

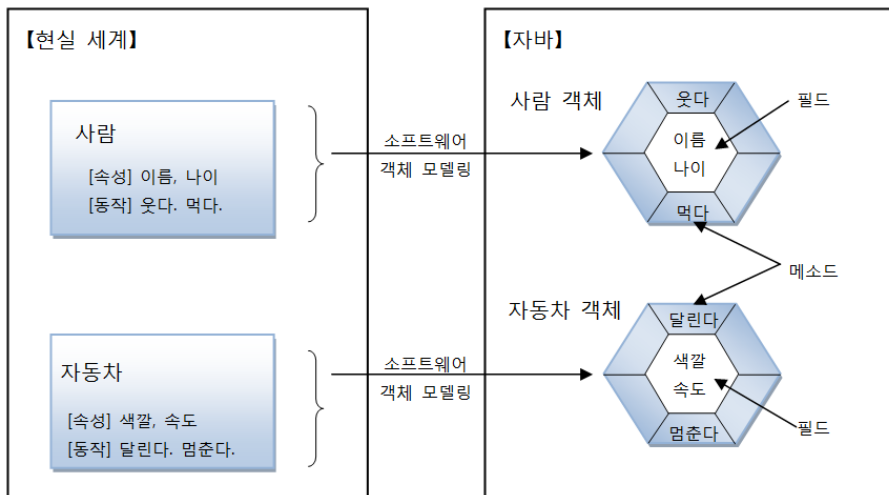
06장 클래스

6.1 객체 지향 프로그래밍

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

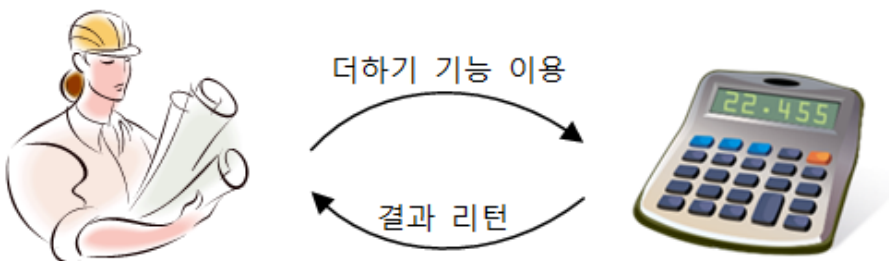
6.1.1 객체란?

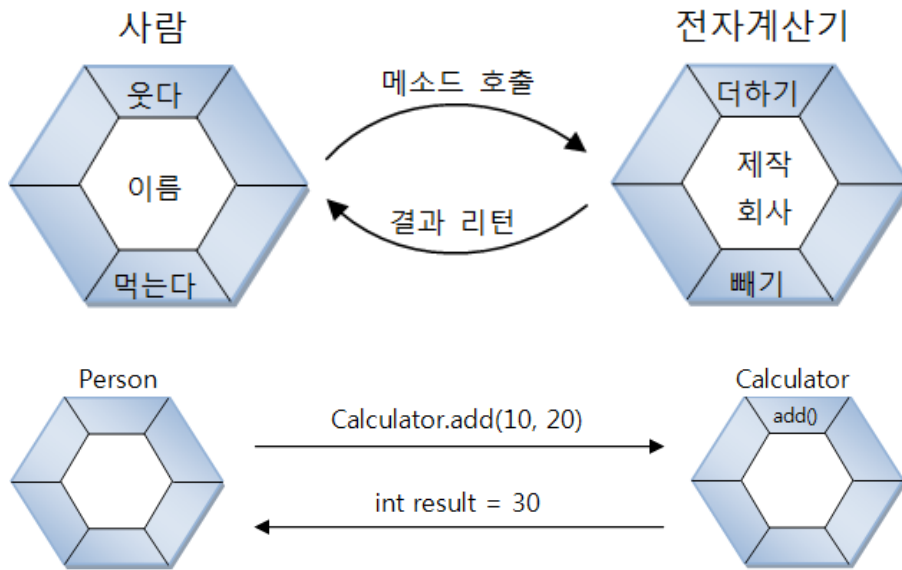
- 물리적으로 존재하거나 추상적으로 생각할 수 있는 것 중에서 자신의 속성을 가지고 있고, 다른 것과 식별 가능한 것을 말한다.



6.1.2 객체의 상호작용

- 객체들은 각각 독립적으로 존재하고, 다른 객체와 서로 상호작용하면서 동작한다.

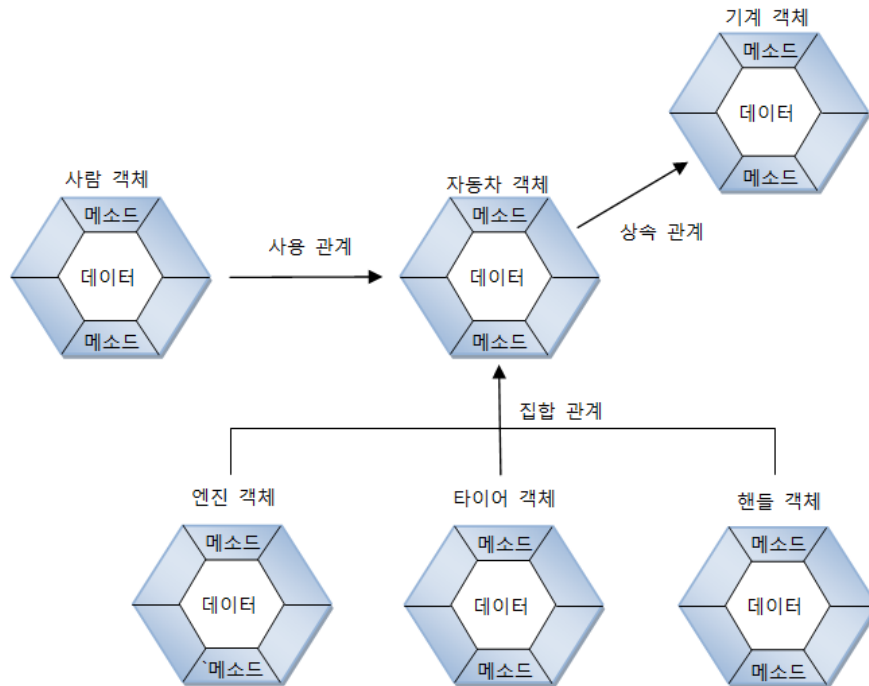




```
int result = Calculator.add(10, 20); //리턴한 값을 int 변수에 저장
```

6.1.3 객체 간의 관계

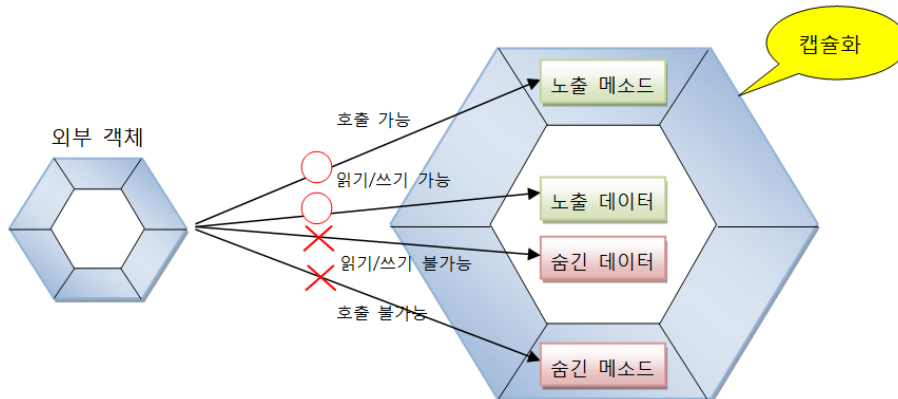
- 객체는 개별적으로 사용될 수 있지만, 대부분 다른 객체와 관계를 맺고 있다.
- 관계의 종류
 - 집합 관계: 완제품과 부품의 관계
 - 사용 관계: 객체가 다른 객체를 사용하는 관계
 - 상속 관계: 종류 객체와 구체적인 사물 객체 관계



