JAVA

- 객체지향

클래스와 인스턴스

클래스의 정의와 인스턴스의 생성

객체지향언어의 특징

▶ 기존의 프로그래밍언어와 크게 다르지 않다.
- 기존의 프로그래밍 언어에 몇 가지 규칙을 추가한 것일 뿐이다.

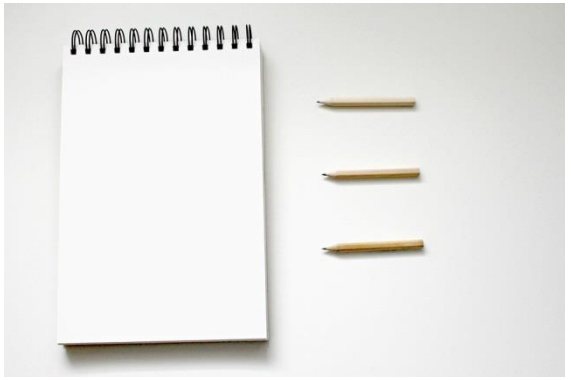
▶ 코드의 재 사용성이 높다.
- 새로운 코드를 작성할 때 기존의 코드를 이용해서 쉽게 작성할 수 있다.

▶ 코드의 관리가 쉬워졌다.
- 코드간의 관계를 맺어줌으로써 보다 적은 노력으로 코드변경이 가능하다.

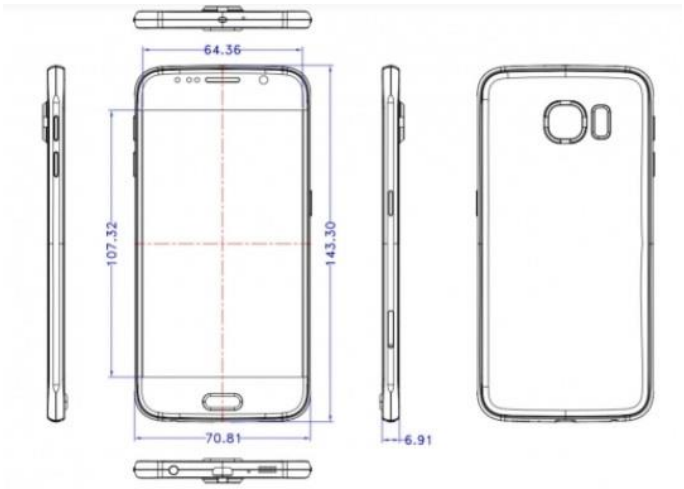
▶ 신뢰성이 높은 프로그램의 개발을 가능하게 한다.
- 제어자와 메서드를 이용해서 데이터를 보호하고, 코드의 중복을 제거하여 코드의 불일치로 인한 오류를 방지할 수 있다.

우리 주변의 객체를

찾아 봅시다,



클래스와 객체

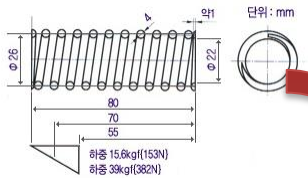


속성과 기능을 가진다.

속성
개별적으로 가지는 값 또는 데이터

기능
속성을 변경하거나 제품의 고유한 기능

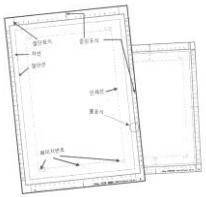
클래스와 객체

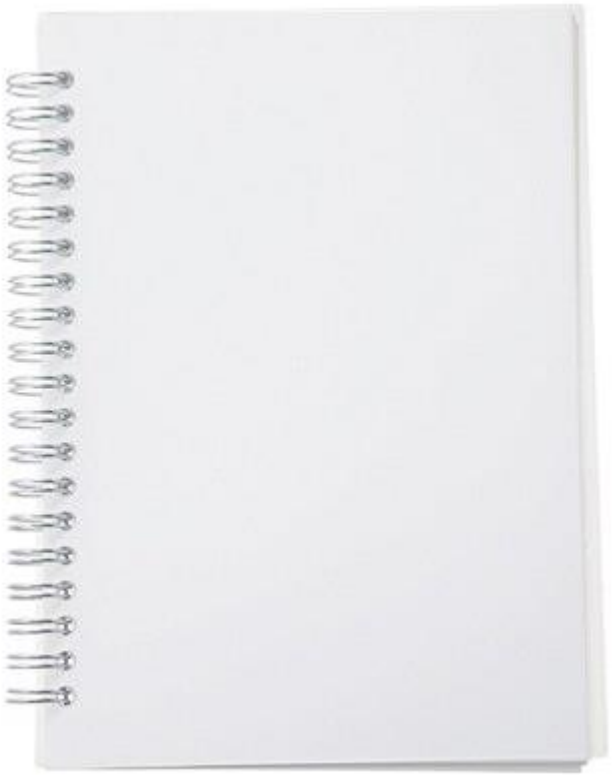


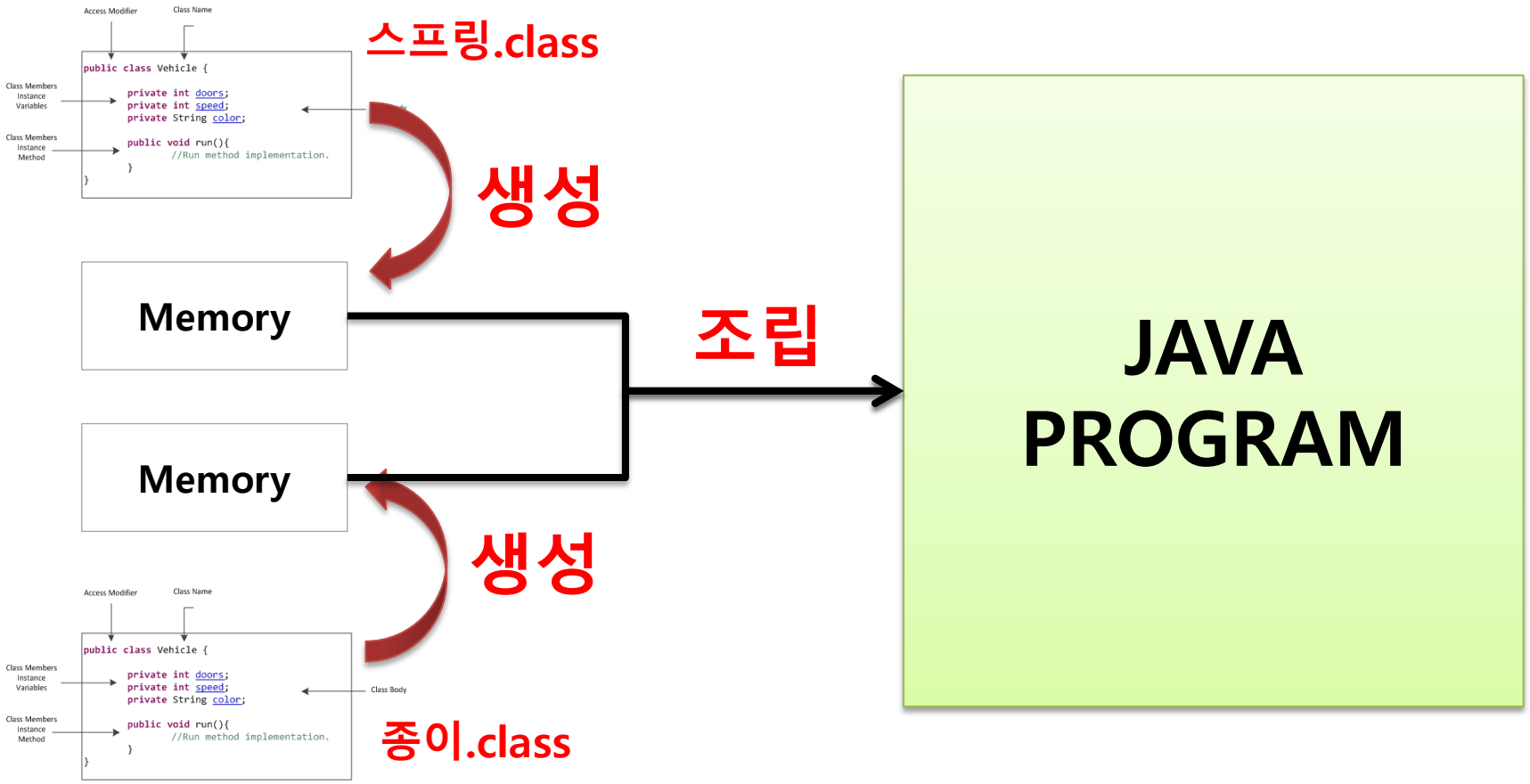
생성

조립

생성






Access Modifier
Class Name
public class Vehicle {
Class Members Instance Variables
private int doors;
private int speed;
private String color;
Class Members Instance Method
public void run(){
//Run method implementation.
}
}
스프링.class
생성
Memory
조립
Memory
JAVA PROGRAM
생성
Access Modifier
Class Name
public class Vehicle {
Class Members Instance Variables
private int doors;
private int speed;
private String color;
Class Body
Class Members Instance Method
public void run(){
//Run method implementation.
}
}
종이.class

