6장. 클래스

이것이 자바다(http://cafe.naver.com/thisjava)

한빛미디어 Hanbit Media, Inc.

Contents

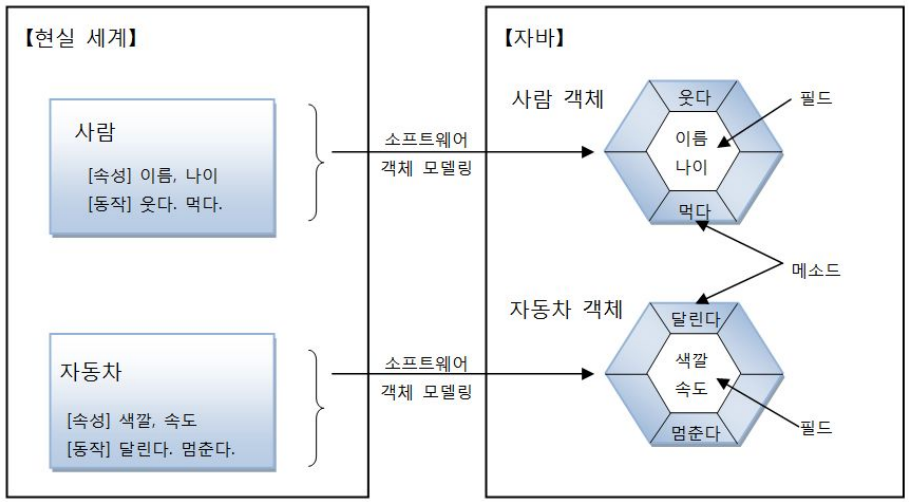
1절. 객체 지향 프로그래밍

❖ 객체 지향 프로그래밍

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

❖ 객체(Object)란?

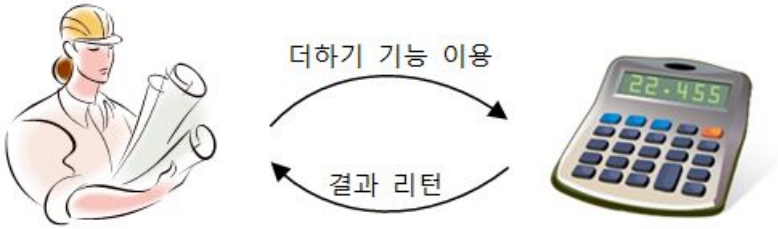
- 물리적으로 존재하는 것 (자동차, 책, 사람)
- 추상적인 것(회사, 날짜) 중에서 자신의 속성과 동작을 가지는 모든 것
- 객체는 필드(속성) 과 메소드(동작)로 구성된 자바 객체로 모델링 가능

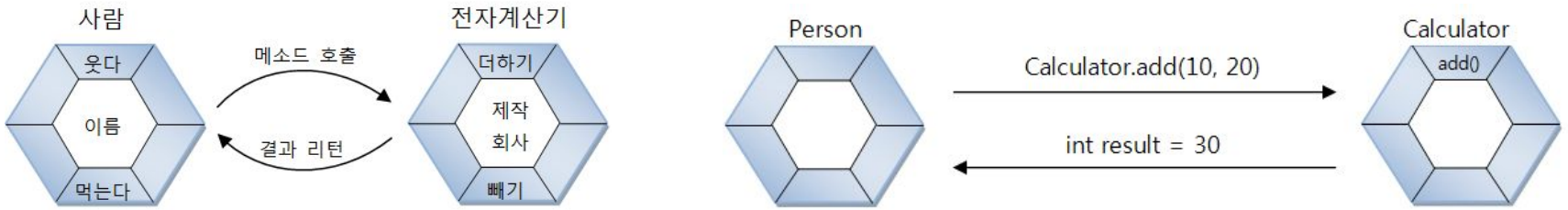


1절. 객체 지향 프로그래밍

❖ 객체의 상호 작용

- 객체들은 서로 간에 기능(동작)을 이용하고 데이터를 주고 받음

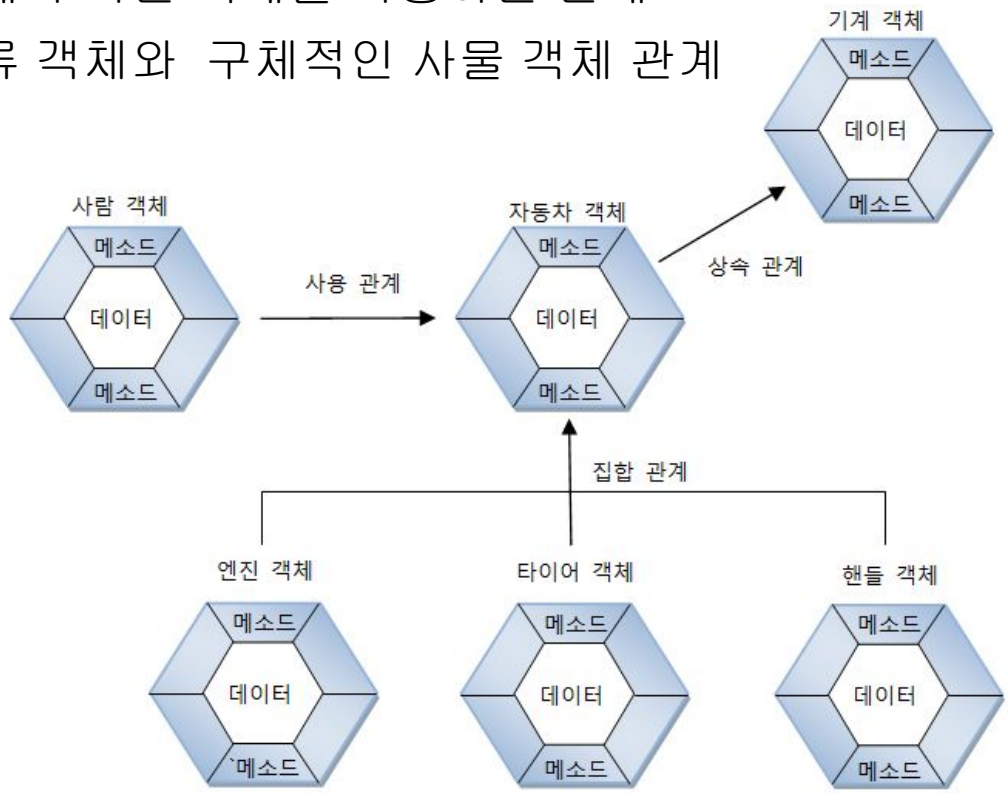




1절. 객체 지향 프로그래밍

❖ 객체간의 관계

- 객체 지향 프로그램에서는 객체는 다른 객체와 관계를 맺음
- 관계의 종류
  - 집합 관계: 완성품과 부품의 관계
  - 사용 관계: 객체가 다른 객체를 사용하는 관계
  - 상속 관계: 종류 객체와 구체적인 사물 객체 관계

