4. 클래스와 객체

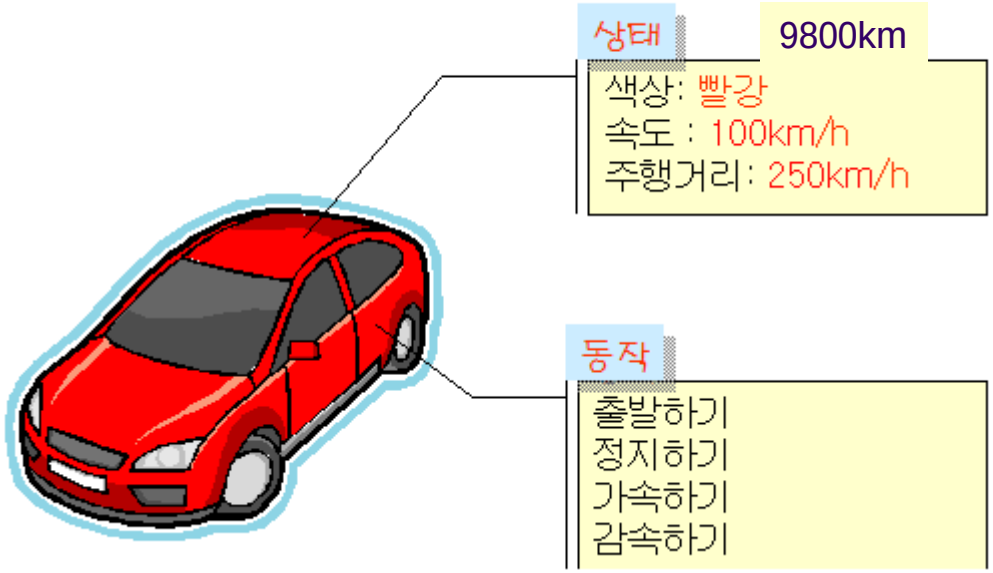
충남대학교
컴퓨터공학과

학습 내용

- 객체지향의 이해
  - 객체
  - 메시지
  - 클래스
  - 객체지향의 장점
  - String 클래스
- 클래스와 객체
  - 객체의 일생
  - 메소트
  - 필드
  - UML
  - 생성자
  - 정적 변수와 메소드
  - 접근제어
  - This
  - 클래스 간의 관계

객체란?

- 객체(Object)는 상태와 동작을 가지고 있다.
- 객체의 상태(state)는 객체의 특징값(속성)이다.
- 객체의 동작(behavior) 또는 행동은 객체가 취할 수 있는 동작

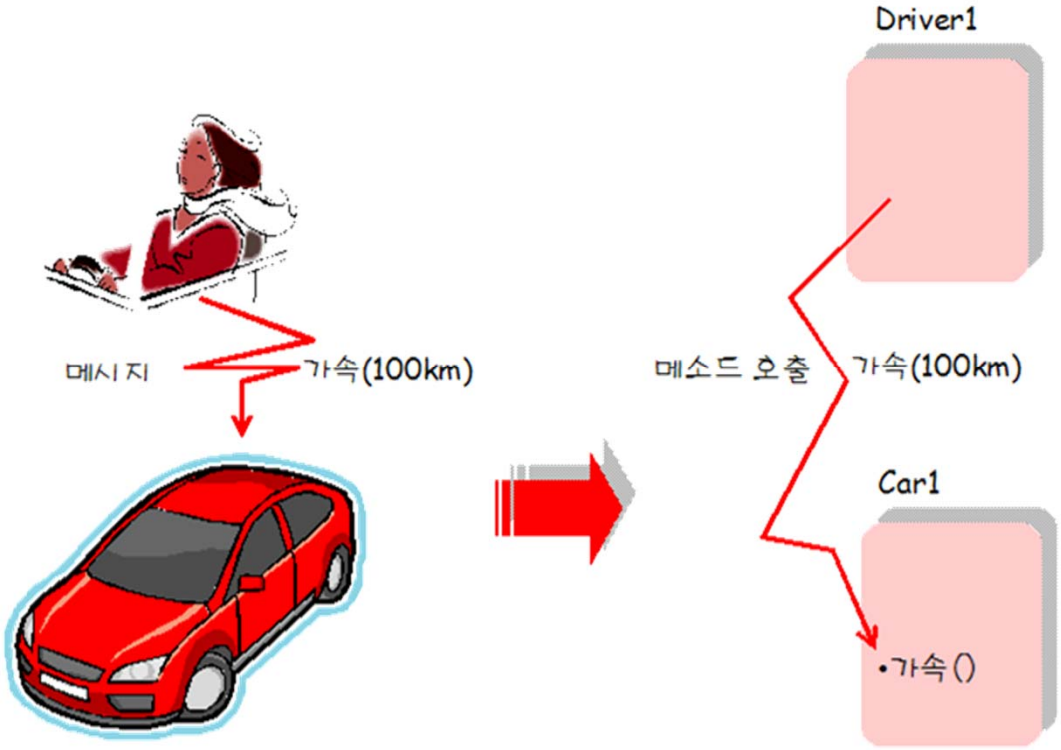


객체의 예

객체	상태	동작
전 구	종류, 색상	켜지기, 꺼지기
라디오	색상	켜지기, 꺼지기
강아지	종류	짖기, 달리기, 물기
자전거	종류, 색상	출발하기, 정지하기, 가속하기, 감속하기
사 자	성별	짖기, 달리기

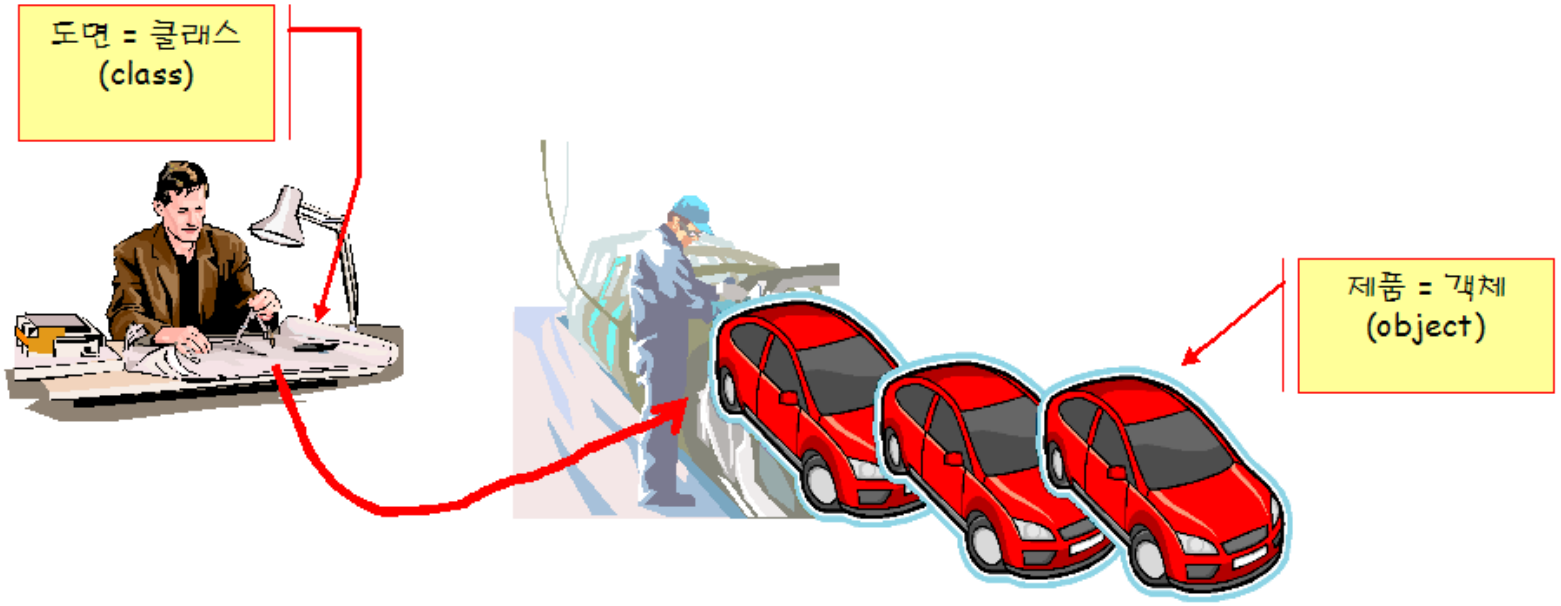
메시지

- 소프트웨어 객체는 메시지(message)를 통해 다른 소프트웨어 객체와 통신하고 서로 상호 작용한다.



클래스

- 클래스(class): 객체를 만드는 설계도
- 클래스로부터 만들어지는 각각의 객체를 특별히 그 클래스의 **인스턴스(instance)**라고도 한다.



자동차 클래스

상태

색상: 빨강
속도 : 100km/h
주행거리: 250km/h
9800km

동작

가속하기
감속하기

```
public class Car
{
        int color;
        int speed;
        int mileage;

        void speedUp(int s) {
                speed += s;
        }
        void speedDown(int s) {
                speed -= s;
        }
}
```

변수

메소드

클래스 내부

Key Point

클래스 내부는 상태를 나타내기 위한 데이터(data) 선언들과 행동을 정의하는 메소드(method) 선언들로 구성된다.

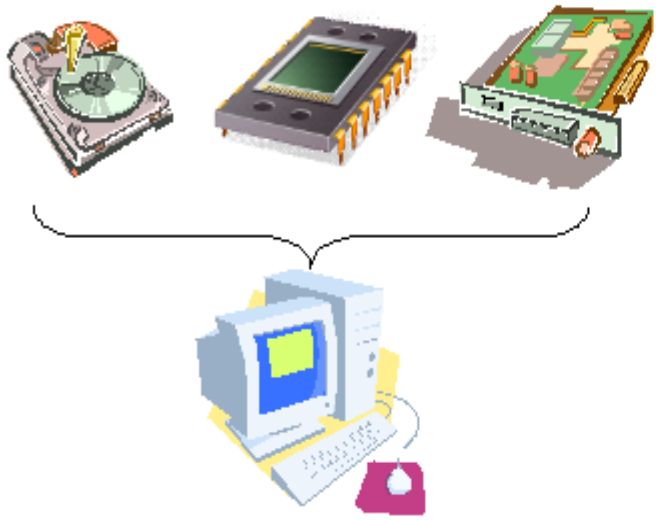
- 데이터는 상태를 표현하기 위한 상수, 변수 등을 의미하고
- 메소드는 그 객체의 행동을 표현하는 함수 혹은 프로시저라고 할 수 있다.

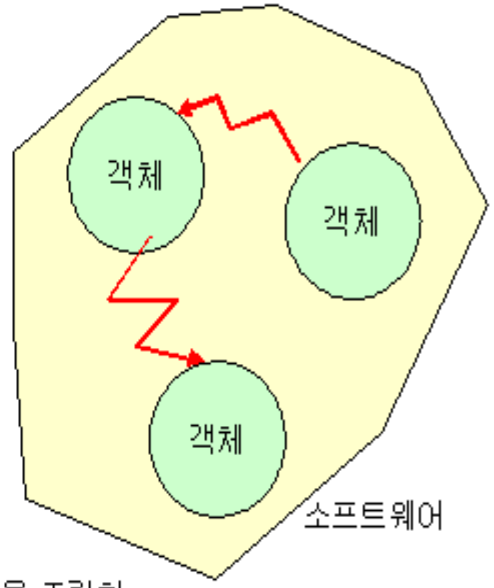
객체 지향의 장점

- 신뢰성있는 소프트웨어를 쉽게 작성할 수 있다.
- 코드를 재사용하기 쉽다.
- 업그레이드가 쉽다.
- 디버깅이 쉽다.

소프트웨어 작성이 쉽다

- 부품을 구입하여 컴퓨터를 조립하듯이 소프트웨어를 작성할 수 있다.





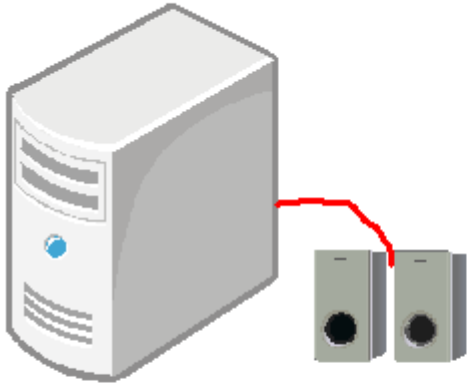
부품을 조립하여 제품을 만들듯이 객체들을 조립하여 소프트웨어를 만든다.

코드의 재사용

업그레이드가 쉽다.

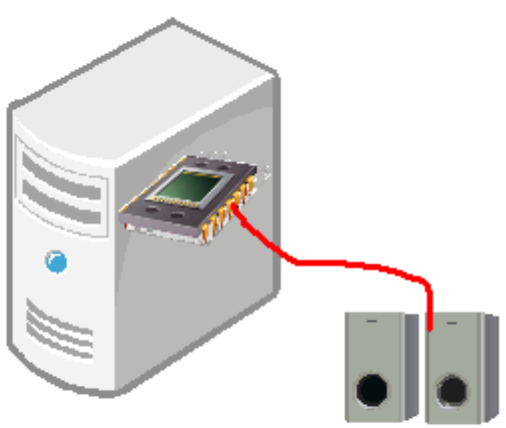
- 라이브러리가 업그레이드되면 쉽게 바꿀 수 있다.
- 정보 은닉이 가능하기 때문에 업그레이드 가능

O



만약 외부의 표준 오디오 단자를 이용하였으면 내부의 사운드 카드를 변경할 수 있다.