6.1.4 객체 지향 프로그래밍의 특징

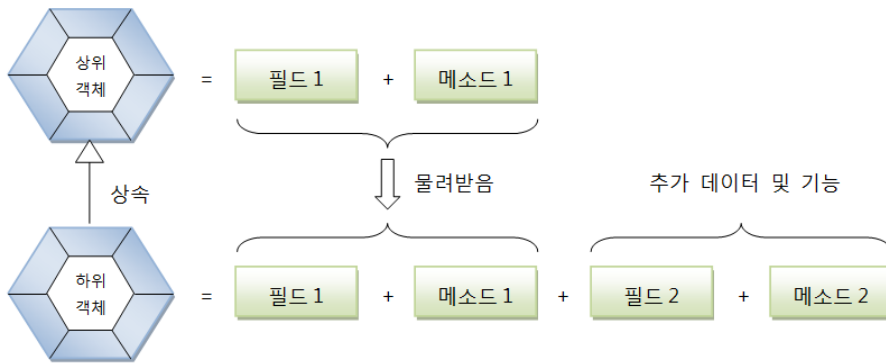
(1) 캡슐화(Encapsulation)

- 객체의 필드, 메소드를 하나로 묶고, 실제 구현 내용을 감추는 것을 말한다. 캡슐화하여 보호하는 이유는 외부의 잘못된 사용으로 인해 객체가 손상되지 않도록 하는데 있다.



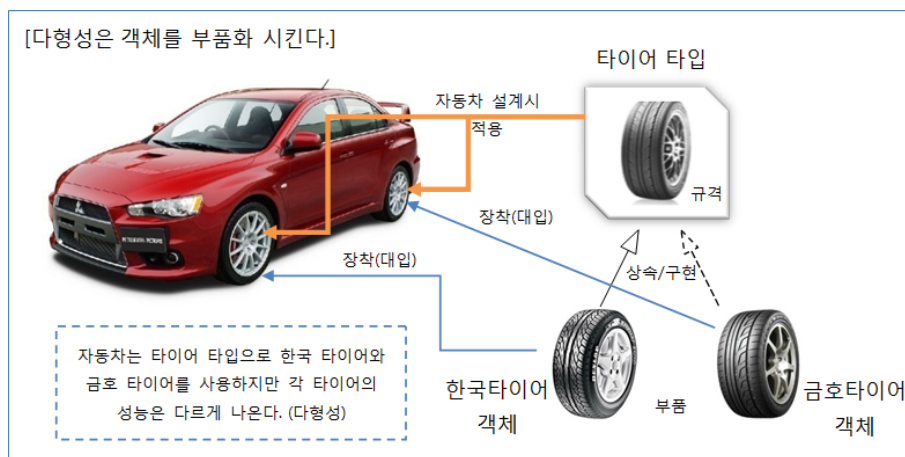
(2) 상속(Inheritance)

- 상위 객체는 자기가 가지고 있는 필드와 메소드를 하위 객체에게 물려주어 하위 객체가 사용할 수 있도록 해준다.



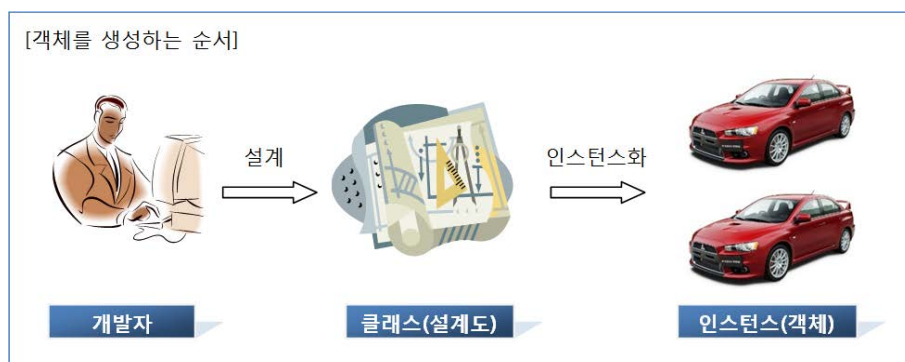
(3) 다형성(Polymorphism)

- 같은 타입이지만 실행 결과가 다양한 객체를 이용할 수 있는 성질을 말한다.



6.2 객체와 클래스

- 클래스에는 객체를 생성하기 위한 필드와 메소드가 정의되어 있다. 클래스로부터 만들어진 객체를 해당 클래스의 인스턴스(instance)라고 한다.



6.3 클래스 선언

■ 자바 식별자 작성 규칙에 따라야 한다.

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

```
public class Car {
}

class Tire {
}
```

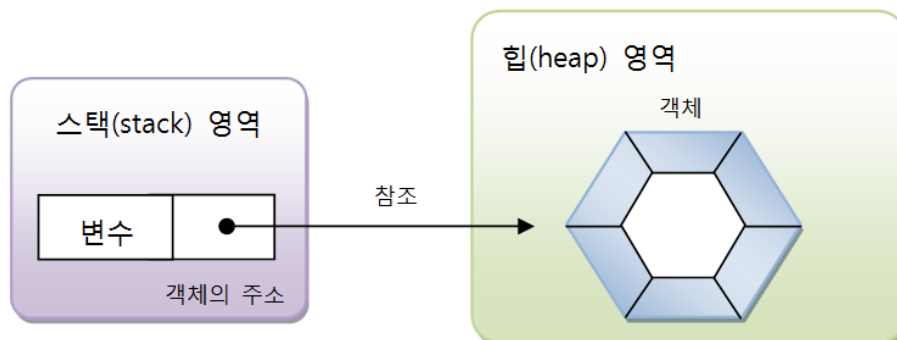
6.4 객체 생성과 클래스 변수

■ new 연산자

- 기본형: new 클래스();
- 객체 생성 역할: 클래스()는 생성자를 호출하는 코드, 생성된 객체는 힙 메모리 영역에 생성
- new 연산자는 객체를 생성 후, 객체 생성 번지 리턴

■ 클래스 변수

- 기본형: 클래스 변수;
변수 = new 클래스();
클래스 변수 = new 클래스(); //한 개의 실행문으로 작성할 수 있다.
- new 연산자에 의해 리턴 된 객체의 번지 저장 (참조 타입 변수)
- 힙 영역의 객체를 사용하기 위해 사용
- 예) Student s1 = new Student();



■ 클래스 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

6.5 클래스의 구성 멤버

- 필드(Field): 속성
- 생성자(Constructor): class명과 동일해야 한다. 클래스에 정의되지 않았어도 컴파일러가 자동으로 생성한다. 필드(멤버변수)를 초기화 시킨다.
- 메소드(Method): 객체의 동작(기능)

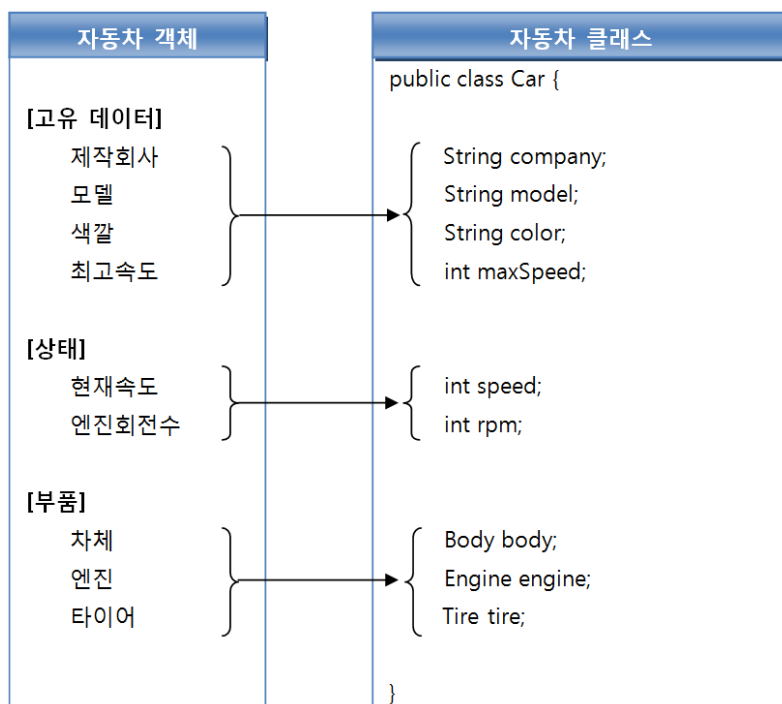
클래스 선언과 객체 생성

```
public class 클래스명 {
    필드;
    생성자;
    메소드;
}

클래스 변수 = new 클래스();
// 스택영역에 저장된 클래스 변수는 힙 영역에 생성된 객체의 주소값을 저장한다.
```