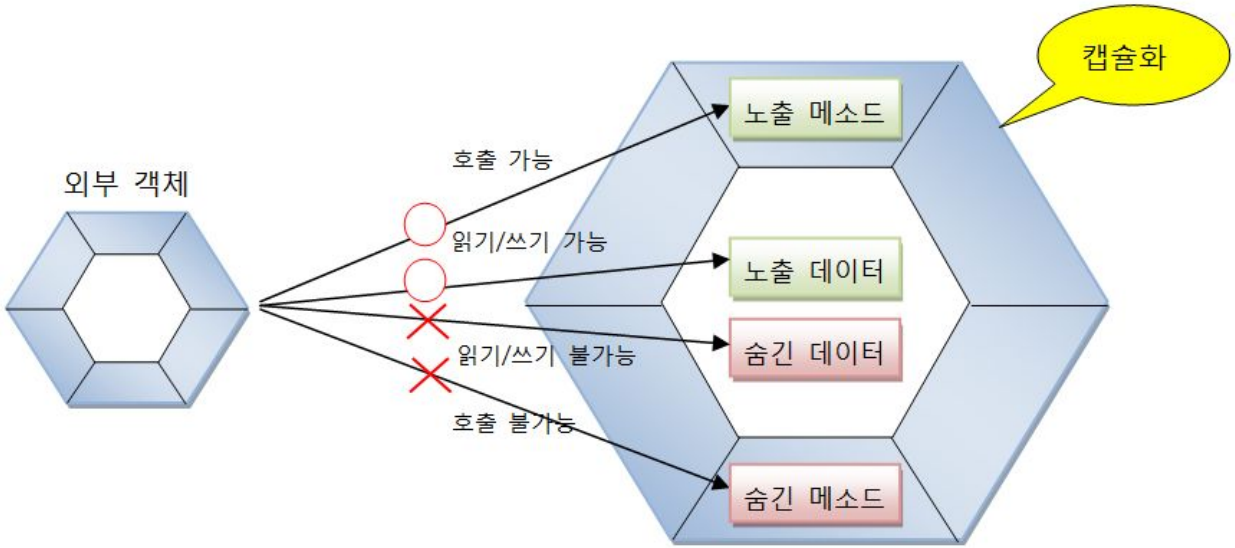


1절. 객체 지향 프로그래밍

❖ 객체 지향 프로그래밍의 특징

▪ 캡슐화

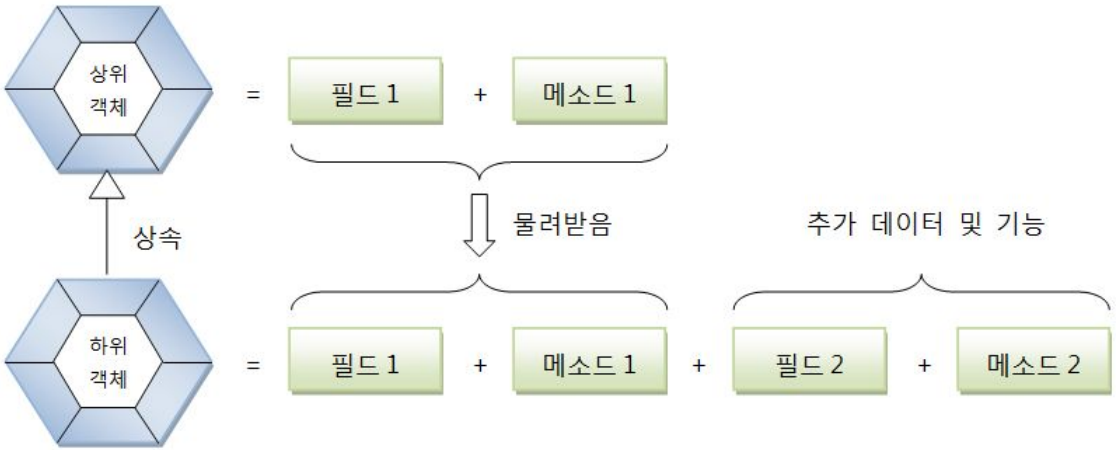
- 객체의 필드, 메소드를 하나로 묶고, 실제 구현 내용을 감추는 것
- 외부 객체는 객체 내부 구조를 알지 못하며 객체가 노출해 제공하는 필드와 메소드만 이용 가능
- 필드와 메소드를 캡슐화하여 보호하는 이유는 외부의 잘못된 사용으로 인해 객체가 손상되지 않도록
- 자바 언어는 캡슐화된 멤버를 노출시킬 것인지 숨길 것인지 결정하기 위해 접근 제한자(Access Modifier) 사용



1절. 객체 지향 프로그래밍

❖ 객체 지향 프로그래밍의 특징

- 상속
  - 상위(부모) 객체의 필드와 메소드를 하위(자식) 객체에게 물려주는 행위
  - 하위 객체는 상위 객체를 확장해서 추가적인 필드와 메소드를 가질 수 있음
  - 상속 대상: 필드와 메소드
  - 상속의 효과
    - 상위 객체를 재사용해서 하위 객체를 빨리 개발 가능
    - 반복된 코드의 중복을 줄임
    - 유지 보수의 편리성 제공
    - 객체의 다형성 구현



1절. 객체 지향 프로그래밍

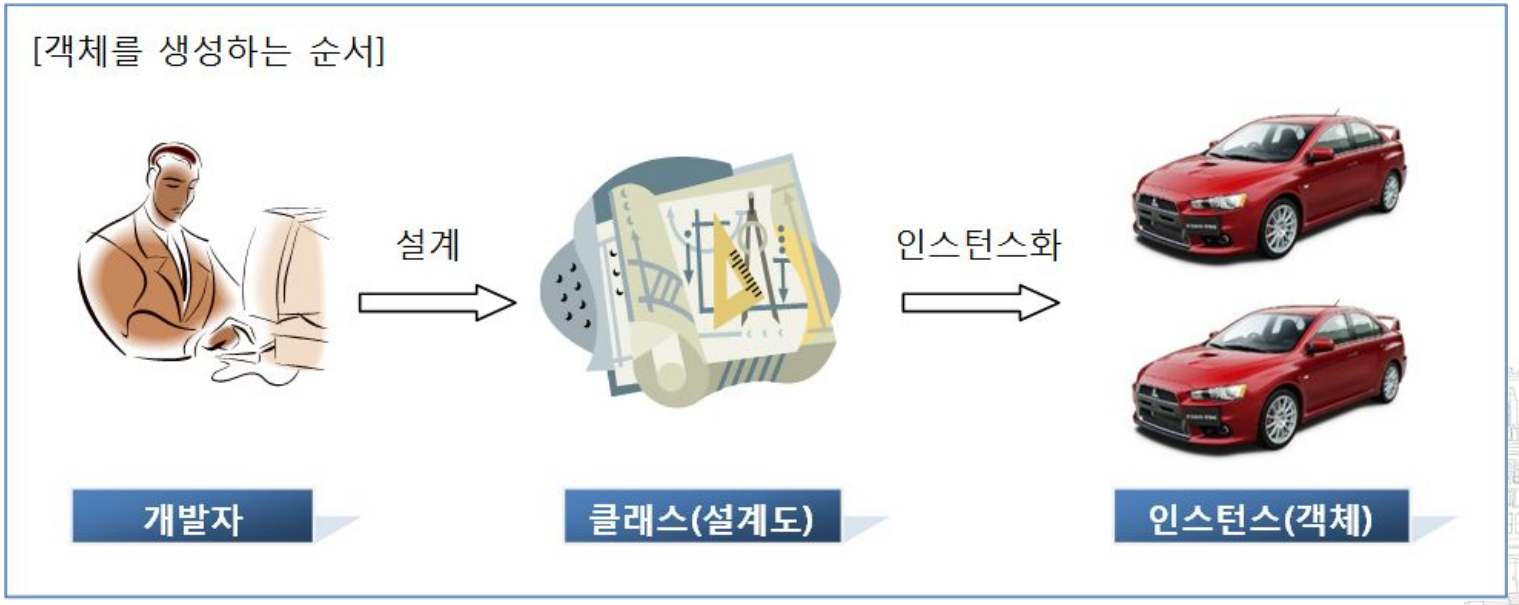
❖ 객체 지향 프로그래밍의 특징

- 다형성 (Polymorphism)
  - 같은 타입이지만 실행 결과가 다양한 객체를 대입할 수 있는 성질
    - 부모 타입에는 모든 자식 객체가 대입
    - 인터페이스 타입에는 모든 구현 객체가 대입
  - 효과
    - 객체를 부품화시키는 것 가능
    - 유지보수 용이

2절. 객체와 클래스

❖ 객체(Object)와 클래스(Class)

- 현실세계: 설계도 □ 객체
- 자바: 클래스 □ 객체
- 클래스에는 객체를 생성하기 위한 필드와 메소드가 정의
- 클래스로부터 만들어진 객체를 해당 클래스의 인스턴스(instance)
- 하나의 클래스로부터 여러 개의 인스턴스를 만들 수 있음



3절. 클래스 선언

❖ 클래스의 이름

- 자바 식별자 작성 규칙에 따라야

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)

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

```
Calculator, Car, Member, ChatClient, ChatServer, Web_Browser
```