Tv.java

```
class Tv {
    // Tv의 속성(멤버변수)
    String color;
    boolean power;
    int channel;

    // Tv의 기능(메서드)
    void power() {   power = !power; }
    void channelUp() {     ++channel; }
    void channelDown() {    --channel; }
}
```

TvTest.java

```
class TvTest {
    public static void main(String args[]) {
        Tv t;
        t = new Tv();
        t.channel = 7;
        t.channelDown();
        System.out.println("현재 채널은 " + t.channel + " 입니다.");
    }
}
```

Compile

JVM

Byte코드

Byte코드

클래스와 객체의 정의와 용도

- 클래스의 정의 – 클래스란 객체를 정의해 놓은 것이다.
- 클래스의 용도 – 클래스는 객체를 생성하는데 사용된다.

- 객체의 정의 – 실제로 존재하는 것. 사물 또는 개념.
- 객체의 용도 – 객체의 속성과 기능에 따라 다름.

클래스	객체
제품 설계도	제품
TV설계도	TV
붕어빵기계	붕어빵

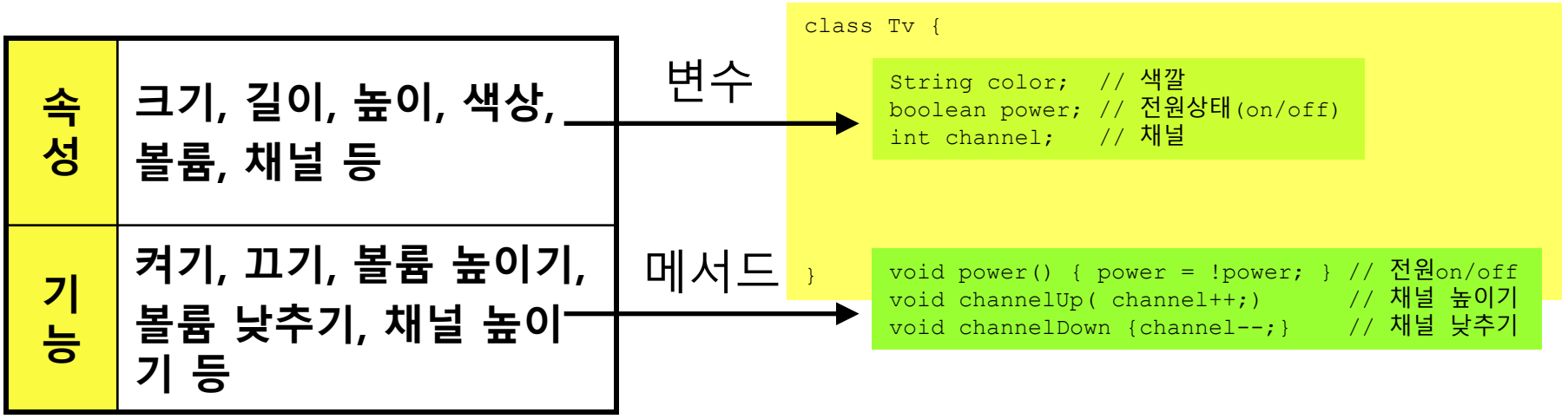
객체의 속성과 기능

▶ 객체는 속성과 기능으로 이루어져 있다.

- 객체는 속성과 기능의 집합이며,
  속성과 기능을 객체의 멤버(member, 구성요소)라고 한다.

▶ 속성은 변수로, 기능은 메서드로 정의한다.

- 클래스를 정의할 때 객체의 속성은 변수로, 기능은 메서드로 정의한다.

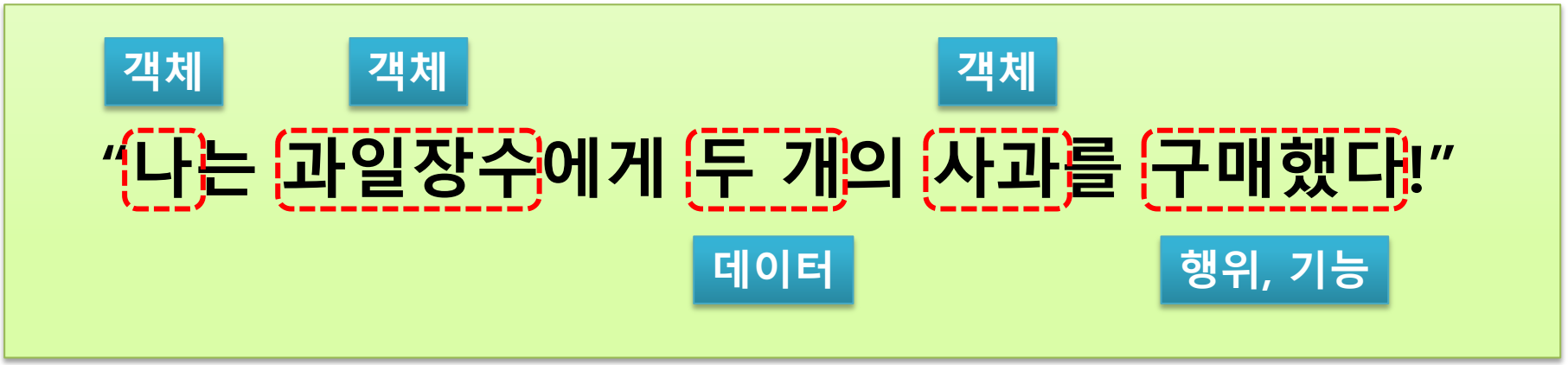


객체지향 프로그래밍과 객체

객체(Object)

사전적 의미: 물건 또는 대상

객체지향 프로그래밍: 객체 중심의 프로그래밍



객체지향 프로그래밍에서는 나, 과일장수, 사과라는 객체를 등장 시켜서 두 개의 사과구매라는 행위를 실체화 한다.

객체를 이루는 것은 데이터와 기능입니다.

과일 장수 객체의 표현

과일장수는 과일을 팝니다. - 행위

과일장수는 사과 20개, 오랜지 10개를 보유하고 있습니다. - 상태

과일장수의 과일판매 수익은 50000원 입니다. - 상태

과일 장수의 데이터 표현

보유하고 있는 사과의 수 : int numOfApple

판매수익 : int myMoney

객체를 이루는 것은 데이터와 기능입니다.

- 과일 장수의 행위 표현

```
int saleApple ( int money )  // 사과 구매금액이 매소드의 인자로 전달
{
        int num = money/1000; // 사과가 개당 1000원 이라 가정
        numOfApple -= num;    // 사과의 개수가 줄어 듬.
        myMoney += money;     // 판매 수익이 발생 함.
        return num;           // 실제 구매가 발생한 사과의 수를 반환
}
```

클래스(class)라는 틀 기반의 객체 생성

- 객체 생성에 앞서 선행되어야 하는 클래스의 정의

```
class FruitSeller
{
    int numOfApple=20;
    int myMoney=0;

    public int SaleApple(int money)
    {
        int num = money/1000;
        numOfApple -= num;
        myMoney += money;
        return num;
    }
}
```

변수 선언

메소드 정의

FruitSeller라는
이름의 틀 정의

‘과일장수’ 클래스 정의와 키워드 final

```
class FruitSeller
{
    final int APPLE_PRICE=1000;
    int numOfApple=20;
    int myMoney=0;

    public int saleApple(int money)
    {
        int num=money/APPLE_PRICE;
        numOfApple-=num;
        myMoney+=money;
        return num;
    }
    public void showSaleResult()
    {
        System.out.println("남은 사과 : " + numOfApple);
        System.out.println("판매 수익 : " + myMoney);
    }
}
```

과일 판매자 관점에서의 과일장수!