X



만약 내부의 오디오 제어칩의 단자에 연결하였으면 내부의 사운드 카드를 변경할 수 없다.

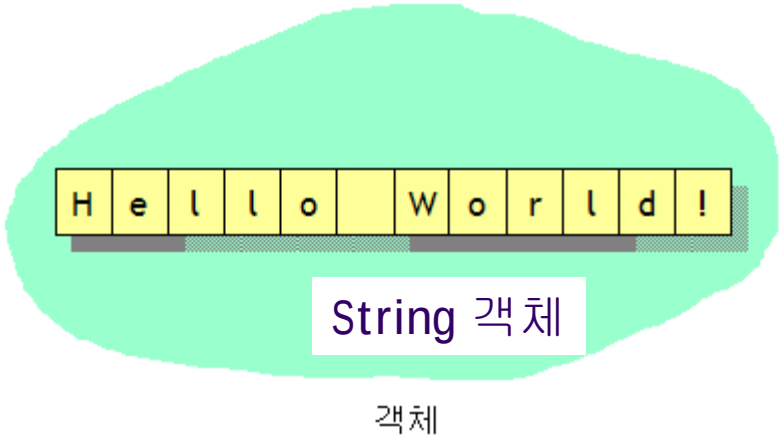
쉬운 디버깅

- 예를 들어서 절차 지향 프로그램에서 하나의 변수를 1000개의 함수가 사용하고 있다고 가정해보자. -> 하나의 변수를 1000개의 함수에서 변경할 수 있다.

- 객체 지향 프로그램에서 100개의 클래스가 있고 클래스당 10개의 메소드를 가정해보자. -> 하나의 변수를 10개의 메소드에서 변경할 수 있다.

- 어떤 방법이 디버깅이 쉬울까?

문자열 객체

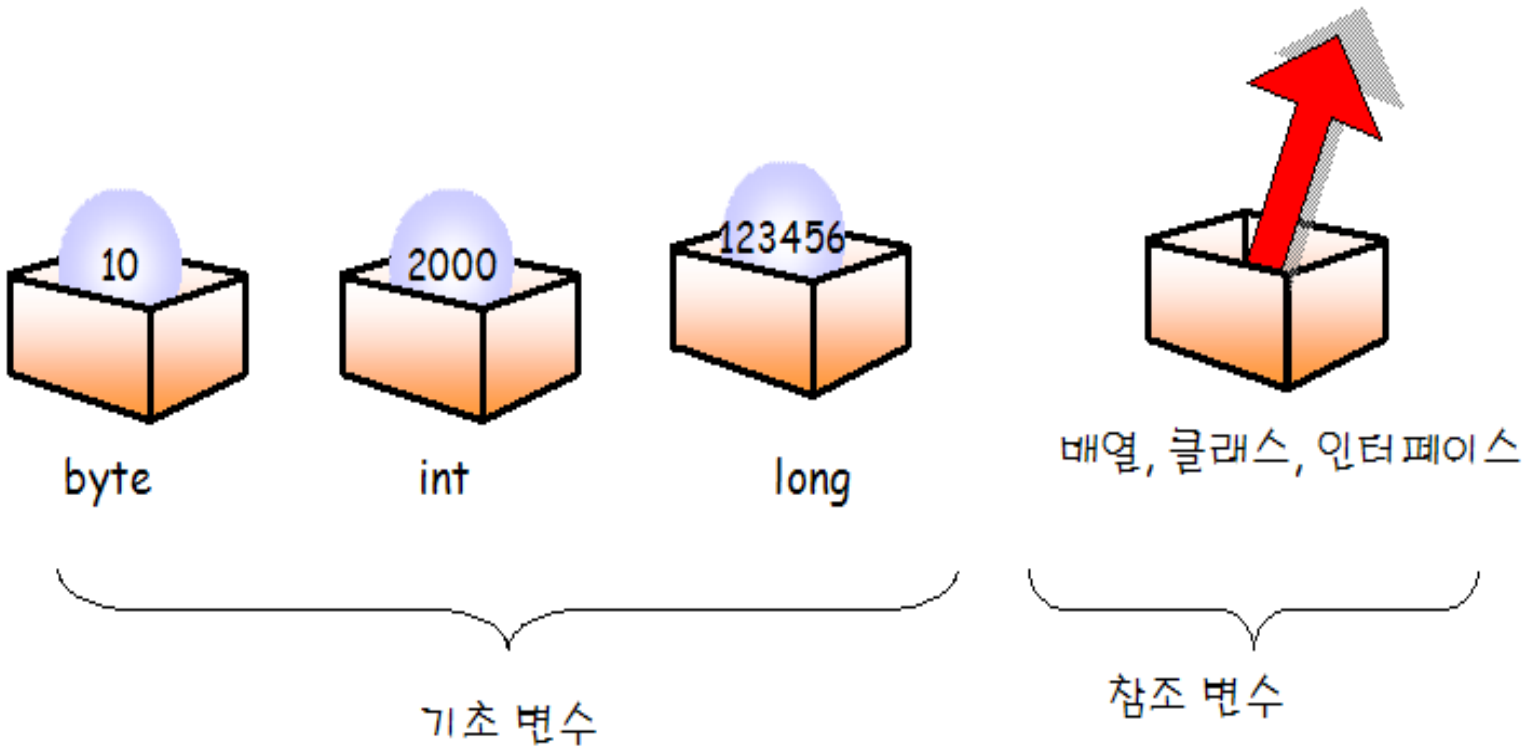


클래스에서 객체를 생성하는 방법

- 단 하나의 방법만이 존재한다.
- String s = new String("Hello World");

- 실체화**(Instantiation)**
  - new 연산자를 이용하여 객체를 생성하는 것을 말하며
  - 객체는 클래스의 **실체(instance)**라고 한다.

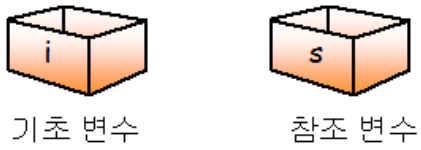
기초 변수와 참조 변수



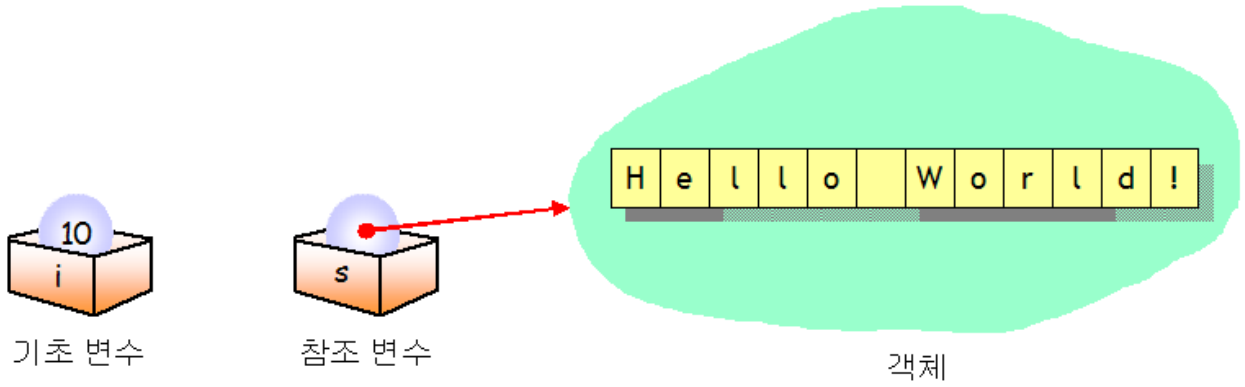
기초 변수와 참조 변수의 비교

```
int     i;      // 기초 변수
String s;       // 참조 변수
```

위와 같이 정의하면 두 변수 모두 처음에는 데이터를 담고 있지 않다. 즉 초기화가 되지 않은 상태이다.

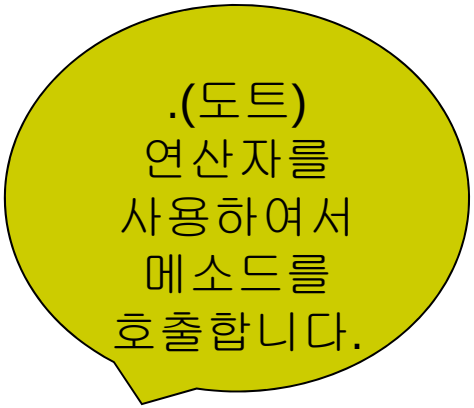



```
i = 10;                                 // 기초 변수에 값을 대입
s = new String("Hello World!"); // 객체를 생성하고 참조 변수에 객체의 주소를 대입
```

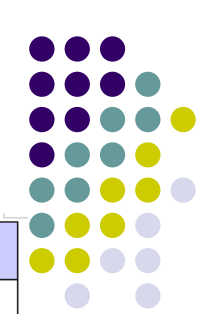


객체의 메소드의 호출

- String s = "Hello World!";
- **int** size = s.length(); // size는 12가 된다.



String 클래스의 메소드



메소드 요약	
char	**charAt**(int index) 지정된 인덱스에 있는 문자를 반환한다.
int	**compareTo**(String anotherString) 사전적 순서로 문자열을 비교한다.앞에 있으면 -1, 같으면 0, 뒤에 있으면 1이 반환된다.
String	**concat**(String str) 주어진 문자열을 현재의 문자열 뒤에 붙인다.
boolean	**equals**(Object anObject) 주어진 객체와 현재의 문자열을 비교한다.
boolean	**equalsIgnoreCase**(String anotherString) 대소문자를 무시하고 비교한다.
boolean	**isEmpty**() length()가 0이면 true를 반환한다.
int	**length**() 현재 문자열의 길이를 반환한다.
String	**replace**(char oldChar, char newChar) 주어진 문자열에서 oldChar를 newChar로 변경한, 새로운 문자열을 생성하여 반환한다.
String	**substring**(int beginIndex, int endIndex) 현재 문자열의 일부를 반환한다.
String	**toLowerCase**() 문자열의 문자들을 모두 소문자로 변경한다.
String	**toUpperCase**() 문자열의 문자들을 모두 대문자로 변경한다.

메소드 사용의 예

StringTest.java

```
public class StringTest
{
   public static void main (String[] args)
   {
      String proverb = "A barking dog";          // new 연산자 생략
      String s1, s2, s3, s4;      // 참조 변수로서 메소드에서 반환된 참조값을 받는다.

      System.out.println ("문자열의 길이 =" + proverb.length());

      s1 = proverb.concat (" never Bites!");      // 문자열 결합
      s2 = proverb.replace ('B', 'b');            // 문자 교환
      s3 = proverb.substring (2, 5);              // 부분 문자열 추출
      s4 = proverb.toUpperCase();                 // 대문자로 변환

      System.out.println(s1);
      System.out.println(s2);
      System.out.println(s3);
      System.out.println(s4);
   }
}
```

출력결과

```
문자열의 길이 =13
A barking dog never Bites!
A barking dog
bar
A BARKING DOG
```

Math 클래스 메소드

메소드	설명
abs(double a) abs(float a) abs(int a) abs(long a)	절대값(absolute value): a의 절대값을 리턴
sqrt(double a)	제곱근(square root): a의 제곱근을 리턴
exp(double a)	지수승(exponentiation): e의 a 승 값을 리턴
sin(double a) cos(double a) tan(double a)	삼각함수(trigonometric function): 사인, 코사인, 탄젠트 값을 리턴
pow(double a, double b)	거듭제곱(power): a의 b승 값을 리턴한다.
random()	난수발생기: 0.0과 1.0 사이의 난수 double 값을 리턴

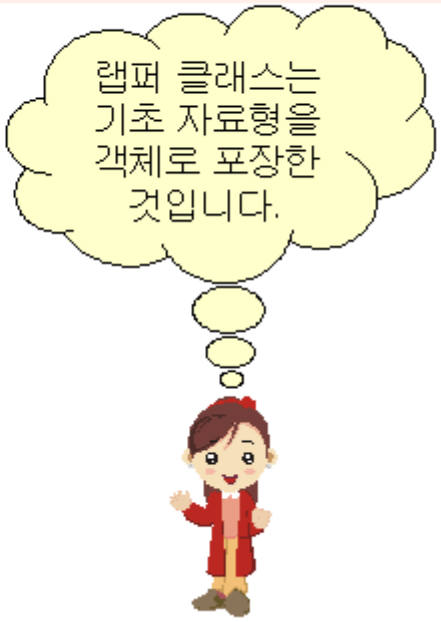
- value = Math.cos(90) + Math.sqrt(2);

```java
/**************************************************************
*  CaseSensitive.java
*
*  String의 메소드 사용을 보여주기 위한 프로그램
**************************************************************/

/**
* 스트링을 대문자, 소문자로 변환하고 또한 두 스트링을 비교한다.
*/
public class CaseSensitive {

    public static void main(String[] args) {

        String a = "Playing With Java";
        String b = "playing with java";

        System.out.println("원래 스트링 : " + a);
        System.out.println("대문자 스트링 : " + a.toUpperCase());
        System.out.println("소문자 스트링 : " + a.toLowerCase() + "\n");

        System.out.println("원래 스트링  : " + a);
        System.out.println("비교 스트링  : " + b);

        if (a.equals(b)) {
            System.out.println("equal(): TRUE 리턴");
        } else {
            System.out.println("equal(): FALSE 리턴");
        }

        if (a.equalsIgnoreCase(b)) {
            System.out.println("equalIgnoreCase(): TRUE 리턴");
        } else {
            System.out.println("equalIgnoreCase(): FALSE 리턴");
        }
    }
}
```