3절. 클래스 선언

❖ 클래스 선언과 컴파일

- 소스 파일 생성: 클래스이름.java (대소문자 주의)
- 소스 작성

```
public class 클래스이름 {


}
```

클래스이름.class

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

Car.java

```
public class Car {

}

class Tire {

}
```

Car.class

Tire.class

4절. 객체 생성과 클래스 변수

❖ new 연산자

- 객체 생성 역할

```
new 클래스();
```

  - 클래스()는 생성자를 호출하는 코드
  - 생성된 객체는 힙 메모리 영역에 생성

- new 연산자는 객체를 생성 후, 객체 생성 번지 리턴

4절. 객체 생성과 클래스 변수

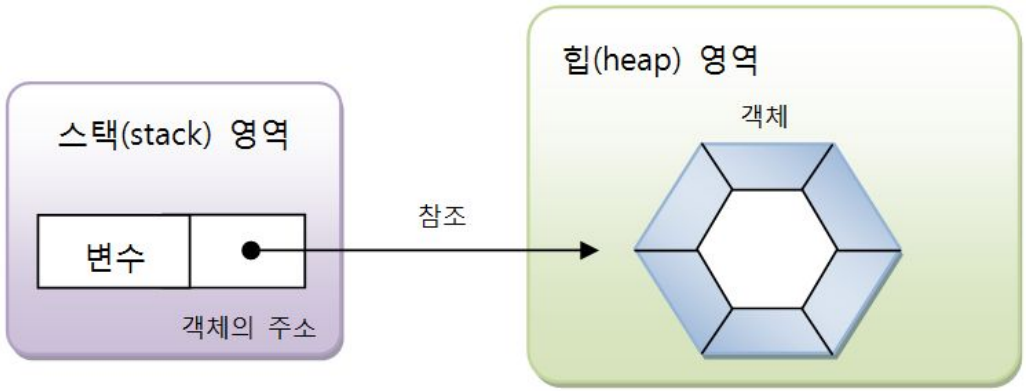
❖ 클래스 변수

- new 연산자에 의해 리턴 된 객체의 번지 저장 (참조 타입 변수)

- 힙 영역의 객체를 사용하기 위해 사용

```
클래스 변수;
변수 = new 클래스();

클래스 변수 = new 클래스();
```



4절. 객체 생성과 클래스 변수

❖ 클래스의 용도

- 라이브러리(API: Application Program Interface) 용
  - 자체적으로 실행되지 않음
  - 다른 클래스에서 이용할 목적으로 만든 클래스

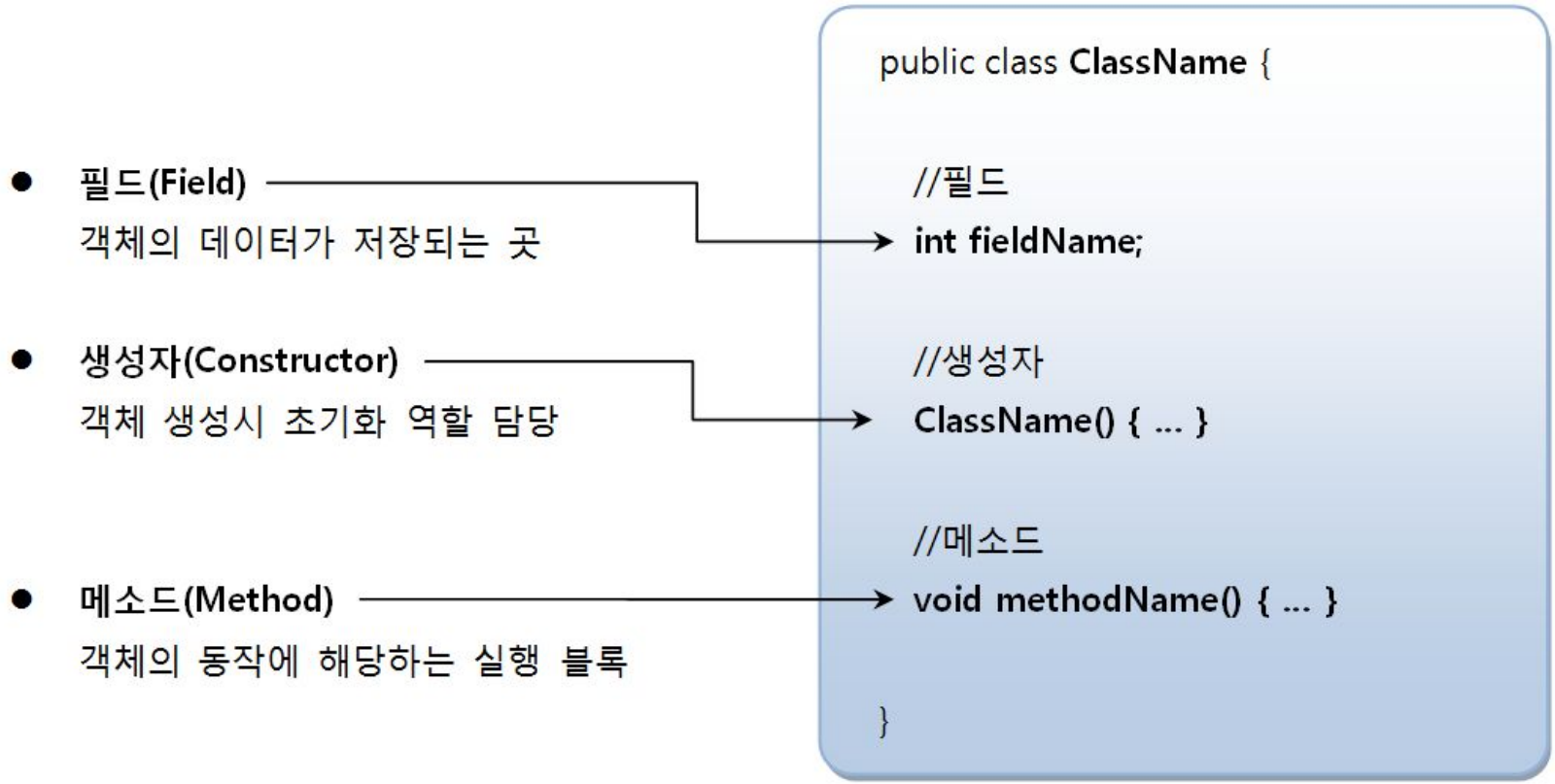
- 실행용
  - main() 메소드를 가지고 있는 클래스로 실행할 목적으로 만든 클래스