‘나(me)’ 클래스 정의

```
class FruitBuyer
{
    int myMoney=5000;
    int numOfApple=0;

    public void buyApple(FruitSeller seller, int money)
    {
        numOfApple+=seller.saleApple(money);
        myMoney-=money;
    }
    public void showBuyResult()
    {
        System.out.println("현재 잔액 : " + myMoney);
        System.out.println("사과 개수 : " + numOfApple);
    }
}
```

과일 구매자 관점에서의 나!

클래스를 기반으로 객체 생성하기

클래스로부터 인스턴스를 생성하는 것.

클래스

틀

```
class FruitSeller
{
    final int APPLE_PRICE=1000;
    int numOfApple=20;
    int myMoney=0;
    public int saleApple(int money)
    {
        . . . . .
    }
    public void showSaleResult( )
    {
        . . . . .
    }
}
```

인스턴스화

instantiation

객체 (인스턴스)

실체

```
final int APPLE_PRICE=1000;
int numOfApple=20;
int myMoney=0;
```

변수

```
public int saleApple(int money)
{
    . . . . .
}
public void showSaleResult( )
{
    . . . . .
}
```

메소드

객체와 인스턴스

우리가 궁극적으로 사용 하고자 하는 것은

클래스가 아니라 객체가 가지는 변수와 메서드 이고,

객체를 사용하려면

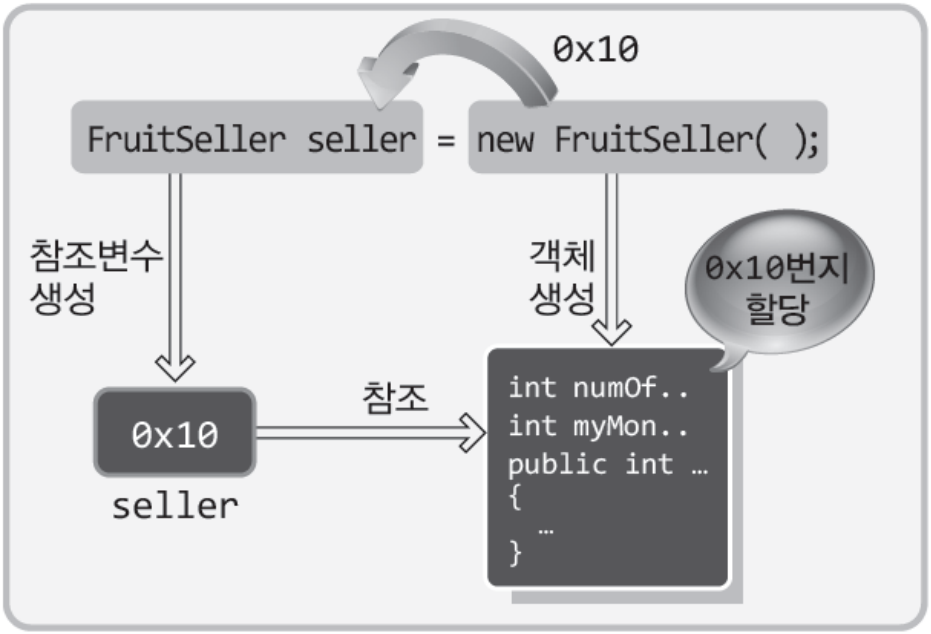
먼저 클래스로부터 객체를 생성해야 한다.

클래스를 기반으로 객체 생성하기

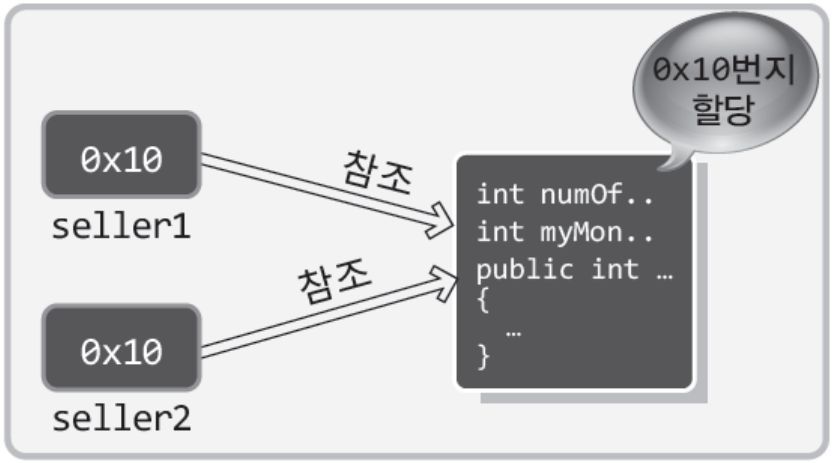
참조변수의 선언	인스턴스의 생성
Class이름 변수이름(식별자)	new Class이름()

```
FruitSeller seller = new FruitSeller();
FruitBuyer buyer = new FruitBuyer();
```

객체 생성과 참조의 관계를 정확히 말하라!



참조변수의 역할




```
FruitSeller seller1 = new FruitSeller();
FruitSeller seller2 = seller1;
```

생성된 객체의 접근 방법

객체 내에 존재하는 변수(속성)의 접근 : 참조

```
FruitSeller seller    =   new FruitSeller();
seller.numOfApple  =  20 ;
```

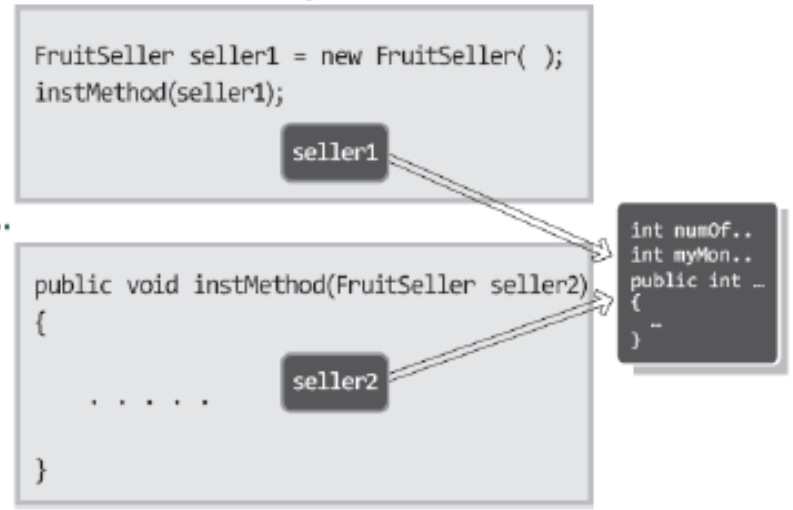
객체 내에 존재하는 메서드(기능)의 접근 : 호출

```
FruitSeller seller    =   new FruitSeller();
seller.saleApple(10);
```

참조 변수와 메소드의 관계

```
public void myMethod()
{
    FruitSeller seller1 = new FruitSeller();
    instMethod(seller1);
    . . . .
}
```

```
public void instMethod(FruitSeller seller2)
{
    . . . . .
}
```



참조변수와 메소드의 관계확인을 위한 예제

```
class Number
{
        int num=0;   // 인스턴스 변수라고 한다.

        public void addNum(int n) // 인스턴스 메소드라고 한다.
        {
                num+=n;
        }

        public int getNumber() // 인스턴스 메소드라고 한다.
        {
                return num;
        }
}
```

참조변수와 메소드의 관계확인을 위한 예제

```
class PassInstance
{
	public static void main(String[] args)
	{
		Number nInst=new Number();
		System.out.println("메소드 호출 전: "+nInst.getNumber());

		simpleMethod(nInst);
		System.out.println("메소드 호출 후: "+nInst.getNumber());

	}
	public static void simpleMethod(Number numb)
	{
		numb.addNum(12);
	}
}
```

```
메소드 호출 전 : 0
메소드 호출 후 : 12
```

참조변수의null 초기화

```
MyInstance myInst = null; // MyInstance라는 크래스의 참조변수 myInst의 선언

if (myInst==null) {
        System.out.println("참조변수 myInst는 팜조하는 객체가 없다. ");
}
```

null 은 아무것도 참조하지 않음을 의미하는 키워드.

