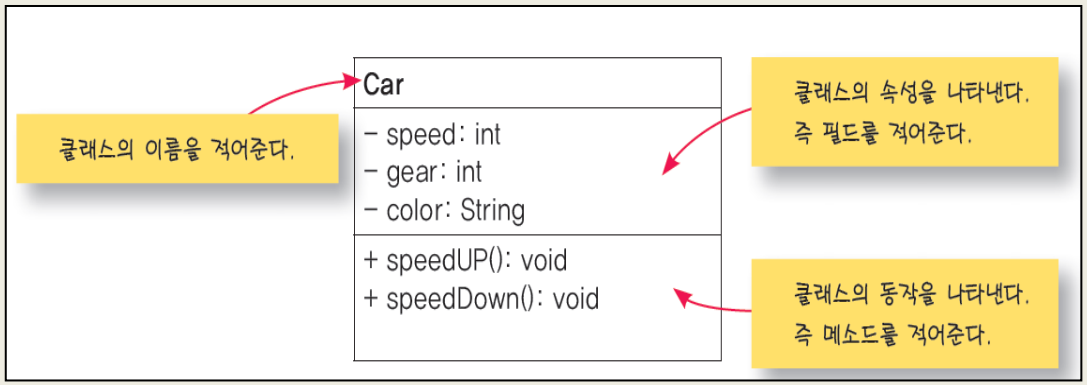
제 4 장
클래스와 객체 Part-2
접근지정자, 생성자

UML-unified modeling language

- UML(Unified Modeling Language):
  - 클래스를 위시하여 객체지향 설계 시에 사용되는 일반적인 모델링 언어
- UML을 사용하면 소프트웨어를 본격적으로 작성하기 전에 구현하고자 하는 시스템을 시각화하여 검토할 수 있다



+	Public
–	Private
#	Protected
/	Derived
~	Package

- 필드나 메소드의 이름 앞에는 가시성 표시자(visibility indicator)가 올 수 있다
- 객체간의 연결을 다양한 관계를 나타내는 화살표를 이용하여 구현한다

클래스 간의 관계

- UML에서 사용되는 화살표의 종류

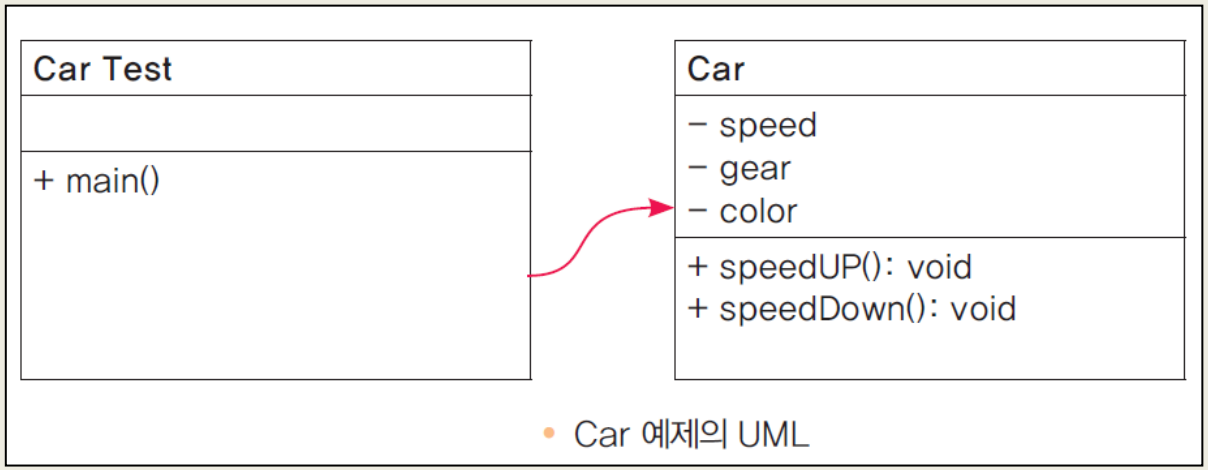
관계	화살표
일반화(generalization), 상속(inheritance)	
구현(realization)	
구성관계(composition)	
집합관계(aggregation)	
유향 연관(direct association)	
양방향 연관(bidirectional association)	
의존(dependency)	

- 필드나 메소드의 이름 앞에는 가시성 표시자(visibility indicator)가 올 수 있다

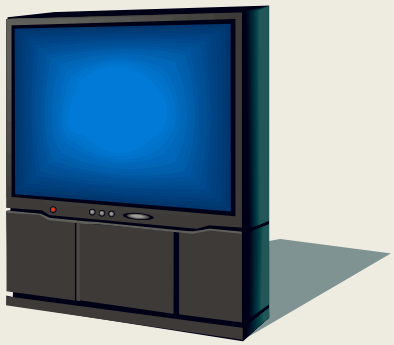
+	Public
–	Private
#	Protected
/	Derived
~	Package

예

- Car 예제를 UML로 그려보면 다음과 같다.