6.6 필드

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



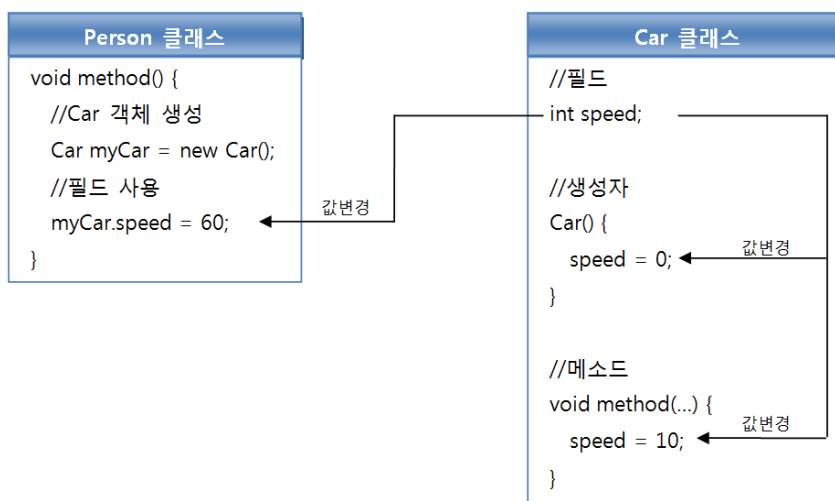
6.6.1 필드 선언

- 기본형: 타입 필드 [= 초기값];
- 예) String company = "현대자동차";
String model = "그랜저";
int maxSpeed = 300;
- 초기값이 지정되지 않은 필드들은 객체 생성 시 자동으로 기본 초기값으로 설정된다. 다음 표는 필드 타입별 기본 초기값을 보여준다.

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

6.6.2 필드 사용

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



6.7 생성자(Constructor)

- new 연산자에 의해 호출되어 객체의 초기화 담당
 - 필드의 값 설정
 - 메소드 호출해 객체를 사용할 수 있도록 준비하는 역할 수행
- 생성자를 실행시키지 않고는 클래스로부터 객체를 만들 수 없다.

6.7.1 기본 생성자

- 모든 클래스는 생성자가 반드시 존재하며 하나 이상 가질 수 있다.
- 생성자 선언을 생략하면 컴파일러는 다음과 같이 중괄호 {} 블록 내용이 비어있는 기본 생성자(Default Constructor)를 바이트 코드에 자동 추가시킨다.

소스 파일(Car.java)	→	바이트 코드 파일(Car.class)
<pre>public class Car { } </pre>		<pre>public class Car { public Car() {} //자동 추가 } 기본 생성자 </pre>

6.7.2 생성자 선언

- 디폴트 생성자 대신 개발자가 직접 선언
- 기본형: 클래스(매개변수선언, ...) {
 - //객체의 초기화 코드
- 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);
```

6.7.3 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


```
public class Korean {
    String nation = "대한민국";
    String name;
    String ssn;
}
Korean k1 = new Korean();

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");
```

- 생성자의 매개 변수와 클래스의 필드명 같은 경우 this 사용

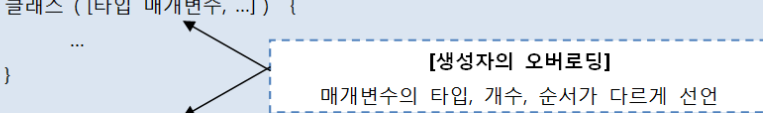
```
public Korean(String name, String ssn) {
    this.name = name;
    this.ssn = ssn;
}
```

6.7.4 생성자 오버로딩(Overloading)

- 매개변수의 타입, 개수, 순서가 다른 생성자 여러 개 선언

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

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



```
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);
```

6.7.5 다른 생성자 호출(this())

- 생성자 오버로딩되면 생성자 간의 중복된 코드 발생
- 생성자에서 다른 생성자를 호출할 때에는 this() 코드를 사용한다.
- 생성자 실행문의 첫번째 줄에 위치해야 한다.

[Car.java] 다른 생성자를 호출해서 중복 코드 줄이기

```
package sec07.exam04_other_constructor_call;

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

    //생성자
    Car() {
    }

    Car(String model) {
        this(model, null, 0);
    }

    Car(String model, String color) {
        this(model, color, 0);
    }

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

[꿀팁] this., this(), super., super() 사용법

- this. : 생성자와 메소드 안에서 멤버변수와 매개변수 이름이 동일한 경우에 주로 사용하며 생략하게 되면 매개변수명과 구별되지 않아 값 전달이 되지 않는다.

// this: 내부 레퍼런스 변수
예) class ClassName{

```
int a, b; // 필드(멤버변수)
public ClassName(int a, int b){ // 매개변수 이름이 동일한 경우
    this.a = a;
    this.b = b;
}
```

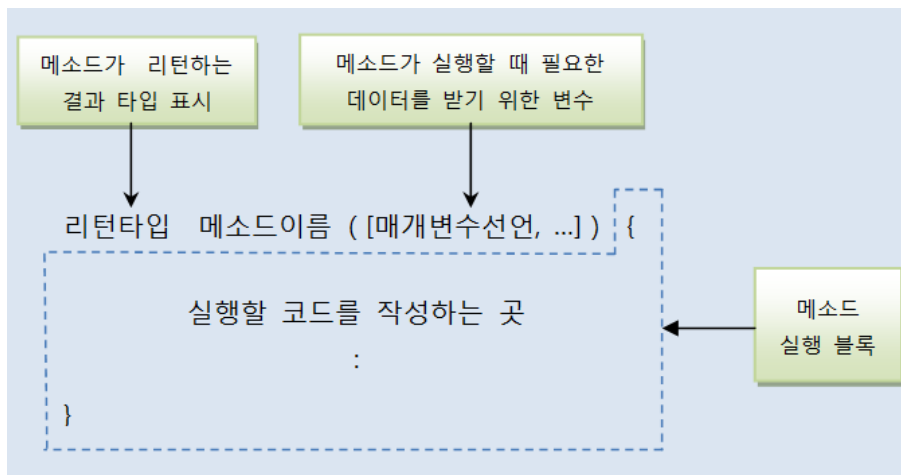
- `this()` : 같은 클래스안에 있는 생성자를 호출할때 사용
- `super.` : 부모 클래스의 (은닉된) 필드와 메소드를 호출할 때 사용하며, 자식 클래스의 메소드 안에서만 사용 가능하다.
예) `super.x` `super.parentPrn()`

- `super()` : 부모 클래스의 생성자를 호출할 때 사용하며, 자식 클래스의 생성자 안에서 첫문장에 기술해야 한다.

6.8 메소드(Method)

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

6.8.1 메소드 선언



(1) 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);
```

(2) 메소드 이름

- 숫자로 시작하면 안 되고, \$와 _를 제외한 특수 문자를 사용하지 말아야 한다.
- 관례적으로 메소드명은 소문자로 작성한다.
- 서로 다른 단어가 혼합된 이름이라면 뒤이어 오는 단어의 첫머리 글자는 대문자로 작성한다.

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

(3) 매개 변수 선언

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

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

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

(4) 매개 변수의 수를 모를 경우

- 매개 변수의 개수를 알 수 없는 경우에 매개 변수를 배열 타입으로 선언하여 사용한다.
- 메소드의 매개변수를 "...로 선언하게 되면, 메소드 호출 시 넘겨준 값의 수에 따라 자동으로 배열이 생성되고 매개값으로 사용된다.

