

# 자바 프로그래밍

## 제6장 클래스, 메소드 심층 탐구

# 학습 내용

## 학습목차

- 01 접근 제어
- 02 접근자와 설정자
  - LAB 안전한 배열 만들기
- 03 생성자
  - LAB Television 생성자
  - LAB 상자를 나타내는 Box 클래스 작성
- 04 생성자 오버로딩
  - LAB 날짜를 나타내는 Date 클래스 작성하기
  - LAB 시간을 나타내는 Time 클래스 작성하기
  - LAB 원을 나타내는 Circle 클래스 작성하기
- 05 다른 필드 초기화 방법
- 06 메소드로 객체 전달하고 반환하기
  - LAB 배열에 저장된 값의 평균 구하기
  - LAB 같은 크기의 Box인지 확인하기
- 07 정적 멤버
  - LAB 직원 클래스 작성하기
- 08 내장 클래스
  - LAB 내부 클래스의 사용예

클래스를 사용하기가  
조금 복잡하네요!

네. 먼저 클래스와 객체의  
개념을 확실히 이해하는 것이  
중요합니다. 다른 것들은 차츰  
익숙해질 것입니다.



# 접근 제어

- 클래스 안의 변수나 메소드를 누구나 사용할 수 있게 하면 어떻게 될까? 많은 문제가 발생할 것이다.

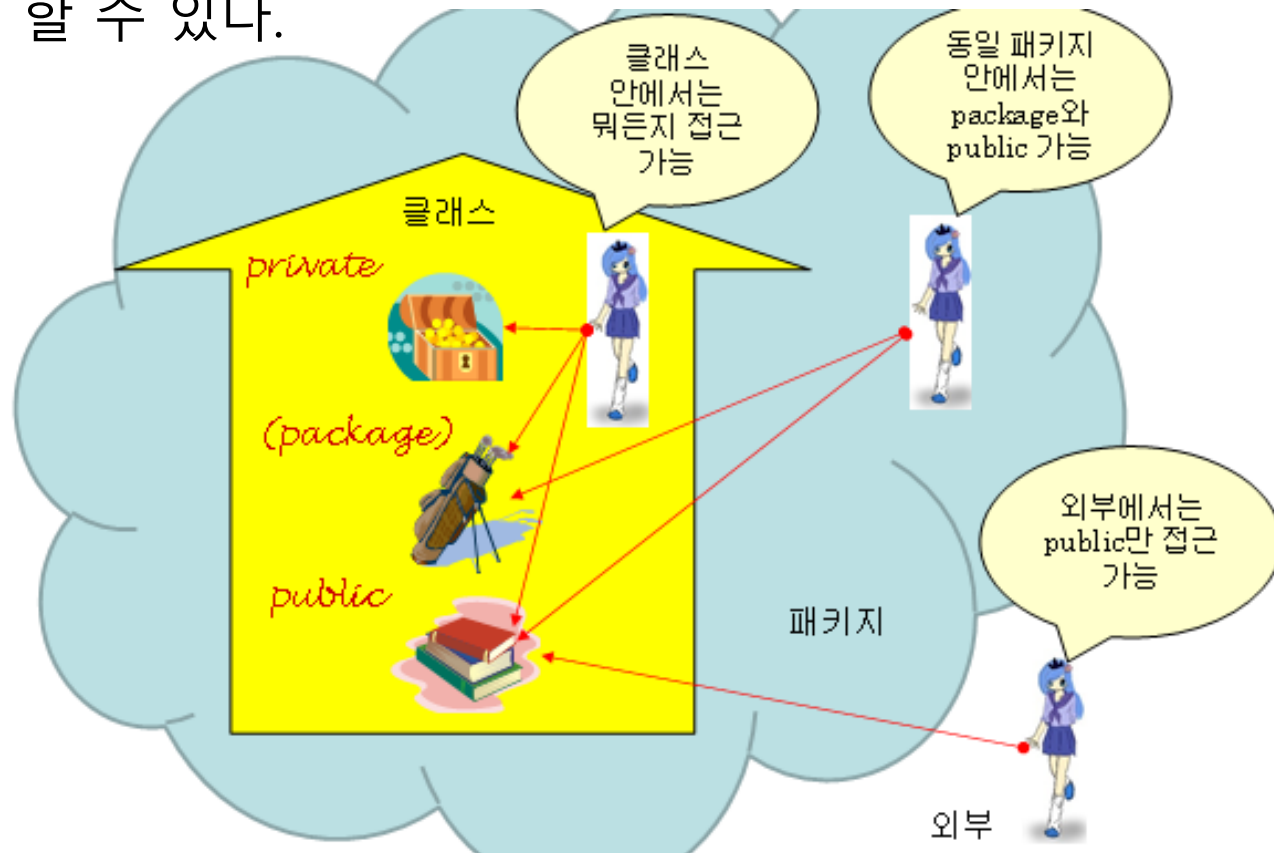
```
// 작성자: kim
public class EmployeeTest {
    public static void main(String[] args) {
        Employee e = new Employee();
        e.regNumber = 20010101;
        e.regNumber *= 2;
        Company c = new Company();
        c.hire(e);
        System.out.println(e.regNumber);
    }
}
```

```
// 작성자: park
public class Employee {
    String name;    // 이름
    int regNumber; // 주민등록번호
    ...
}
```

```
// 작성자: lee
public class Company {
    Employee e1;
    void hire(Employee e) {
        e1 = e;
        e1.regNumber = 100;
    }
    ...
}
```

# 접근 제어

- 접근 제어(access control): 다른 클래스가 특정한 필드나 메소드에 접근하는 것을 제어하는 것
  - 올바르게 정의되고 권한이 있는 메소드만 데이터를 사용할 수 있게 할 수 있다.



# 접근 제어

- 클래스의 멤버(필드, 메소드 등)를 선언할 때, 접근 지정자를 사용하여 멤버별 접근 제어 가능

접근 지정자	클래스	패키지	자식 클래스	전체 세계
public	O	O	O	O
protected	O	O	O	X
없음(디폴트)	O	O	X	X
private	O	X	X	X