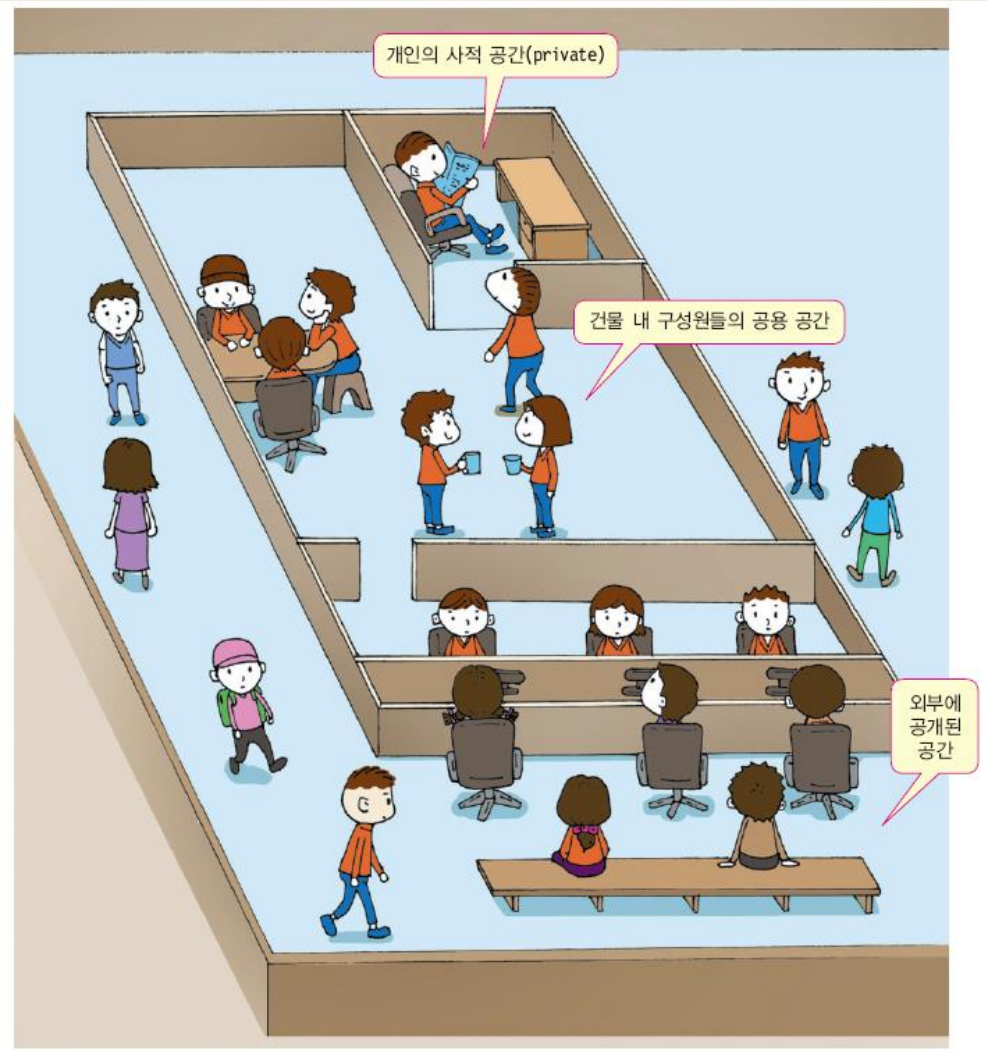


Car 예제의 UML



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

접근 지정자(Access Modifier)



클래스 접근 지정자

- 클래스 앞에 올 수 있는 접근 지정자
- 2가지 → public, 무지정(default, package-private)

  - **public** 접근 지정자
    - 다른 모든 클래스가 접근 가능

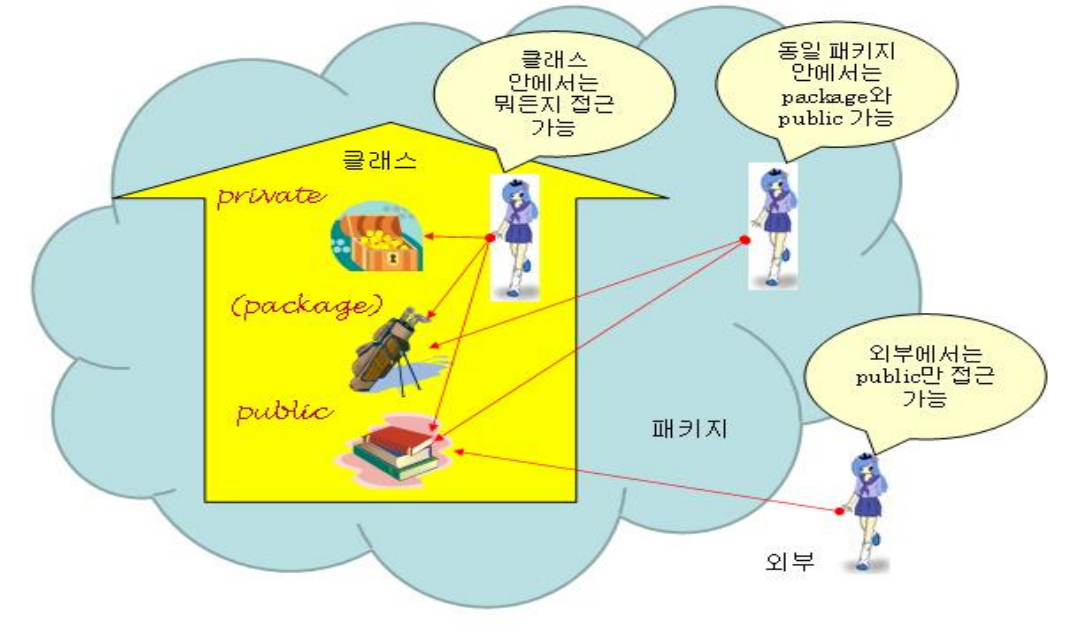
```
public class Person {}
```

  - 접근 지정자 생략 (default 접근 지정자)
    - 또는 package-private라고도 함
    - 같은 패키지 내에 있는 클래스에서만 접근 가능
      - 다른 말로 같은 디렉토리에 있는 클래스끼리 접근 가능

```
class Person {}
```

멤버 접근 지정자

- 클래스 내의 필드나 메소드들을 모든 다른 객체들에게 open 하는 것이 좋은가? -> 많은 문제가 발생
- (예) 국가 기밀 서류를 누구나 보도록 방치하면 어떻게 될까?
- 접근 제어(access control):
  - 다른 클래스가 자신 혹은 다른 클래스의 특정한 필드나 메소드에 접근하는 것을 제어



default (또는 package-private)	•같은 패키지 내에서 접근 가능
public	•패키지 내부, 외부 클래스에서 접근 가능
private	•정의된 클래스 내에서만 접근 가능 •상속 받은 하위 클래스에서도 접근 불가
protected	•같은 패키지 내에서 접근 가능 •다른 패키지에서 접근은 불가하나 상속을 받은 경우 하위 클래스에서는 접근 가능

멤버에 접근하는 클래스	멤버의 접근 지정자			
	default	private	protected	public
같은 패키지의 클래스	O	X	O	O
다른 패키지의 클래스	X	X	X	O

접근 지정자의 이해

public 접근 지정자 사례

```
class A {
    void f() {
        B b = new B();
        b.n = 3;
        b.g();
    }
}
```

패키지 P

```
public class B {
    public int n;
    public void g() {
        n = 5;
    }
}
```

```
class C {
    public void k() {
        B b = new B();
        b.n = 7;
        b.g();
    }
}
```