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

5절. 클래스의 구성 멤버

❖ 클래스의 구성 멤버

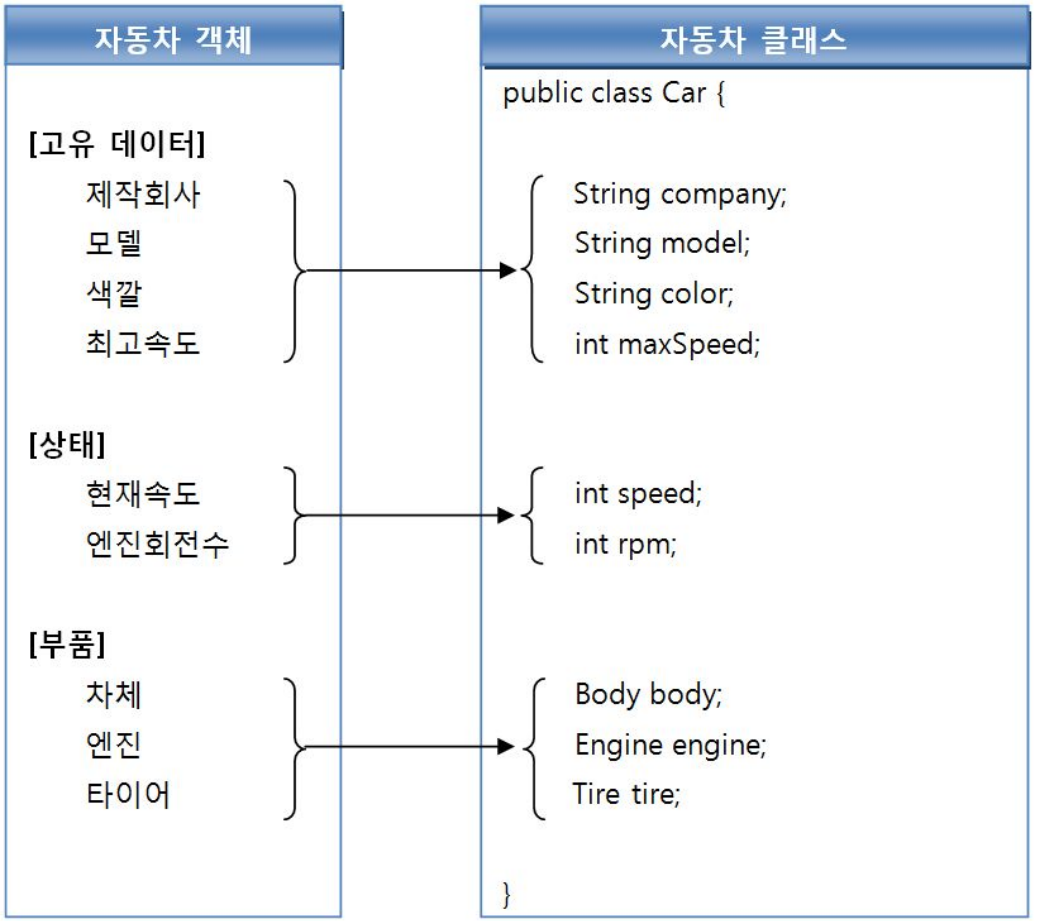
- 필드(Field)
- 생성자(Constructor)
- 메소드(Method)



6절. 필드(field)

❖ 필드의 내용

- 객체의 고유 데이터
- 객체가 가져야 할 부품 객체
- 객체의 현재 상태 데이터



6절. 필드(field)

❖ 필드 선언

```
타입 필드 [ = 초기값] ;
```

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

6절. 필드(field)

❖ 필드의 기본 초기값

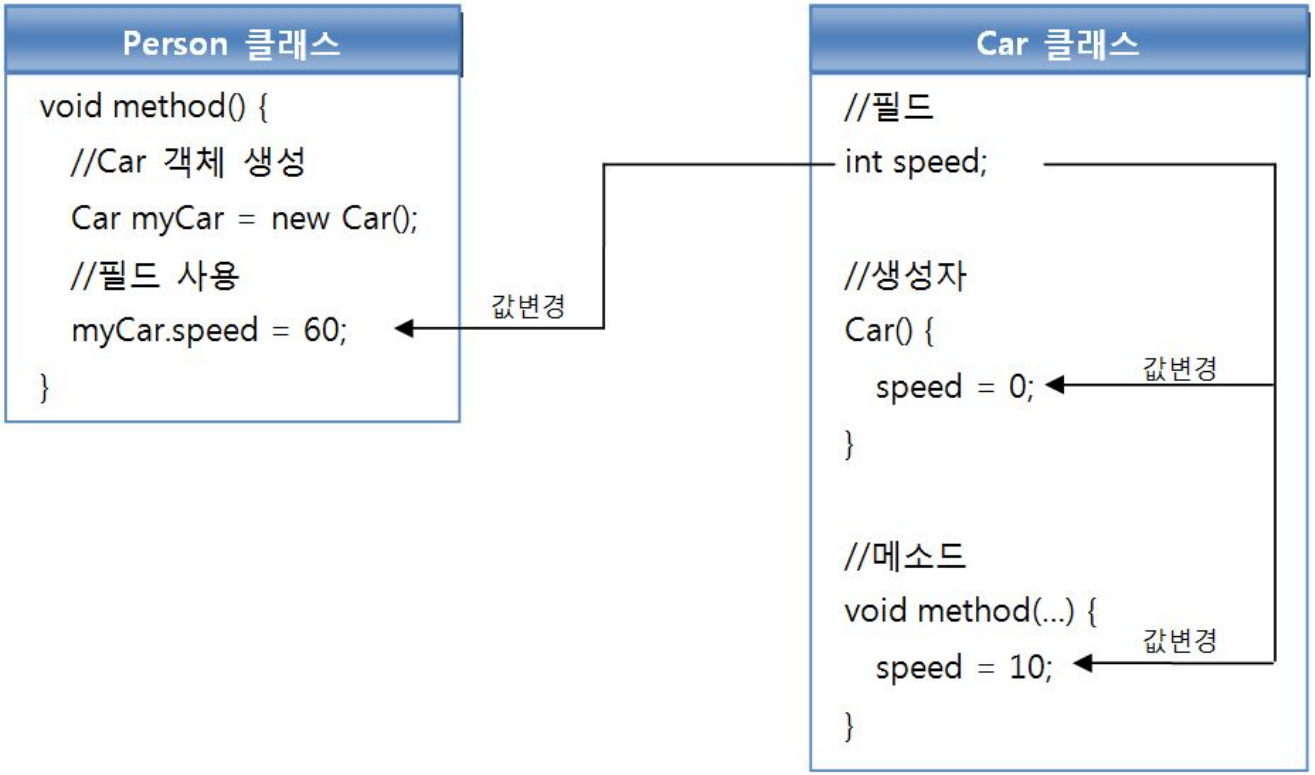
- 초기값 지정되지 않은 필드
  - 객체 생성시 자동으로 기본값으로 초기화

분류		데이터 타입	초기값
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

6절. 필드(field)

❖ 필드 사용

- 필드 값을 읽고, 변경하는 작업을 말한다.
- 필드 사용 위치
  - 객체 내부: "필드이름" 으로 바로 접근
  - 객체 외부: "변수.필드이름"으로 접근



7절. 생성자(Constructor)

❖ 생성자

- new 연산자에 의해 호출되어 객체의 초기화 담당

```
new 클래스();
```

  - 필드의 값 설정
  - 메소드 호출해 객체를 사용할 수 있도록 준비하는 역할 수행

❖ 기본 생성자(Default Constructor)

- 모든 클래스는 생성자가 반드시 존재하며 하나 이상 가질 수 있음
- 생성자 선언을 생략하면 컴파일러는 다음과 같은 기본 생성자 추가

```
[public] 클래스() { }
```

소스 파일(Car.java)

```
public class Car {

}
```

→

바이트 코드 파일(Car.class)

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

기본 생성자

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

기본 생성자

7절. 생성자(Constructor)

❖ 생성자 선언

- 디폴트 생성자 대신 개발자가 직접 선언

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

} 생성자 블록

- 개발자 선언한 생성자 존재 시 컴파일러는 기본 생성자 추가하지 않음
  - new 연산자로 객체 생성시 개발자가 선언한 생성자 반드시 사용

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

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

7절. 생성자(Constructor)

❖ 필드 초기화

- 초기값 없이 선언된 필드는 객체가 생성될 때 기본값으로 자동 설정

- 다른 값으로 필드 초기화하는 방법
  - 필드 선언할 때 초기값 설정

  - 생성자의 매개값으로 초기값 설정

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

  - 매개 변수와 필드명 같은 경우 this 사용 (p.206~208)

7절. 생성자(Constructor)

❖ 생성자 다양화해야 하는 이유

- 객체 생성할 때 외부 값으로 객체를 초기화할 필요
- 외부 값이 어떤 타입으로 몇 개가 제공될 지 모름 - 생성자도 다양화

❖ 생성자 오버로딩(Overloading) (p.208~211)

- 매개변수의 타입, 개수, 순서가 다른 생성자 여러 개 선언

```
public class 클래스 {
    클래스 ( [타입 매개변수, ...] ) {
        ...
    }
    클래스 ( [타입 매개변수, ...] ) {
        ...
    }
}
```

[생성자의 오버로딩]
매개변수의 타입, 개수, 순서가 다르게 선언

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

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

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

7절. 생성자(Constructor)

❖ 다른 생성자 호출(this())