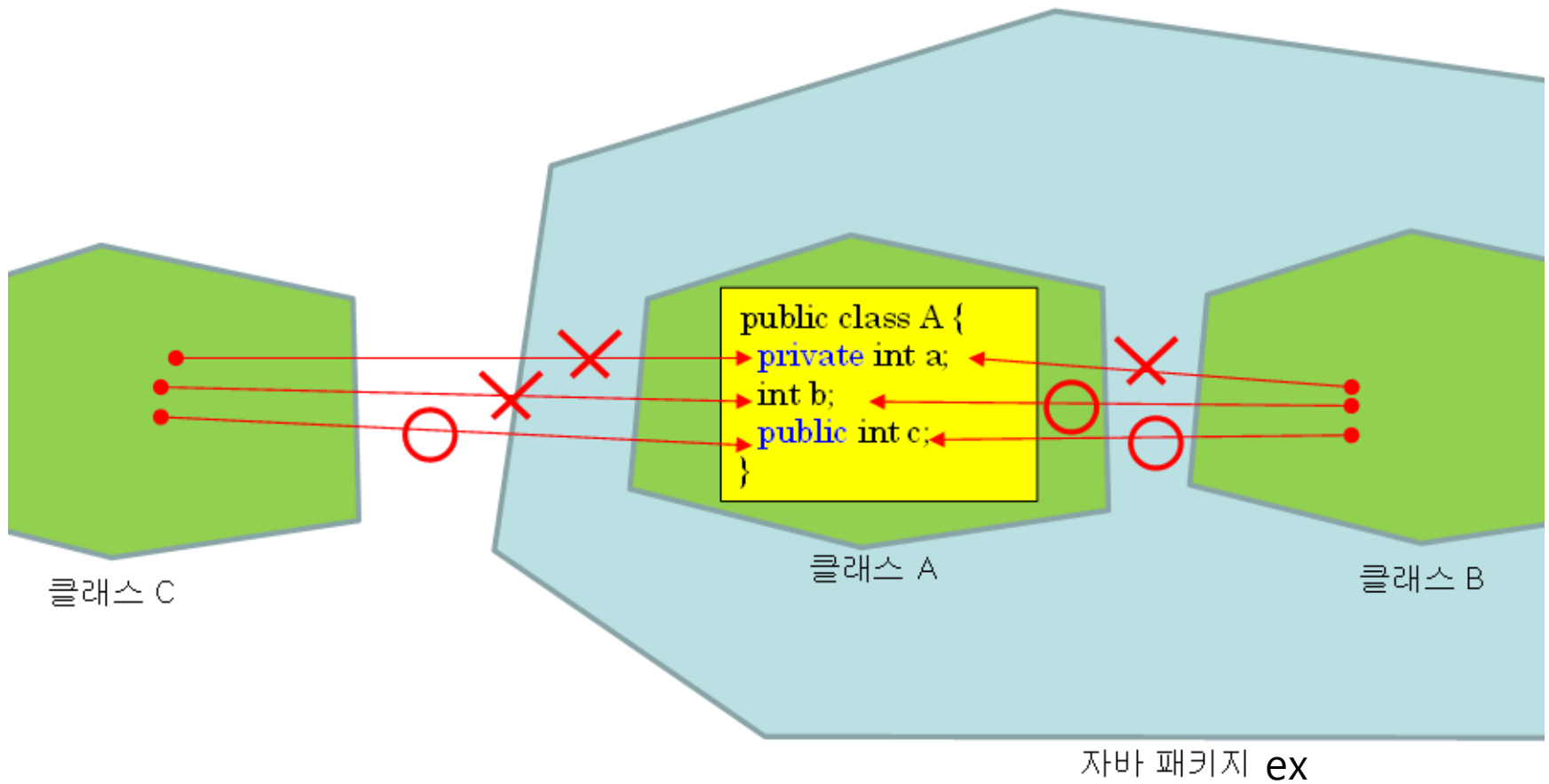
protected  
는 7장에  
서 다룸

## 예: 접근 지정자

```
package ex;
public class A {
    private int a;      // 전용 : 클래스 A 내에서 접근 가능
    int b;              // 디폴트 : 패키지 ex 내에서 접근 가능
    public int c;       // 공용 : 어디서나 접근 가능
}
```

```
package ex;
public class B {
    public static void main(String[] args) {
        A obj = new A();
        obj.a = 10;      // 에러! 클래스 B는 클래스 A의 전용 멤버 접근 불가
        obj.b = 20;      // 같은 패키지이므로 B는 A의 디폴트 멤버 접근 가능
        obj.c = 30;      // A의 공용 멤버는 어디서나 접근할 수 있음
    }
}
```

## 예: 접근 지정자



```
public class A {  
    private int a;  
    public int x;  
  
    public void print() {  
        System.out.println(a + x);  
    }  
    private void add() {  
        a = a + x;  
    }  
    public void set() {  
        a = 1;  
        x = 1;  
        add();  
        print();  
    }  
}
```

```
public class B {  
    public static void main(String[] args) {  
        A obj = new A();  
        obj.x = 3;  
        obj.set();  
        obj.print();  
    }  
}
```

(2) 클래스 A 외부에서는 public 멤버인 x, print(), set()만 접근 가능하다.

(1) 클래스 A의 멤버는 a, x, print(), add(), set()이며, 클래스 A 내에서는 모든 멤버에 접근 가능하다.



# 예: 접근 지정자

```
package ex;
public class A {
    private int a; // 전용
    int b;
    public int c;

    public void set() { // 공용
        a = 10; // 전용 필드 접근
        b = 20;
        c = 30;
        add(); // 전용 메소드 호출
    }

    private void add() { // 전용
        a++; // 전용 필드 접근
    }
}
```

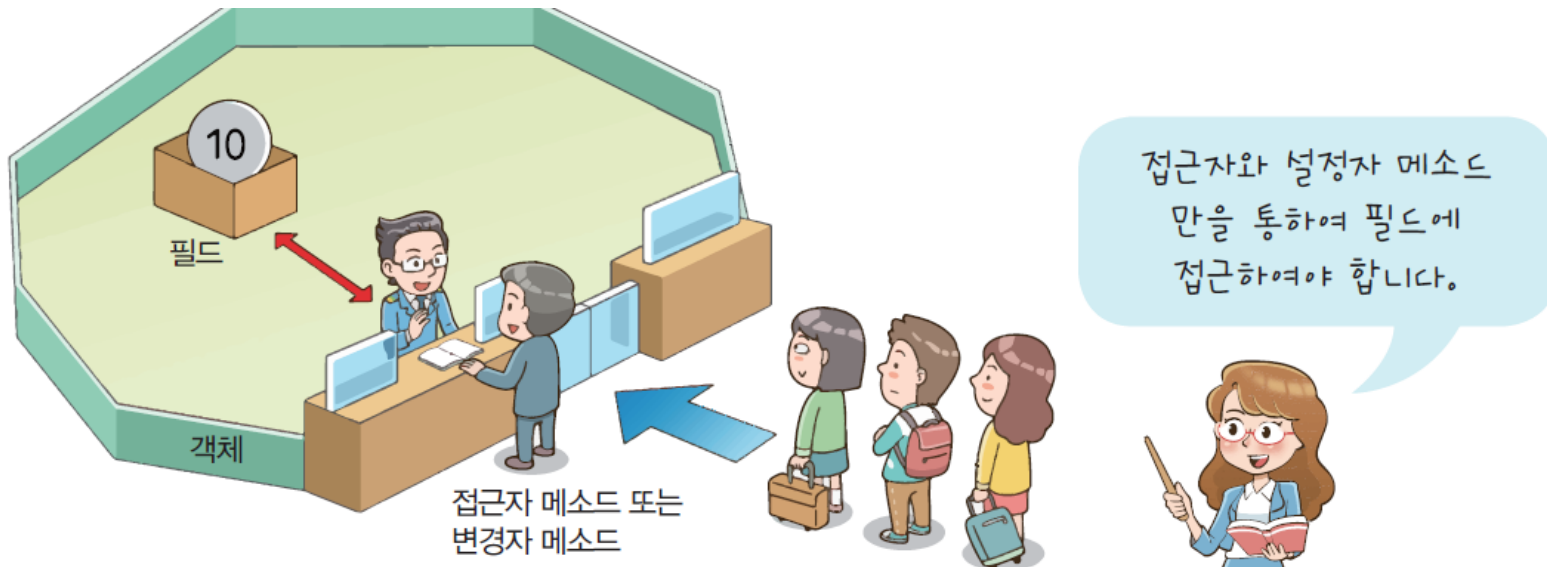
```
package ex;
public class B {
    public static void main(String[] args) {
        A obj = new A();
        obj.a = 10; // 에러!
        obj.b = 20;
        obj.c = 20;
        obj.add(); // 에러!
        obj.set();
    }
}
```

공용 메소드는 클래스 외부에서도 사용할 수 있도록 기능을 제공. service method 역할

전용 메소드는 클래스 내부의 다른 메소드를 보조. support method 역할

# 접근자와 설정자

- 정보 은닉(information hiding)
  - 구현의 세부 사항을 클래스 안에 감춤
  - 이를 위해 클래스 안의 데이터를 private로 선언하여 외부에서 마음대로 변경하지 못하게 함
  - 대신, public 메소드를 제공하고, 이를 호출하여 데이터를 이용하도록 함