랩퍼 클래스

- 기초 자료형을 객체로 포장하여 주는 클래스

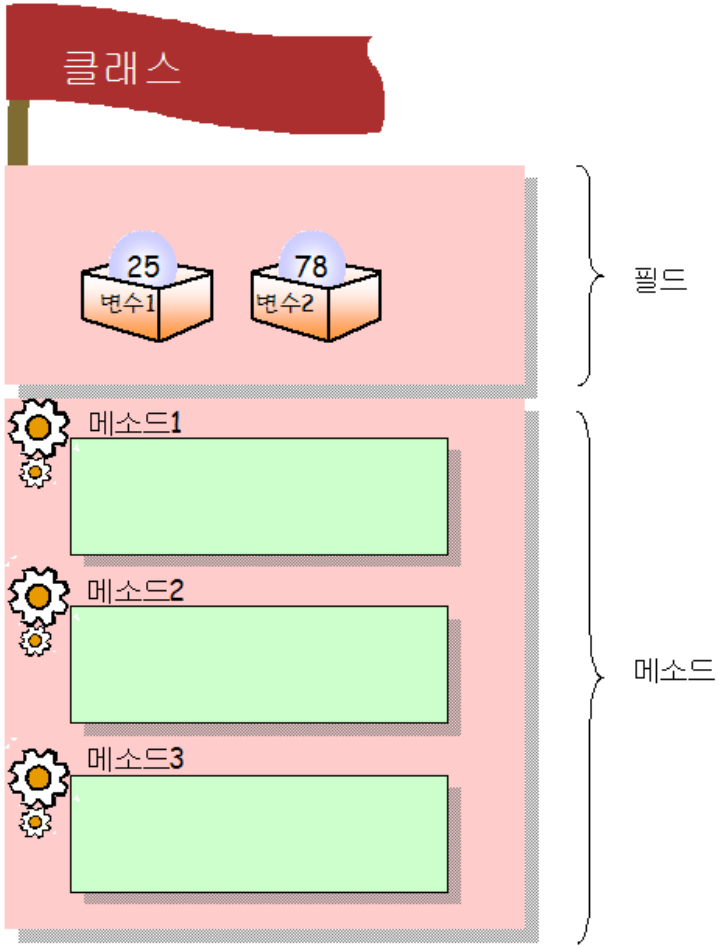
```
Integer obj = new Integer(10);
```



기본 타입을 위한 랩퍼 클래스

기본 타입	랩퍼 클래스
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

클래스의 구성



- 클래스(class)는 객체의 설계도라 할 수 있다.
- 클래스는 필드와 메소드로 이루어진다.
- 필드(field)는 객체의 속성을 나타낸다.
- 메소드(method)는 객체의 동작을 나타낸다.

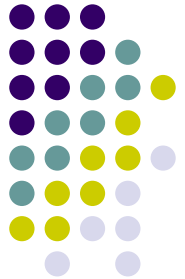
클래스 정의의 예

```
class Car {

    // 필드 정의
    public int speed; // 속도
    public int mileage; // 주행거리
    public String color; // 색상

    // 메소드 정의
    public void speedUp() { // 속도 증가 메소드
            speed += 10;
    }

    public void speedDown() { // 속도 감소 메소드
            speed -= 10;
    }

    public String toString() { // 객체의 상태를 문자열로 반환하는 메소드
            return "속도: " + speed + " 주행거리: " + mileage + " 색상: " + color;
    }
}
```

필드 정의!

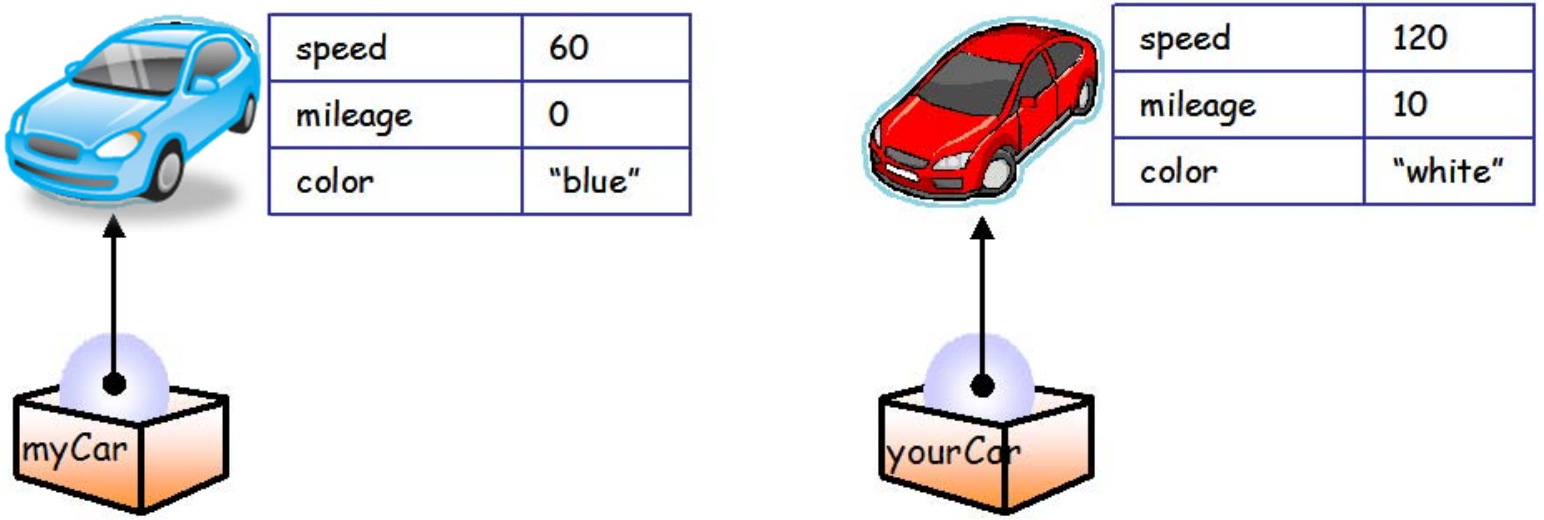
메소드 정의!

테스트 클래스



```
public class CarTest {
    public static void main(String[] args) {
        Car myCar = new Car();        // 첫번째 객체 생성
        Car yourCar = new Car(); // 두번째 객체 생성

        myCar.speed = 60;  // 객체의 필드 변경
        myCar.mileage = 0; // 객체의 필드 변경
        myCar.color = "blue";          // 객체의 필드 변경

        myCar.speedUp();               // 객체의 메소드 호출
        System.out.println(myCar);

        yourCar.mileage = 10;          // 객체의 필드 변경
        yourCar.speed = 120;    // 객체의 필드 변경
        yourCar.color = "white";  // 객체의 필드 변경

        yourCar.speedDown();           // 객체의 메소드 호출
        System.out.println(yourCar);
    }
}
```

속도: 70 주행거리: 0 색상: blue
속도: 110 주행거리: 10 색상: white



객체의 일생

Car c = new Car();

c.speedUp();

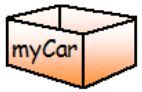
c = null;

객체의 생성

```
Car     myCar;              // ① 참조 변수를 선언
myCar = new  Car();         // ② 객체를 생성하고 ③ 참조값을 myCar에 저장
```

① 참조 변수 선언

Car 타입의 객체를 참조할 수 있는 변수 myCar를 선언한다.



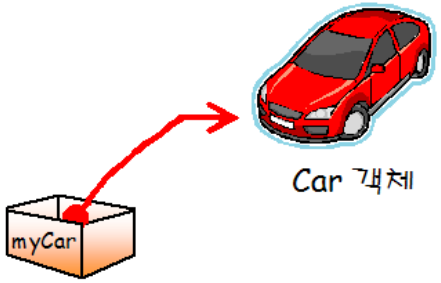
② 객체 생성

new 연산자를 이용하여 객체를 생성하고 객체 참조값을 반환한다.

Car 객체

③ 참조 변수와 객체의 연결

생성된 새로운 객체의 참조값을 myCar라는 참조 변수에 대입한다.



Car 객체

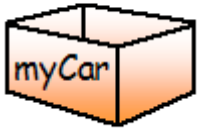
참조변수

주의

```
Car     myCar;
```

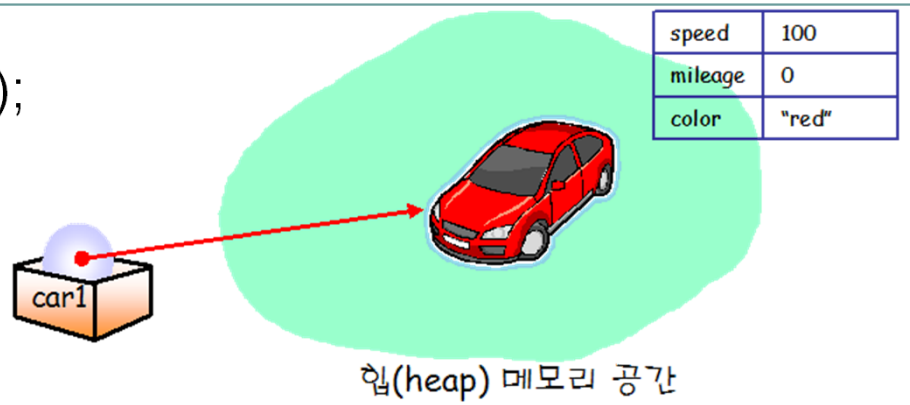
위의 문장으로 객체가
생성되는 것은 아님!!!

객체를 가리키는 참조값을
담을 수 있는 변수만 생성됨.

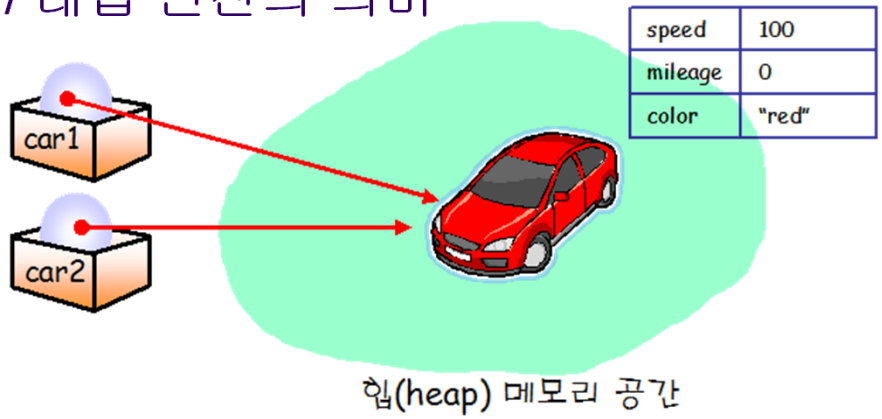


참조 변수와 대입 연산

- Car car1 = new Car();

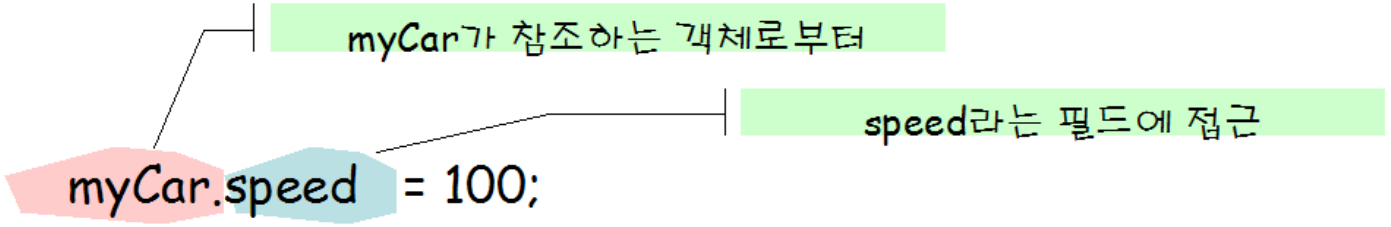


- Car car2 = car1; // 대입 연산의 의미



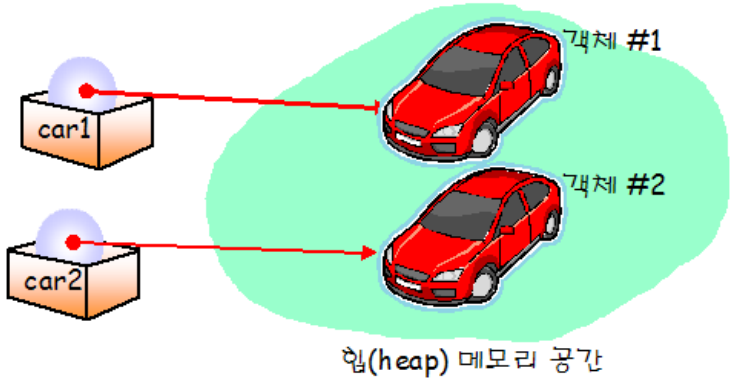
객체의 사용

- 객체를 이용하여 필드와 메소드에 접근할 수 있다.