사과장수 시뮬레이션 완료!

```
class FruitSeller
{
        int numOfApple=20;
        int myMoney=0;
        final int APPLE_PRICE=1000;

        public int saleApple(int money)
        {
                int num=money/APPLE_PRICE;
                numOfApple-=num;
                myMoney+=money;
                return num;
        }
        public void showSaleResult()
        {
                System.out.println("남은 사과: " + numOfApple);
                System.out.println("판매 수익: " + myMoney);
        }
}
```

사과장수 시뮬레이션 완료!

```
class FruitBuyer
{
        int myMoney=5000;
        int numOfApple=0;

        public void buyApple(FruitSeller seller, int money)
        {
                numOfApple+=seller.saleApple(money);
                myMoney-=money;
        }
        public void showBuyResult()
        {
                System.out.println("현재 잔액: " + myMoney);
                System.out.println("사과 개수: " + numOfApple);

        }
}
```

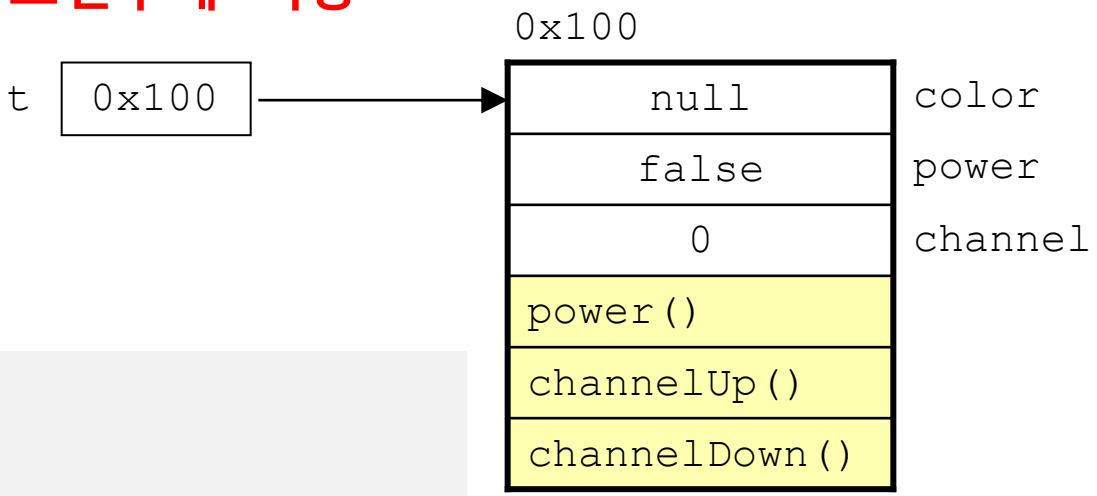
사과장수 시뮬레이션 완료!

```
class FruitSalesMain1
{
	public static void main(String[] args)
	{
		FruitSeller seller = new FruitSeller();
		FruitBuyer buyer = new FruitBuyer();
		buyer.buyApple(seller, 2000);

		System.out.println("과일 판매자의 현재 상황");
		seller.showSaleResult();

		System.out.println("과일 구매자의 현재 상황");
		buyer.showBuyResult();
	}
}
```

```
과일 판매자의 현재 상황
남은 사과 : 18
판매 수익 : 2000
과일 구매자의 현재 상황
남은 사과 : 3000
판매 수익 : 2
```

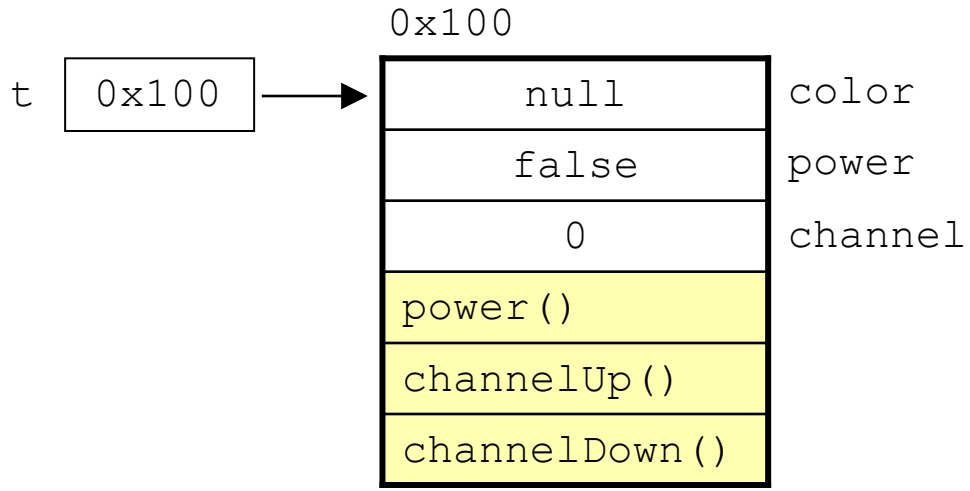
인스턴스의 생성과 사용

```
Tv t;     // 객체생성 후, 생성된 객체의
          // 주소를 참조변수에 저장
t = new Tv();

Tv t = new Tv();
```




```
class Tv {
     String color;  // 색깔
     boolean power; // 전원상태(on/off)
     int channel;   // 채널
     void power() { power = !power; } // 전원on/off
     void channelUp( channel++;)      // 채널 높이기
     void channelDown {channel--;}    // 채널 낮추기
}
```

인스턴스의 생성과 사용




```
Tv t;
t = new Tv();
t.channel = 7;
t.channelDown();
System.out.println(t.channel);
```

인스턴스의 생성과 사용

```
class Tv {
    // Tv의 속성(멤버변수)
    String color;           // 색상
    boolean power;                  // 전원상태(on/off)
    int channel;            // 채널

    // Tv의 기능(메서드)
    void power() {   power = !power; }   // TV를 켜거나 끄는 기능을 하는 메서드
    void channelUp() {     ++channel; }   // TV의 채널을 높이는 기능을 하는 메서드
    void channelDown() {    --channel; }  // TV의 채널을 낮추는 기능을 하는 메서드
}

class TvTest {
    public static void main(String args[]) {
        Tv t;                   // Tv인스턴스를 참조하기 위한 변수 t를 선언
        t = new Tv();           // Tv인스턴스를 생성한다.
        t.channel = 7;          // Tv인스턴스의 멤버변수 channel의 값을 7로 한다.
        t.channelDown();        // Tv인스턴스의 메서드 channelDown()을 호출한다.
        System.out.println("현재 채널은 " + t.channel + " 입니다.");
    }
}
```

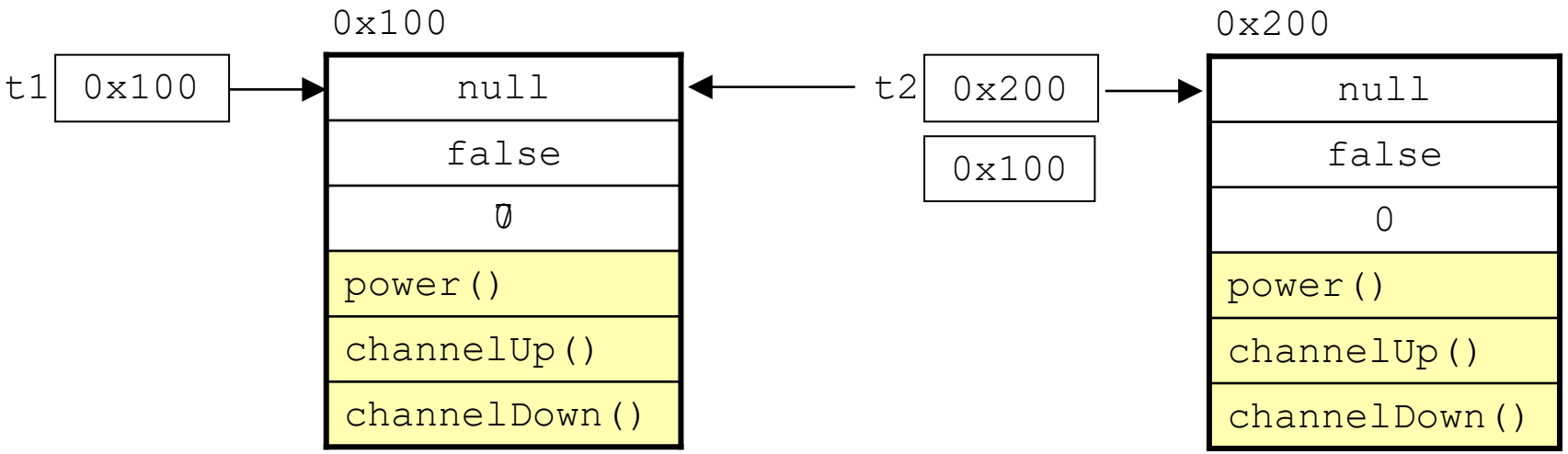
현재 체널은 6 입니다.