- 생성자 오버로딩되면 생성자 간의 중복된 코드 발생
- 초기화 내용이 비슷한 생성자들에서 이러한 현상을 많이 볼 수 있음
  - 초기화 내용을 한 생성자에 몰아 작성
  - 다른 생성자는 초기화 내용을 작성한 생성자를 this(…)로 호출

```
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

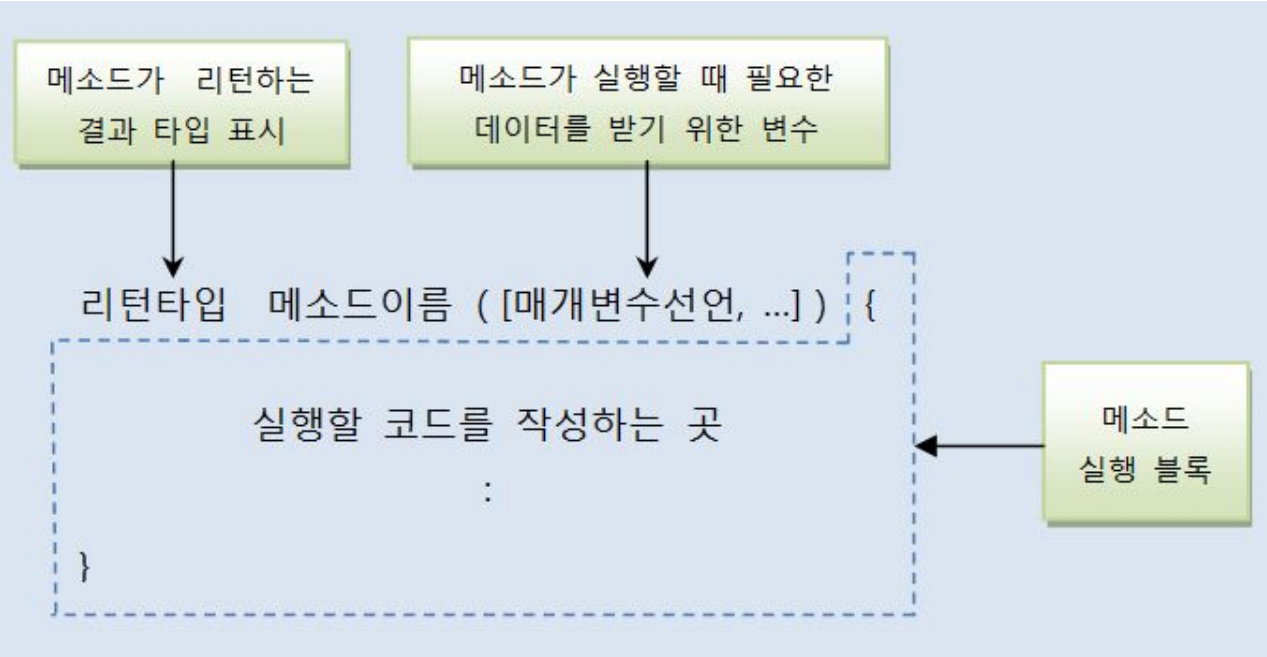
중복 코드

8절. 메소드(method)

❖ 메소드란?

- 객체의 동작(기능)
- 호출해서 실행할 수 있는 중괄호 { } 블록
- 메소드 호출하면 중괄호 { } 블록에 있는 모든 코드들이 일괄 실행

❖ 메소드 선언



8절. 메소드(method)

❖ 메소드 리턴 타입

- 메소드 실행된 후 리턴하는 값의 타입

- 메소드는 리턴값이 있을 수도 있고 없을 수도 있음

[메소드 선언]

```
void powerOn() { ... }

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

[메소드 호출]

```
powerOn();

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

❖ 메소드 이름

- 자바 식별자 규칙에 맞게 작성

8절. 메소드(method)

❖ 메소드 매개변수 선언

- 매개변수는 메소드를 실행할 때 필요한 데이터를 외부에서 받기 위해 사용
- 매개변수도 필요 없을 수 있음

[메소드 선언]

```
void powerOn() { ... }

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

[메소드 호출]

```
powerOn();

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

```
byte b1 = 10;

byte b2 = 20;

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

8절. 메소드(method)

❖ 리턴(return) 문 (p.221~224)

- 메소드 실행을 중지하고 리턴값 지정하는 역할

- 리턴값이 있는 메소드
  - 반드시 리턴(return)문 사용해 리턴값 지정해야

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

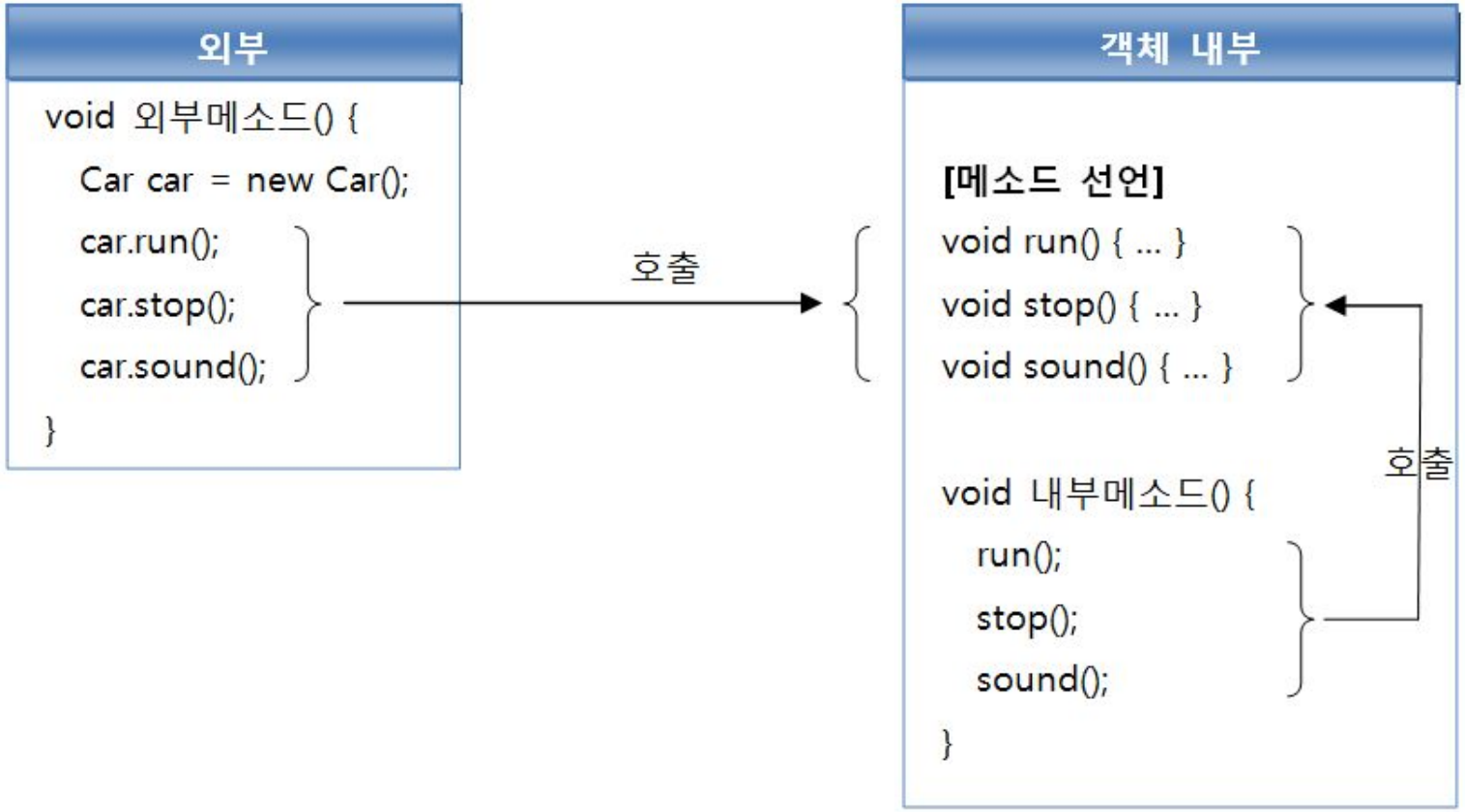
```
boolean isLeftGas() {
    if(gas==0) {
        System.out.println("gas 가 없습니다.");
        return false;
    }
    System.out.println("gas 가 있습니다.");
    return true;
}
```