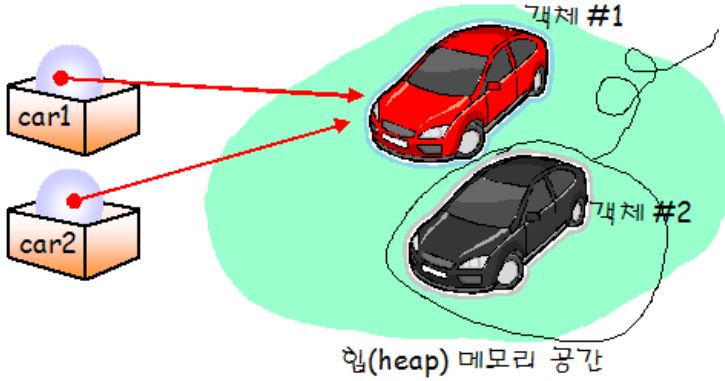


객체의 소멸

```
Car car1 = new Car();    // 첫번째 객체
Car car2 = new Car();    // 두번째 객체
```



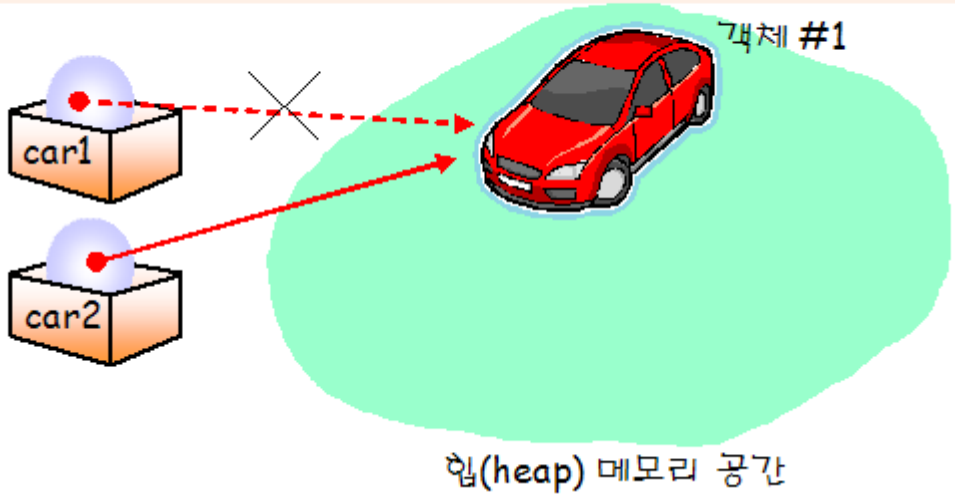

```
car2 = car1;    // car1과 car2는 같은 객체를 가리킨다.
                // car2가 가리켰던 객체는 쓰레기 수집기에 의하여 수거된다.
```



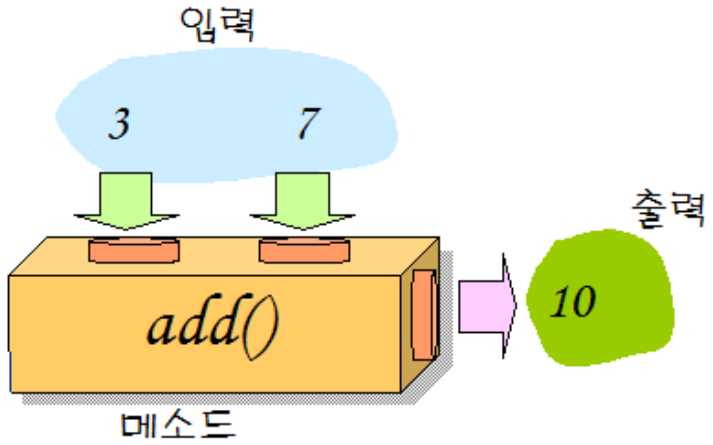
객체는 참조가 없어지면 소멸!!

객체의 소멸

```
car1 = null;      // 객체 1은 아직도 활성화된 참조가 있기 때문에 소멸되지 않는다.
```



메소드



메소드 선언은 다음과 같은 형식을 가진다.

```
[수식자] 반환형 myMethod ( 매개변수 목록 ) {
        // 문장들
        ...
}
```

메소드 선언

- 메소드 머리**(method header)**와 메소드 본체**(method body)**로 구성
- 메소드 머리는 리턴 타입, 메소드 이름, 매개변수 리스트로 구성
- 매개변수 리스트는 각 매개변수의 타입(type)과 이름을 선언

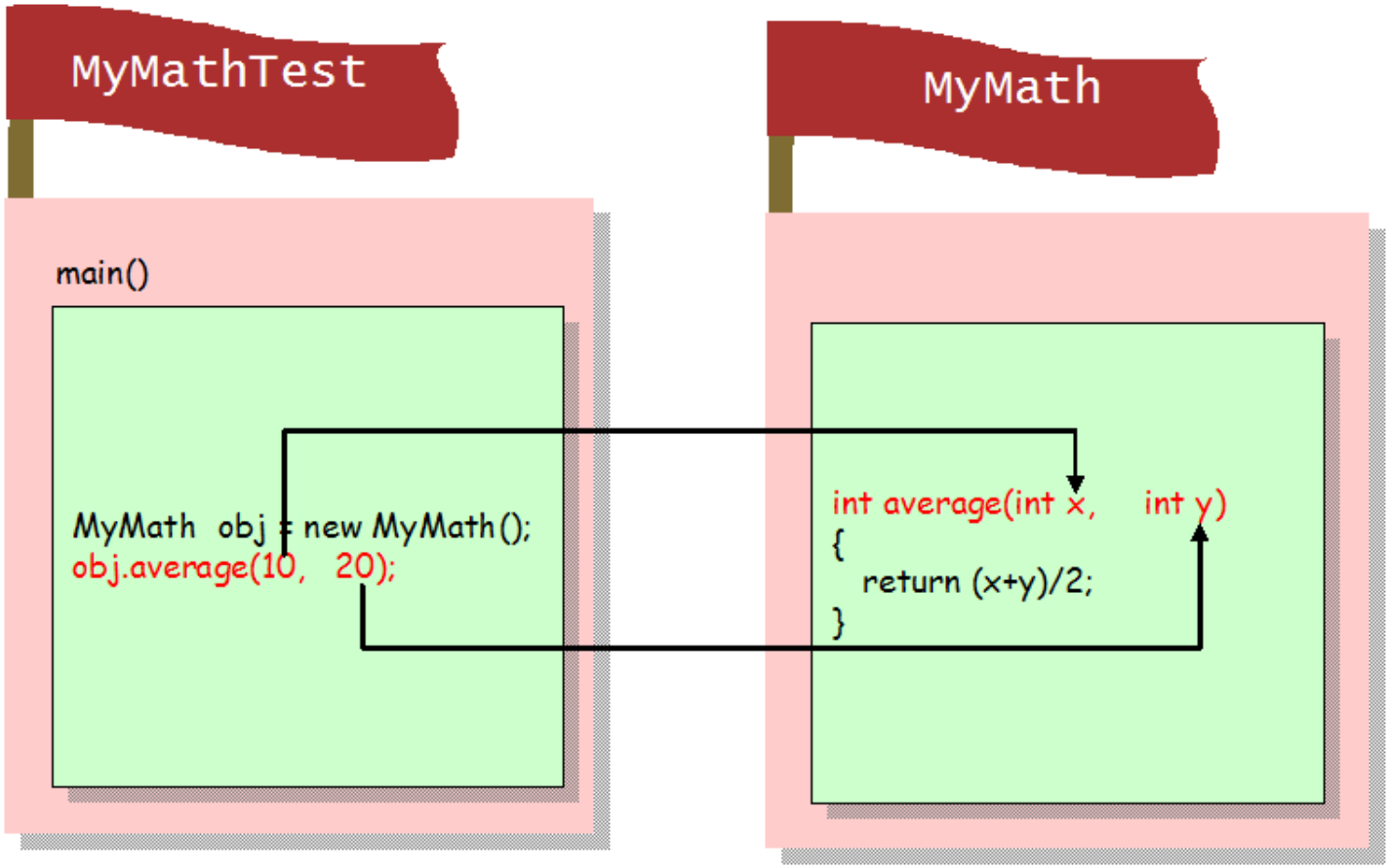
```
//  메소드에 대한 주석
public void move(double dx, double dy)
{
}
```

메소드머리

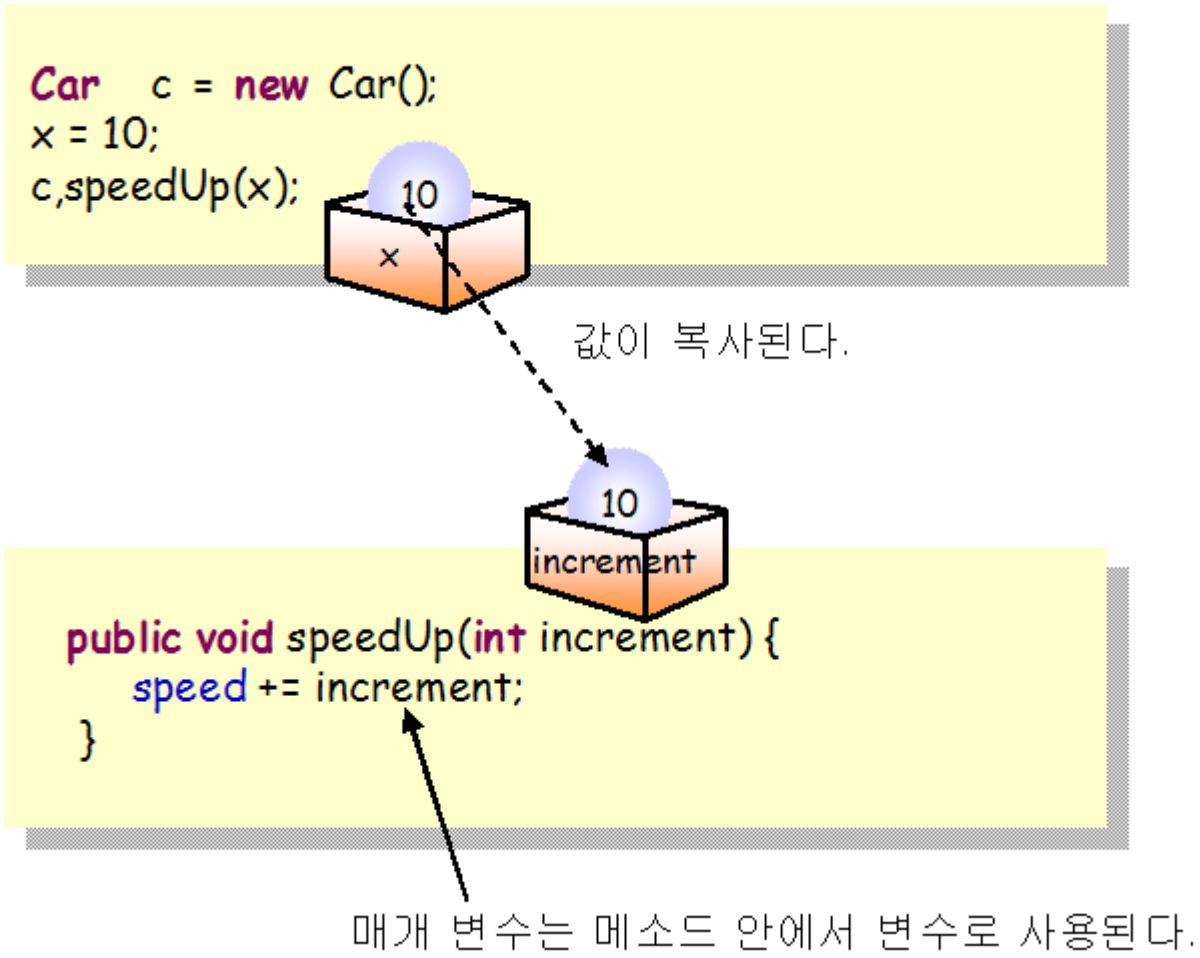
메소드본체

```
public void move(double dx, double dy)
{
   x = x + dx;
   y = y + dy;
}
```