인스턴스의 생성과 사용

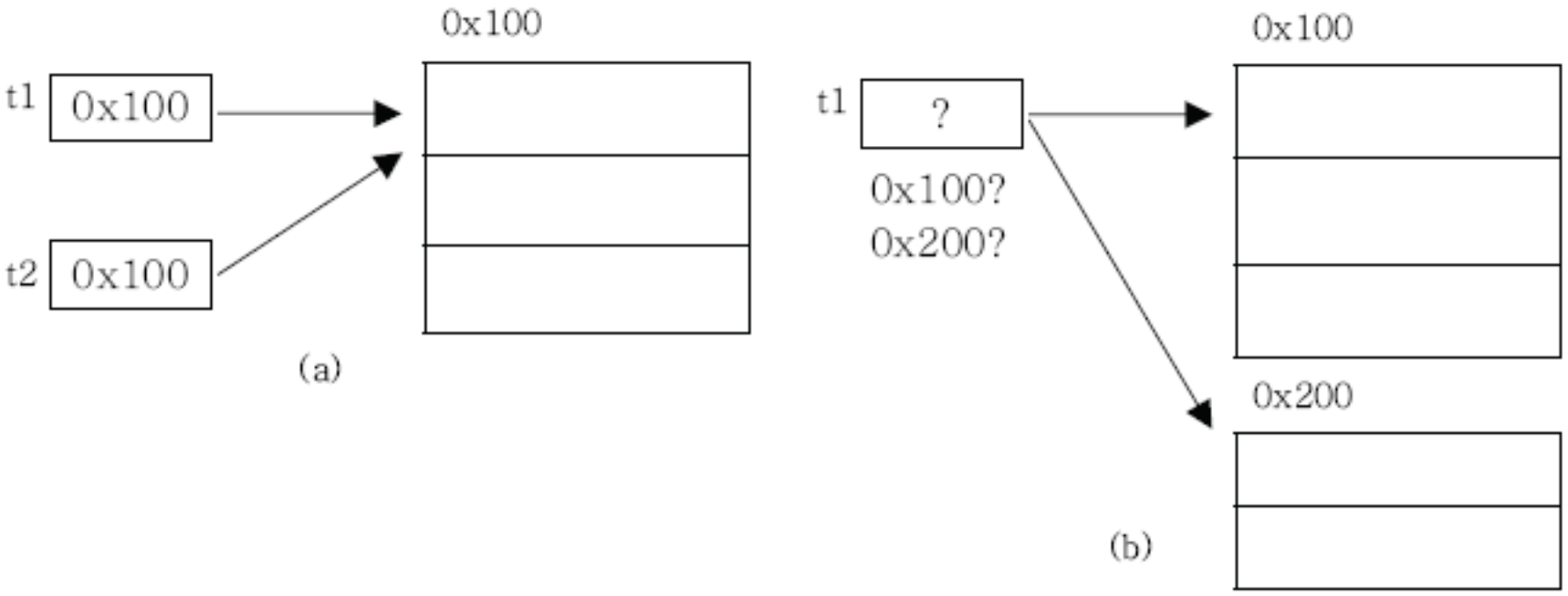
```
class TvTest2 {
        public static void main(String args[]) {
                Tv t1 = new Tv();
                Tv t2 = new Tv();
                System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
                System.out.println("t2의 channel값은 " + t2.channel + "입니다.");

                t1.channel = 7;       // channel 값을 7으로 한다.
                System.out.println("t1의 channel값을 7로 변경하였습니다.");

                System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
                System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
        }
}
```

인스턴스의 생성과 사용

```
Tv t1 = new Tv();
Tv t2 = new Tv();
t2 = t1;
t1.channel = 7;
System.out.println(t1.channel);
System.out.println(t2.channel);
```



인스턴스의 생성과 사용



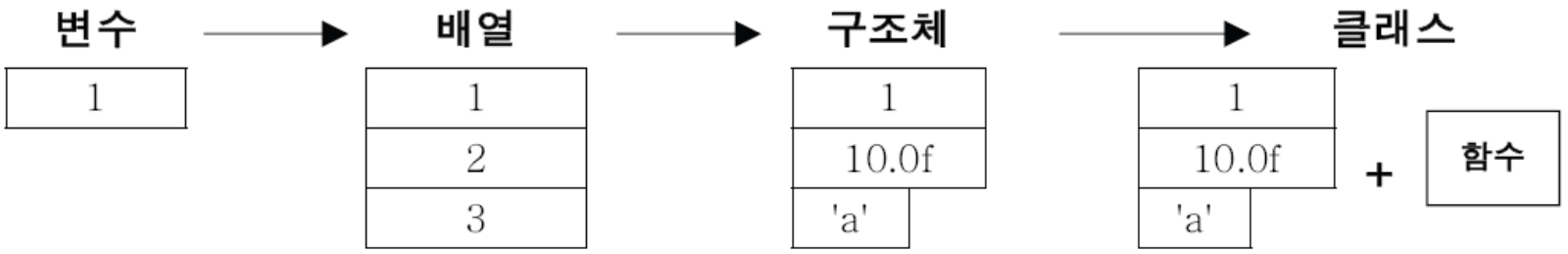
(a) 하나의 인스턴스를 여러 개의 참조변수가 가리키는 경우(가능)
(b) 여러 개의 인스턴스를 하나의 참조변수가 가리키는 경우(불가능)

[그림6-2] 참조변수와 인스턴스의 관계

인스턴스의 생성과 사용

```
class Tv {
    // Tv의 속성(멤버변수)
    String color;                   // 색상
    boolean power;                  // 전원상태(on/off)
    int channel;                    // 채널
    // Tv의 기능(메서드)
    void power() {   power = !power; }   /* TV를 켜거나 끄는 기능을 하는 메서드 */
    void channelUp() {      ++channel; }   /* TV의 채널을 높이는 기능을 하는 메서드 */
    void channelDown() {    --channel; } /* TV의 채널을 낮추는 기능을 하는 메서드 */
}
class TvTest2 {
        public static void main(String args[]) {
                Tv t1 = new Tv();
                Tv t2 = new Tv();
                System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
                System.out.println("t2의 channel값은 " + t2.channel + "입니다.");

                t2 = t1                 // t1이 저장하고 있는 값(주소)을 t2에 저장한다.
                t1.channel = 7;       // channel 값을 7으로 한다.
                System.out.println("t1의 channel값을 7로 변경하였습니다.");

                System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
                System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
        }
}
```

클래스의 또 다른 정의



- 변수 – **하나의 데이터**를 저장할 수 있는 공간
- 배열 – **같은 타입의 여러 데이터**를 저장할 수 있는 공간
- 구조체 – **타입에 관계없이 서로 관련된 데이터**들을 저장할 수 있는 공간
- 클래스 – **데이터와 함수의 결합**(구조체+함수)

클래스의 또 다른 정의

사용자 정의 타입(User-defined type)

- 프로그래머가 직접 새로운 타입을 정의할 수 있다.
- 서로 관련된 값을 묶어서 하나의 타입으로 정의한다.

```
class Time {
        int hour;
        int minute;
        int second;
}
```

```
int hour;
int minute;
int second;
```

→

```
Time t = new Time();
```

```
int hour1, hour2, hour3 ;
int minute1, minute2. minute3;
int second1, secound2, second3;
```

→

```
Time t1 = new Time();
Time t2 = new Time();
Time t3 = new Time();
```

```
int[] hour = new int[3];
int[] minute = new int[3];
int[] second = new int[3];
```

→

```
Time[] t = new Time[3];
t[0] = new Time();
t[1] = new Time();
t[2] = new Time();
```

문제

문제1.

밑변과 높이 정보를 지정할 수 있는 Triangle 클래스를 정의하자.