```
int sum1(int[] values) { }
int sum2(int ... values) { }

//메소드 호출
int[] values = {1,2,3};
int result1 = sum1(values);
int result1 = sum1(new int[] {1,2,3,4,5});
int result2 = sum2(1,2,3);
```

```
int result2 = sum2(1,2,3,4,5);
```

[Computer.java] 매개 변수의 수를 모를 경우

```
package sec08.exam01_method_declaration;

public class Computer {
    int sum1(int[] values) {
        int sum = 0;
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }
        return sum;
    }

    int sum2(int... values) {
        int sum = 0;
        for (int i = 0; i < values.length; i++) {
            sum += values[i];
        }
        return sum;
    }
}
```

[ComputerExample.java] 매개 변수의 수를 모를 경우

```
package sec08.exam01_method_declaration;

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

        int[] values1 = { 1, 2, 3 };
        int result1 = myCom.sum1(values1);
        System.out.println("result1: " + result1);

        int result2 = myCom.sum1(new int[] { 1, 2, 3, 4, 5 });
        System.out.println("result2: " + result2);

        int result3 = myCom.sum2(1, 2, 3);
        System.out.println("result3: " + result3);

        int result4 = myCom.sum2(1, 2, 3, 4, 5);
        System.out.println("result4: " + result4);
    }
}
```

6.8.2 리턴(return)문

- 메소드 실행을 중지하고 리턴값 지정하는 역할

(1) 리턴값이 있는 메소드

- 반드시 리턴(return)문 사용해서 리턴값을 지정해야 한다.
- 리턴값은 리턴 타입이거나 리턴 타입으로 변환될 수 있어야 한다.

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

int plus(int x, int y) {
    byte result = (byte)(x + y);
    return result; // byte와 short은 int로 자동 타입 변환된다.
}
```

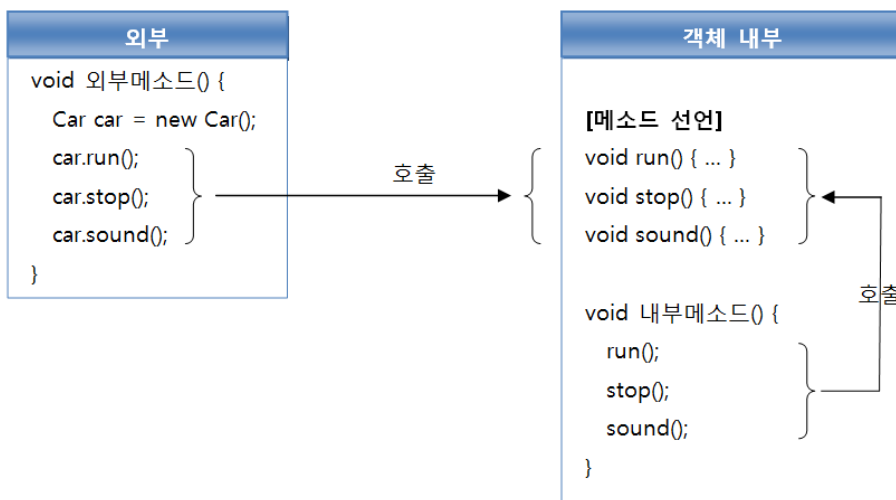
(2) 리턴값이 없는 메소드(void)

- 메소드 실행을 강제 종료시킨다.

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

6.8.3 메소드 호출

- 클래스 내부: 메소드 이름으로 호출
- 클래스 외부: 객체 생성 후, 참조 변수를 이용해 호출



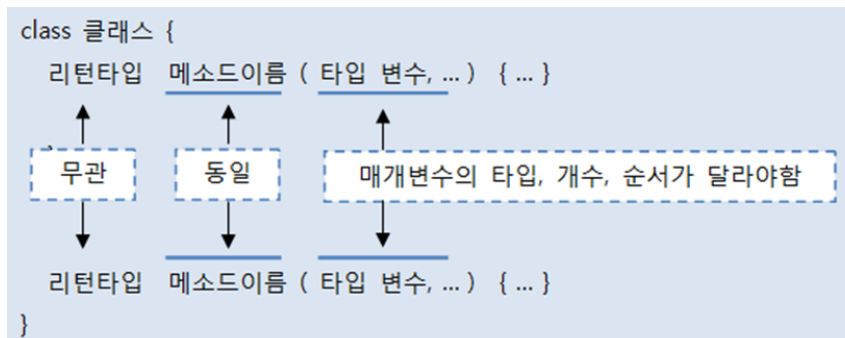
[꿀팁] Call by value vs. Call by reference

```
// call by value (실제 값에 의한 메소드 호출)
static void check01(int a){
    System.out.println("전달된 값="+a);
}

// call by reference (주소값 전달에 의한 메소드 호출)
static void check03(String s){
    System.out.println("전달된 값="+s);
}
```

6.8.4 메소드 오버로딩(Method Overloading)

- 1개의 클래스에 동일한 이름의 메소드를 여러개 정의하는 것



```
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);

[Calculator.java]

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

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

6.9 인스턴스 멤버와 this

- 인스턴스(instance) 멤버
 - 객체(인스턴스)를 생성한 후 사용할 수 있는 필드와 메소드를 말하는데, 이들을 각각 인

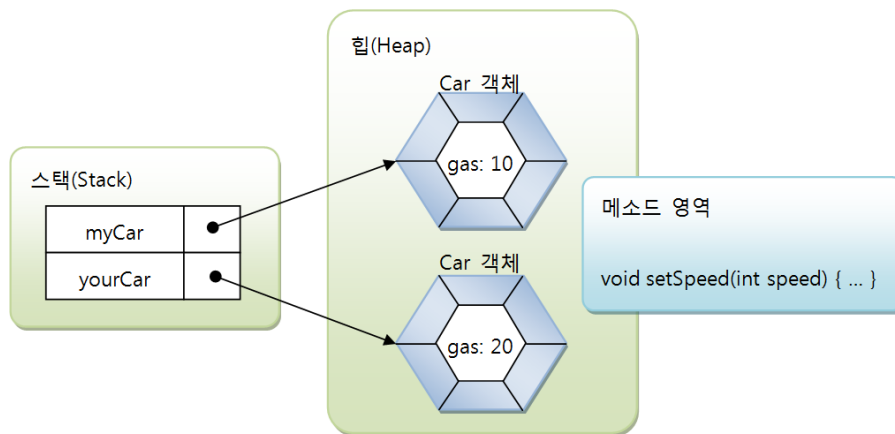
스턴스 필드, 인스턴스 메소드라고 부른다.

- 인스턴스 필드와 메소드는 객체에 소속된 멤버이기 때문에 객체 없이는 사용할 수 없다.

```
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);
```



■ this

- 자신을 "나"라고 하듯이, 객체는 자신을 "this"라고 한다. 따라서 this.model은 자신이 가지고 있는 model 필드라는 뜻이다.
- 주로 생성자와 메소드의 매개 변수 이름이 필드와 동일한 경우, 인스턴스 멤버인 필드임을 명시하고자 할 때 사용한다.

[Car.java] 인스턴스 멤버와 this

```
package sec09.exam01_instance_member;

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

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

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



```
void run() {
    for (int i = 10; i <= 50; i += 10) {
        this.setSpeed(i);
        System.out.println(this.model + "가 달립니다.(시속:" + this.speed +
"km/h)");
    }
}
```

6.10 정적 멤버와 static

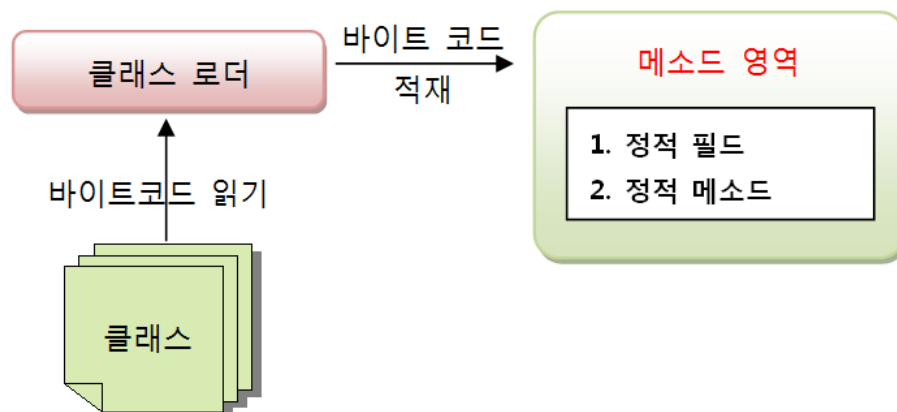
- 클래스에 고정된 필드와 메소드 - 정적 필드, 정적 메소드
- 정적 멤버는 클래스에 소속된 멤버
 - 객체 내부에 존재하지 않고, 메소드 영역에 존재
 - 정적 멤버는 객체를 생성하지 않고 클래스로 바로 접근해 사용

6.10.1 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, ... ) { ... }
}
```



6.10.2 정적 멤버 사용

- 클래스 이름과 함께 도트(.) 연산자로 접근
- 기본형: 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);

//바람직하지 못한 사용, 이클립스에서는 경고 표시가 나타난다.
Calculator myCalcu = new Calculator();
double result1 = 10 * 10 * myCalcu.pi;
int result2 = myCalcu.plus(10, 5);
```

6.10.3 정적 초기화 블록

- 자바는 정적 필드의 복잡한 초기화 작업을 위해서 정적 블록(static block)를 제공한다.