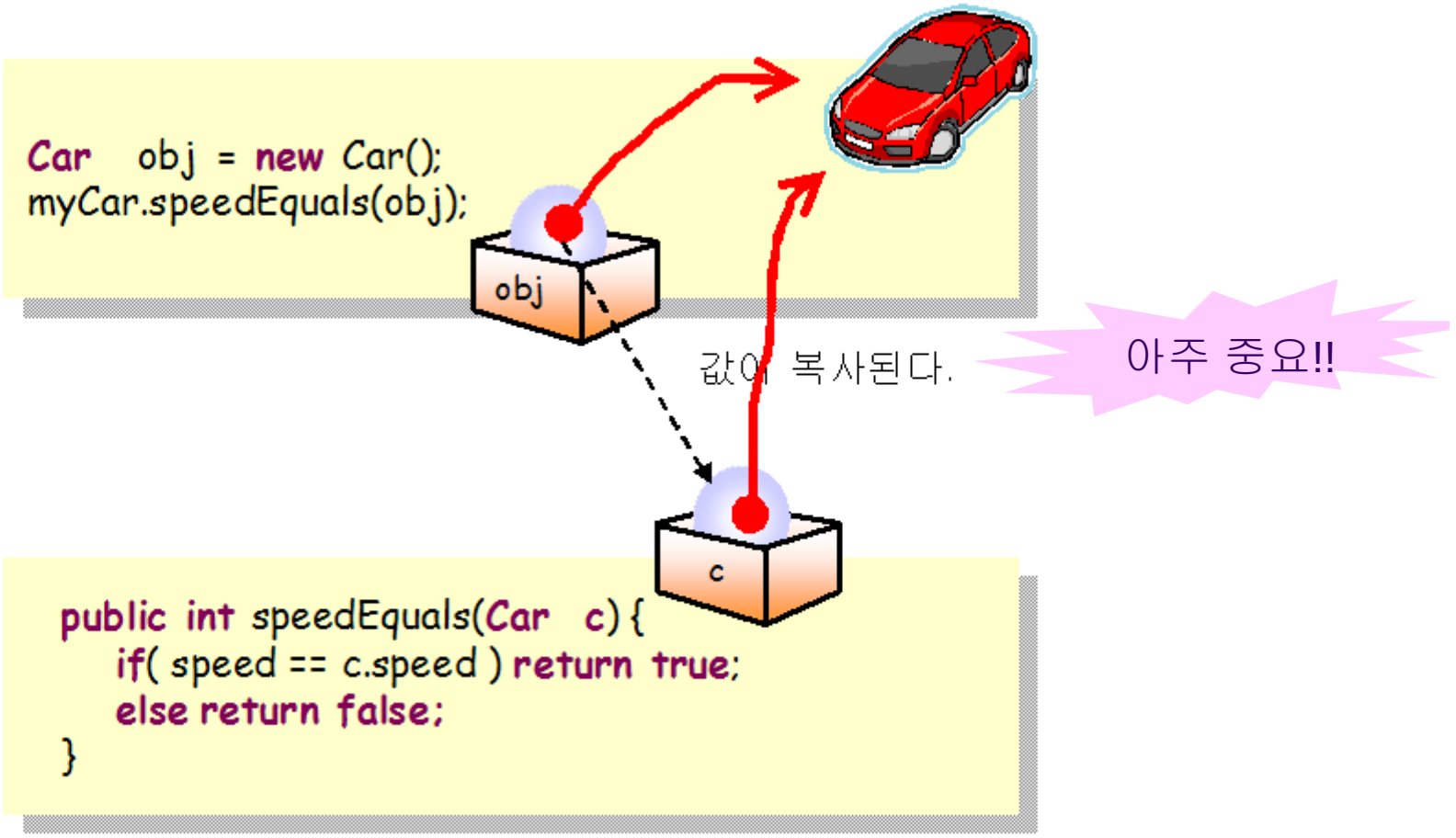
매개 변수


MyMathTest
main()
MyMath obj = new MyMath();
obj.average(10, 20);
MyMath
int average(int x, int y)
{
return (x+y)/2;
}

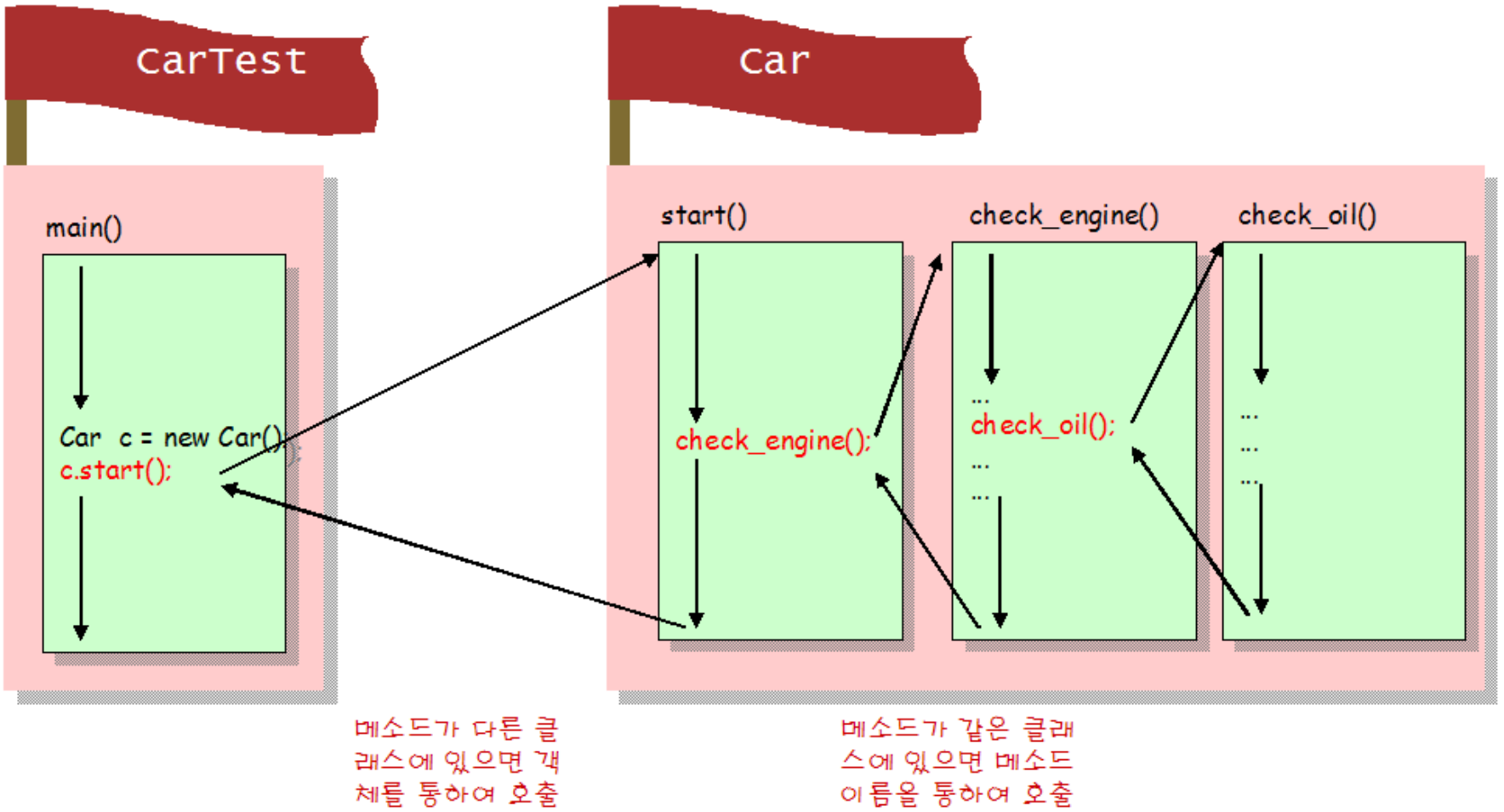
값에 의한 전달


Car c = new Car();
x = 10;
c.speedUp(x);
10
x
값이 복사된다.
10
increment
public void speedUp(int increment) {
speed += increment;
}
매개 변수는 메소드 안에서 변수로 사용된다.

매개 변수가 객체인 경우


Car obj = new Car();
myCar.speedEquals(obj);
obj
값이 복사된다.
아주 중요!!
c
public int speedEquals(Car c){
if(speed == c.speed) return true;
else return false;
}

메소드 호출


CarTest
Car
main()
Car c = new Car();
c.start();
start()
check_engine()
check_oil()
check_engine();
...
check_oil();
...
...
...
...
...
메소드가 다른 클래스에 있으면 객체를 통하여 호출
메소드가 같은 클래스에 있으면 메소드 이름을 통하여 호출

메소드 호출의 예제

```
import java.util.*;
class DiceGame {

    int diceFace;
    int userGuess;

    private void RollDice()
    {
            diceFace = (int)(Math.random() * 6) + 1;
    }
    private int getUserInput(String prompt)
    {
            System.out.println(prompt);
            Scanner s = new Scanner(System.in);
            return s.nextInt();
    }
```

메소드 호출의 예제

```
private void checkUserGuess()
    {
        if( diceFace == userGuess )
            System.out.println("맞았습니다");
        else
            System.out.println("틀렸습니다");
    }
    public void startPlaying()
    {
        int userGuess = getUserInput("예상값을 입력하시오: ");
        RollDice();
        checkUserGuess();
    }
}

public class DiceGameTest {
     public static void main(String[] args) {
        DiceGame game = new DiceGame();
        game.startPlaying();
     }
}
```

결과 화면

```
예상값을 입력하시오:
3
틀렸습니다
```

중복 메소드

- 메소드 오버라이딩(method overriding)

```
// 정수값을 제곱하는 메소드
public int square(int i)
{
        return i*i;
}

// 실수값을 제곱하는 메소드
public double square(double i)
{
        return i*i;
}
```

- 메소드 호출시 매개 변수를 보고 일치하는 메소드가 호출된다.
- 만약 square(3.14)와 같이 호출되면 컴파일러는 매개 변수의 개수, 타입, 순서 등을 봐서 두 번째 메소드를 호출한다.

중복 메소드 예제

```
class Car {

    // 필드 선언
    private int speed; // 속도

    // 중복 메소드: 정수 버전
    public void setSpeed(int s) {
        speed = s;
        System.out.println("정수 버전 호출");
    }

    // 중복 메소드: 실수 버전
    public void setSpeed(double s) {
        speed = (int)s;
        System.out.println("실수 버전 호출");
    }
}
```

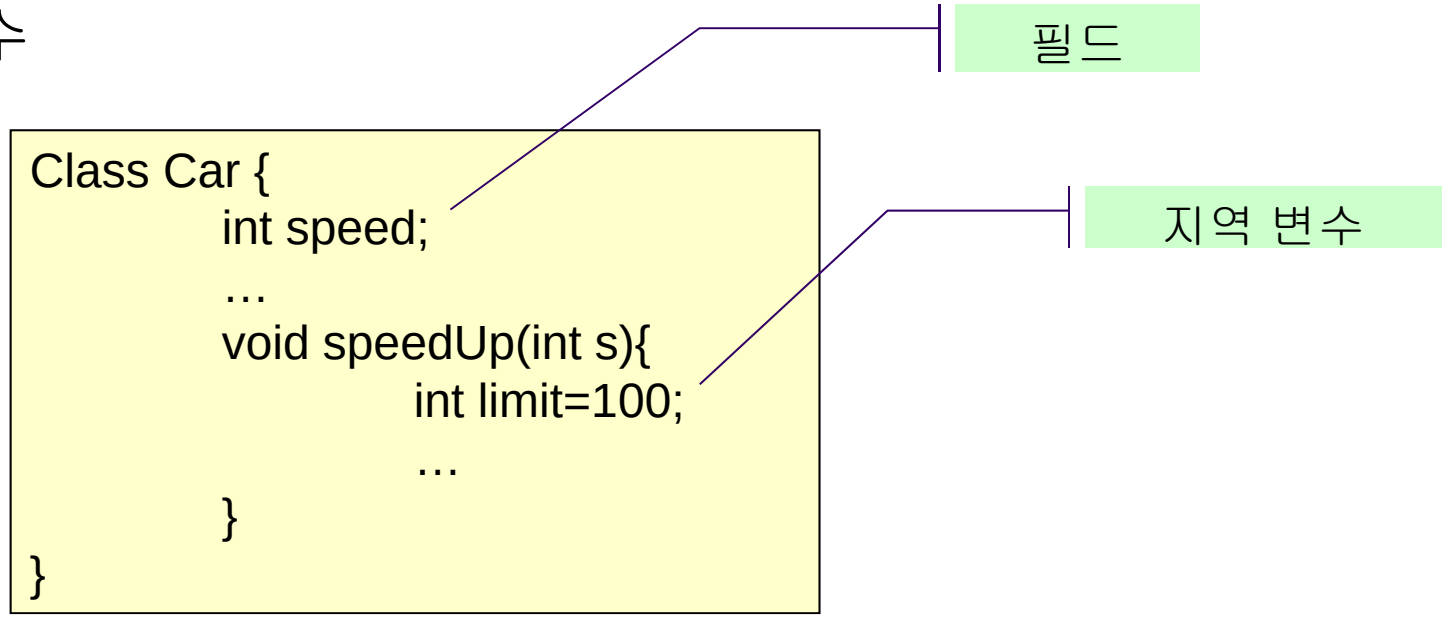
중복 메소드 예제

```
public class CarTest1 {
	public static void main(String[] args) {
		Car myCar = new Car(); // 첫번째 객체 생성

		myCar.setSpeed(100); // 정수 버전 메소드 호출
		myCar.setSpeed(79.2); // 실수 버전 메소드 호출
	}
}
```