끝으로 밑변과 높이 정보를 변경시킬 수 있는 메서드와 삼각형의 넓이를 계산해서 반환하는 메서드도 함께 정의하자.

문제

문제2.

다음조건을 만족하는 클래스를 구성하자. 구슬치기와 딱지치기

어린아이가 소유하고 있는 구슬의 개수 정보를 담을 수 있다.

놀이를 통한 구슬을 주고받음을 표현하는 메서드가 존재한다.

두 번째 조건은 두 아이가 구슬치기를 하는 과정에서 구슬의 잃고 얻음을 의미하는 것이다.

조건을 만족하는 클래스를 정의하였다면, 다음조건을 만족하는 인스턴스를 각각 생성하자.

어린이 1의 보유자산 → 구슬 15개

어린이 2의 보유자산 → 구슬 9개

인스턴스의 생성이 완료되면 다음의 상황을 main 메서드 내에서 시뮬레이션 하자.

“1차 게임에서 어린이 1은 어린이 2의 구슬 2개를 획득한다”

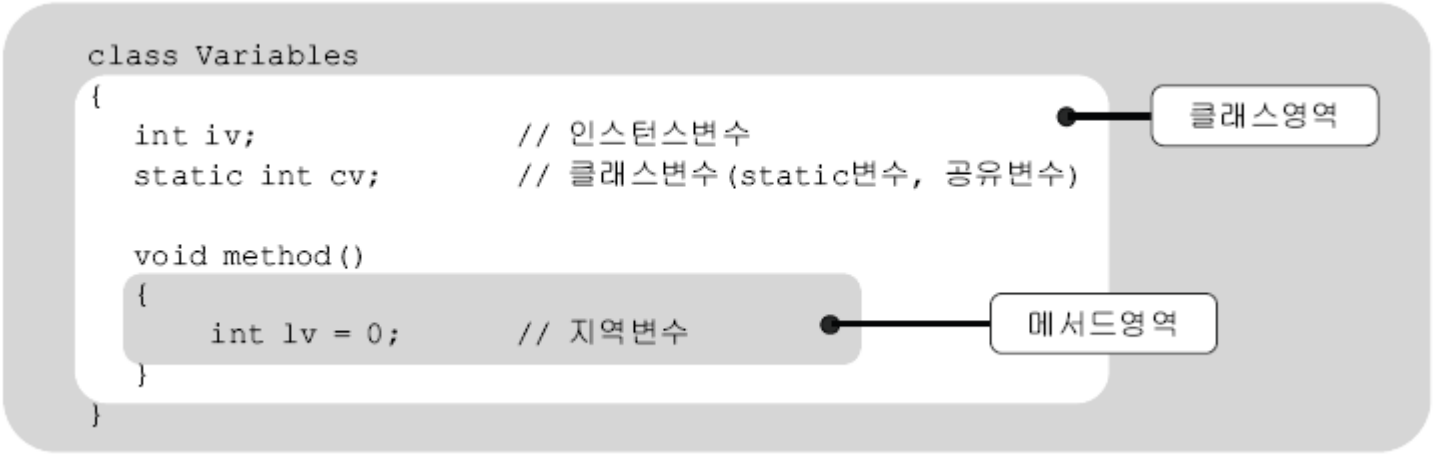
“2차 게임에서 어린이 2는 어린이 1의 구슬 7개를 획득한다.”

마지막으로 각각의 어린이의 보유 구슬의 개수를 출력하고 프로그램 종료.

변수와 메서드

선언위치에 따른 변수의 종류

"변수의 선언위치가 변수의 종류와 범위(scope)을 결정한다."



변수의 종류	선언위치	생성시기
클래스변수	클래스 영역	클래스가 메모리에 올라갈 때
인스턴스변수		인스턴스 생성시
지역변수	메서드 영역	변수 선언문 수행 시

선언위치에 따른 변수의 종류

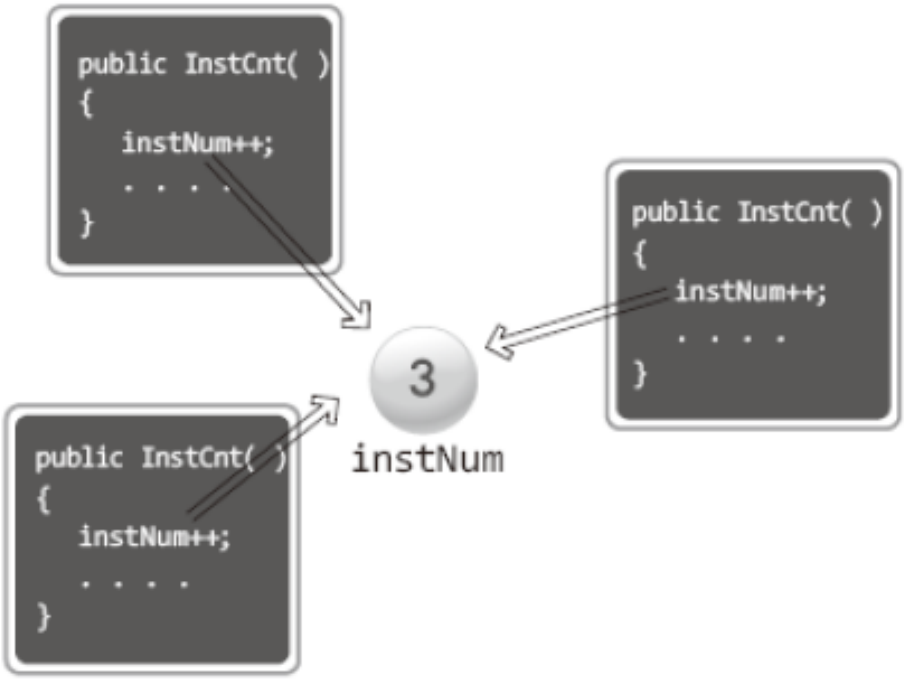
- 인스턴스변수(instance variable)
  - 각 인스턴스의 개별적인 저장공간. 인스턴스마다 다른 값 저장가능
  - 인스턴스 생성 후, '참조변수.인스턴스변수명'으로 접근
  - 인스턴스를 생성할 때 생성되고, 참조변수가 없을 때 가비지컬렉터에 의해 자동제거됨

- 클래스변수(class variable)
  - 같은 클래스의 모든 인스턴스들이 공유하는 변수
  - 인스턴스 생성 없이 '클래스이름.클래스변수명'으로 접근
  - 클래스가 로딩될 때 생성되고 프로그램이 종료될 때 소멸

- 지역변수(local variable)
  - 메서드 내에 선언되며, 메서드의 종료와 함께 소멸
  - 조건문, 반복문의 블럭{} 내에 선언된 지역변수는 블럭을 벗어나면 소멸

static 변수 (클래스 변수)

- static 변수
  - 인스턴스의 생성과 상관없이 초기화되는 변수
  - 하나만 선언되는 변수
  - Public으로 선언되면 누구나 어디서든 접근이 가능!



static 변수의 접근방법

접근하는 위치에 따라 접근 형태, 방식이 다를 수 있다.

```
class AccessWay
{
        static int num=0;

        AccessWay()
        {
                incrCnt();
        }
        public void incrCnt(){ num++; }        // 클래스 내부 접근 방법
}

class ClassVarAccess
{
        public static void main(String[] args)
        {
                AccessWay way=new AccessWay();
                way.num++;                     // 인스턴스 이름을 이용한 접근방법
                AccessWay.num++;
                System.out.println("num="+AccessWay.num);
        }                                      // 클래스 이름을 이용한 접근방법
}
```

static 변수의 초기화 시점

- JVM은 실행과정에서 필요한 클래스의 정보를 메모리에 로딩한다.
- 바로 이 Loading 시점에서 static 변수가 초기화 된다.

```
class InstCnt
{
    static int instNum=100;

    public InstCnt()
    {
        instNum++;
        System.out.println("인스턴스 생성 : "+instNum);
    }
}
class StaticValNoInst
{
    public static void main(String[] args)
    {
        InstCnt.instNum-=15;
        System.out.println(InstCnt.instNum);
    }
}
```