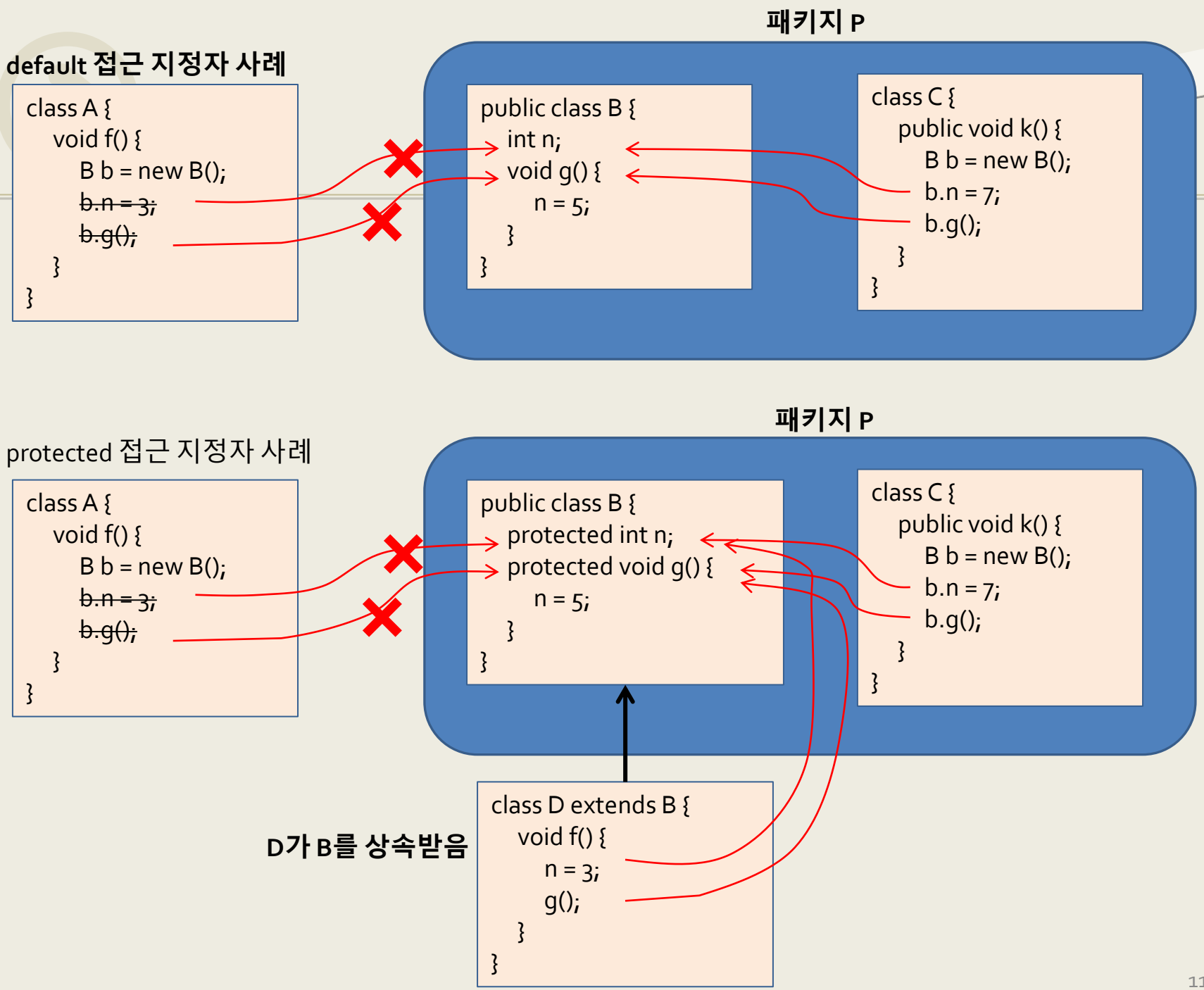
private 접근 지정자 사례

```
class A {
    void f() {
        B b = new B();
        b.n = 3;
        b.g();
    }
}
```

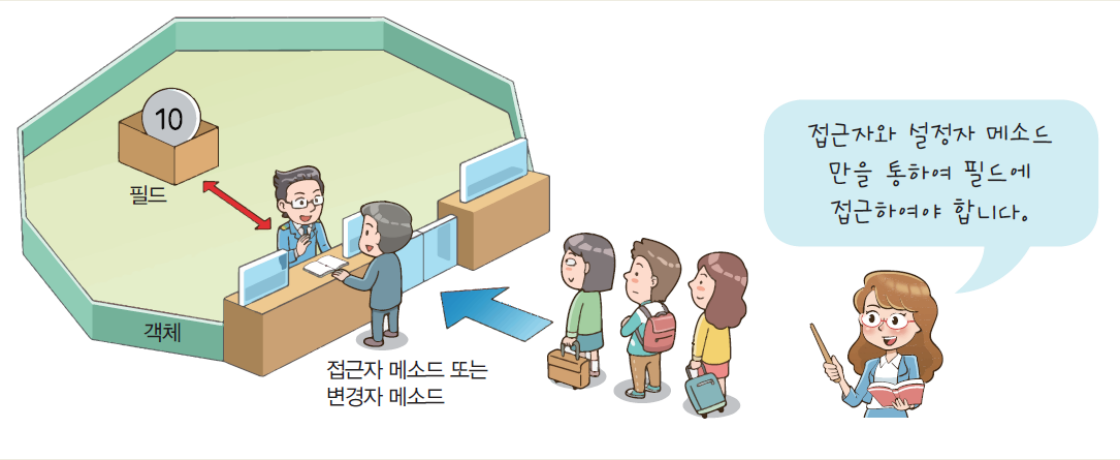
패키지 P

```
public class B {
    private int n;
    private void g() {
        n = 5;
    }
}
```

```
class C {
    public void k() {
        B b = new B();
        b.n = 7;
        b.g();
    }
}
```


패키지 P
default 접근 지정자 사례
class A {
void f() {
B b = new B();
b.n = 3;
b.g();
}
}
public class B {
int n;
void g() {
n = 5;
}
}
class C {
public void k() {
B b = new B();
b.n = 7;
b.g();
}
}
패키지 P
protected 접근 지정자 사례
class A {
void f() {
B b = new B();
b.n = 3;
b.g();
}
}
public class B {
protected int n;
protected void g() {
n = 5;
}
}
class C {
public void k() {
B b = new B();
b.n = 7;
b.g();
}
}
D가 B를 상속받음
class D extends B {
void f() {
n = 3;
g();
}
}

- Private와 같은 접근지정자로 제한을 준 멤버를 접근하고자 하는 경우, 설정자(set---)와 접근자(get---)를 구현하여 사용하도록 함
- 설정자(mutator) : setter 메소드
  - 필드의 값을 설정하는 메소드
  - **setXXX**() 형식
- 접근자(accessor) : getter 메소드
  - 필드의 값을 반환하는 메소드
  - **getXXX()** 형식



예제 : 접근 지정자의 사용

다음의 소스를 컴파일 해보고 오류가 난 이유를 설명하고 오류를 수정하시오.

```
class Sample {
   public int a;
   private int b;
   int c;
}

public class AccessEx {
   public static void main(String[] args) {
      Sample aClass = new Sample();
      aClass.a = 10;
      aClass.b = 10;
      aClass.c = 10;
   }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        The field Sample.b is not visible

        at AccessEx.main(AccessEx.java:11)
```

예제 결과

오류가 수정된 소스

```
class Sample {
   public int a;
   private int b;
   int c;
   public int getB() {
      return b;
   }
   public void setB(int value) {
      b = value;
   }
}
public class AccessEx {
   public static void main(String[] args)
{
      Sample aClass = new Sample();
      aClass.a = 10;
      aClass.setB(10);
      aClass.c = 10;
   }
}
```

- Sample 클래스의 a와 c는 각각 public, default 지정자로 선언이 되었으므로 같은 패키지에 속한 AccessEx 클래스에서 접근이 가능
- b는 private으로 선언이 되었으므로 AccessEx 클래스에서는 접근이 불가능
- private 접근 지정자를 갖는 멤버는 클래스 내부에 get/set 메소드를 만들어서 접근한다.

예 제

```
public class Account {
        private int regNumber;
        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 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() + "입니다.");
        }
}
```

이름은 Tom 통장 잔고는 100000입니다.