# 접근자와 설정자

- 접근자(accessor; getter)
  - 필드의 값을 반환하는 메소드
  - 메소드 이름은 일반적으로 get필드명()
- 설정자(mutator; setter)
  - 필드의 값을 설정하는 메소드
  - 메소드 이름은 일반적으로 set필드명()
- 예) 필드명이 age이면
  - 접근자는 getAge()
  - 설정자는 setAge()

# 예: 텔레비전

- 접근자/설정자를 두지 않은 예

```
public class Television {  
    int channel;  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Television tv = new Television();  
        tv.channel = 11;           // tv의 채널을 11로 지정  
        int c = tv.channel;      // tv의 채널을 알아냄  
        System.out.println(c);  
    }  
}
```

# private 멤버 접근 오류

- channel 필드를 private로 지정해보자.

```
public class Television {  
    private int channel;  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Television tv = new Television();  
        tv.channel = 11;           // 에러  
        int c = tv.channel;      // 에러  
        System.out.println(c);  
    }  
}
```

# 해결 방법: getter와 setter

```
public class Television {  
    private int channel;  
  
    public int getChannel() {                // getter  
        return channel;  
    }  
    public void setChannel(int newChannel) {    // setter  
        channel = newChannel;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Television tv = new Television();  
        _____;                // tv.channel = 11; 대신  
        int c = _____;        // int c = tv.channel; 대신  
        System.out.println(c);  
    }  
}
```

# 해결 방법: getter와 setter

```
public class Television {  
    private int channel;  
  
    public int getChannel() {                // getter  
        return channel;  
    }  
    public void setChannel(int newChannel) { // setter  
        channel = newChannel;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Television tv = new Television();  
        tv.setChannel(11);                // tv.channel = 11; 대신  
        int c = tv.getChannel();        // int c = tv.channel; 대신  
        System.out.println(c);  
    }  
}
```

## 예: 은행계좌

```
public class Account {  
    private String name;  
    private int balance;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String newName) {  
        name = newName;  
    }  
    public int getBalance() {  
        return balance;  
    }  
    public void setBalance(int newBalance) {  
        balance = newBalance;  
    }  
}
```



# 현재 객체를 나타내는 this

- 메소드에서 this는 그 메소드를 실행하고 있는 현재 객체를 나타낸다.

```
public class Account {  
    private String name;  
    private int balance;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getBalance() {  
        return balance;  
    }  
    public void setBalance(int balance) {  
        this.balance = balance;  
    }  
}
```

```
public class Account {  
    private String name;  
    private int balance;  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public int getBalance() {  
        return balance;  
    }  
    public void setBalance(int balance) {  
        this.balance = balance;  
    }  
}
```

이클립스의 Source > Generate  
Getters and Setters 메뉴를 이  
용하여 정의할 수도 있음

```
public class AccountTest {  
    public static void main(String[] args) {  
        Account obj = new Account();  
        obj.setName("Tom");  
        obj.setBalance(100000);  
        System.out.println("이름은 " + obj.getName()  
            + " 통장 잔고는 " + obj.getBalance());  
    }  
}
```

이름은 Tom 통장 잔고는 100000

# 접근자와 설정자는 왜 사용하는가?

- 접근자와 설정자를 사용해야만 구현 세부사항을 숨길 수 있어서 나중에 클래스를 업그레이드할 때 편하다.
- 설정자에서 매개 변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있다.
- 필요할 때마다 필드값을 계산하여 반환할 수 있다.
- 읽기만 가능한 필드로 만들고 싶으면 그 필드에 설정자는 없이 접근자만 제공하면 된다.

## 예: 접근자와 설정자의 장점

- 접근자/설정자를 사용하지 않고 필드에 직접 접근하는 경우

```
public class Student {  
    int age;  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Student s = new Student();  
        s.age = -10;    // 학생의 나이가 -10?  
    }  
}
```

## 예: 접근자와 설정자의 장점

- 설정자는 필드의 값을 변경하려는 외부의 시도를 주의 깊게 검사할 수 있다.

```
public class Student {  
    private int age;  
    public void setAge(int age) {  
        if (age < 0) this.age = 0;  
        else this.age = age;  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Student s = new Student();  
        s.setAge(-10);  
    }  
}
```

# 생성자(constructor)

- 생성자는 객체가 **생성될 때** 필드에 초기값을 제공하고 필요한 **초기화 절차**를 실행하는 특수한 메소드
  - 이름: 클래스 이름과 동일
  - 리턴 타입: 없음(void도 아님)

형식

```
public class Car {  
    Car() {  
        ...  
    }  
}
```

클래스 이름과 동일한 메소드가 바로 생성자입니다. 여기서 객체의 초기화를 담당합니다.



# 생성자의 예

```
public class MyCounter {  
    private int counter;
```

```
    public MyCounter() {  
        counter = 1;  
    }
```

// 생성자

```
    public int getCounter() {  
        return counter;  
    }
```

```
}
```

```
public class MyCounterTest {  
    public static void main(String[] args) {  
        MyCounter obj1 = new MyCounter();  
        MyCounter obj2 = new MyCounter();  
        System.out.println(obj1.getCounter());  
        System.out.println(obj2.getCounter());  
    }  
}
```

// 생성자 호출  
// 생성자 호출

1  
1

# 매개변수를 가지는 생성자

```
public class MyCounter {  
    private int counter;  
  
    public MyCounter(int value) {  
        counter = value;  
    }  
  
    public int getCounter() {  
        return counter;  
    }  
}
```

```
public class MyCounterTest {  
    public static void main(String[] args) {  
        MyCounter obj1 = new MyCounter(100);  
        MyCounter obj2 = new MyCounter(200);  
        System.out.println(obj1.getCounter());  
        System.out.println(obj2.getCounter());  
    }  
}
```

100  
200



## 예: 자동차 클래스

```
public class Car {  
    private String color;  
    private int speed;  
  
    public Car() {                // 생성자  
        color = "red";  
        speed = 100;  
    }  
    ...  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car c1 = new Car();      // 생성자 호출  
        ...  
    }  
}
```

# 생성자 메소드 오버로딩

```
public class Car {  
    private String color;  
    private int speed;  
  
    public Car(String c, int s) {    // 생성자 1 – 매개변수가 있는 생성자  
        color = c;  
        speed = s;  
    }  
  
    public Car() {                  // 생성자 2  
        color = "red";  
        speed = 0;  
    }  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car c1 = new Car("blue", 100);    // 생성자 1 호출  
        Car c2 = new Car();              // 생성자 2 호출  
    }  
}
```

# 기본 생성자(default constructor)

- 클래스에서 생성자를 하나도 정의하지 않는 경우
  - 메소드의 몸체 부분이 비어있는 기본 생성자가 자동적으로 만들어진다.

```
public class Car {  
    private String color;  
    private int speed;  
}
```



```
public Car() {  
}
```

Car 클래스에 이와 같은  
기본 생성자가 있는 것임

```
public class CarTest {  
    public static void main(String[] args) {  
        Car c1 = new Car();    // 디폴트 생성자 호출  
    }  
}
```

# 주의할 점

- 생성자를 하나라도 정의하면 기본 생성자는 자동적으로 만들어지지 않는다.

```
public class Car {  
    private String color;  
    private int speed;  
  
    public Car(String c, int s)  
        color = c;  
        speed = s;  
    }  
}
```



```
public Car() {  
}
```