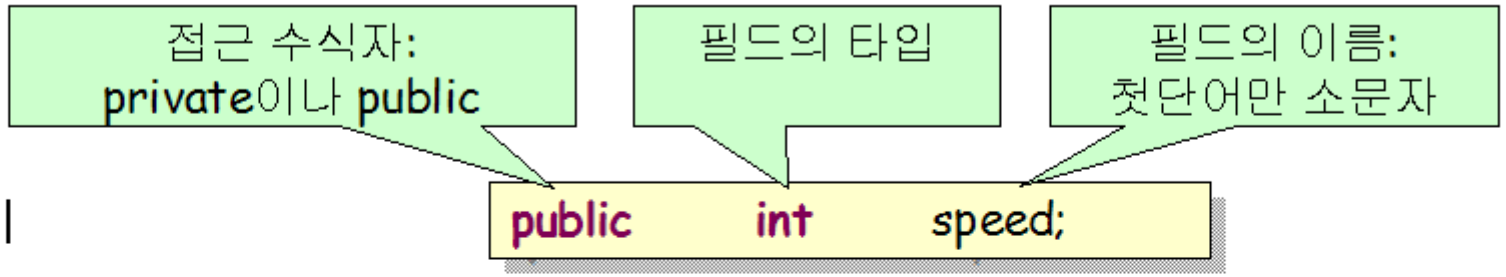
```
정수 버전 호출
실수 버전 호출
```

자바에서의 변수의 종류

- 필드(field) 또는 인스턴스 변수: 클래스 안에서 선언되는 멤버 변수
- 지역 변수(local variable): 메소드나 블록 안에서 선언되는 변수



필드의 선언



필드의 접근 수식자는 어떤 클래스가 필드에 접근할 수 있는지를 표시한다.

- public : 이 필드는 모든 클래스로부터 접근가능하다.
- private : 클래스 내부에서만 접근이 가능하다.

필드의 사용 범위

Date.java

```
public class Date {

        public void printDate() {
                System.out.println(year + "." + month + "." + day);
        }

        public int getDay() {
                return day;
        }

        // 필드 선언
        public int year;
        public String month;
        public int day;
}
```

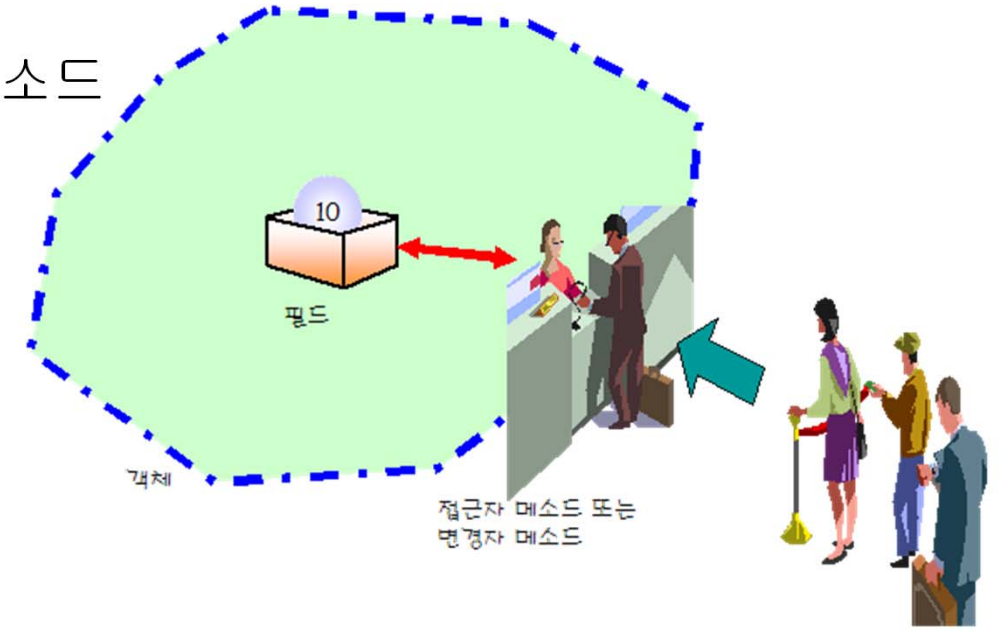
선언 위치와는 상관없이 어디서나 사용이 가능하다.

정보 은닉 (Information Hiding)

- 클래스 메소드를 사용하는 프로그래머는 메소드가 어떻게 구현되었는지 알 필요가 없다.
  - 단지, 해당 메소드가 어떤 기능을 하는가를 알면 된다.
- 정보 은닉:
  - 구현의 세부 사항을 숨기고 메소드의 기능/역할 만을 사용할 수 있도록 설계하는 것
- 그러므로 What과 How를 분리하여 메소드를 설계해야 함.

설정자와 접근자

- 설정자(mutator)
  - 필드의 값을 설정하는 메소드
  - setXXX() 형식
- 접근자(accessor)
  - 필드의 값을 반환하는 메소드
  - getXXX() 형식



설정자와 접근자의 예

CarTest2.java

```
class Car {

        // 필드 선언
        private int speed; // 속도
        private int mileage; // 주행거리
        private String color; // 색상

        // 접근자 선언
        public int getSpeed() {
                return speed;
        }

        // 설정자 선언
        public void setSpeed(int s) {
                speed = s;
        }
```

설정자와 접근자의 사용

```
public class CarTest2 {
        public static void main(String[] args) {
                // 객체 생성
                Car myCar = new Car();

                // 설정자 메소드 호출
                myCar.setSpeed(100);
                myCar.setMileage(0);
                myCar.setColor("red");

                // 접근자 메소드 호출
                System.out.println("현재 자동차의 속도는 " + myCar.getSpeed());
                System.out.println("현재 자동차의 주행거리는 " + myCar.getMileage());
                System.out.println("현재 자동차의 속도는 " + myCar.getColor());
        }
}
```

설정자와 접근자는 왜 사용하는가?

- 설정자에서 매개 변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있다.
- 필요할 때마다 필드값을 계산하여 반환할 수 있다.
- 접근자만을 제공하면 자동적으로 읽기만 가능한 필드를 만들 수 있다.

```
public void setSpeed(int s)
{
        if( s < 0 )
                speed = 0;
        else
                speed = s;
}
```

필드의 초기화

- 필드값은 선언과 동시에 초기화 될 수 있다.

```
public class ClassRoom {
                public static int capacity = 60; //60으로 초기화
                private boolean use = false;  // false로 초기화
}
```

주의

지역 변수는 사용하기 전에 반드시 초기화를 하여야 한다. 자바에서는 지역 변수를 선언하고 초기화하지 않으면 오류가 발생한다.

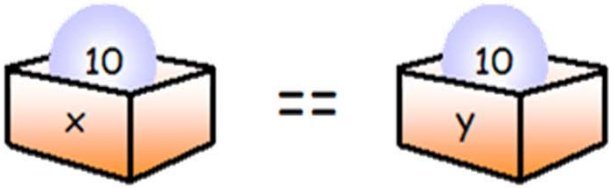
```
class BugProgram {
    ...
    public int getAverage(int x, int y)
    {
        int sum;
        sum += x;          // 초기화 되지 않은 지역 변수를 사용하면 오류!
        sum += y;
        return sum/2;
    }
}
```

실행결과

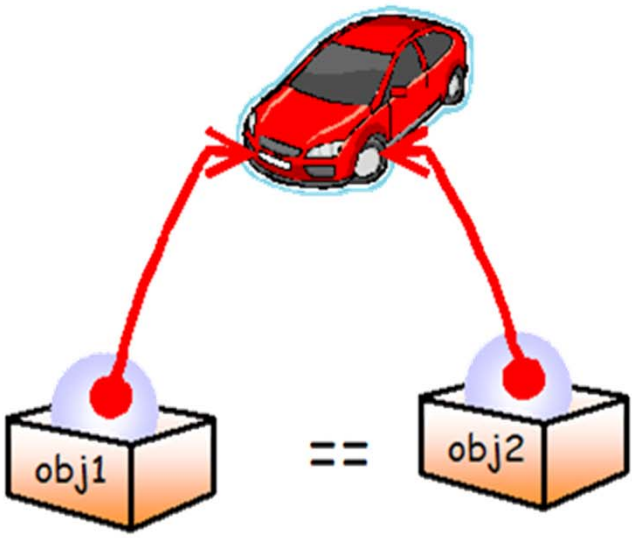
```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The local variable sum may not have been initialized
        at BugProgram.getAverage(BugProgram.java:6)
        at Test.main(Test.java:5)
```

변수와 변수의 비교

- “변수1 == 변수2”의 의미



기초형 변수의 경우
값이 같으면 true

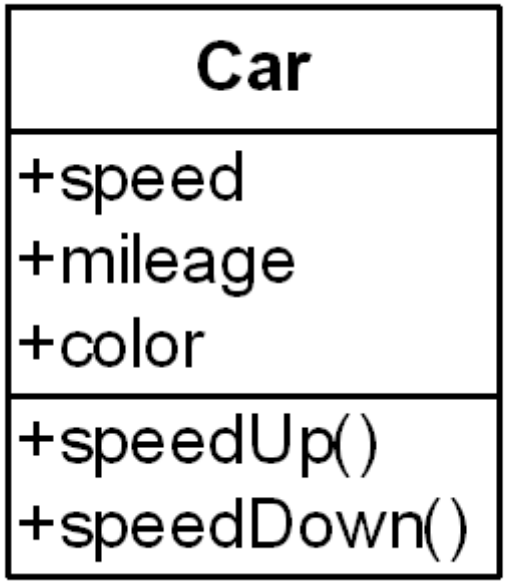


참조형 변수의 경우
같은 객체를 가리키면 true

- 참조형 변수의 경우, 객체의 내용이 같다는 의미가 아니다.

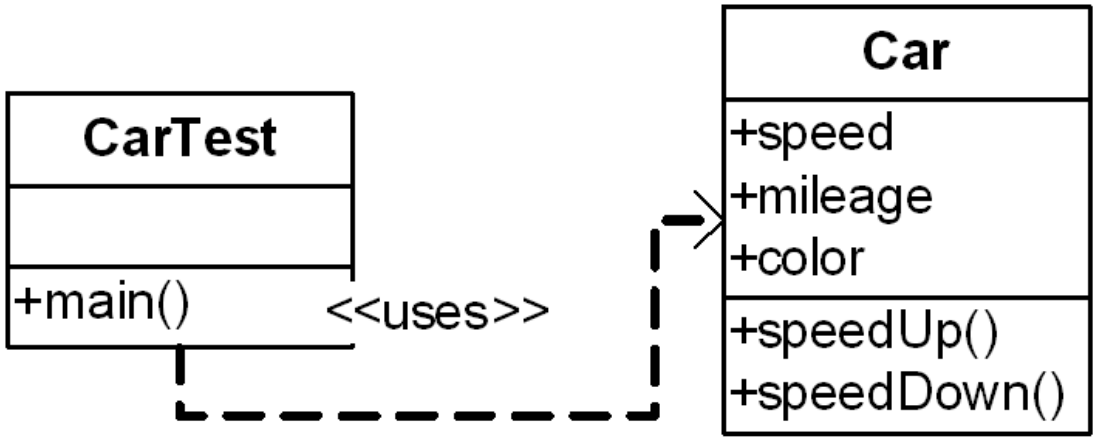
UML

- UML(Unified Modeling Language)



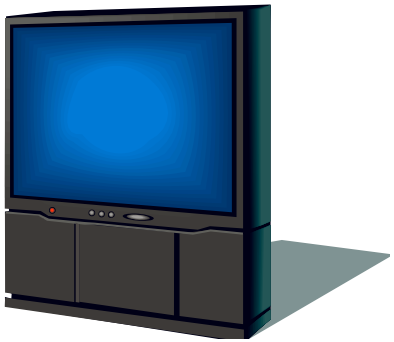
클래스와 클래스의 관계

상속(inheritance)	
인터페이스 상속(interface inheritance)	
의존(dependency)	
집합(aggregation)	
연관(association)	
유향 연관(direct association)	



중간점검

1. TV를 나타내는 클래스를 정의하고 UML의 클래스 다이어그램으로 표현하여 보라.

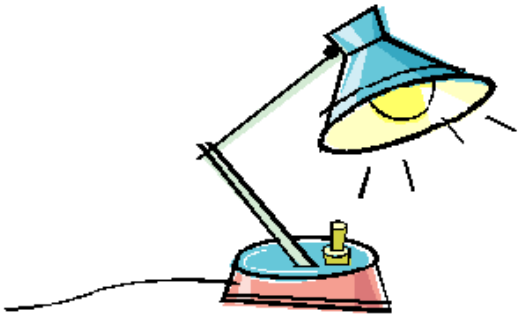


Television
-isOn -volume -channel
+setChannel() +getChannel() +setVolume() +getVolume() +turnOn() +turnOff() +toString()

예제

집에서 사용하는 데스크 램프를 클래스로 작성하여 보면 다음과 같다.