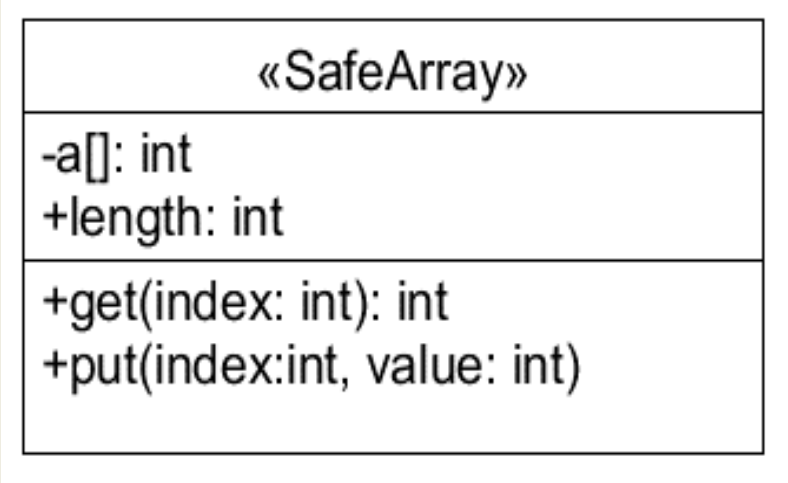
설정자와 접근자 장점

- 접근자와 설정자를 사용해야만 나중에 클래스를 업그레이드할 때 편하다.
- 접근자에서 매개 변수를 통하여 잘못된 값이 넘어오는 경우, 이를 사전에 차단할 수 있다.
- 접근자만을 제공하면 자동적으로 읽기만 가능한 필드를 만들 수 있다.

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

LAB: 설정자와 접근자 기반의 안전한 배열 만들기

- 만약 인덱스가 배열의 크기를 벗어나게 되면 실행 오류가 발생한다. 따라서 실행 오류를 발생하지 않는 안전한 배열을 작성하여 보자



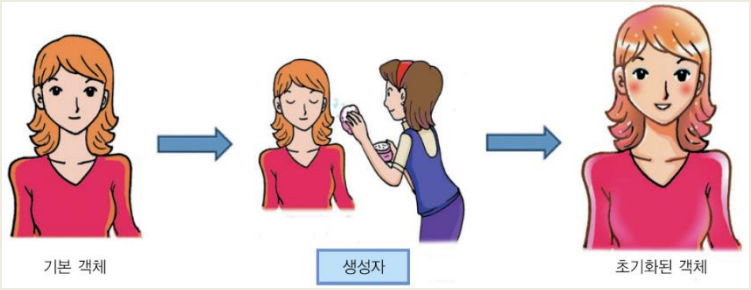
```
public class SafeArray {
        private int a[];
        public int length;
        public SafeArray(int size) {
                a = new int[size];
                length = size;
        }
        public int get(int index) {
                if (index >= 0 && index < length) {
                        return a[index];
                }
                return -1;
        }
        public void put(int index, int value) {
                if (index >= 0 && index < length) {
                        a[index] = value;
                } else
                        System.out.println("잘못된 인덱스 " + index);
        }
}
```

```
public class SafeArrayTest {
        public static void main(String args[]) {
                SafeArray array = new SafeArray(3);

                for (int i = 0; i < (array.length + 1); i++) {
                        array.put(i, i * 10);
                }
        }
}
```

잘못된 인덱스 3

생성자(Constructor)

- 생성자의 특징
  - 생성자는 메소드
  - 생성자의 이름은 클래스 이름과 동일
  - 객체가 생성될 때에 필드에게 초기값을 제공하고 필요한 초기화 절차를 실행하는 메소드
  - 생성자는 new를 통해 객체를 생성할 때만 호출됨
  - 생성자도 오버로딩 가능
  - 생성자는 리턴 타입을 지정할 수 없다.
  - 생성자는 하나 이상 정의되어야 함
    - 개발자가 생성자를 하나도 정의하지 않으면 자동으로 기본 생성자가 정의됨
      - 컴파일러에 의해 자동 생성
      - 기본 생성자를 디폴트 생성자(default constructor)라고도 함



생성자 정의와 생성자 호출

생성자 명은
클래스 이름과 동일

생성자는 리턴 타입 없음

생성자 오버로딩 가능

new 문장과
일치하는 생성자 호출

```
class Sample {
   int id;
   public  Sample(int x) {
      this.id = x;
   }
   public  Sample() {
      this.id = 0;
   }

   public  void set(int x) {this.id = x;}
   public  int get() {return this.id;}

   public static void main(String [] args) {
      Sample ob1 = new Sample(3);
      Sample ob2 = new Sample();
      Sample s; // 생성자 호출하지 않음
   }
}
```

예제 : 생성자 정의와 호출

클래스 Book을 String title, String author, int ISBN의 3개의 필드를 갖도록 정의하라.

```
public class Book {
   String title;
   String author;
   int ISBN;
   public Book(String title, String author, int ISBN) {
      this.title = title;
      this.author = author;
      this.ISBN = ISBN;
   }

   public static void main(String [] args) {
      Book javaBook = new Book("Java JDK", "황기태", 3333);
   }
}
```

기본 생성자

- 기본 생성자(default constructor)
  - 클래스에 생성자가 하나도 정의되지 않은 경우
  - 컴파일러에 의해 자동으로 생성
    - 인자 없는 생성자
    - 아무 작업 없이 단순 리턴
- 디폴트 생성자라고도 부름

```
class DefaultConstructor{
    int x;
    public  void setX(int x) {this.x = x;}
    public  int getX() {return x;}

    public static void main(String [] args) {
        DefaultConstructor p= new DefaultConstructor();
        p.setX(3);
    }
}
```

개발자가 작성한 코드

```
class DefaultConstructor{
    int x;
    public  void setX(int x) {this.x = x;}
    public  int getX() {return x;}

    public DefaultConstructor() { }

    public static void main(String [] args) {
        DefaultConstructor p= new DefaultConstructor();
        p.setX(3);
    }
}
```

컴파일러에 의해 자동 삽입된 기본 생성자

컴파일러가 자동으로 기본 생성자를 삽입한 코드

기본 생성자가 자동 생성되지 않는 경우

- 클래스에 생성자가 하나라도 존재하면 자동으로 기본 생성자가 생성되지 않음

컴파일러가 기본 생성자를 자동 생성하지 않음

public DefaultConstructor() { }

```
class DefaultConstructor{
   int x;
   public  void setX(int x) {this.x = x;}
   public  int getX() {return x;}

   public DefaultConstructor(int x) {
      this.x = x;
   }
   public static void main(String [] args) {
      DefaultConstructor p1= new DefaultConstructor(3);
      int n = p1.getX();

      DefaultConstructor p2= new DefaultConstructor();
      p2.setX(5);
   }
}
```

컴파일 오류.
해당하는 생성자가 없음 !!!