  - return 문 뒤에 실행문 올 수 없음
- 리턴값이 없는 메소드
  - 메소드 실행을 강제 종료 시키는 역할

8절. 메소드(method)

❖ 메소드 호출

- 메소드는 클래스 내·외부의 호출에 의해 실행
  - 클래스 내부: 메소드 이름으로 호출 (p.225~228)
  - 클래스 외부: 객체 생성 후, 참조 변수를 이용해 호출 (p.228~229)



8절. 메소드(method)

❖ 메소드 오버로딩(Overloading)

- 클래스 내에 같은 이름의 메소드를 여러 개 선언하는 것
- 하나의 메소드 이름으로 다양한 매개값 받기 위해 메소드 오버로딩
- 오버로딩의 조건: 매개변수의 타입, 개수, 순서가 달라야

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

무관 / 동일 / 매개변수의 타입, 개수, 순서가 달라야함

```
void println() { .. }
void println(boolean x) { .. }
void println(char x) { .. }
void println(char[] x) { .. }
void println(double x) { .. }
void println(float x) { .. }
void println(int x) { .. }
void println(long x) { .. }
void println(Object x) { .. }
void println(String x) { .. }
```

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

```
plus(10, 20);
```

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

```
plus(10.5, 20.3);
```

```
int x = 10;
double y = 20.3;
plus(x, y);
```

?

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

9절. 인스턴스 멤버와 this

❖ 인스턴스 멤버란?

- 객체(인스턴스) 마다 가지고 있는 필드와 메소드
  - 이들을 각각 인스턴스 필드, 인스턴스 메소드라고 부름

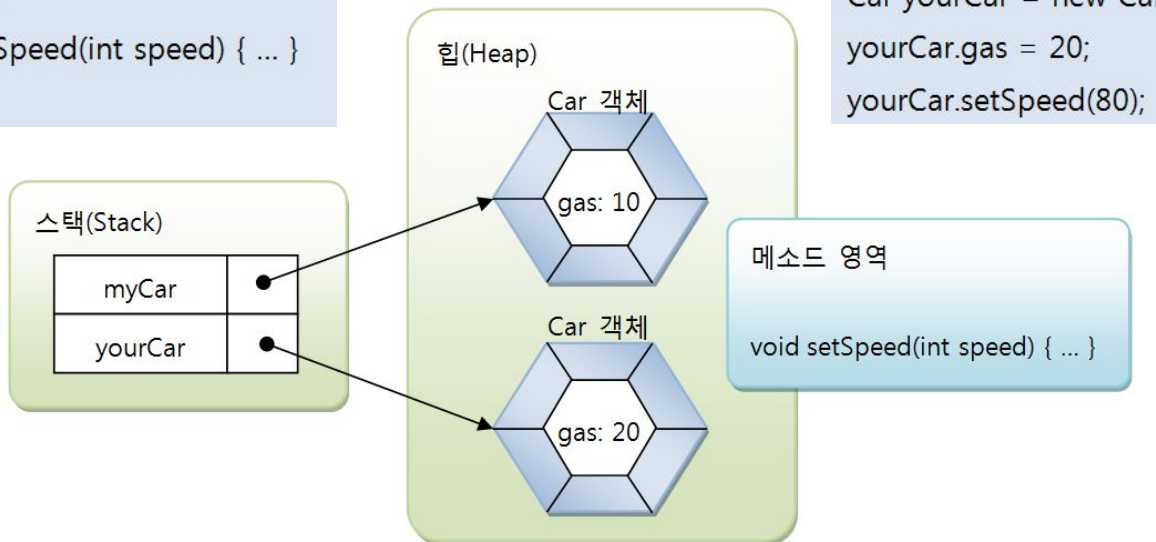
- 인스턴스 멤버는 객체 소속된 멤버이기 때문에 객체가 없이 사용불가

```
public class Car {
   //필드
   int gas;

   //메소드
   void setSpeed(int speed) { ... }
}
```

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

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



9절. 인스턴스 멤버와 this

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;
}
```

10절. 정적 멤버와 static

❖ 정적(static) 멤버란?

- 클래스에 고정된 필드와 메소드 - 정적 필드, 정적 메소드

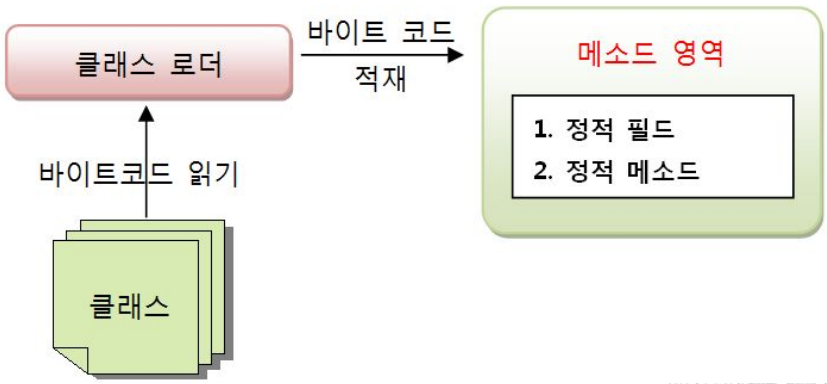
- 정적 멤버는 클래스에 소속된 멤버
  - 객체 내부에 존재하지 않고, 메소드 영역에 존재
  - 정적 멤버는 객체를 생성하지 않고 클래스로 바로 접근해 사용

❖ 정적 멤버 선언

- 필드 또는 메소드 선언할 때 static 키워드 붙임

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

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



10절. 정적 멤버와 static

❖ 정적 멤버 사용

- 클래스 이름과 함께 도트(.) 연산자로 접근

```
클래스.필드;
클래스.메소드( 매개값, ... );
```

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

[바람직한 사용]

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

[바람직하지 못한 사용]

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

10절. 정적 멤버와 static

❖ 인스턴스 멤버 선언 vs 정적 멤버 선언의 기준

- 필드
  - 객체 마다 가지고 있어야 할 데이터 □ 인스턴스 필드
  - 공용적인 데이터 □ 정적 필드

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

10절. 정적 멤버와 static

❖ 정적 초기화 블록

- 클래스가 메소드 영역으로 로딩될 때 자동으로 실행하는 블록

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

- 정적 필드의 복잡한 초기화 작업과 정적 메소드 호출 가능
- 클래스 내부에 여러 개가 선언되면 선언된 순서대로 실행

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

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

10절. 정적 멤버와 static

❖ 정적 메소드와 정적 블록 작성시 주의할 점

- 객체가 없어도 실행 가능
- 블록 내부에 인스턴스 필드나 인스턴스 메소드 사용 불가
- 객체 자신의 참조인 this 사용 불가
  - EX) main()

```
//인스턴스 필드와 메소드
int field1;
void method1() { ... }

//정적 필드와 메소드
static int field2;
static void method2() { ... }
```

```
//정적 블록
static {
    field1 = 10;        (x)
    method1();          (x)
    field2 = 10;        (o)
    method2();          (o)
}

//정적 메소드
static void Method3 {
    this.field1 = 10;       (x)
    this.method1();         (x)
    field2 = 10;            (o)
    method2();              (o)
}
```

컴파일 에러

컴파일 에러

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

10절. 정적 멤버와 static

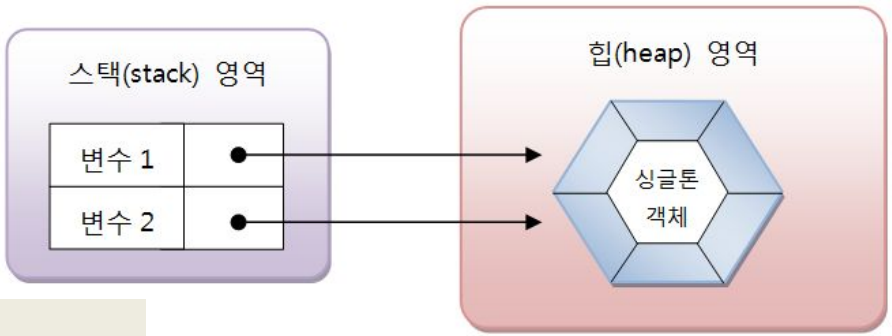
- 싱글톤(Singleton)
  - 하나의 애플리케이션 내에서 단 하나만 생성되는 객체

- 싱글톤을 만드는 방법
  - 외부에서 new 연산자로 생성자를 호출할 수 없도록 막기
    - private 접근 제한자를 생성자 앞에 붙임
  - 클래스 자신의 타입으로 정적 필드 선언
    - 자신의 객체를 생성해 초기화
    - private 접근 제한자 붙여 외부에서 필드 값 변경 불가하도록
  - 외부에서 호출할 수 있는 정적 메소드인 getInstance() 선언
    - 정적 필드에서 참조하고 있는 자신의 객체 리턴

10절. 정적 멤버와 static

❖ 싱글톤 얻는 방법