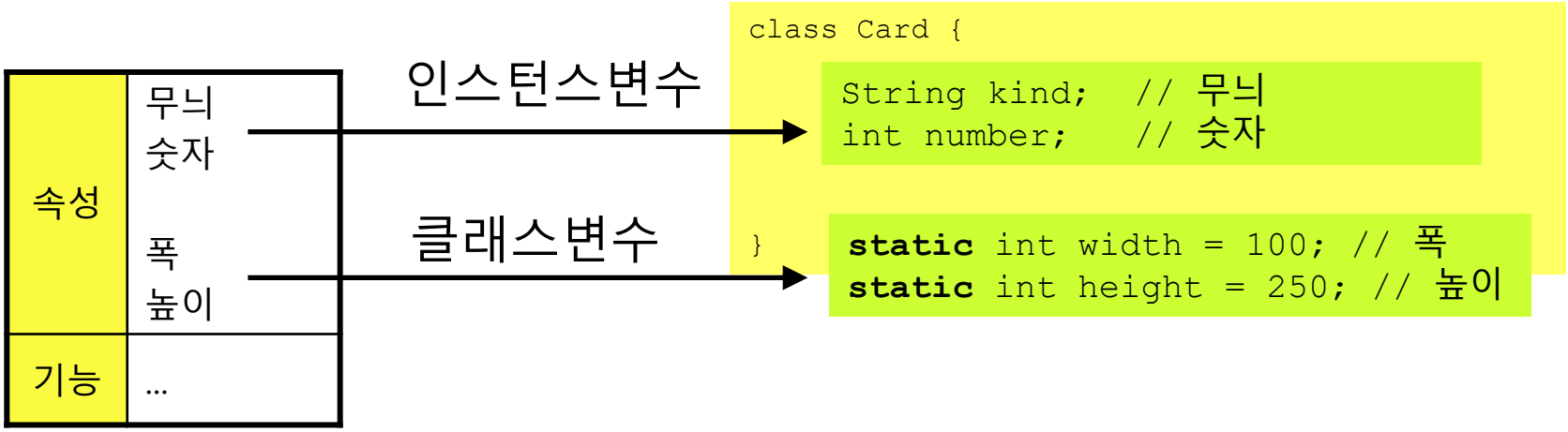
이 예제에서는 한번의 인스턴스 생성이 진행되지 않았다. 즉, 인스턴스 생성과 static 변수와는 아무런 상관이 없다.

static 변수의 초기화 시점

```
class InstCnt
{
	static int instNum=100;

	public InstCnt()
	{
		instNum++;
		System.out.println("인스턴스 생성: "+instNum);
	}
}

class StaticValNoInst
{
	public static void main(String[] args)
	{
		InstCnt.instNum-=15;
		System.out.println(InstCnt.instNum);
	}
}
```

클래스변수와 인스턴스변수

"인스턴스변수는 인스턴스가 생성될 때마다 생성되므로 인스턴스마다 각기 다른 값을 유지할 수 있지만, 클래스변수는 모든 인스턴스가 하나의 저장공간을 공유하므로 항상 공통된 값을 갖는다."



변수

```
class CardTest{
        public static void main(String args[]) {
                System.out.println("Card.width = " + Card.width);
                System.out.println("Card.height = " + Card.height);

                Card c1 = new Card();
                c1.kind = "Heart";
                c1.number = 7;

                Card c2 = new Card();
                c2.kind = "Spade";
                c2.number = 4;

                System.out.println("c1은 " + c1.kind + ", " + c1.number + "이며,
크기는 (" + c1.width + ", " + c1.height + ")" );
                System.out.println("c2는 " + c2.kind + ", " + c2.number + "이며,
크기는 (" + c2.width + ", " + c2.height + ")" );
                System.out.println("c1의 width와 height를 각각 50, 80으로 변경합
니다.");
```

변수

```
                c1.width = 50;
                c1.height = 80;

                System.out.println("c1은 " + c1.kind + ", " + c1.number + "이며,
크기는 (" + c1.width + ", " + c1.height + ")" );
                System.out.println("c2는 " + c2.kind + ", " + c2.number + "이며,
크기는 (" + c2.width + ", " + c2.height + ")" );
        }
 }

class Card {
        String kind ;                   // 카드의 무늬 - 인스턴스 변수
        int number;                     // 카드의 숫자 - 인스턴스 변수
        static int width = 100;         // 카드의 폭   - 클래스 변수
        static int height = 250;        // 카드의 높이 - 클래스 변수
}
```

기본형 매개변수와 참조형 매개변수

- 기본형 매개변수 – 변수의 값을 읽기만 할 수 있다.(read only)
- 참조형 매개변수 – 변수의 값을 읽고 변경할 수 있다.(read & write)

메서드

```
class Data { int x; }

class ParameterTest {
        public static void main(String[] args) {
                Data d = new Data();
                d.x = 10;
                System.out.println("main() : x = " + d.x);

                change(d.x);
                System.out.println("After change(d.x)");
                System.out.println("main() : x = " + d.x);
        }

        static void change(int x) {  // 기본형 매개변수
                x = 1000;
                System.out.println("change() : x = " + x);
        }
}
```

메서드

```
class Data { int x; }

class ParameterTest2 {
	public static void main(String[] args) {

		Data d = new Data();
		d.x = 10;
		System.out.println("main() : x = " + d.x);

		change(d);
		System.out.println("After change(d)");
		System.out.println("main() : x = " + d.x);

	}

	static void change(Data d) { // 참조형 매개변수
		d.x = 1000;
		System.out.println("change() : x = " + d.x);
	}
}
```

메서드

```
class ParameterTest3 {
	public static void main(String[] args) {

		int[] x = {10};  // 크기가 1인 배열. x[0] = 10;
		System.out.println("main() : x = " + x[0]);

		change(x);
		System.out.println("After change(x)");
		System.out.println("main() : x = " + x[0]);

	}

	static void change(int[] x) { // 참조형 매개변수
		x[0] = 1000;
		System.out.println("change() : x = " + x[0]);
	}
}
```

클래스메서드(static메서드)와 인스턴스메서드

- 인스턴스메서드
  - 인스턴스 생성 후, '참조변수.메서드이름()'으로 호출
  - 인스턴스변수나 인스턴스메서드와 관련된 작업을 하는 메서드
  - 메서드 내에서 인스턴스변수 사용가능

- 클래스메서드(static메서드)
  - 객체생성없이 '클래스이름.메서드이름()'으로 호출
  - 인스턴스변수나 인스턴스메서드와 관련없는 작업을 하는 메서드
  - 메서드 내에서 인스턴스변수 사용불가
  - 메서드 내에서 인스턴스변수를 사용하지 않는다면 static을 붙이는 것을 고려한다.

static 메서드 (클래스 메서드)

- static 메소드의 기본적인 특성과 접근방법은 static 변수와 동일하다.

```
class NumberPrinter
{
    public static void showInt(int n){ System.out.println(n); }
    public static void showDouble(double n) { System.out.println(n); }
}

class ClassMethod
{
    public static void main(String[] args)
    {
        NumberPrinter.showInt(20);
        NumberPrinter np=new NumberPrinter();
        np.showDouble(3.15);
    }
}
```

클래스의 이름을 통한 호출

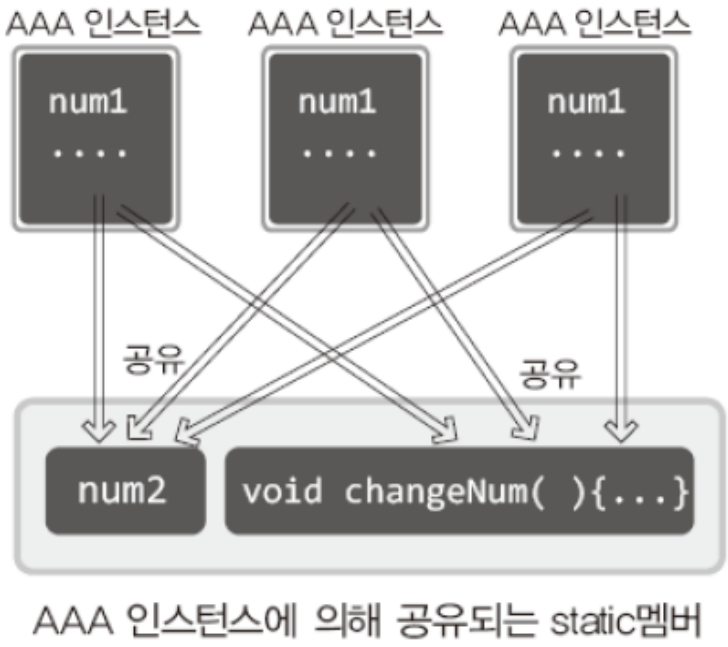
인스턴스의 이름을 통한 호출

이 예제에서 보이듯이 인스턴스를 생성하지 않아도 static 메소드는 호출 가능하다.

static 메소드의 인스턴스 접근? 말이 안된다!