this 레퍼런스

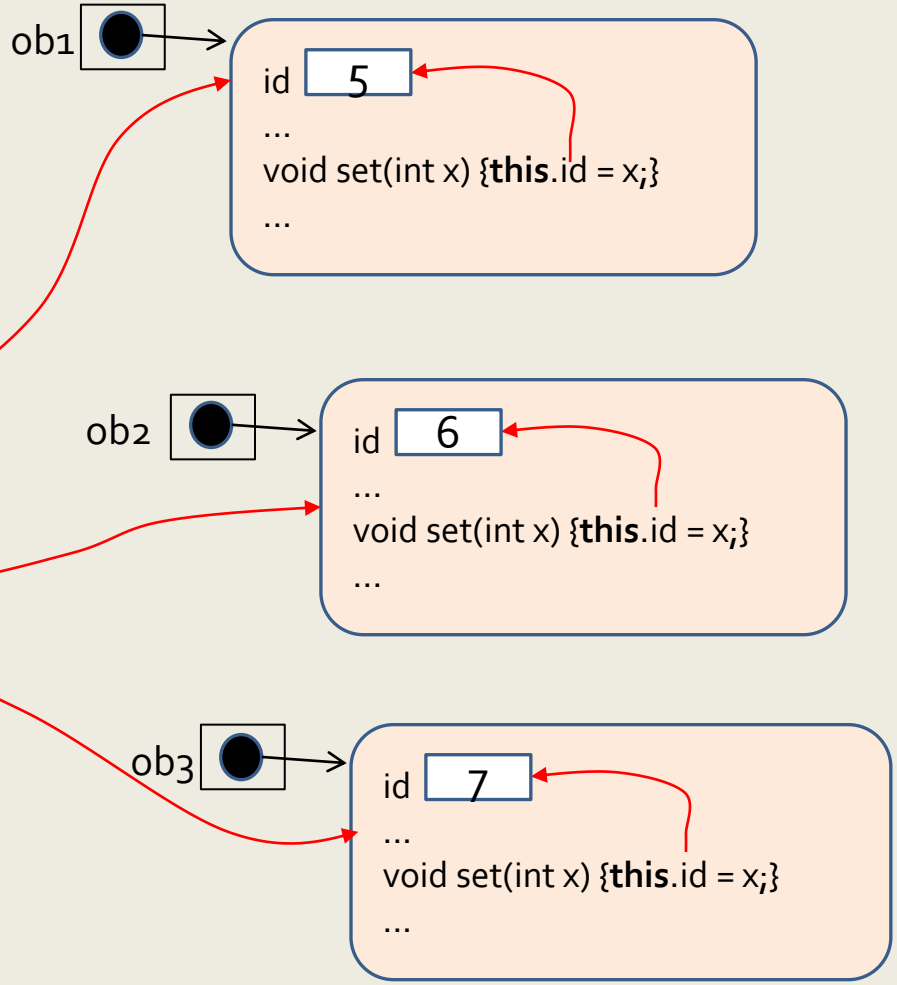
- this의 기초 개념
  - 현재 객체 자기 자신을 가리킨다.
    - 자기 자신에 대한 레퍼런스
    - 같은 클래스 내에서 클래스 멤버, 변수를 접근할 때 객체 이름이 없으면 묵시적으로 this로 가정
- this의 필요성
  - 객체의 멤버 변수와 메소드 변수의 이름이 같은 경우
  - 객체 자신을 메소드에 전달 또는 반환할 때

```
class Samp {
    int id;
    public Samp(int id) {this.id = id; }
    public void set(int id) {this.id = id; }
    public int get() {return id; }
}
```

- 메소드나 생성자에서 this는 현재 객체를 나타낸다

```
public class Point {
    public int x = 0;
    public int y = 0;

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

```
class Samp {
    int id;
    public Samp(int x) {this.id = x;}
    public  void set(int x) {this.id = x;}
    public  int get() {return this.id;}

    public static void main(String [] args) {
      Samp ob1 = new Samp(3);
      Samp ob2 = new Samp(3);
      Samp ob3 = new Samp(3);

       ob1.set(5);
       ob2.set(6);
       ob3.set(7);
    }
}
```



필드 초기화 방법

- 필드 선언시 초기화

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

- 초기화 블록(instance initializer block)과 정적(static) 초기화 블록

```
public class Car {
        int speed;
        Car() {
                System.out.println("속도는 " + speed);
        }
        {
                speed = 100;
        }
        public static void main(String args[]) {
                Car c1 = new Car();
                Car c2 = new Car();
        }
}
```

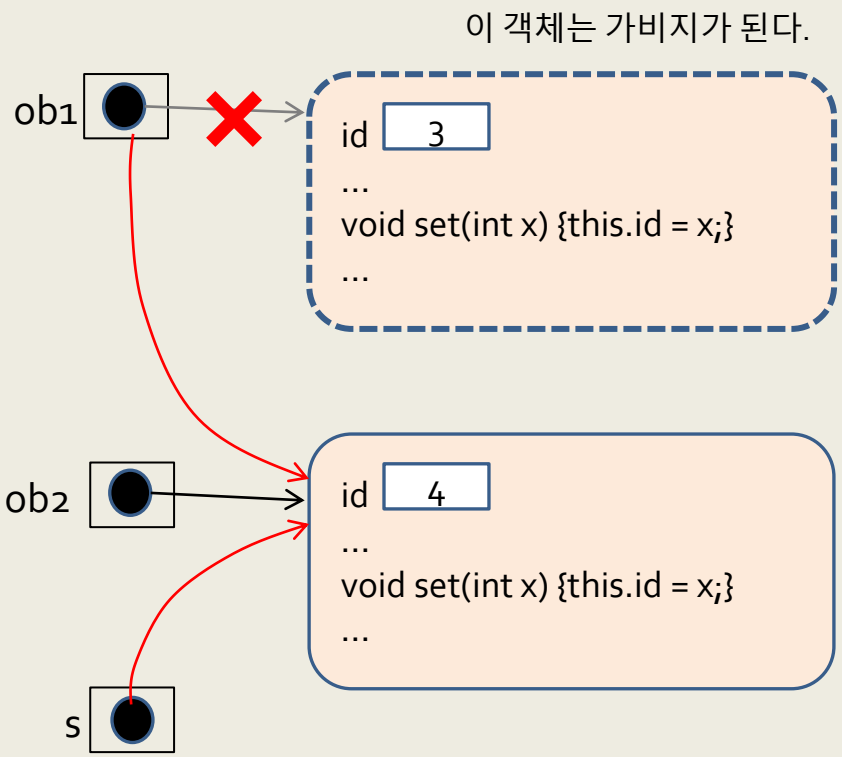
예 제