```
클래스 변수 1 = 클래스.getInstance();
클래스 변수 2 = 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.");
}
```

11절. final 필드와 상수(static final)

❖ final 필드

- 최종적인 값을 갖고 있는 필드 = 값을 변경할 수 없는 필드
- final 필드의 딱 한번의 초기값 지정 방법
  - 필드 선언 시
  - 생성자

```
public class Person {
    final String nation = "Korea";
    final String ssn;
    String name;

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

11절. final 필드와 상수(static final)

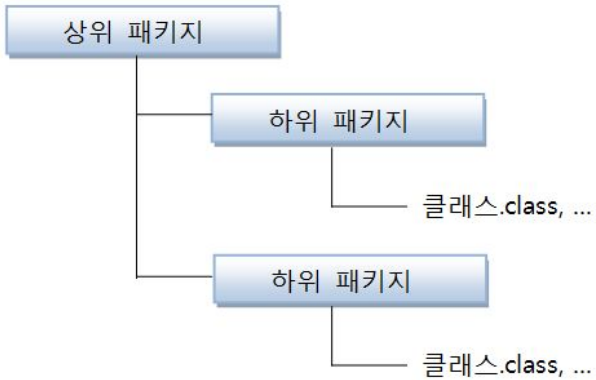
- 상수(static final)
  - 상수 = 정적 final 필드
    - final 필드:
      - 객체마다 가지는 불변의 인스턴스 필드
    - 상수(static final):
      - 객체마다 가지고 있지 않음
      - 메소드 영역에 클래스 별 로 관리되는 불변의 정적 필드
      - 공용 데이터로서 사용
  - 상수 이름은 전부 대문자로 작성
  - 다른 단어가 결합되면 _ 로 연결

12절. 패키지(package)

❖ 패키지란?

- 클래스를 기능별로 묶어서 그룹 이름을 붙여 놓은 것
  - 파일들을 관리하기 위해 사용하는 폴더(디렉토리)와 비슷한 개념
  - 패키지의 물리적인 형태는 파일 시스템의 폴더

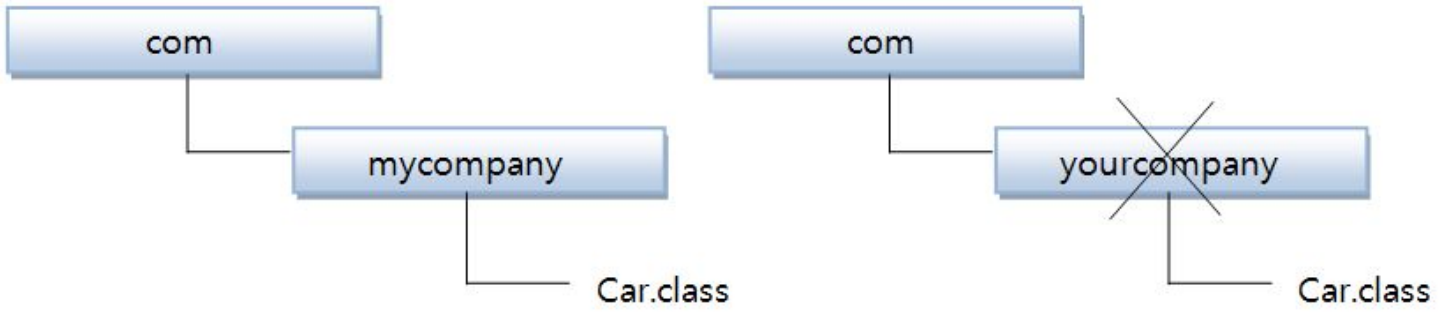
- 클래스 이름의 일부
  - 클래스를 유일하게 만들어주는 식별자
  - 전체 클래스 이름 = 상위패키지.하위패키지.클래스
  - 클래스명이 같아도 패키지명이 다르면 다른 클래스로 취급



12절. 패키지(package)

❖ 패키지란?

- 클래스 선언할 때 패키지 결정
  - 클래스 선언할 때 포함될 패키지 선언
  - 클래스 파일은(~.class) 선언된 패키지와 동일한 폴더 안에서만 동작
  - 클래스 파일은(~.class) 다른 폴더 안에 넣으면 동작하지 않음



12절. 패키지(package)

❖ import 문

- 패키지 내에 같이 포함된 클래스간 클래스 이름으로 사용 가능
- 패키지가 다른 클래스를 사용해야 할 경우
  - 패키지 명 포함된 전체 클래스 이름으로 사용

```
pckage com.mycompany;

public class Car {
    com.hankook.Tire tire = new com.hankook.Tire();
}
```