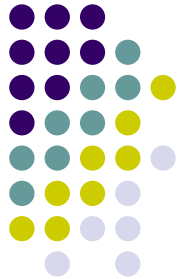
DeskLamp
-isOn : bool
+turnOn() +turnOff()



예제

```
class DeskLamp {
    // 인스턴스 변수 정의
    private boolean isOn; // 켜짐이나 꺼짐과 같은 램프의 상태

    // 메소드 정의
    public void turnOn() // 램프를 켠다.
    {
        isOn = true;
    }

    public void turnOff() // 램프를 끈다.
    {
        isOn = false;
    }

    public String  toString() {
        return "현재 상태는 " + (isOn == true ? "켜짐" : "꺼짐");
    }
}
```

예제



```
public class DeskLampTest {
    public static void main(String[] args) {
        // 역시 객체를 생성하려면 new 예약어를 사용한다.
        DeskLamp myLamp = new DeskLamp();

        // 객체의 메소드를 호출하려면 도트 연산자인 .을 사용한다.
        myLamp.turnOn();
        System.out.println(myLamp);
        myLamp.turnOff();
        System.out.println(myLamp);

    }
}
```

```
현재 상태는 켜짐
현재 상태는 꺼짐
```

생성자(constructor)

- 생성자(contructor): 객체가 생성될 때에 필드에게 초기값을 제공하고 필요한 초기화 절차를 실행하는 메소드
- 클래스와 같은 이름을 갖는 특수 메소드
- new 연산자에 의해 객체가 생성될 때 자동으로 실행된다.
- 주로 객체를 초기화하는데 사용된다.
- 주의 : 생성자는 리턴 타입이 없으며 리턴 타입을 사용하면 컴파일 오류

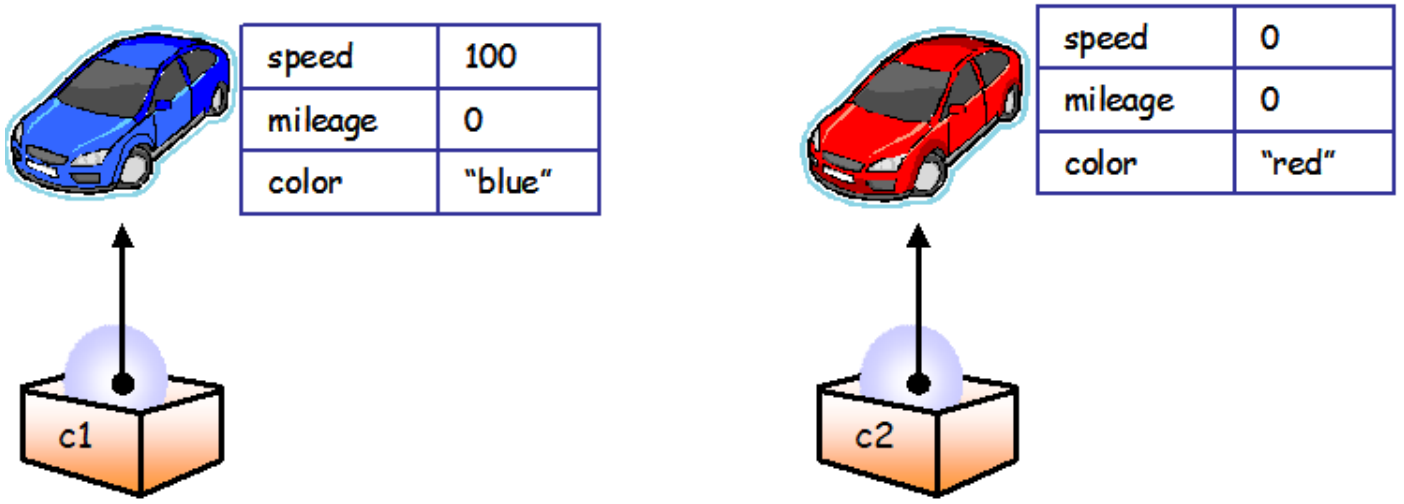
생성자의 예제

CarTest.java

```
class Car {
        public int speed;          // 속도
        public int mileage;                // 주행 거리
        public String color;       // 색상

        // 첫 번째 생성자
        public Car(int s, int m, String c) {
                speed = s;
                mileage = m;
                color = c;
        }
        // 두 번째 생성자
        public Car() {
              speed = mileage = 0;
          color = "red";
        }
}
```

생성자의 예제

```
public class CarTest {
        public static void main(String args[]) {
                Car c1 = new Car(100, 0, "blue"); // 첫 번째 생성자 호출
                Car c2 = new Car(); // 두 번째 생성자 호출
        }
}
```



디폴트 생성자

- 만약 클래스 작성시에 생성자를 하나도 만들지 않는 경우에는 자동적으로 메소드의 몸체 부분이 비어있는 생성자가 만들어진다.

CarTest1.java

```
class Car {
        public int speed;               // 속도
        public int mileage;             // 주행 거리
        public String color; // 색상
}
public class CarTest1 {
        public static void main(String args[]) {
            Car c1 = new Car(); // 디폴트 생성자 호출
        }
}
```

생성자에서 메소드 호출

Car.java

```
public class Car {
      public int speed; // 속도
      public int mileage; // 주행 거리
      public String color; // 색상

      // 첫 번째 생성자
      public Car(int s, int m, String c) {
          speed = s;
          mileage = m;
          color = c;
      }
      // 색상만 주어진 생성자
      public Car(String c) {
          this(0, 0, c);              // 첫 번째 생성자를 호출한다.
      }
}
```

예제: Date 클래스

DateTest.java

```
import java.util.Scanner;

class Date {
        private int year;
        private String month;
        private int day;

        public Date() {                    // 기본 생성자
                month = "1월";
                day = 1;
                year = 2009;
        }

        public Date(int year, String month, int day) {     // 생성자
                setDate(year, month, day);
        }

        public Date(int year) {                            // 생성자
                setDate(year, "1월", 1);
        }
```

예제: Date 클래스

```
        public void setDate(int year, String month, int day) {
                this.month = month;                    // this는 현재 객체를 가리킨다.
                this.day = day;
                this.year = year;
        }
}
public class DateTest {

        public static void main(String[] args) {
                Date date1=new Date(2009,"3월", 2);        // 2009.3.2
                Date date2=new Date(2010);                 // 2010.1.1
                Date date3=new Date();                     // 2009.1.1
        }
}
```

Time 클래스

TimeTest.java


```
class Time {
        private int hour; // 0 - 23
        private int minute; // 0 - 59
        private int second; // 0 - 59

        // 첫 번째 생성자
        public Time() {
                this(0, 0, 0);
        }

        // 두 번째 생성자
        public Time(int h, int m, int s) {
                setTime(h, m, s);
        }

        // 시간 설정 함수
        public void setTime(int h, int m, int s) {
                hour = ((h >= 0 && h < 24) ? h : 0); // 시간 검증
                minute = ((m >= 0 && m < 60) ? m : 0); // 분 검증
                second = ((s >= 0 && s < 60) ? s : 0); // 초 검증
        }
```

```
        // "시:분:초"의 형식으로 출력
        public String toString() {
                return String.format("%02d:%02d:%02d", hour, minute, second);
        }
}
public class TimeTest {
        public static void main(String args[]) {
                // Time 객체를 생성하고 초기화한다.
                Time time = new Time();

                System.out.print("기본 생성자 호출 후 시간:  ");
                System.out.println(time.toString());

                // 두 번째 생성자 호출
                Time time2 = new Time(13, 27, 6);
                System.out.print("두번째 생성자 호출 후 시간: ");
                System.out.println(time2.toString());

                // 올바르지 않은 시간으로 설정해본다.
                Time time3 = new Time(99, 66, 77);
                System.out.print("올바르지 않은 시간 설정 후 시간: ");
                System.out.println(time3.toString());
        }
}
```

실행결과

```
기본 생성자 호출 후 시간:  00:00:00
두번째 생성자 호출 후 시간: 13:27:06
올바르지 않은 시간 설정 후 시간: 00:00:00
```

Circle 클래스

CircleTest.java

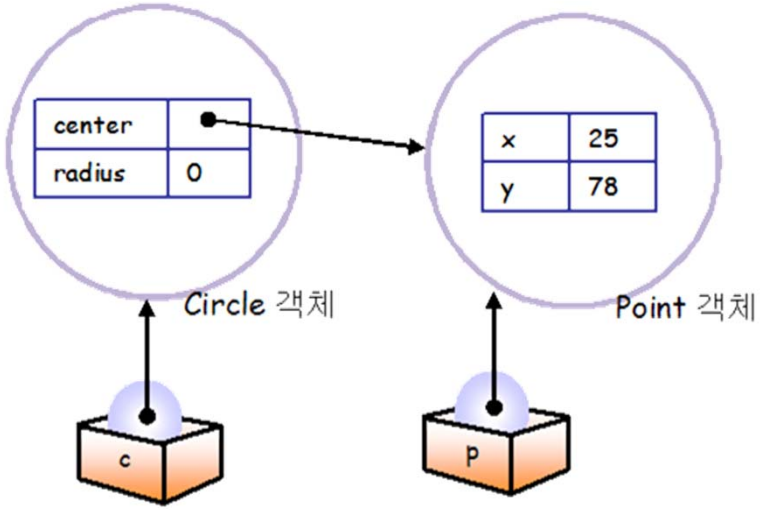
```
class Point {
        public int x;
        public int y;

        // 생성자
        public Point(int a, int b) {
                x = a;
                y = b;
        }
}
```

Circle 클래스

```
class Circle {
        public int radius = 0;
        public Point center; // Point 참조 변수가 필드로 선언되어 있다.

        // 생성자
        public Circle() {
                center = new Point(0, 0);
        }

        public Circle(int r) {
                center = new Point(0, 0);
                radius = r;
        }

        public Circle(Point p, int r) {
                center = p;
                radius = r;
        }
}
```

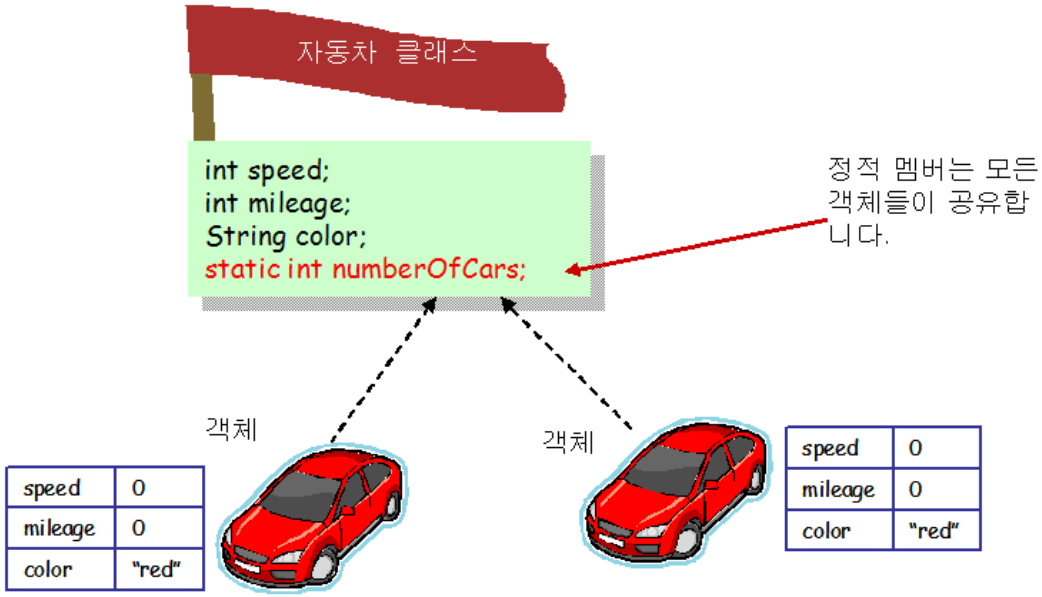
Circle 클래스

```
public class CircleTest {
        public static void main(String args[]) {
                // Circle 객체를 생성하고 초기화한다.
                Point p = new Point(25, 78);
                Circle c = new Circle(p, 10);
        }
}
```

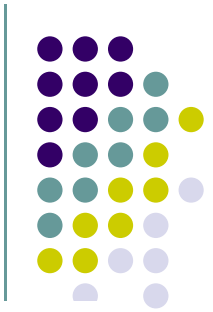


정적 변수 (Static Variable)

- 인스턴스 변수(instance variable): 객체마다 하나씩 있는 변수
- 정적 변수(static variable):
  - 모든 객체를 통틀어서 하나만 있는 변수
  - 클래스 내에 모든 실체(객체)가 공유해서 사용하는 멤버 변수
  - 객체가 아니라 클래스 내에 static 변수를 위한 기억공간이 생성됨.