```
Class InstExam {
        static      int      a = 0;                  // (1) a=0
                    int      b = a++;                // (6) b=17
        {
                    a = a + 2;                       // (5) a=16
        }
        static      int      c = a;                  // (2) c=0
        static {
                    a = a + 5;                       // (3) a = 5
        }
        int         d = a;                           // (7) d=16
        public Test() {
                    System.out.println("a =  " + a);
                    System.out.println("b =  " + b);
                    System.out.println("c =  " + c);
                    System.out.println("d =  " + d);
        }
        static {
                    a = a + 9;                       // (4) a=14
        }
}
```

객체의 치환

* 객체의 치환은 객체가 복사되는 것이 아니며 레퍼런스가 복사된다.

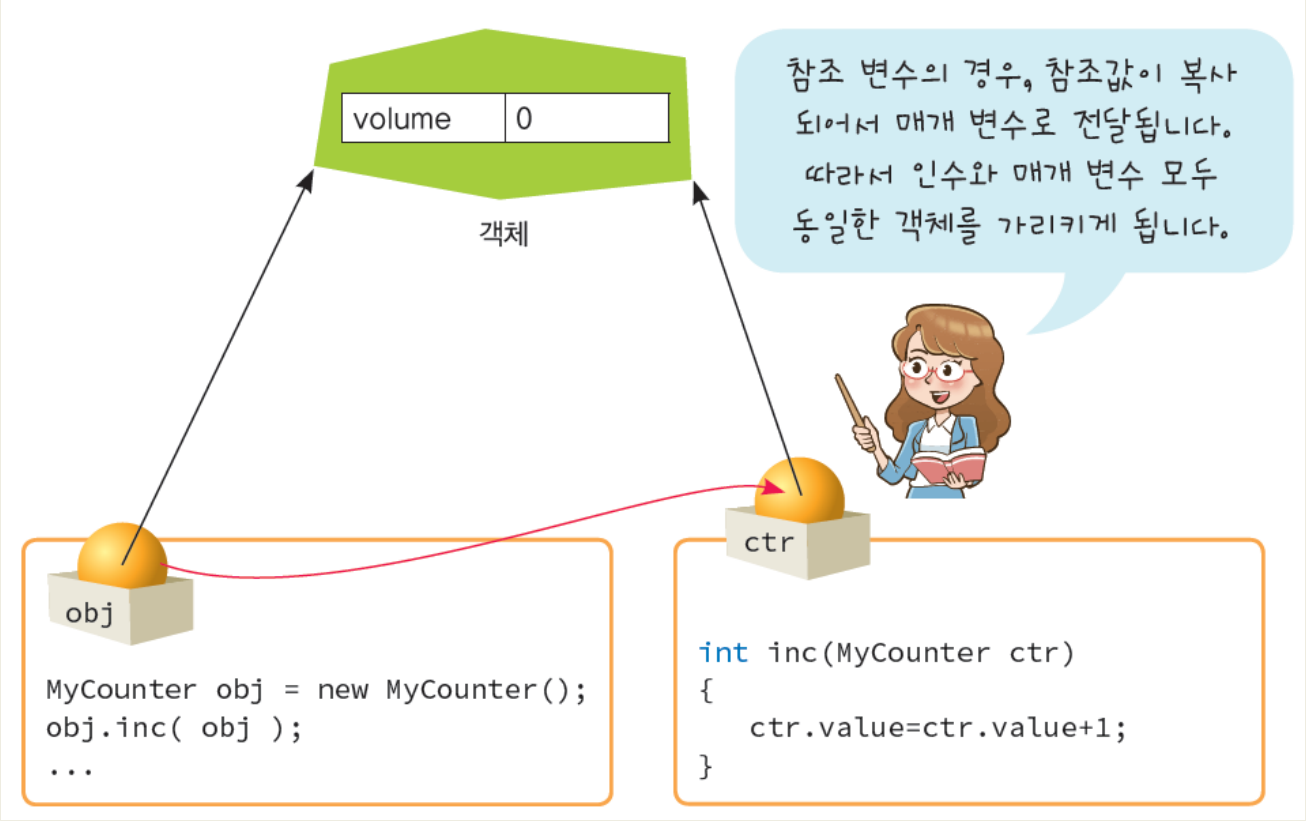
```
class Samp {
    int id;
    public Samp(int x) {this.id = x;}
    public  void set(int x) {this.id = x;}
    public  int get() {return this.id;}

    public static void main(String [] args) {
      Samp ob1 = new Samp(3);
      Samp ob2 = new Samp(4);
      Samp s;

        s = ob2;
        ob1 = ob2; // 객체의 치환
        System.out.println("ob1.id="+ob1.id);
        System.out.println("ob2.id="+ob2.id);
    }
}
```




```
ob1.id=4
ob2.id=4
```

메소드로 객체가 전달되는 경우

- 객체를 메소드로 전달하게 되면 객체가 복사되어 전달되는 것이 아니고 참조 변수의 값이 복사되어서 전달된다.



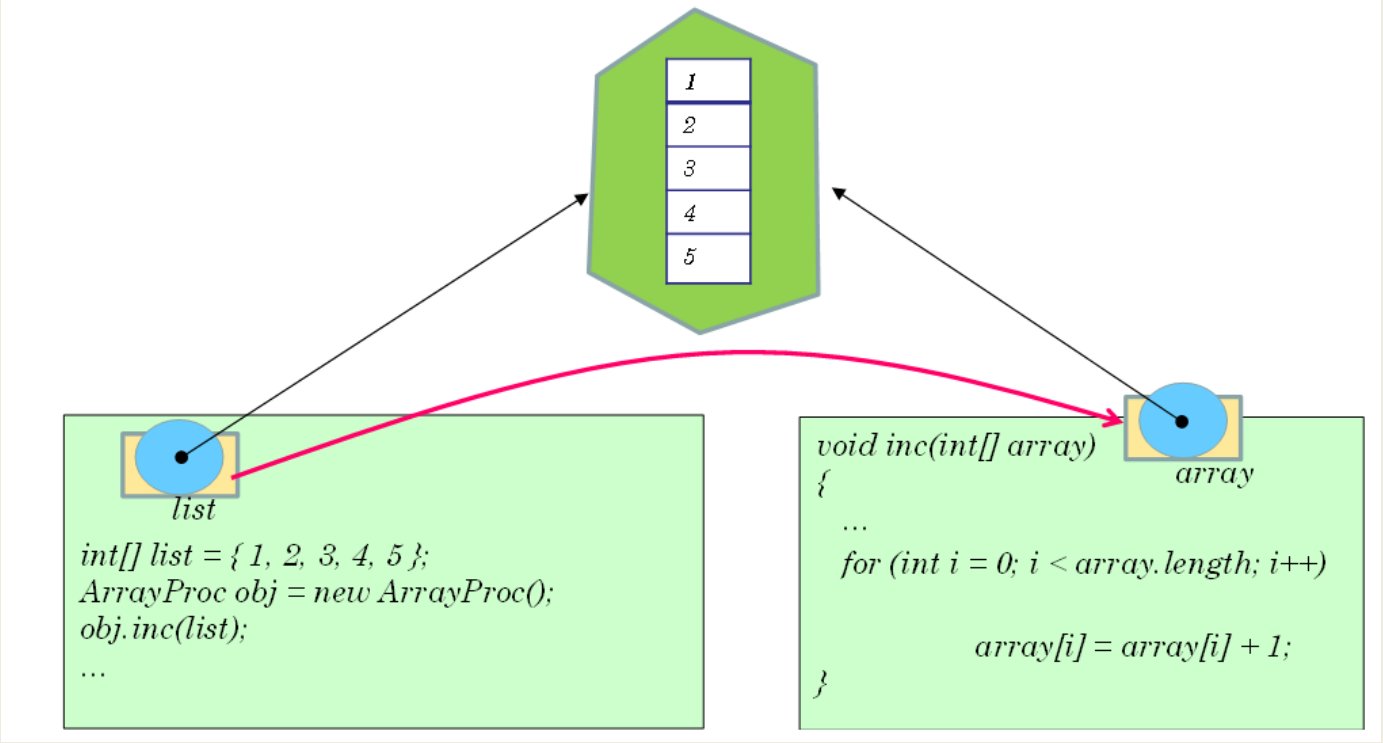
```
class MyCounter {
        int value = 0;
        void inc(MyCounter ctr) {
                ctr.value = ctr.value + 1;
        }
}
```

```
public class MyCounterTest2 {
        public static void main(String args[]) {
                MyCounter obj = new MyCounter();