  - Import 문으로 패키지를 지정하고 사용

```
package com.mycompany;

import com.hankook.Tire;
[ 또는 import com.hankook.*; ]

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

Source>Organize imports (단축키: Ctrl+Shift+O)

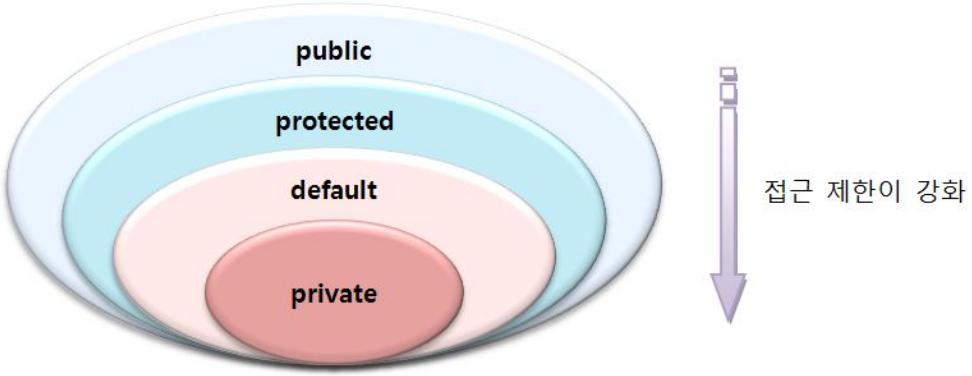
1. Window>Preference>Java>Code Style>Organize imports 를 선택
2. Number of imports needed for .*의 99 를 1 로 변경하고 [OK] 버튼을 클릭한다.
3. 다시한번 Ctrl+Shift+O 를 클릭한다.

13절. 접근 제한자

❖ 접근 제한자(Access Modifier)

- 클래스 및 클래스의 구성 멤버에 대한 접근을 제한하는 역할
  - 다른 패키지에서 클래스를 사용하지 못하도록 (클래스 제한)
  - 클래스로부터 객체를 생성하지 못하도록 (생성자 제한)
  - 특정 필드와 메소드를 숨김 처리 (필드와 메소드 제한)

- 접근 제한자의 종류

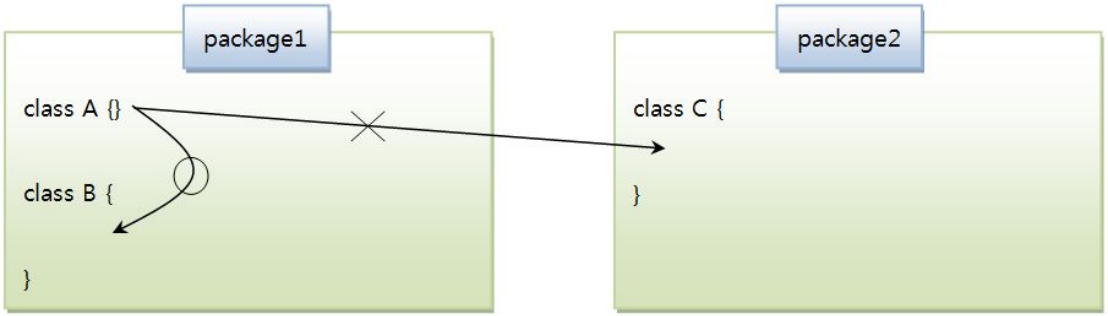


접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스

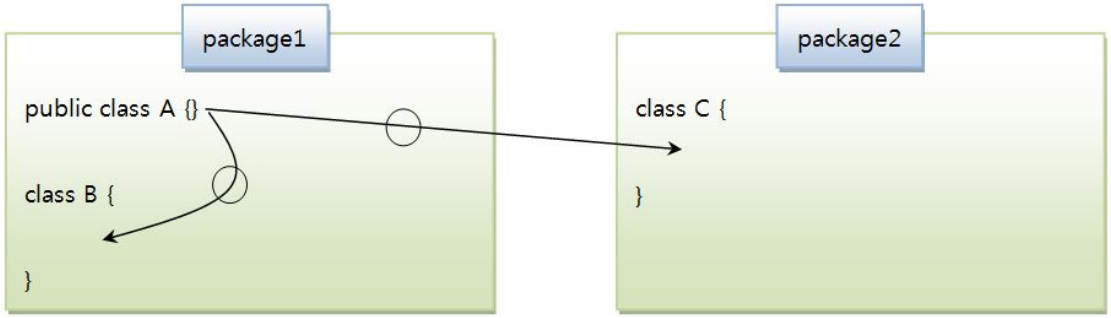
13절. 접근 제한자

❖ 클래스의 접근 제한

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



- public
  - 다른 개발자가 사용할 수 있도록 라이브러리 클래스로 만들 때 유용



13절. 접근 제한자

- 생성자 접근 제한 (p.260~262)
  - 생성자가 가지는 접근 제한에 따라 호출 여부 결정

- 필드와 메소드의 접근 제한 (p.262~264)
  - 클래스 내부, 패키지 내, 패키지 상호간에 사용할 지 고려해 선언

14절. Getter와 Setter

- 클래스 선언할 때 필드는 일반적으로 private 접근 제한
  - 읽기 전용 필드가 있을 수 있음 (Getter의 필요성)
  - 외부에서 엉뚱한 값으로 변경할 수 없도록 (Setter의 필요성)

- Getter
  - private 필드의 값을 리턴 하는 역할 - 필요할 경우 필드 값 가공
  - getFieldName() 또는 isFieldName() 메소드
    - 필드 타입이 boolean 일 경우 isFieldName()

- Setter
  - 외부에서 주어진 값을 필드 값으로 수정
    - 필요할 경우 외부의 값을 유효성 검사
  - setFieldName(타입 변수) 메소드
    - 매개 변수 타입은 필드의 타입과 동일

15절. 어노테이션(Annotation)

❖ 어노테이션(Annotation)이란?

- 프로그램에게 추가적인 정보를 제공해주는 메타데이터(metadata)

- 어노테이션 용도
  - 컴파일러에게 코드 작성 문법 에러 체크하도록 정보 제공
  - 소프트웨어 개발 툴이 빌드나 배치 시 코드를 자동 생성하게 정보 제공
  - 실행 시(런타임시) 특정 기능 실행하도록 정보 제공

15절. 어노테이션(Annotation)

❖ 어노테이션 타입 정의와 적용

- 어노테이션 타입 정의
  - 소스 파일 생성: AnnotatoinName.java
  - 소스 파일 내용

```
public @interface AnnotationName {
}
```

- 어노테이션 타입 적용

```
@AnnotationName
```

```
@Override
public void toString() { ... }
```

15절. 어노테이션(Annotation)

❖ 기본 엘리먼트 value

```
public @interface AnnotationName {
  String value();
  int elementName() default 5;
}
```

기본 엘리먼트 선언

- 어노테이션 적용할 때 엘리먼트 이름 생략 가능

```
@AnnotationName("값");
```

- 두 개 이상의 속성을 기술할 때에는 value=값 형태로 기술

```
@AnnotationName(value="값", elementName=3);
```

15절. 어노테이션(Annotation)

❖ 어노테이션 적용 대상

- 코드 상에서 어노테이션을 적용할 수 있는 대상
- java.lang.annotation.ElementType 열거 상수로 정의

ElementType 열거 상수	적용 대상
TYPE	클래스, 인터페이스, 열거 타입
ANNOTATION_TYPE	어노테이션
FIELD	필드
CONSTRUCTOR	생성자
METHOD	메소드
LOCAL_VARIABLE	로컬 변수
PACKAGE	패키지

15절. 어노테이션(Annotation)

❖ 어노테이션 유지 정책

- 어노테이션 적용 코드가 유지되는 시점을 지정하는 것
- java.lang.annotation.RetentionPolicy 열거 상수로 정의

RetentionPolicy 열거 상수	설명
SOURCE	소스상에서만 어노테이션 정보를 유지한다. 소스 코드를 분석할 때만 의미가 있으며, 바이트 코드 파일에는 정보가 남지 않는다.
CLASS	바이트 코드 파일까지 어노테이션 정보를 유지한다. 하지만 리플렉션을 이용해서 어노테이션 정보를 얻을 수는 없다.
RUNTIME	바이트 코드 파일까지 어노테이션 정보를 유지하면서 리플렉션을 이용해서 런타임에 어노테이션 정보를 얻을 수 있다.

- 리플렉션(Reflection): 런타임에 클래스의 메타 정보를 얻는 기능
  - 클래스가 가지고 있는 필드, 생성자, 메소드, 어노테이션의 정보를 얻을 수 있음
  - 런타임 시 어노테이션 정보를 얻으려면 유지 정책을 RUNTIME으로 설정

15절. 어노테이션(Annotation)

❖ 런타임시 어노테이션 정보 사용하기

- 클래스에 적용된 어노테이션 정보 얻기
  - 클래스.class 의 어노테이션 정보를 얻는 메소드 이용

- 필드, 생성자, 메소드에 적용된 어노테이션 정보 얻기
  - 다음 메소드 이용해 java.lang.reflect 패키지의 Field, Constructor, Method 클래스의 배열 얻어냄

리턴타입	메소드명(매개변수)	설명
Field[]	getFields()	필드 정보를 Field 배열로 리턴
Constructor[]	getConstructors()	생성자 정보를 Contructor 배열로 리턴
Method[]	getDeclaredMethods()	메소드 정보를 Method 배열로 리턴

15절. 어노테이션(Annotation)

❖ 런타임시 어노테이션 정보 사용하기

- Field, Constructor, Method가 가진 다음 메소드 호출
  - 어노테이션 정보를 얻기 위한 메소드

리턴타입	메소드명(매개변수)
boolean	isAnnotationPresent(Class<? extends Annotation> annotationClass)
	지정한 어노테이션이 적용되었는지 여부. Class 에서 호출했을 경우 상위 클래스에 적용된 경우에도 true 를 리턴한다.
Annotation	getAnnotation(Class<T> annotationClass)
	지정한 어노테이션이 적용되어 있으면 어노테이션을 리턴하고 그렇지 않다면 null 을 리턴한다. Class 에서 호출했을 경우 상위 클래스에 적용된 경우에도 어노테이션을 리턴한다.
Annotation[]	getAnnotations()
	적용된 모든 어노테이션을 리턴한다. Class 에서 호출했을 경우 상위 클래스에 적용된 어노테이션도 모두 포함한다. 적용된 어노테이션이 없을 경우 길이가 0 인 배열을 리턴한다.
Annotation[]	getDeclaredAnnotations()
	직접 적용된 모든 어노테이션을 리턴한다. Class 에서 호출했을 경우 상위 클래스에 적용된 어노테이션은 포함되지 않는다.