정적 변수의 예



- *Car.java*

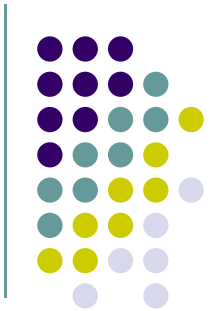
```
public class Car {
        private int speed;
        private int mileage;
        private String color;

        // 자동차의 시리얼 번호
        private int id;

        // 실체화된 Car 객체의 개수를 위한 정적 변수
        private static int numberOfCars = 0;

        public Car(int s, int m, String c) {
                speed = s;
                mileage = m;
                color = c;

                // 자동차의 개수를 증가하고 id 번호를 할당한다.
                id = ++numberOfCars;
        }
}
```

정적 메소드 (Static Method)

- 정적 메소드(static method): 객체를 생성하지 않고 사용할 수 있는 메소드
- (예) Math 클래스에 들어 있는 각종 수학 메소드 들

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

정적 메소드의 예



CarTest3.java

```
class Car {
        private int speed;
        private int mileage;
        private String color;
         // 자동차의 시리얼 번호
        private int id;
         // 실체화된 Car 객체의 개수를 위한 정적 변수
        private static int numberOfCars = 0;

        public Car(int s, int m, String c) {
                speed = s;
                mileage = m;
                color = c;
                 // 자동차의 개수를 증가하고 id 번호를 할당한다.
                id = ++numberOfCars;
        }
       // 정적 메소드
        public static int getNumberOfCars() {
                return numberOfCars; // OK!
        }
}
```

정적 메소드의 예

```
public class CarTest3 {
        public static void main(String args[]) {
                Car c1 = new Car(100, 0, "blue");          // 첫 번째 생성자 호출
                Car c2 = new Car(0, 0, "white"); // 첫 번째 생성자 호출
                int n = Car.getNumberOfCars();   // 정적 메소드 호출
                System.out.println("지금까지 생성된 자동차 수 = " + n);
        }
}
```

실행결과

```
지금까지 생성된 자동차 수 = 2
```

상수

- 공간을 절약하기 위하여 정적 변수로 선언된다.

```
public class Car {
   ...
   static final int MAX_SPEED = 350;
   ...
}
```

EmployeeTest.java

```
import java.util.*;

class Employee {
        private String name;
        private double salary;

        private static int count = 0;      // 정적 변수

       // 생성자
        public Employee(String n, double s) {
              name = n;
              salary = s;
              count++; // 정적 변수인 count를 증가
        }

       // 객체가 소멸될 때 호출된다.
        protected void finalize() {
              count--; // 직원이 하나 줄어드는 것이므로 count를 하나 감소
        }

        // 정적 메소드
        public static int getCount() {
              return count;
        }
}
```

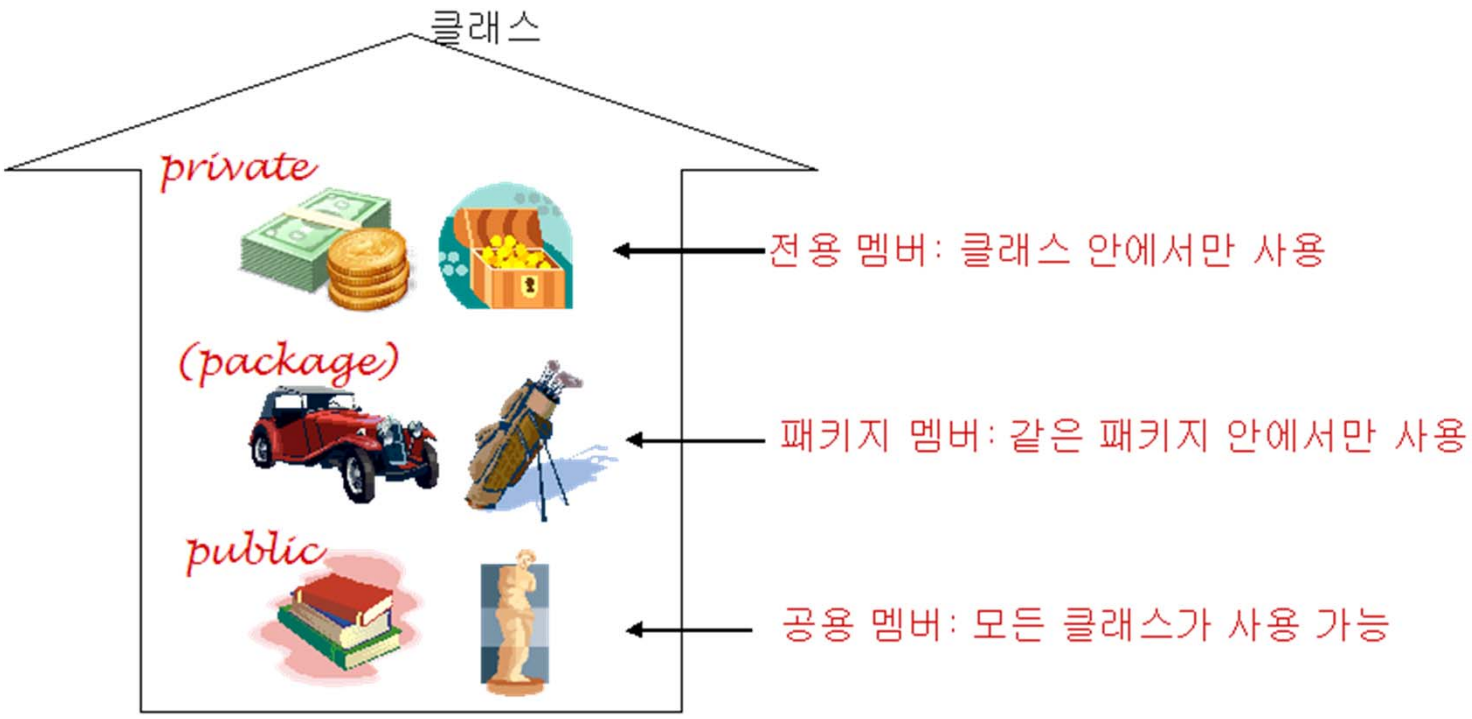
```
public class EmployeeTest {
        public static void main(String[] args) {
                Employee e1,e2,e3;
                e1 = new Employee("김철수", 35000);
                e2 = new Employee("최수철", 50000);
                e3 = new Employee("김철호", 20000);

                int n = Employee.getCount();
                System.out.println("현재의 직원수=" + n);
        }
}
```

실행결과

```
현재의 직원수=3
```

접근 제어

- 접근 제어(access control): 다른 클래스가 특정한 필드나 메소드에 접근하는 것을 제어하는 것

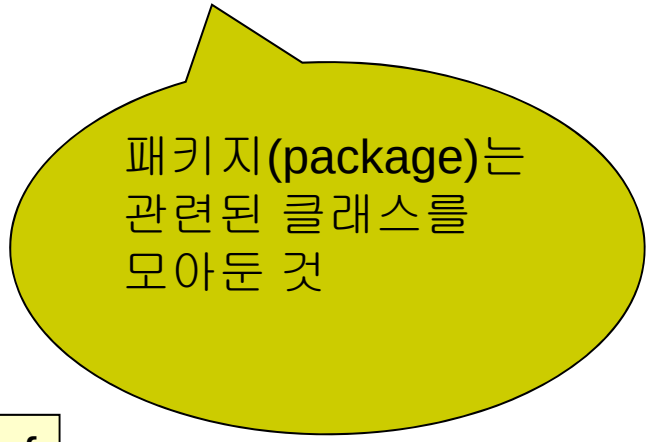


접근 제어의 종류

- 클래스 수준에서의 접근 제어
- 멤버 수준에서의 접근 제어

클래스 수준에서의 접근 제어

- public: 다른 모든 클래스가 사용할 수 있는 공용 클래스
- package: 수식자가 없으면: 같은 패키지 안에 있는 클래스들만이 사용




```
public class myClass {
        ...
}
```

```
class myClass {
        ...
}
```

멤버 수준에서의 접근 제어

분류	접근 지정자	클래스 내부	같은 패키지내의 클래스	다른 모든 클래스
전용 멤버	private	O	X	X
패키지 멤버	없음	O	O	X
공용 멤버	public	O	O	O

EmployeeTest.java

```
import java.util.*;
class Employee {
        private String name;          // private 로 선언
        private int salary;           // private 로 선언
        int age;                      // package 로 선언

        // 생성자
        public Employee(String n, int a, double s) {
            name = n;
            age = a;
            salary = s;
        }
         // 직원의 이름을 반환
        public String getName() {
            return name;
        }
         // 직원의 월급을 반환
        private int getSalary() {     // private 로 선언
            return salary;
        }
         // 직원의 나이를 반환
        int getAge() {                // package로 선언
            return age;
        }
}
```

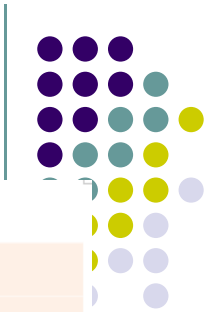
```
public class EmployeeTest {
    public static void main(String[] args) {
        Employee e;
        e = new Employee("홍길동", 0, 3000);
        e.salary = 300; // 오류! private 변수
        e.age = 26;              // 같은 패키지이므로 OK
        int sa = e.getSalary();  // 오류! private 메소드
        String s = e.getName(); // OK!
        int a = e.getAge();      // 같은 패키지이므로 OK
    }
}
```

실행결과

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        The field Employee.salary is not visible
        The method getSalary() from the type Employee is not visible
        at EmployeeTest.main(EmployeeTest.java:8)
```

this

- this는 지금 호출되고 있는 객체 자신을 참조한다.
  - this.member = 10;
- 생성자를 호출할 때도 사용된다.
  - this(10, 20);



PersonTest.java

```
class Person {
        String lastName;
        String firstName;

        String getLastName() {
            return lastName;
        }

        String getFirstName() {
            return firstName;
        }

        public Person(String lastName, String firstName) {
            this.lastName = lastName;          // this는 현재 객체를 가리킨다.
            this.firstName = firstName;        // this는 현재 객체를 가리킨다.
        }

        public StringbuildName() {
            return String.format("%s %s\n", this.getLastName(), getFirstName());   // ①
        }
}
```

```java
public class PersonTest {
    public static void main(String args[]) {
        Person person = new Person("홍", "길동");
        System.out.println(person.buildName());
    }
}
```

클래스와 클래스 간의 관계

- 사용(use): 하나의 클래스가 다른 클래스를 사용한다.
- 집합(has-a): 하나의 클래스가 다른 클래스를 포함한다.
- 상속(is-a): 하나의 클래스가 다른 클래스를 상속한다.

사용 관계

- 클래스 A의 메소드에서 클래스 B의 메소드들을 호출한다.

Complex.java

```
public class Complex {
    private double real;
    private double imag;

    public Complex(double r, double i) {
        real = r;
        imag = i;
    }
    ...
    double getReal() {
        return real;
    }
    double getImag() {
        return imag;
    }
    public Complex add(Complex c) {          // 객체 참조를 매개 변수로 받는다.
        double resultReal = real + c.getReal();
        double resultImag = real + c.getImag();
        return new Complex(resultReal, resultImag);
    }
}
```

집합 관계

- 클래스 A안에 클래스 B가 포함된다.

AlarmClockTest.java

```
class Time {
        private int time;
        private int minute;
        private int second;

        public Time(int t, int m, int s) {
            time = t;
            minute = m;
            second = s;
        }
}
```

```
public class AlarmClock {
    private Time currentTime;
    private Time alarmTime;

    publicAlarmClock(Time a, Time c) {
        alarmTime = a;
        currentTime = c;
    }
}

public class AlarmClockTest {
    public static void main(String args[]) {
        Time alarm = new Time(6, 0, 0);
        Time current = new Time(12, 56, 34);
        AlarmClock c = new AlarmClock(alarm, current);

        System.out.println(c);
    } // end main
} // end class
```

패키지 (Package)

- 패키지는 한 폴더에 그룹으로 함께 묶인 클래스들의 모음
- 폴더의 이름이 패키지의 이름
- 각 클래스들은 분할된 파일로 구성 됨
- 각 클래스는 파일의 시작에 자신이 속한 패키지를 정의하고 있음
- 클래스들은 import 라는 문장을 이용하여 패키지로 묶인 클래스들을 사용

대표적인 **Java API** 패키지

패키지	목적
java.lang	일반적인 지원
java.applet	애플릿
java.awt	그래픽 사용자 인터페이스
javax.swing	추가 그래픽 기능
java.net	네트워크 통신
java.util	유틸리티
javax.xml.parsers	XML 문서 처리

패키지 내의 클래스 사용 방법

- 완전히 지정된 이름(fully qualified name) 사용

```
java.util.Scanner scan = new java.util.Scanner();
```

  - 이 방법은 매번 사용할 때마다 긴 이름을 사용해야 한다는 단점이 있다.

- import 후 클래스 이름만 사용하는 것이다.

```
import java.util.Scanner;
Scanner scan = new Scanner();
```

- 패키지 내의 모든 클래스들을 import

```
import java.util.*;
```