                System.out.println("obj.value = " + obj.value);
                obj.inc(obj);
                System.out.println("obj.value = " + obj.value);
        }
}
```

```
obj.value = 0
obj.value = 1
```

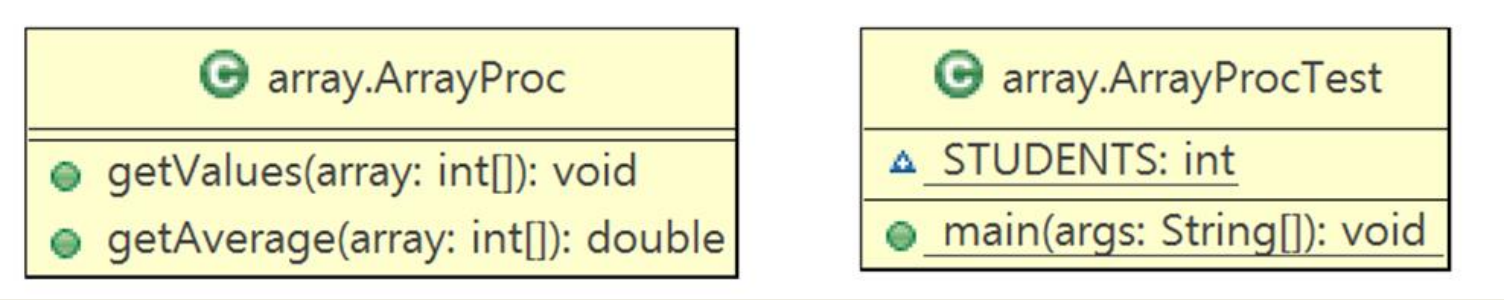
메소드로 배열이 전달되는 경우

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



예 제

- 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
```

```java
import java.util.Scanner;
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;
        }
}
```

```java
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));
        }
}
```

박스크기 구별: 객체를 반환하는 메소드 예

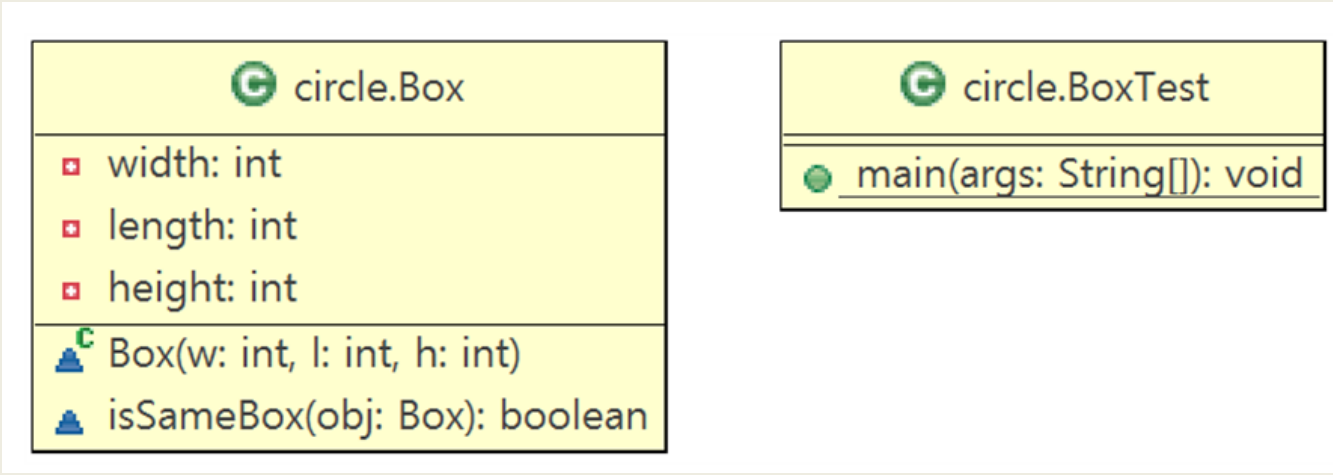
```
public class Box {
        int width, length, height;
        int volume;

        Box(int w, int l, int h) {
                width = w;
                length = l;
                height = h;
                volume = w * l * h;
        }

        Box whosLargest(Box box1, Box box2) {
                if (box1.volume > box2.volume)
                        return box1;
                else
                        return box2;
        }
 }
```

```
public class BoxTest {
        public static void main(String args[]) {
                Box obj1 = new Box(10, 20, 50);
                Box obj2 = new Box(10, 30, 30);

                Box largest = obj1.whosLargest(obj1, obj2);
                System.out.println("(" + largest.width + "," +
largest.length + ","
                                        + largest.height + ")");
        }
}
```

```
(10,20,50)
```

같은 크기의 Box인지 확인하기

- 2개의 박스가 같은 치수인지를 확인하는 메소드 isSameBox()를 작성하여 보자. 만약 박스의 크기가 같으면 true를 반환하고 크기가 다르면 false를 반환한다. isSameBox()의 매개 변수는 객체 참조 변수가 된다.



```
public class Box {
        int width, length, height;

        Box(int w, int l, int h) {
                width = w;
                length = l;
                height = h;
        }

        boolean isSameBox(Box obj) {
                if ((obj.width == width) & (obj.length == length)
                                & (obj.height == height))
                        return true;
                else
                        return false;
        }
}
```

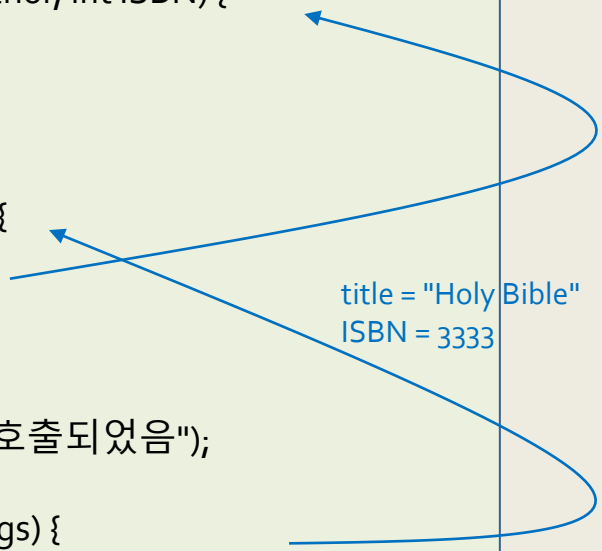
```
public class BoxTest {
        public static void main(String args[]) {
                Box obj1 = new Box(10, 20, 50);
                Box obj2 = new Box(10, 20, 50);

                System.out.println("obj1 == obj2 : " + obj1.isSameBox(obj2));
        }
}
```

obj1 == obj2 : true

this(), 생성자에서 다른 생성자 호출

- this()
  - 같은 클래스의 다른 생성자 호출 시 사용
  - 생성자 내에서만 사용 가능
    - 다른 메소드에서는 사용 불가
  - 반드시 생성자 코드의 제일 처음에 수행

```
public class Book {
    String title;
    String author;
    int ISBN;

    public Book(String title, String author, int ISBN) {
    this.title = title;
    this.author = author;
    this.ISBN = ISBN;
    }
    public Book(String title, int ISBN) {
        this(title, "Anonymous", ISBN);
    }
    public Book() {
        this(null, null, 0);
        System.out.println("생성자가 호출되었음");
    }
    public static void main(String [] args) {
        Book javaBook = new Book("Java JDK", "황기태", 3333);
        Book holyBible = new Book("Holy Bible", 1);
        Book emptyBook = new Book();
    }
}
```



this() 사용 실패 예와 생성자 오버로딩

```
public Book() {
   System.out.println("생성자가 호출되었음");
   this(null, null, 0); // 생성자의 첫 번째 문장이 아니기 때문에 컴파일 오류
}
```