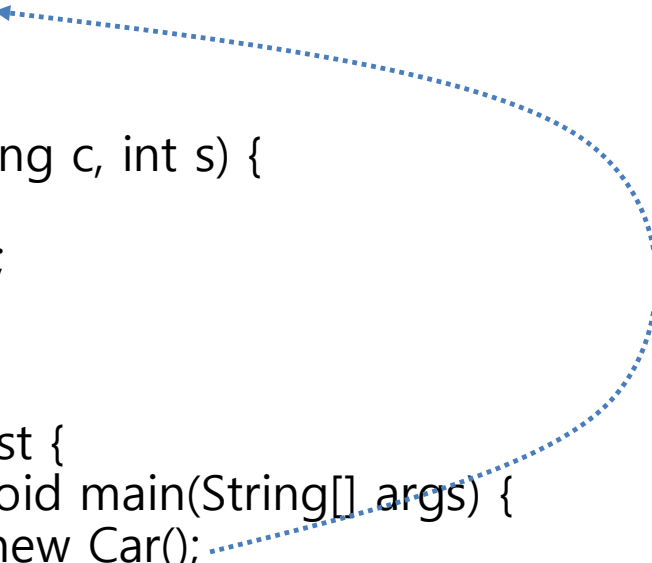
```
public class CarTest {  
    public static void main(String[] args) {  
        Car c1 = new Car();    // 에러!!  
    }  
}
```

"The constructor Car()  
is undefined"

# 앞의 에러 해결책

- 명시적으로 기본 생성자를 정의한다.

```
public class Car {  
    private String color;  
    private int speed;  
  
    public Car() {  
    }  
  
    public Car(String c, int s) {  
        color = c;  
        speed = s;  
    }  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car c1 = new Car();  
    }  
}
```



# this로 현재 객체 나타내기


- 메소드나 생성자에서 this는 현재 객체를 나타낸다.

```
public class Point {  
    private int x;  
    private int y;  
  
    // 생성자  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

# 생성자 this()

- 생성자는 다른 생성자를 호출할 수 있다.
  - 단, 생성자 이름이 아니라 **this(...)** 라는 형식으로 호출

```
public class Car {  
    private String color;  
    private int speed;  
  
    public Car(String c, int s) {    // 생성자 1  
        color = c;  
        speed = s;  
    }  
  
    public Car() {                  // 생성자 2  
        this("red", 100);          // 생성자 1을 호출한다.  
    }  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car myCar = new Car();  
    }  
}
```



# 생성자 this()

```
public class Rectangle {  
    private int x, y;  
    private int width, height;  
  
    public Rectangle() {  
        this(0, 0, 1, 1);  
    }  
  
    public Rectangle(int width, int height) {  
        this(0, 0, width, height);  
    }  
  
    public Rectangle(int x, int y, int width, int height) {  
        this.x = x;  
        this.y = y;  
        this.width = width;  
        this.height = height;  
    }  
    // ...  
}
```

```
public Rectangle() {  
    x = 0;  
    y = 0;  
    width = 1;  
    height = 1;  
}
```

}  
와 같은 역할을 함



# 생성자 호출 에러 예

- 생성자는 아무 데서나 호출할 수는 없다.
  - this() 호출은 생성자 메소드의 첫 문장이어야 함

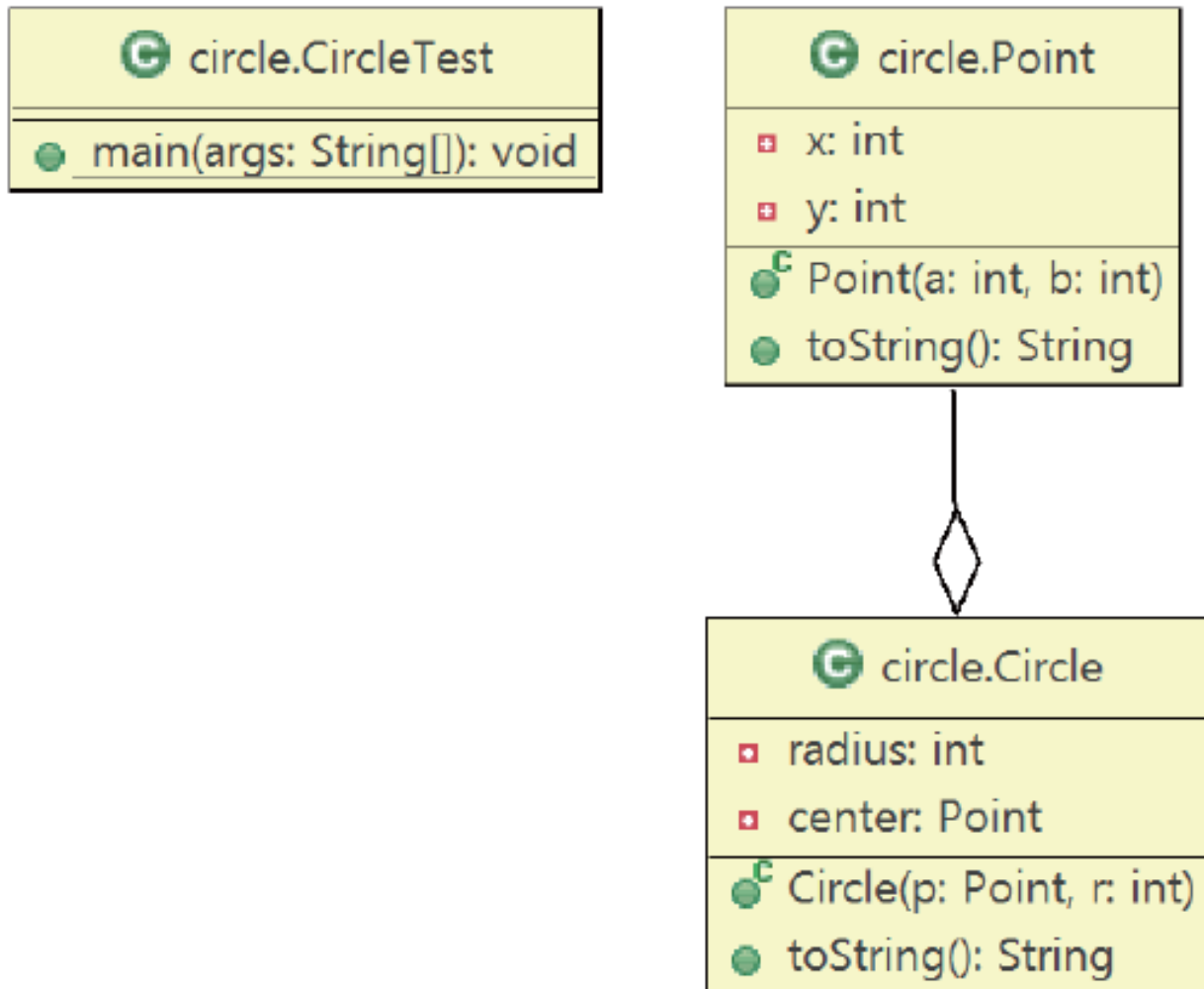
```
public class Car {  
    ...  
    public Car() {  
    }  
    public Car(String c) {  
        Car();                // 에러  
        color = c;  
    }  
    public Car(String c, int s) {  
        speed = s;  
        this();                // 에러  
    }  
}  
  
public class CarTest {  
    public static void main(String[] args) {  
        Car c1 = new Car();  
        Car();                // 에러  
    }  
}
```

"The method Car() is undefined for the type Car"

"Constructor call must be the first statement in a constructor"

"The method Car() is undefined for the type Main"

# LAB: 원을 나타내는 Circle 클래스