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

- 정적 블록은 클래스가 메모리로 로딩될 때 자동적으로 실행된다. 클래스 내부에 여러 개가 선언되면 선언된 순서대로 실행된다.

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

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

6.10.4 정적 메소드와 블록 선언 시 주의할 점

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

```
public class ClassName {
    //인스턴스 필드와 메소드
    int field1;
    void method1() {...}
    //정적 필드와 메소드
    static int fields2;
    static void method2() {...}

    //정적 블록
    static {
        field1 = 10; //(x), 컴파일 에러
        method1(); //(x)
        fields2 = 10; //(o)
        method2(); //(o)
    }

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

6.10.5 싱글톤(Singleton)

- 싱글톤(Singleton)
 - 하나의 애플리케이션 내에서 단 하나만 생성되는 객체
- 싱글톤을 만드는 방법
 - 첫째, 클래스 자신의 타입으로 정적 필드 선언
 - 자신의 객체를 생성해 초기화
 - private 접근 제한자 붙여 외부에서 필드 값 변경 불가하도록
 - 둘째, 외부에서 new 연산자로 생성자를 호출할 수 없도록 막기
 - private 접근 제한자를 생성자 앞에 붙임
 - 셋째, 외부에서 호출할 수 있는 정적 메소드인 getInstance() 선언
 - 정적 필드에서 참조하고 있는 자신의 객체 리턴

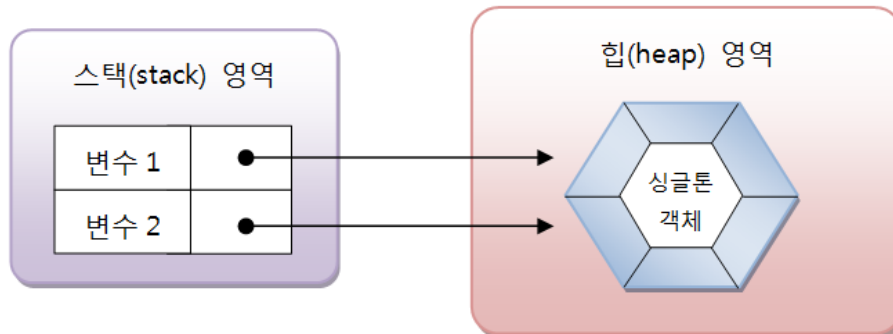
```
public class 클래스 {
    //정적 필드
    private static 클래스 singleton = new 클래스();

    //생성자
    private 클래스() {}
}
```

```
//정적 메소드
static 클래스 getInstance() {
    return singleton;
}
}
```

■ 싱글톤 얻는 방법

클래스 변수 1 = 클래스.getInstance();
클래스 변수 2 = 클래스.getInstance();



[Singleton.java] 싱글톤

```
package sec10.exam04_singleton;

public class Singleton {
    private static final Singleton singleton = new Singleton();
    // private static Singleton singleton;

    private Singleton() {
    }

    static Singleton getInstance() {
        // if (singleton == null ) { singleton = new Singleton(); }
        return singleton;
    }
}
```

[SingletonExample.java] 싱글톤 객체

```
package sec10.exam04_singleton;

public class SingletonExample {
    public static void main(String[] args) {
        /*
        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.");
    }
}

```

[꿀팁] Java에서 DB 접속 방법

1. JDBC

- 사용자가 요청을 할 때마다 매번 드라이버를 로드하고 커넥션 객체를 생성하여 연결하고 종료한다.

2. DBCP(커넥션풀)

- Java에서 DB연결시에 주로 사용한다.
- 풀 속에 미리 커넥션이 생성되어 있기 때문에 커넥션을 생성하는 데 드는 연결 시간이 소비되지 않는다.
- 커넥션을 계속해서 재사용하기 때문에 생성되는 커넥션 수가 많지 않다.
- 커넥션 풀을 사용하면 커넥션을 생성하고 닫는 시간이 소모되지 않기 때문에 그만큼 어플리케이션의 실행 속도가 빨라지며, 또한 한 번에 생성될 수 있는 커넥션 수를 제어하기 때문에 동시 접속자 수가 몰려도 웹 어플리케이션이 쉽게 다운되지 않는다.
- 커넥션 풀을 너무 크게 해놓으면 당연히 메모리 소모가 클 것이고, 적게 해놓으면 커넥션이 많이 발생할 경우 대기시간이 발생할 것이다.

3. Singleton

- 한 번의 접속만으로 모두 처리하므로 자원의 사용(메모리, DB Session수)을 줄일 수 있다.
- Many DB drivers are not thread safe. Using a singleton means that if you have many threads, they will all share the same connection. The singleton pattern does not give you thread safety. It merely allows many threads to easily share a "global" instance.

[꿀팁] 싱글톤 동기화와 늦은 초기화(Lazy Initialization) 기법

- 일반적인 싱글톤 기법(이른 초기화)은 객체를 사용하기 전에 미리 생성해 놓는다. 만약 이러한 클래스가 수백개가 있고 클라이언트는 이러한 클래스를 사용하지 않고 작업이 종료된다면 괜한 자원만 소비된 셈이다.
- 실제로 필요할 때까지 객체를 생성하지 않는 방법 -> 불필요한 부하가 걸리지 않는다.

[SingletonLazyInit.java] Singleton Lazy Initialization 기법 및 동기화

```

01 package sec10.exam04_singleton;
02
03 public class SingletonLazyInit {
04     private volatile static SingletonLazyInit singleton; // DCL을 위한 volatile 키워드 이용
05
06     private SingletonLazyInit() {
07     }
08
09     public static SingletonLazyInit getInstance() {
10         if (singleton == null) {
11             synchronized (SingletonLazyInit.class) { // 생성할때만 동기화 시킴
12                 if (singleton == null) { // DCL(Double-Checking Locking)
13                     singleton = new SingletonLazyInit();

```

```

14     }
15     }
16     }
17     return singleton;
18 }
19 }

```

6.11 final 필드와 상수

6.11.1 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;
    }
}

```

[FinalTest01.java] final 필드

```

class FinalMember {
    final int a = 10; // final 필드로 선언한다.

    public void setA(int a) {
        this.a = a; // 값을 변경할 수 없어 에러가 발생한다.
    }
}

public class FinalTest01 {
    public static void main(String[] args) {
        FinalMember ft = new FinalMember();
    }
}

```

```

        final int a = 1000;
        ft.setA(100);
        System.out.println(a);
    }
}

```

6.11.2 상수(constant, static final)

- 상수는 여러 가지 값으로 초기화 될 수 없고 공용성(static)을 띠며 불변(final)해야 한다.
- 상수명은 모두 대문자로 작성하는 것이 관례이다.

```

//기본형
static final 타입 상수 [= 초기값];

static final double PI = 3.14159;
static final double EARTH_SURFACE_AREA;

