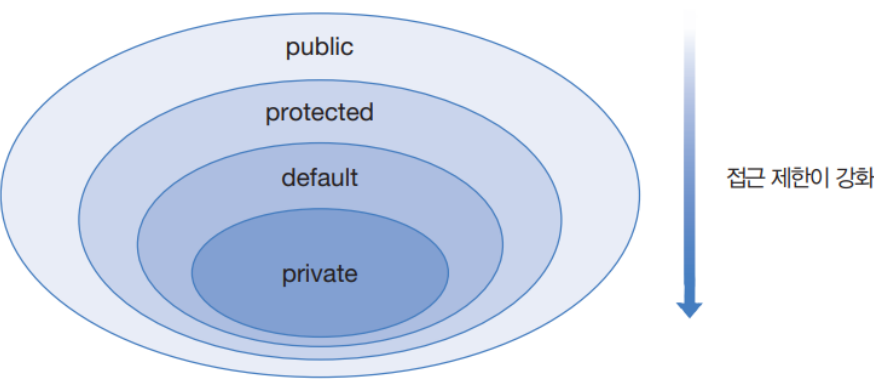
- import 문은 패키지 선언과 클래스 선언 사이에 작성. import 키워드 뒤에는 사용하고자 하는 클래스의 전체 이름을 기술

접근 제한자

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



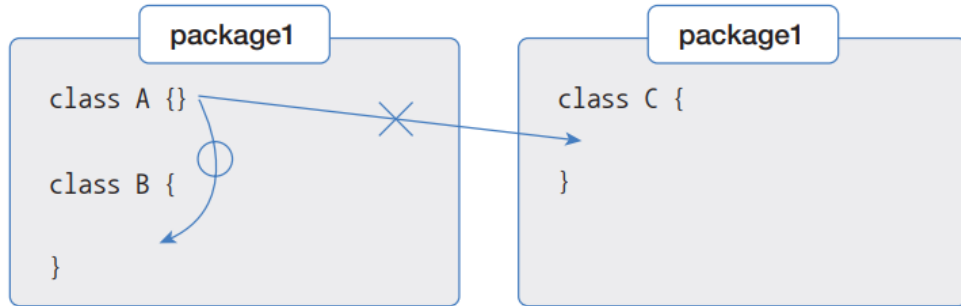
- 접근 제한자는 public, protected, private의 세 가지 종류



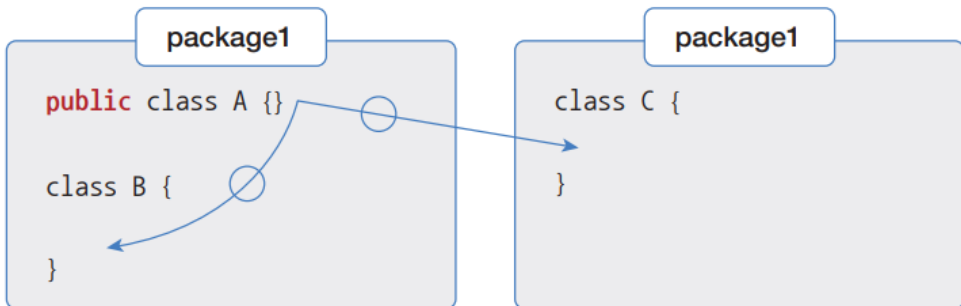
접근 제한자	제한 대상	제한 범위
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	같은 패키지이거나, 자식 객체만 사용 가능 (7장 상속에서 자세히 설명)
(default)	클래스, 필드, 생성자, 메소드	같은 패키지
private	필드, 생성자, 메소드	객체 내부

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



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.

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	모든 패키지에서 필드를 읽고 변경할 수 있다.
	메소드(...)	모든 패키지에서 메소드를 호출할 수 있다.
	필드	같은 패키지에서만 필드를 읽고 변경할 수 있다.
	메소드(...)	같은 패키지에서만 메소드를 호출할 수 있다.
private	필드	클래스 내부에서만 필드를 읽고 변경할 수 있다.
	메소드(...)	클래스 내부에서만 메소드를 호출할 수 있다.

Setter

- 데이터를 검증해서 유효한 값만 필드에 저장하는 메소드

Getter

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

```
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

싱글톤 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;  
    }  
}
```