```
public class Circle {  
    private int radius;  
    private Point center;  
  
    public Circle(Point center, int radius) {  
        this.radius = radius;  
        this.center = center;  
    }  
    @Override  
    public String toString() {  
        return "Circle [radius=" + radius + ", center=" + center + "];"  
    }  
}
```

```
public class Point {  
    private int x, y;  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    @Override  
    public String toString() {  
        return "Point [x=" + x + ", y=" + y + "];"  
    }  
}
```

```
public class CircleTest {  
    public static void main(String[] args) {  
        Point p = new Point(25, 78);  
        Circle c = new Circle(p, 10);  
        System.out.println(c);  
    }  
}
```

Circle [radius=10, center=Point [x=25, y=78]]

# 필드 초기화 방법

- 생성자에서 초기화하는 방법
- 필드 선언시 초기화하는 방법
  - 초기값이 미리 알려져 있고, 한 줄에 적을 수 있는 경우

```
public class Hotel {  
    private int capacity = 10;           // 10으로 초기화  
    private boolean full = false;       // false로 초기화  
    ...  
}
```

- 인스턴스 초기화 블록을 사용하는 방법
  - 어떤 생성자가 선택되든지 공통의 초기화 코드를 실행하고자 할 때
  - 무명 클래스 초기화할 때

무명 클래스는 9장에서 다룸

# 필드 초기화 방법

- 인스턴스 초기화 블록(instance initializer block)을 사용한 필드 초기화
  - 인스턴스 초기화 블록은 모든 생성자 앞부분에 복사된다.

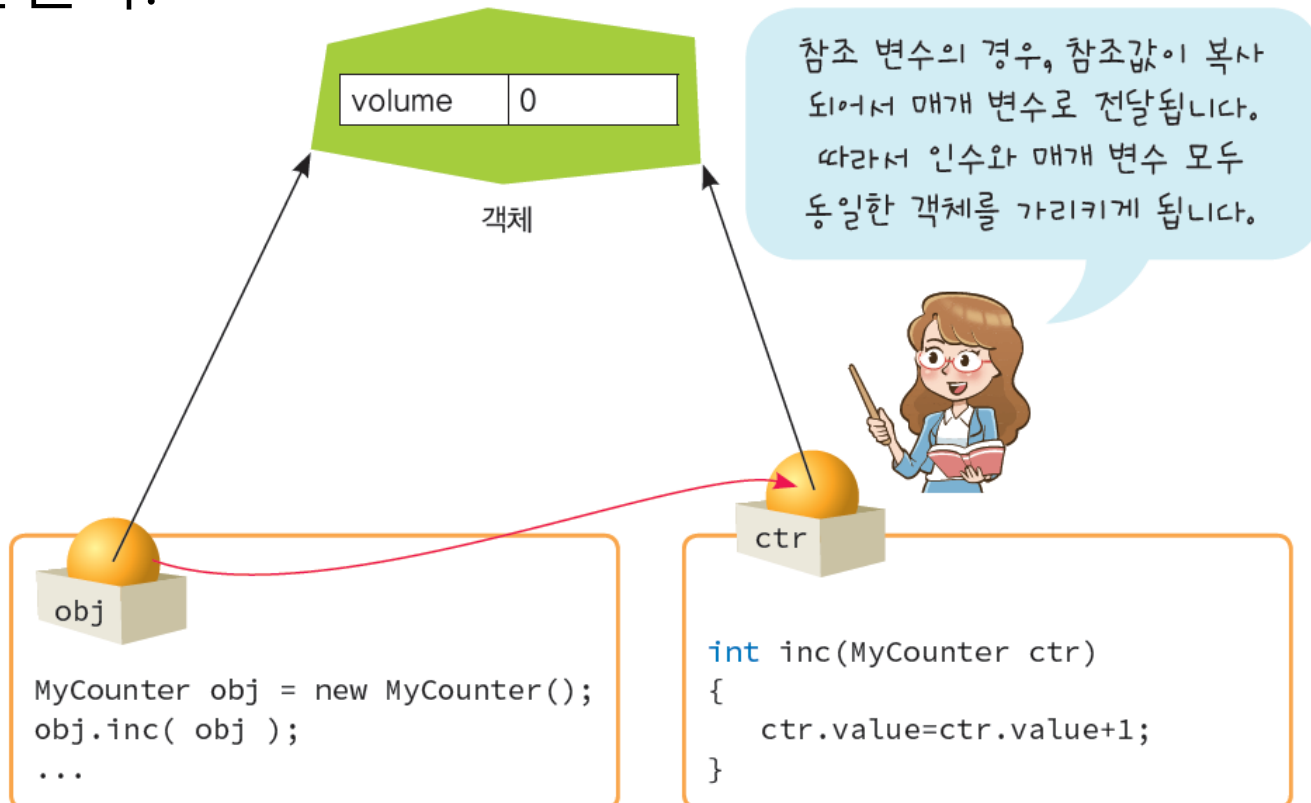
```
public class Car {  
    private int speed;  
  
    public Car() {  
        System.out.println("속도는 " + speed);  
    }  
  
    // 인스턴스 초기화 블록  
    {  
        speed = 100;  
    }  
    ...  
}
```

# 메소드로 객체 전달하고 반환하기

- 매개변수가 객체인 경우
- 매개변수가 배열인 경우
- 객체를 리턴하는 메소드

# 매개변수가 객체인 경우

- 메소드의 매개변수가 클래스 타입인 경우, 객체가 복사되어 전달되는 것이 아니고 객체의 참조값이 복사되어 전달된다.



# 예: 객체 매개변수

```
public class Point {  
    private int x;  
    public Point(int x) {  
        this.x = x;  
    }  
    public void switch(Point p) {  
        int tmp = x;  
        x = x.p;  
        x.p = tmp;  
    }  
    public int getX() {  
        return x;  
    }  
}
```

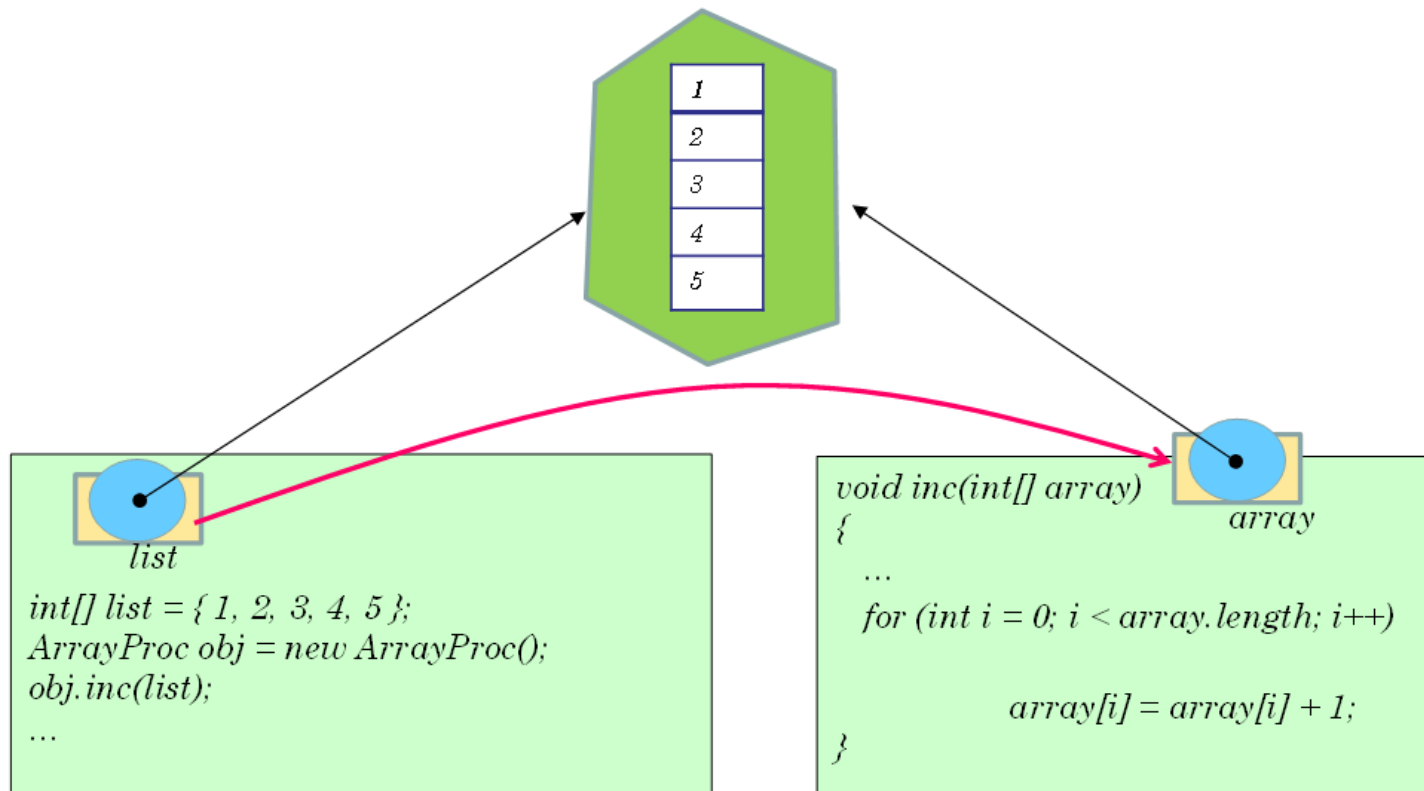
3  
5  
5  
3