```

```

[Earth.java] static final

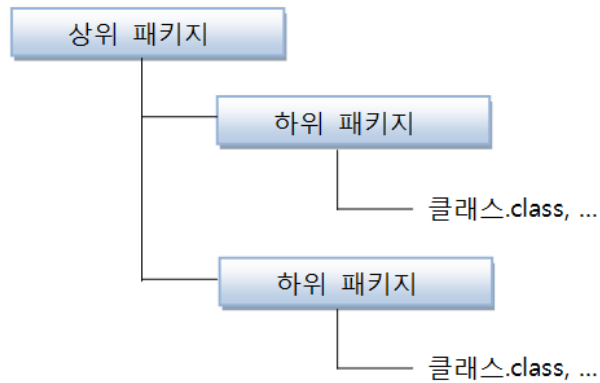
public class Earth {
    static final double EARTH_RADIUS = 6400;
    static final double EARTH_SURFACE_AREA;

    static {
        EARTH_SURFACE_AREA = 4 * Math.PI * EARTH_RADIUS * EARTH_RADIUS;
        // Math.PI는 자바에서 제공하는 상수
    }
}

```

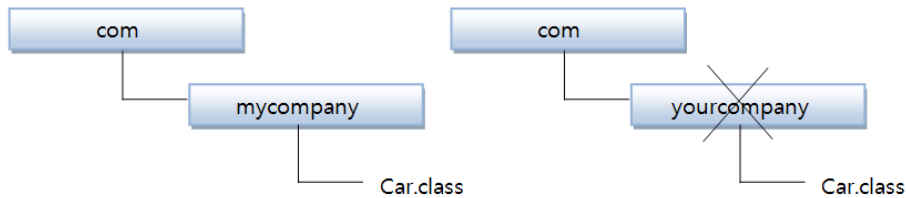
6.12 패키지(package)

- 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



■ 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



6.12.1 패키지 선언

```
package 상위패키지.하위패키지;

public class ClassName { ... }
```

6.12.2 패키지 선언이 포함된 클래스 컴파일

```
C:\Temp>javac -d . Application.java -encoding UTF8
C:\Temp>java sec12.exam01_package_compile.Application
애플리케이션을 실행합니다.
C:\Temp>
```

6.12.3 이클립스에서 패키지 생성과 클래스 생성

- 프로젝트의 src 폴더를 선택하고 메뉴에서 [File -> New -> Package]를 선택하면 된다. 또는 마우스 오른쪽 버튼을 클릭해서 [New -> Package]를 선택해도 좋다.

6.12.4 import문

- 패키지가 다른 클래스를 사용해야 할 경우에 Import 문으로 패키지를 지정하고 사용한다.

- 이클립스에는 개발자가 import문을 작성하지 않아도 사용된 클래스를 조사해서 필요한 import문을 자동적으로 추가하는 기능이 있다.
 - 메뉴: Source > Organize imports
 - 단축키: Ctrl + Shift + O

```
package com.mycompany;

import com.hankook.Tire;

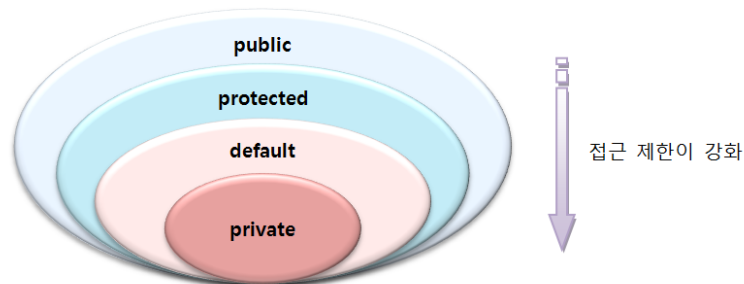
public class Car {
    Tire tire = new Tire();
}
```

6.13 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 종류와 적용할 대상
 - 빨간색 부분은 같은 패키지(default) 혹은 자식 클래스(protected)인 경우에 해당한다.

접근 제한자	같은 클래스	같은 패키지	자식 클래스	다른 패키지
private	0	X	X	X
생략(default)	0	0	X (0)	X
protected	0	0	0	X (0)
public	0	0	0	0

()는 상속관계인 경우이다.



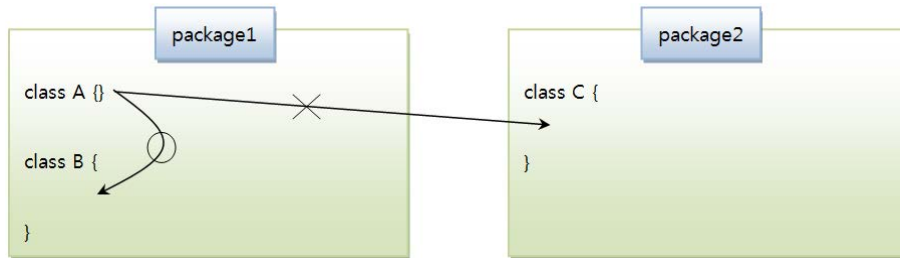
접근 제한	적용 대상	접근할 수 없는 클래스
public	클래스, 필드, 생성자, 메소드	없음
protected	필드, 생성자, 메소드	자식 클래스가 아닌 다른 패키지에 소속된 클래스
default	클래스, 필드, 생성자, 메소드	다른 패키지에 소속된 클래스
private	필드, 생성자, 메소드	모든 외부 클래스

6.13.1 클래스의 접근 제한

- 클래스에 적용할 수 있는 접근 제한은 public과 default 단 두가지이다.

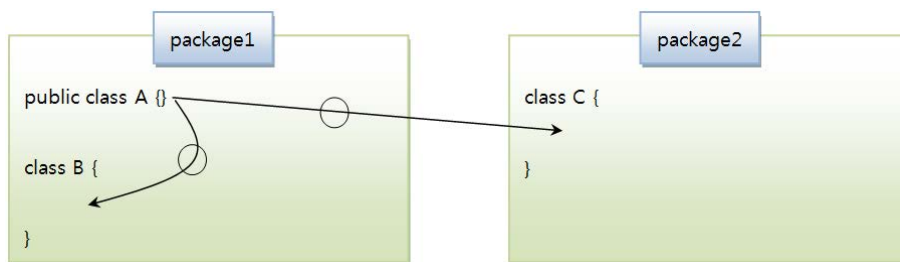
(1) default 접근 제한

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



(2) public 접근 제한

- 다른 개발자가 사용할 수 있도록 라이브러리 클래스로 만들 때 유용



6.13.2 생성자의 접근 제한

- 생성자가 가지는 접근 제한에 따라 호출 여부 결정

```
public class ClassName {
    public ClassName(...) {...}
    protected ClassName(...) {...}
    ClassName(...) {...}
    private ClassName(...) {...}
}
```

6.13.3 필드와 메소드의 접근 제한

- 클래스 내부, 패키지 내, 패키지 상호간에 사용할 지 고려해 선언

[A.java] 필드와 메소드의 접근 제한

```
package sec13.exam03_field_method_access.package1;

public class A {
    //필드
    public int field1;
```

```

    int field2;
    private int field3;

    //생성자
    public A() {
        field1 = 1;
        field2 = 1;
        field3 = 1;

        method1();
        method2();
        method3();
    }

    //메소드
    public void method1() {}
    void method2() {}
    private void method3() {}
}

```

[B.java] 필드와 메소드의 접근 제한

```

package sec13.exam03_field_method_access.package1;

public class B {
    public B() {
        A a = new A();
        a.field1 = 1; // (0)
        a.field2 = 1; // (0)
        a.field3 = 1; // (X)

        a.method1(); // (0)
        a.method2(); // (0)
        a.method3(); // (X)
    }
}

```

[C.java] 필드와 메소드의 접근 제한

```

package sec13.exam03_field_method_access.package2;

import sec13.exam03_field_method_access.package1.A;

public class C {
    public C() {
        A a = new A();
        a.field1 = 1; // (0)
        a.field2 = 1; // (X)
        a.field3 = 1; // (X)

        a.method1(); // (0)
        a.method2(); // (X)
        a.method3(); // (X)
    }
}

```

6.14 Getter와 Setter 메소드

- 객체 데이터를 객체 외부에서 직접적으로 마음대로 읽고 변경할 경우, 객체의 무결성이 깨어질 수 있어 은닉(private)시키고 매개값을 검증할 수 있는 메소드를 공개(public)하여 데이터에 접근을 유도한다.
 - Getter
 - private 필드의 값을 리턴 하는 역할
 - getFieldName() 또는 isFieldName() 메소드 : 필드 타입이 boolean일 경우에는 get으로 시작하지 않고 is로 시작하는 것이 관례이다.
 - Setter
 - 외부에서 주어진 값을 필드 값으로 수정 -> 필요할 경우 외부의 값을 유효성 검사
 - setName(타입 변수) 메소드