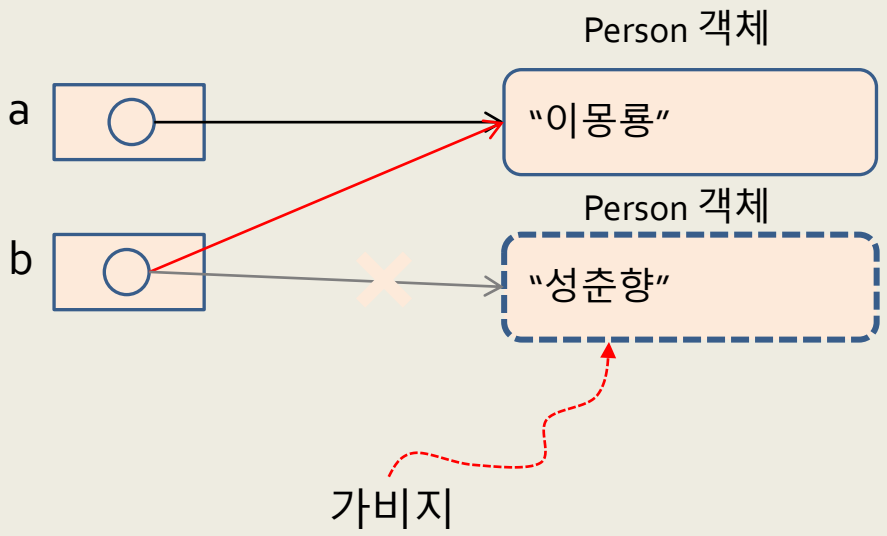
```
public class Rectangle {
        private int x, y;
        private int width, height;

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

객체의 소멸과 가비지

- 객체 소멸
  - new에 의해 생성된 객체 메모리를 자바 가상 기계에게 되돌려 주는 행위
  - 가용 메모리에 포함시킴
- 자바에서 객체 삭제 기능 없음
  - 개발자에게는 매우 다행스러운 기능
    - C/C++에서는 할당 받은 객체를 개발자가 프로그램 내에서 삭제해야 함
- 가비지(garbage)
  - 객체에 대한 레퍼런스가 없어지면 객체는 가비지(garbage)가 됨
  - 자바 가상 기계의 가비지 컬렉터가 가비지 메모리를 반환

가비지 사례

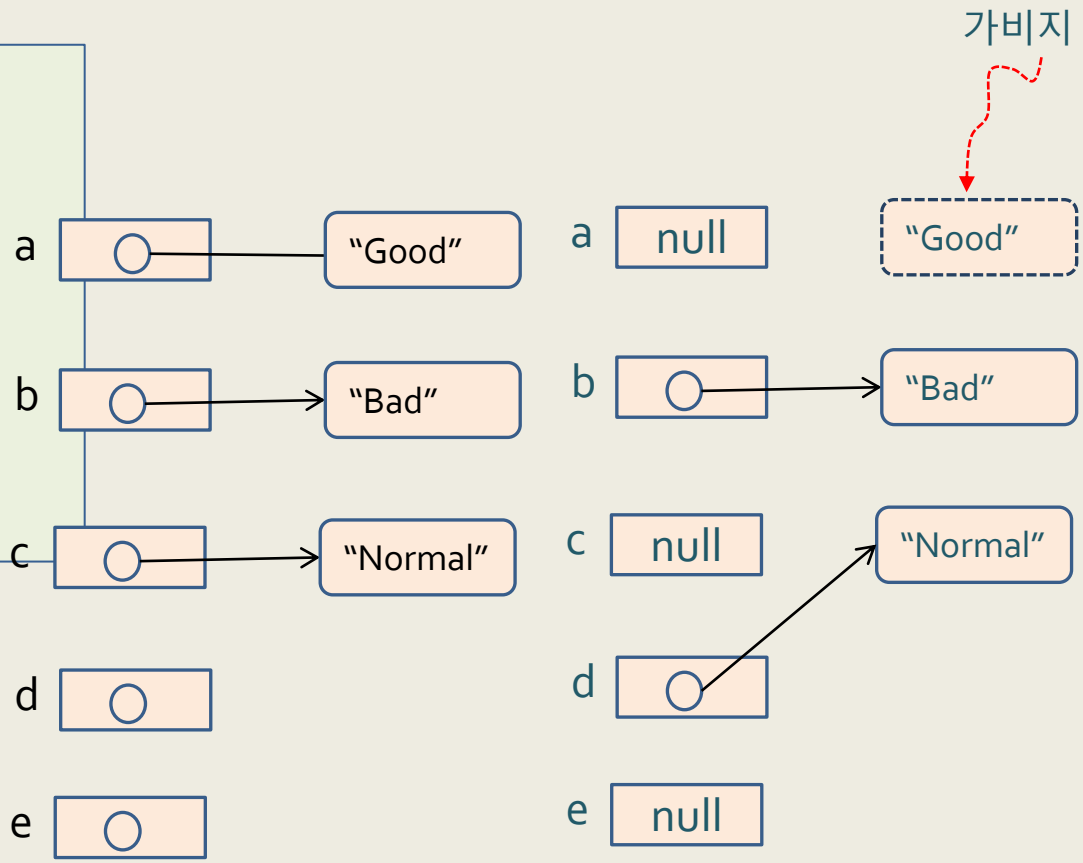
```
Person a, b;
a = new Person("이몽룡");
b = new Person("성춘향");
b = a; // b가 가리키던 객체는가비지가 됨
```



예제 : 가비지 발생

다음 소스에서 언제 가비지가 발생하는지 설명하라.

```
public class GarbageEx {
    public static void main(String[] args) {
        String a = new String("Good");
        String b = new String("Bad");
        String c = new String("Normal");
        String d, e;
        a = null;
        d = c;
        c = null;
    }
}
```



(a) 초기 객체 생성시

(b) 코드 전체 실행 후

가비지(Garbage) 컬렉션

- 가비지 컬렉션
  - 자바에서는 가비지들을 자동 회수, 가용 메모리 공간으로 이동하는 행위
  - 자바 가상 기계 내에 포함된 가비지 컬렉터(garbage collector)에 의해 자동 수행
- 개발자에 의한 강제 가비지 컬렉션
  - System 또는 Runtime 객체의 gc() 메소드 호출

```
System.gc(); // 가비지 컬렉션 작동 요청
```

  - 자바 가상 기계에 강력한 가비지 컬렉션을 요청.
  - 그러나 자바 가상 기계가 가비지 컬렉션 시점을 전적으로 판단