```
public class Main {  
    public static void main(String[] args) {  
        Point p1 = new Point(3);  
        Point p2 = new Point(5);  
  
        System.out.println(p1.getX());  
        System.out.println(p2.getX());  
  
        p1.switch(p2);  
  
        System.out.println(p1.getX());  
        System.out.println(p2.getX());  
    }  
}
```



# 매개변수가 배열인 경우

- 배열도 객체이기 때문에 메소드에 배열 매개변수를 전달하는 것은 배열 참조 변수를 복사하는 것이다.



# 예: 배열에 저장된 값의 평균 구하기

- 사용자로부터 값을 받아서 배열에 채운 후에 배열에 저장된 모든 값의 평균을 구하여 출력하는 프로그램을 작성하여 보자.



```
성적을 입력하시오:10
성적을 입력하시오:20
성적을 입력하시오:30
성적을 입력하시오:40
성적을 입력하시오:50
평균은 = 30.0
```

```
public class ArrayProc {
    public void getValues(int[] array) {
        Scanner scan = new Scanner(System.in);
        for (int i = 0; i < array.length; i++) {
            System.out.print("성적을 입력하시오:");
            array[i] = scan.nextInt();
        }
    }
    public double getAverage(int[] array) {
        double total = 0;
        for (int i = 0; i < array.length; i++)
            total += array[i];
        return total / array.length;
    }
}
```

```
public class ArrayProcTest {
    final static int STUDENTS = 5;
    public static void main(String[] args) {
        int[] scores = new int[STUDENTS];
        ArrayProc obj = new ArrayProc();
        obj.getValues(scores);
        System.out.println("평균은 " + obj.getAverage(scores));
    }
}
```

## 예: 객체를 리턴하는 메소드

```
public class Box {  
    private int width, length, height;  
    public Box(int width, int length, int height) {  
        this.width = width;  
        this.length = length;  
        this.height = height;  
    }  
    public Box copyBox() {  
        Box b = new Box(width, length, height);  
        return b;  
    }  
    ...  
}
```

```
public class BoxTest {  
    public static void main(String[] args) {  
        Box b1 = new Box(1, 2, 3);  
        Box b2 = b1.copyBox();  
    }  
}
```

# 예: 객체를 매개변수로 받는 메소드

```
public class Box {  
    private int width, length, height;  
    public Box(int width, int length, int height) {  
        this.width = width;  
        this.length = length;  
        this.height = height;  
    }  
    public int getVolume() {  
        return width * length * height;  
    }  
    public boolean equalVolume(Box other) {  
        return getVolume() == other.getVolume();  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Box b1 = new Box(1, 2, 3);  
        Box b2 = new Box(6, 1, 1);  
        System.out.println(b1.equalVolume(b2));  
    }  
}
```

# 정적 멤버

- 클래스의 정적 멤버(static member)는 그 클래스에 속한 모든 객체들을 통틀어서 하나만 존재하며, 모든 객체들이 공유한다.

**date** : 인스턴스 변수

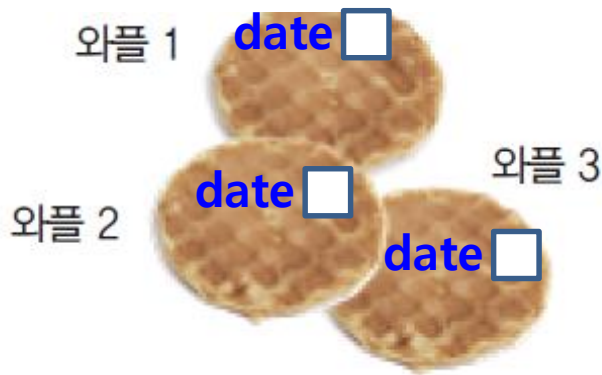
**price** : 정적 변수



클래스



객체생성



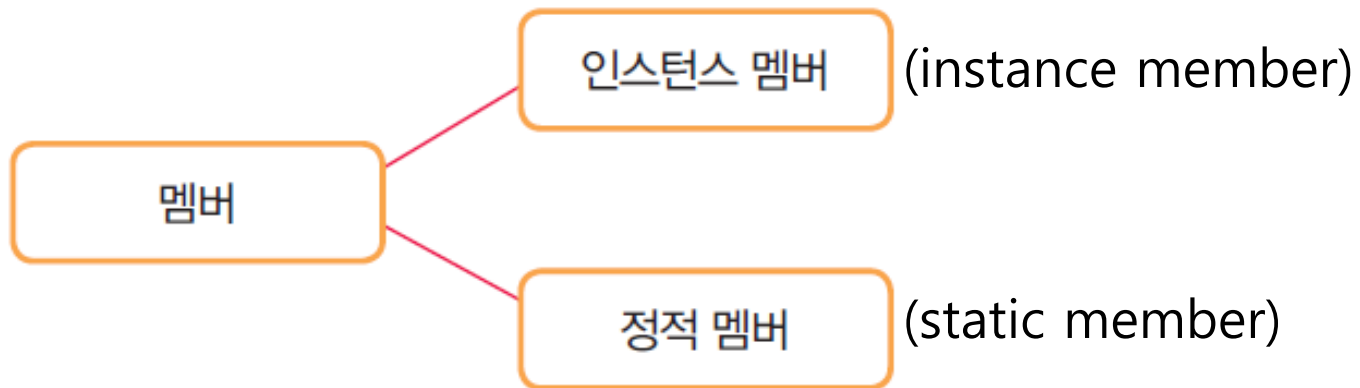
객체

정적 멤버는 클래스당 하나만 생성되어서 모든 객체가 공유합니다.



# 인스턴스 멤버 vs 정적 멤버

- 클래스의 멤버(필드, 메소드 등)는 인스턴스 멤버와 정적 멤버로 나뉘어진다.



# 인스턴스 변수

- 인스턴스 변수(instance variable)는 객체(인스턴스)마다 별도로 생성된다.
  - 예: 인스턴스 변수 channel, volume, onOff