[Car.java] Getter와 Setter 메소드 선언

```
package sec14.exam01_getter_setter;

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

    // 생성자

    // 메소드
    public int getSpeed() {
        return speed;
    }

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

    public boolean isStop() {
        return stop;
    }

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

[과제] 회원정보 입력

1. 키보드를 통해서 각 회원들의 정보를 입력 받는 클래스(MemberInput)를 작성한다. 이때 키보드를 입력한 회원의 정보는 새로운 회원정보를 저장하는 클래스(MemberInfo01)의 필드에 저장을 시킨후 출력하는 프로그램을 작성하라. (단, 2명 이상의 회원 정보를 입력 받아서 처리할 수 있도록 한다.)

[MemberInput.java]

```
01 package verify;
02
```

```

03 import java.util.Scanner;
04
05 public class MemberInput {
06
07     public static void main(String[] args) {
08         // 객체 배열
09         MemberInfo01[] m = new MemberInfo01[5];
10         int i = 0;
11
12         // 아래에 코드를 입력하세요.
13
14     }
15 }
16
17 class MemberInfo01 {
18     private String name;
19     private int age;
20     private String email;
21     private String address;
22
23     // 아래에 코드를 입력하세요.
24
25 }

```

6.15 어노테이션(Annotation)

- 어노테이션은 메타데이터라고 볼 수 있다. 메타데이터란 컴파일 과정과 실행 과정에서 코드를 어떻게 컴파일하고 처리할것인지를 알려주는 정보이다.
- 어노테이션은 다음 세 가지 용도로 사용된다.
 - 컴파일러에게 코드 문법 에러를 체크하도록 정보를 제공
 - 소프트웨어 개발 툴이 빌드나 배치 시 코드를 자동으로 생성할 수 있도록 정보를 제공
 - 실행 시(런타임 시) 특정 기능을 실행하도록 정보를 제공

```

//기본형
@AnnotationName

```

6.15.1 어노테이션 타입 정의와 적용

- 어노테이션 타입을 정의하는 방법은 인터페이스를 정의하는 것과 유사한다.

```

public @interface AnnotationName {
    String value(); // 기본 엘리먼트 선언
    String elementName1();
    int elementName2() default 5;
}

@AnnotationName("값");
@AnnotationName(elementName1="값",elementName2=3);

```

6.15.2 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

```
// 어노테이션 선언
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
public @interface AnnotationName {
}

// 어노테이션 적용
@AnnotationName
public class ClassName {
    @AnnotationName
    private String fieldName;

    @AnnotationName // X, @Target에 CONSTRUCT가 없어 생성자는 적용 못함
    public ClassName() {}

    @AnnotationName
    public void methodName() {}
}
```

6.15.3 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

```
@Target({ElementType.TYPE, ElementType.FIELD, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface AnnotationName {
}
```

6.15.4 런타임 시 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 Constructor 배열로 리턴
Method[]	getDeclaredMethods()	메소드 정보를 Method 배열로 리턴

- 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.

[PrintAnnotation.java] 어노테이션 정의

```
01 package sec15.exam01_annotation;
02
03 import java.lang.annotation.ElementType;
04 import java.lang.annotation.Retention;
05 import java.lang.annotation.RetentionPolicy;
06 import java.lang.annotation.Target;
07
08 @Target({ElementType.METHOD})
09 @Retention(RetentionPolicy.RUNTIME)
10 public @interface PrintAnnotation {
11     String value() default "-";
12 }
```

```
12         int number() default 15;
13     }
```

[Service.java] 어노테이션 적용 클래스

```
01 package sec15.exam01_annotation;
02
03 public class Service {
04     @PrintAnnotation
05     public void method1() {
06         System.out.println("실행 내용1");
07     }
08
09     @PrintAnnotation("★")
10     public void method2() {
11         System.out.println("실행 내용2");
12     }
13
14     @PrintAnnotation(value="#", number=20)
15     public void method3() {
16         System.out.println("실행 내용3");
17     }
18 }
```

[PrintAnnotationExample.java]

```
01 package sec15.exam01_annotation;
02
03 import java.lang.reflect.Method;
04
05 public class PrintAnnotationExample {
06     public static void main(String[] args) {
07         // Service 클래스로부터 메소드 정보를 얻음
08         Method[] declaredMethods = Service.class.getDeclaredMethods();
09
10         // Method 객체를 하나씩 처리
11         for (Method method : declaredMethods) {
12             // PrintAnnotation이 적용되었는지 확인
13             if (method.isAnnotationPresent(PrintAnnotation.class)) {
14                 // PrintAnnotation 객체 얻기
15                 PrintAnnotation printAnnotation =
16                     method.getAnnotation(PrintAnnotation.class);
17
18                 // 메소드 이름 출력
19                 System.out.println("[ " + method.getName() + " ] ");
20                 // 구분선 출력
21                 for (int i = 0; i < printAnnotation.number(); i++) {
22                     System.out.print(printAnnotation.value());
23                 }
24                 System.out.println();
25
26                 try {
27                     // 메소드 호출
28                     method.invoke(new Service());
29                 } catch (Exception e) {
30                     }
31                 System.out.println();
32             }
33         }
34     }
35 }
```


[과제] 확인문제

1. 객체와 클래스에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 클래스는 객체를 생성하기 위한 설계도(청사진)와 같은 것이다.
- (2) new 연산자로 클래스의 생성자를 호출함으로써 객체가 생성된다.
- (3) 하나의 클래스로 하나의 객체만 생성할 수 있다.
- (4) 객체는 클래스의 인스턴스이다.

2. 클래스의 구성 멤버가 아닌 것은 무엇입니까?

- (1) 필드(field)
- (2) 생성자(constructor)
- (3) 메소드(method)
- (4) 로컬 변수(local variable)

3. 필드, 생성자, 메소드에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 필드는 객체의 데이터를 저장한다.
- (2) 생성자는 객체의 초기화를 담당한다.
- (3) 메소드는 객체의 동작 부분으로, 실행 코드를 가지고 있는 블록이다.
- (4) 클래스는 반드시 필드와 메소드를 가져야 한다.

4. 필드에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 필드는 메소드에서 사용할 수 있다.
- (2) 인스턴스 필드 초기화는 생성자에서 할 수 있다.
- (3) 필드는 반드시 생성자 선언 전에 선언되어야 한다.
- (4) 필드는 초기값을 주지 않더라도 기본값으로 자동 초기화된다.

5. 생성자에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 객체를 생성하려면 생성자 호출이 반드시 필요한 것은 아니다.
- (2) 생성자는 다른 생성자를 호출하기 위해 this()를 사용할 수 있다.
- (3) 생성자가 선언되지 않으면 컴파일러가 기본 생성자를 추가한다.
- (4) 외부에서 객체를 생성할 수 없도록 생성자에 private 접근 제한자를 붙일 수 있다.

6. 메소드에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 리턴값이 없는 메소드는 리턴 타입을 void로 해야 한다.
- (2) 리턴 타입이 있는 메소드는 리턴값을 지정하기 위해 반드시 return문이 있어야 한다.
- (3) 매개값의 수를 모를 경우 "..."를 이용해서 매개 변수를 선언할 수 있다.
- (4) 메소드의 이름은 중복해서 선언할 수 없다.

7. 메소드 오버로딩에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 동일한 이름의 메소드를 여러 개 선언하는 것을 말한다.
- (2) 반드시 리턴 타입이 달라야 한다.
- (3) 매개 변수의 타입, 수, 순서를 다르게 선언해야 한다.
- (4) 매개값의 타입 및 수에 따라 호출될 메소드가 선택된다.

8. 인스턴스 멤버와 정적 멤버에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 정적 멤버는 static으로 선언된 필드와 메소드를 말한다.
- (2) 인스턴스 필드는 생성자 및 정적 블록에서 초기화될 수 있다.
- (3) 정적 필드와 정적 메소드는 객체 생성 없이 클래스를 통해 접근할 수 있다.
- (4) 인스턴스 필드와 메소드는 객체를 생성하고 사용해야 한다.

9. final 필드와 상수(static final)에 대한 설명으로 틀린 것은 무엇입니까?

- (1) final 필드와 상수는 초기값이 저장되면 값을 변경할 수 없다.
- (2) final 필드와 상수는 생성자에서 초기화될 수 있다.
- (3) 상수의 이름은 대문자로 작성하는 것이 관례이다.
- (4) 상수는 객체 생성 없이 클래스를 통해 사용할 수 있다.

10. 패키지에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 패키지는 클래스들을 그룹화시키는 기능을 한다.
- (2) 클래스가 패키지에 소속되려면 패키지 선언을 반드시 해야 한다.
- (3) import문은 다른 패키지의 클래스를 사용할 때 필요하다.
- (4) mycompany 패키지에 소속된 클래스는 yourcompany에 옮겨 놓아도 동작한다.

11. 접근 제한에 대한 설명으로 틀린 것은 무엇입니까?

- (1) 접근 제한자는 클래스, 필드, 생성자, 메소드의 사용을 제한한다.
- (2) public 접근 제한은 아무런 제한 없이 해당 요소를 사용할 수 있게 한다.
- (3) default 접근 제한은 해당 클래스 내부에서만 사용을 허가한다.
- (4) 외부에서 접근하지 못하도록 하려면 private 접근 제한을 해야 한다.

12. 다음 클래스에서 해당 멤버가 필드, 생성자, 메소드 중 어떤 것인지 빈칸을 채우세요.

```
public class Member {  
    private String name; // (    #1    )  
    public Member(String name) { ... } // (    #2    )  
    public void setName(String name) { ... } // (    #3    )  
}
```

13. 현실 세계의 회원을 Member 클래스로 모델링하려고 합니다. 회원의 데이터로는 이름, 나이

다, 비밀번호, 나이가 있습니다. 이 데이터들을 가지는 Member 클래스를 선언해보세요.

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

```
package verify.exam13;

public class Member {
    // 작성 위치
}
```

14. 위에서 작성한 Member 클래스에 생성자를 추가하려고 합니다. 다음과 같이 Member 객체를 생성할 때 name 필드와 id 필드를 외부에서 받은 값으로 초기화하려면 생성자를 어떻게 선언해야 하나요?

```
Member user1 = new Member("홍길동", "hong");
Member user2 = new Member("감자바", "java");
```

[Member.java]

```
package verify.exam14;

public class Member {
    // 작성 위치
}
```

15. MemberService 클래스에 login() 메소드와 logout() 메소드를 선언하려고 합니다. login() 메소드를 호출할 때에는 매개값으로 id와 password를 제공하고, logout() 메소드는 id만 매개값으로 제공합니다. MemberService 클래스와 login(), logout() 메소드를 선언해보세요.

- (1) login() 메소드는 매개값 id가 "hong", 매개값 password가 "12345"일 경우에만 true로 리턴하고 그 이외의 값일 경우에는 false를 리턴하도록 하세요.
- (2) logout() 메소드의 내용은 "로그아웃 되었습니다."가 출력되도록 하세요.

리턴 타입	메소드 이름	매개 변수(타입)
boolean	login	id(String), password(String)
void	logout	id(String)

[MemberService.java]

```
01 package verify.exam15;
02
03 public class MemberService {
04     // 작성 위치
05 }
```

```
06 }
```

[MemberServiceExample.java]

```
01 package verify.exam15;
02
03 public class MemberServiceExample {
04     public static void main(String[] args) {
05         MemberService memberService = new MemberService();
06         boolean result = memberService.login("hong", "12345");
07         if(result) {
08             System.out.println("로그인 되었습니다.");
09             memberService.logout("hong");
10         } else {
11             System.out.println("id 또는 password가 올바르지 않습니다.");
12         }
13     }
14 }
15
16 // 실행 결과
17 // 로그인 되었습니다.
18 // 로그아웃 되었습니다.
```

16. PrinterExample 클래스에서 Printer 객체를 생성하고 println() 메소드를 호출해서 매개값을 콘솔에 출력하려고 합니다. println() 메소드의 매개값으로 int, boolean, double, String 값을 줄 수 있습니다. Printer 클래스에서 println() 메소드를 선언해보세요.

[Printer.java]

```
01 package verify.exam16;
02
03 public class Printer {
04     // 작성 위치
05
06 }
```

[PrinterExample.java]

```
01 package verify.exam16;
02
03 public class PrinterExample {
04     public static void main(String[] args) {
05         Printer printer = new Printer();
06         printer.println(10);
07         printer.println(true);
08         printer.println(5.7);
09         printer.println("홍길동");
10     }
11 }
12
13 // 실행 결과
14 // 10
15 // true
16 // 5.7
17 // 홍길동
```

17. 16번 문제에서는 Printer 객체를 생성하고 println() 메소드를 생성했습니다. Printer 객체를 생성하지 않고 PrinterExample 클래스에서 다음과 같이 호출하려면 Printer 클래스를 어떻게 수정하면 될까요?

[Printer.java]

```
01 package verify.exam17;
02
03 public class Printer {
04     // 작성 위치
05
06 }
```

[PrinterExample.java]

```
01 package verify.exam17;
02
03 public class PrinterExample {
04     public static void main(String[] args) {
05         Printer.println(10);
06         Printer.println(true);
07         Printer.println(5.7);
08         Printer.println("홍길동");
09     }
10 }
11
12 // 실행 결과
13 // 10
14 // true
15 // 5.7
16 // 홍길동
```

18. ShopService 객체를 싱글폰으로 만들고 싶습니다. ShopServiceExample 클래스에서 ShopService의 getInstance() 메소드로 싱글폰을 얻을 수 있도록 ShopService 클래스를 작성해보세요.

[ShopService.java]

```
01 package verify.exam18;
02
03 public class ShopService {
04     // 작성 위치
05
06 }
```

[ShopServiceExample.java]

```
01 package verify.exam18;
02
03 public class ShopServiceExample {
04     public static void main(String[] args) {
05         ShopService obj1 = ShopService.getInstance();
06         ShopService obj2 = ShopService.getInstance();
07
08         if(obj1 == obj2) {
09             System.out.println("같은 ShopService 객체 입니다.");
10         } else {
11             System.out.println("다른 ShopService 객체 입니다.");
12         }
13     }
14 }
```

```

12         }
13     }
14 }
15
16 // 실행 결과
17 // 같은 ShopService 객체 입니다.

```

19. 은행 계좌 객체인 Account 객체는 잔고(balance) 필드를 가지고 있습니다. balance 필드는 음수값이 될 수 없고, 최대 백만 원까지만 저장할 수 있습니다. 외부에서 balance 필드를 마음대로 변경하지 못하도록 하고, $0 \leq \text{balance} \leq 1,000,000$ 범위의 값만 가질 수 있도록 Account 클래스를 작성해 보세요.

- (1) Setter와 Getter를 이용하세요.
- (2) 0과 1,000,000은 MIN_BALANCE와 MAX_BALANCE 상수를 선언해서 이용하세요.
- (3) Setter의 매개값이 음수이거나 백만 원을 초과하면 현재 balance 값을 유지하세요.

[Account.java]

```

01 package verify.exam19;
02
03 public class Account {
04     // 작성 위치
05
06 }

```

[AccountExample.java]

```

01 package verify.exam19;
02
03 public class AccountExample {
04     public static void main(String[] args) {
05         Account account = new Account();
06
07         account.setBalance(10000);
08         System.out.println("현재 잔고: " + account.getBalance());
09
10         account.setBalance(-100);
11         System.out.println("현재 잔고: " + account.getBalance());
12
13         account.setBalance(2000000);
14         System.out.println("현재 잔고: " + account.getBalance());
15
16         account.setBalance(300000);
17         System.out.println("현재 잔고: " + account.getBalance());
18     }
19 }
20
21 // 실행 결과
22 // 현재 잔고: 10000
23 // 현재 잔고: 10000
24 // 현재 잔고: 10000
25 // 현재 잔고: 300000

```