```
class Television {  
    int channel;  
    int volume;  
    boolean onOff;  
}
```

channel	7
volume	9
onOff	trun

객체 A

channel	9
volume	10
onOff	trun

객체 B

channel	11
volume	5
onOff	trun

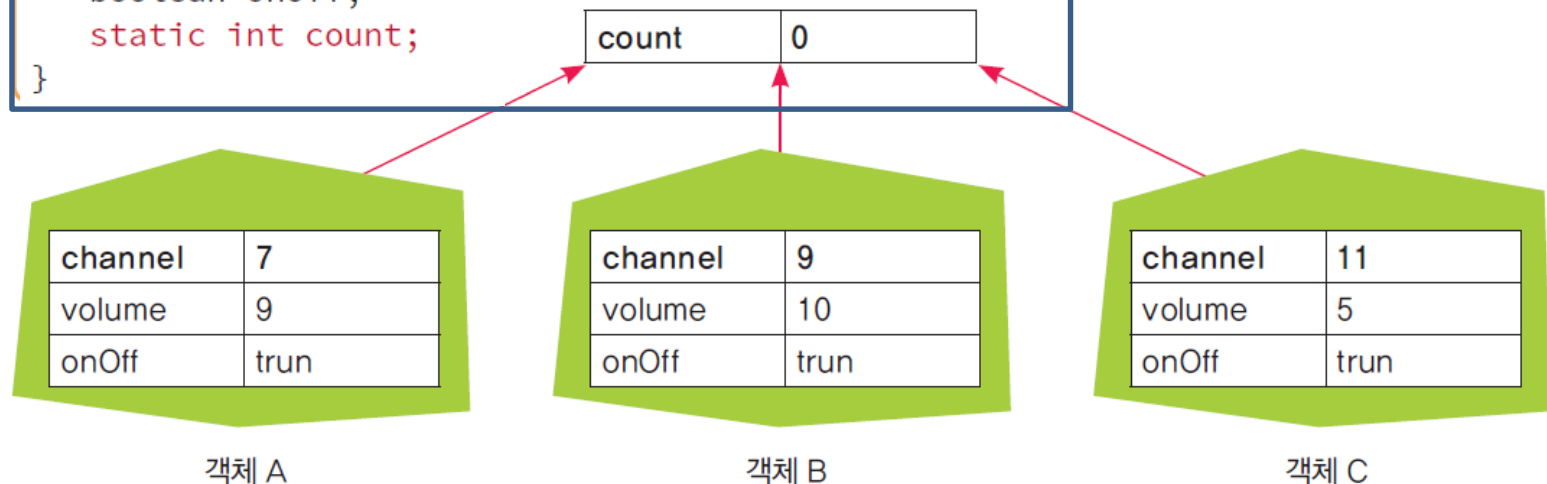
객체 C



# 정적 변수

- 정적 변수(static variable)는 클래스에 속한 클래스 변수
  - 객체 생성 없이도 사용 가능
  - 객체마다 갖는 것이 아님 - 정적 변수 하나를 그 클래스의 객체들이 공유
  - 필드 선언시 static 키워드를 붙임

```
class Television {  
    int channel;  
    int volume;  
    boolean onOff;  
    static int count;  
}
```



# 정적 변수 예제

```
public class Car {  
    private int id;                // 자동차의 시리얼 넘버  
    private static int numbers = 0; // 생성된 자동차 수(정적 변수)  
  
    public Car() {  
        id = ++numbers;           // 자동차 수를 증가하고 id에 대입  
    }  
    public void print() {  
        System.out.println(id);  
    }  
}
```

```
public class CarTest {  
    public static void main(String[] args) {  
        Car c1 = new Car();  
        Car c2 = new Car();  
        c1.print();  
        c2.print();  
    }  
}
```



1  
2

# 정적 변수 예제

```
public class Car {  
    private int speed;  
    private int id;                // 자동차의 시리얼 넘버  
    private static int numbers = 0; // 생성된 자동차 후(정적 변수)  
    public static final int MAX = 100; // 최대 속도(정적 멤버: 상수)  
    public Car() {  
        speed = MAX;  
        id = ++numbers;           // 자동차의 수를 증가하고 id에 대입  
    }  
    public void print() {  
        System.out.println(speed + " " + id);  
    }  
}
```

```
public class CarTest {  
    public static void main(String[] args) {  
        System.out.println(Car.MAX); // 객체 생성 없이 접근 가능  
        Car c1 = new Car();  
        Car c2 = new Car();  
        c1.print();  
        c2.print();  
    }  
}
```

100  
100 1  
100 2

# 정적 메소드

- 정적 메소드(static method)
  - 클래스 메소드
  - 객체를 생성하지 않고 호출할 수 있는 메소드
  - 메소드 선언부에 static 키워드를 붙임
  - 정적 메소드는 인스턴스 변수나 인스턴스 메소드 접근 불가
    - 즉, 정적 메소드는 정적 변수, 정적 메소드만 접근 가능
  - 예) Math 클래스에 들어 있는 각종 수학 메소드들

**double** value = Math.sqrt(9.0);

sqrt()는 Math 클래스  
에 속한 static method

# 정적 메소드 예

```
public class Car {  
    private int id;                // 자동차의 시리얼 넘버  
    private static int numbers = 0; // 자동차 수(정적 변수)  
  
    public Car() {  
        id = ++numbers;           // 자동차 수를 증가하고 id에 대입  
    }  
    public static int getNumbers() { // 정적 메소드  
        return numbers;  
    }  
}
```

```
public class CarTest {  
    public static void main(String[] args) {  
        System.out.println(Car.getNumbers());  
        Car c1 = new Car();  
        Car c2 = new Car();  
        System.out.println(Car.getNumbers());  
    }  
}
```

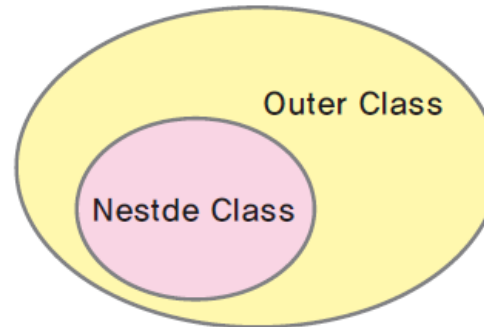
0  
2

# 중첩 클래스(nested class)

- 자바에서는 클래스 안에서 클래스를 정의할 수 있다.

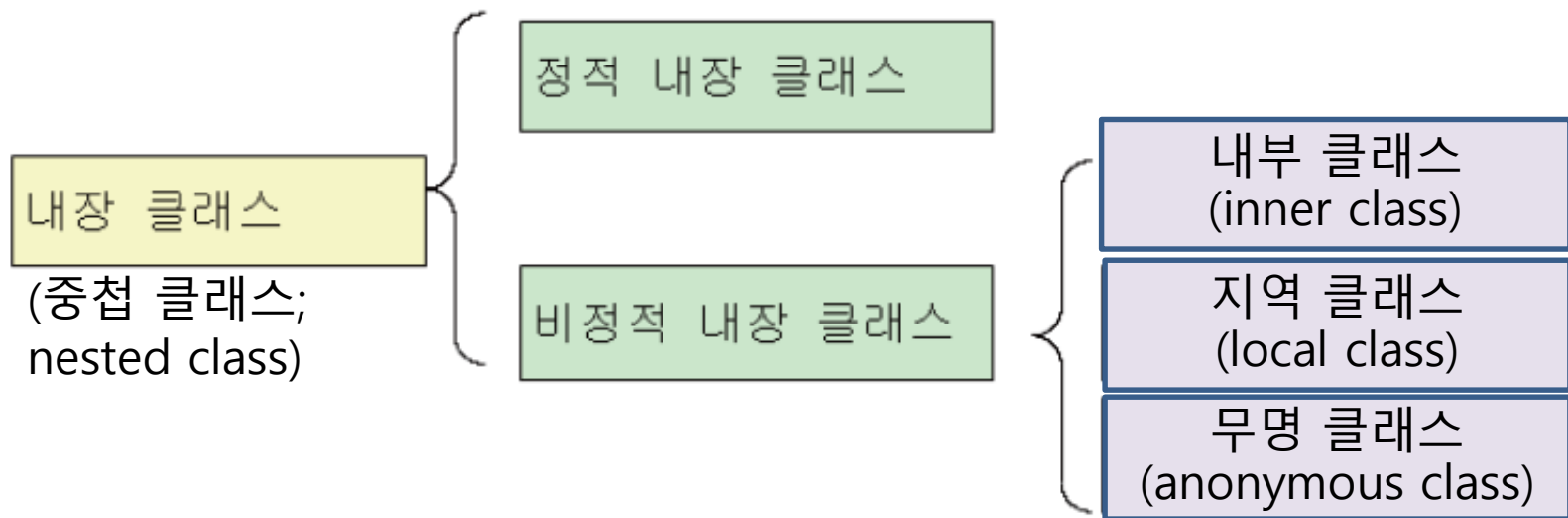
형식

```
class OuterClass {  
    ...  
    class NestedClass {  
        ...  
    }  
}
```



- 중첩 클래스(내장 클래스)를 사용하는 이유
  - 내장 클래스는 외부 클래스의 멤버가 private이어도 접근할 수 있다.
  - 내장 클래스는 외부에서 보이지 않도록 감출 수 있다.

# nested class 분류



# 내부 클래스(inner class)

- 클래스의 멤버처럼 클래스 내부에 정의되는 클래스

전체적인 구조



형식

```
class OuterClass {  
    ...  
    class InnerClass { }  
    ...  
}
```

클래스가 다른 클래스 안에 내장된다.



# inner class 예

```
public class OuterClass {  
    private int value = 10;  
  
    public class InnerClass {           // 내부 클래스  
        public void myMethod() {  
            System.out.println("외부 클래스의 private 변수 값: " + value);  
        }  
    }  
  
    public OuterClass() {  
        InnerClass obj = new InnerClass();  
        obj.myMethod();  
    }  
}
```

```
public class InnerClassTest {  
    public static void main(String[] args) {  
        OuterClass outer = new OuterClass();  
    }  
}
```

외부 클래스의 private 변수 값: 10

# inner class 예

```
public class OuterClass {  
    private int value = 10;  
  
    public class InnerClass {        // inner class  
        public void myMethod() {  
            System.out.println("외부 클래스의 private 변수 값: " + value);  
        }  
    }  
}
```

```
public class InnerClassTest {  
    public static void main(String[] args) {  
        OuterClass outer = new OuterClass();  
        OuterClass.InnerClass inner = outer.new InnerClass();  
        inner.myMethod();  
    }  
}
```

외부 클래스의 private 변수 값: 10

# 참고: inner class

- 게임에서 캐릭터가 여러 가지 아이템을 가지고 있다. 이것을 코드로 구현하여 보자. 아이템은 캐릭터만 사용한다고 가정하자. 그러면 캐릭터를 나타내는 클래스 안에 아이템을 inner class로 정의할 수 있다.

## GameCharacter

list: ArrayList<GameItem>  
add(name: String, type: int, price: int): void  
print(): void

## GameCharacter.GameItem

name: String  
type: int  
price: int  
getPrice(): int  
toString(): String

## GameChracterTest

main(args: String[]): void

```

import java.util.ArrayList;

public class GameCharacter {
    private class Gameltem {
        String name;
        int type;
        int price;
        int getPrice() { return price; }
        @Override
        public String toString() {
            return "Gameltem [name=" + name + ", type=" + type + ", price=" + price + "];"
        }
    }

    private ArrayList<Gameltem> list = new ArrayList<>();

    public void add(String name, int type, int price) {
        Gameltem item = new Gameltem();
        item.name = name;
        item.type = type;
        item.price = price;
        list.add(item);
    }

    public void print() {
        int total = 0;
        for (Gameltem item : list) {
            System.out.println(item);
            total += item.getPrice();
        }
        System.out.println(total);
    }
}

```

```

public class GameChracterTest {
    public static void main(String[] args) {
        GameCharacter charac = new GameCharacter();
        charac.add("Sword", 1, 100);
        charac.add("Gun", 2, 50);
        charac.print();
    }
}

```

# 지역 클래스(local class)

- 메소드(또는 코드 블록) 내부에 정의되는 클래스이다.
- 메소드 밖에서는 local class를 볼 수 없다.
- local class에는 접근 제어자(public, private, ...)를 둘 수 없다.
  - 지역 변수에 접근 제어자를 둘 수 없는 것과 마찬가지.
- local class가 소속된 메소드의 지역 변수 중 final인 것만 참조할 수 있다.
  - 지역 클래스의 인스턴스가 메소드 호출보다 더 오랜기간 동안 존재할 수 있으므로 지역 변수의 자체 복사본을 필요로 함
  - 여러 인스턴스의 지역 변수 복사본들이 서로 다른 값을 가지는 것을 방지하기 위해 final 지역 변수만 참조할 수 있게 함

JDK 8 부터 : 지역변수가 final로 선언되지 않았더라도 변경 코드가 없어 실제로 final로 간주할 수 있으면 참조 가능

```

public class OuterClass {
    private int data = 30;

    public void display() {
        final String msg = "현재의 데이터값은 ";

        class LocalClass {           // local class
            void printMsg() {
                System.out.println(msg + data);
            }
        }

        LocalClass obj = new LocalClass();
        obj.printMsg();
    }
}

```

local class 이름 앞에는  
final과 abstract만 붙일  
수 있다.

```

public class LocalClassTest {
    public static void main(String[] args) {
        OuterClass obj = new OuterClass();
        obj.display();
    }
}

```

현재의 데이터값은 30

# 참고: local class

```
public class LocalClassExample {
    public static void validatePhoneNumber(String phoneNumber1, String phoneNumber2) {
        final int numberLength = 10;

        class PhoneNumber {
            String formattedPhoneNumber = null;
            PhoneNumber(String phoneNumber) {
                String currentNumber = phoneNumber.replaceAll("[^0-9]", "");
                if (currentNumber.length() == numberLength)
                    formattedPhoneNumber = currentNumber;
                else
                    formattedPhoneNumber = null;
            }
            public String getNumber() {
                return formattedPhoneNumber;
            }
        }

        PhoneNumber myNumber1 = new PhoneNumber(phoneNumber1);
        PhoneNumber myNumber2 = new PhoneNumber(phoneNumber2);
        if (myNumber1.getNumber() == null)
            System.out.println("First number is invalid");
        else
            System.out.println("First number is " + myNumber1.getNumber());
        if (myNumber2.getNumber() == null)
            System.out.println("Second number is invalid");
        else
            System.out.println("Second number is " + myNumber2.getNumber());
    }

    public static void main(String[] args) {
        validatePhoneNumber("123-456-7890", "456-7890");
    }
}
```