• static 메소드는 인스턴스에 속하지 않기 때문에 인스턴스 멤버에 접근이 불가능하다!

```
class AAA
{
    int num1;
    static int num2;
    static void changeNum()
    {
        num1++;      // 문제 됨!
        num2++;      // 문제 안됨!
    }
    . . . .
}
```



AAA 인스턴스에 의해 공유되는 static멤버

static 변수 num2의 증가를 어떤 인스턴스를 대상으로 진행해야 하겠는가?

때문에 이는 논리적으로 인식될 수 없는 코드이다.

멤버간의 참조와 호출 – 메서드의 호출

"같은 클래스의 멤버간에는 객체생성이나 참조변수 없이 참조할 수 있다. 그러나 static멤버들은 인스턴스멤버들을 참조할 수 없다."

```
class TestClass {
   void instanceMothod() {}       // 인스턴스메서드
   static void staticMethod() {} // static메서드

   void instanceMothod2() {       // 인스턴스메서드
        instanceMethod();         // 다른 인스턴스메서드를 호출한다.
        staticMethod();           // static메서드를 호출한다.
   }

   static void staticMethod2() { // static메서드
        instanceMethod();         // 에러!!! 인스턴스메서드를 호출할 수 없다.
        staticMethod();           // static메서드는 호출 할 수 있다.
   }
} // end of class
```

멤버간의 참조와 호출 – 변수의 접근

"같은 클래스의 멤버간에는 객체생성이나 참조변수 없이 참조할 수 있다. 그러나 static멤버들은 인스턴스멤버들을 참조할 수 없다."

```
class TestClass2 {
   int iv;           // 인스턴스변수
   static int cv;    // 클래스변수

   void instanceMothod() {        // 인스턴스메서드
        System.out.println(iv);   // 인스턴스변수를 사용할 수 있다.
        System.out.println(cv);   // 클래스변수를 사용할 수 있다.
   }

   static void staticMethod() {   // static메서드
        System.out.println(iv);   // 에러!!! 인스턴스변수를 사용할 수 없다.
        System.out.println(cv);   // 클래스변수를 사용할 수 있다.
   }
} // end of class
```

메서드

```
class MemberCall {
	int iv = 10;
	static int cv = 20;

	int iv2 = cv;
//	static int cv2 = iv;		// 에러. 클래스변수는 인스턴스 변수를 사용할 수 없음.
	static int cv2 = new MemberCall().iv;	// 이처럼 객체를 생성해야 사용가능.

	static void staticMethod1() {
		System.out.println(cv);
//		System.out.println(iv);  // 에러. 클래스메서드에서 인스턴스변수를 사용불가.
		MemberCall c = new MemberCall();
		System.out.println(c.iv);   // 객체를 생성한 후에야 인스턴스변수의 참조가능.
	}

	void instanceMethod1() {
		System.out.println(cv);
		System.out.println(iv); //인스턴스메서드에서는 인스턴스변수를 바로 사용가
능.
	}
```

메서드

```
	static void staticMethod2() {
		staticMethod1();
//		instanceMethod1(); // 에러. 클래스메서드에서는 인스턴스메서드를 호출할 수 없음.
		MemberCall c = new MemberCall();
		c.instanceMethod1(); // 인스턴스를 생성한 후에야 호출할 수 있음.
	}

	void instanceMethod2() {	// 인스턴스메서드에서는 인스턴스메서드와 클래스메서드
		staticMethod1();		//  모두 인스턴스 생성없이 바로 호출이 가능하다.
		instanceMethod1();
	}
}
```

생성자(Constructor)

두 명의 과일장수와 한 명의 구매자

서로 다른 인스턴스의 생성은, 인스턴스 변수의 초기화라는 문제를 고민하게 한다.

- 과일장수1 : 보유하고 있는 사과의 수는 30개이고, 개당 가격은 1,500원
- 과일장수2 : 보유하고 있는 사과의 수는 20개이고, 개당 가격은 1,000원

```
class FruitSeller
{
    int numOfApple;
    int myMoney;
    int APPLE_PRICE;        // 이전 예제에서는 final로 선언되었다!
    . . . . .
    public void initMembers(int money, int appleNum, int price)
    {
        myMoney=money;
        numOfApple=appleNum;
        APPLE_PRICE=price;
    }
}
```

APPLE_PRICE의 값이 변경되어야 하므로 final이 빠진다.

바람직하지 않다!

- final이 빠지므로..
- 초기화 과정의 불편함..

```
public static void main(String[] args)
{
    FruitSeller seller1 = new FruitSeller();
    seller1.initMembers(0, 30, 1500);
    FruitSeller seller2 = new FruitSeller();
    seller2.initMembers(0, 20, 1000);
    FruitBuyer buyer = new FruitBuyer();
    buyer.buyApple(seller1, 4500);
    buyer.buyApple(seller2, 2000);
    . . . . . .
}
```

두 명의 과일장수와 한 명의 구매자

```
class FruitSeller
{
        int numOfApple;
        int myMoney;
        int APPLE_PRICE; // 이전 예제에서는 final로 선언되었다!!!
        public int saleApple(int money)
        {
                int num=money/APPLE_PRICE;
                numOfApple-=num;
                myMoney+=money;
                return num;
        }
        public void showSaleResult()
        {
                System.out.println("남은 사과: " + numOfApple);
                System.out.println("판매 수익: " + myMoney);
        }
        public void initMembers(int money, int appleNum, int price)
        {
                myMoney=money;
                numOfApple=appleNum;
                APPLE_PRICE=price;
        }
}
```

두 명의 과일장수와 한 명의 구매자

```
class FruitBuyer
{
        int myMoney=10000;    // 여기서도 클래스의 변경이 발생!
        int numOfApple=0;

        public void buyApple(FruitSeller seller, int money)
        {
                numOfApple+=seller.saleApple(money);
                myMoney-=money;
        }
        public void showBuyResult()
        {
                System.out.println("현재 잔액: " + myMoney);
                System.out.println("사과 개수: " + numOfApple);
        }
}
```

두 명의 과일장수와 한 명의 구매자

```
class FruitSalesMain2
{
	public static void main(String[] args)
	{
		FruitSeller seller1 = new FruitSeller();
		seller1.initMembers(0, 30, 1500);
		// 값을 초기화 해야 인스턴스 생성 완료!!!
		FruitSeller seller2 = new FruitSeller();
		seller2.initMembers(0, 20, 1000);

		FruitBuyer buyer = new FruitBuyer();
		buyer.buyApple(seller1, 4500);
		buyer.buyApple(seller2, 2000);

		System.out.println("과일 판매자1의 현재 상황");
		seller1.showSaleResult();

		System.out.println("과일 판매자2의 현재 상황");
		seller2.showSaleResult();

		System.out.println("과일 구매자의 현재 상황");
		buyer.showBuyResult();
	}
}
```

```
과일 판매자1의 현재 상황
남은 사과 : 27
판매 수익 : 4500
과일 판매자2의 현재 상황
남은 사과 : 18
판매 수익 : 2000
과일 구매자의 현재 상황
남은 사과 : 3000
판매 수익 : 5
```

생성자(Constructor)

▶ 생성자란?

- 인스턴스가 생성될 때마다 호출되는 '**인스턴스 초기화 메서드**'
- 인스턴스 변수의 초기화 또는 인스턴스 생성시 수행할 작업에 사용
- 몇 가지 조건을 제외하고는 메서드와 같다.
- 모든 클래스에는 반드시 하나 이상의 생성자가 있어야 한다.

* 인스턴스 초기화 – 인스턴스 변수에 적절한 값을 저장하는 것.

```
Card c = new Card();
```

1. 연산자 new에 의해서 메모리(heap)에 Card클래스의 인스턴스가 생성된다.
2. 생성자 Card()가 호출되어 수행된다.
3. 연산자 new의 결과로, 생성된 Card인스턴스의 주소가 반환되어 참조변수 c에 저장된다.

생성자(Constructor)

▸ 생성자의 조건

- 생성자의 이름은 클래스의 이름과 같아야 한다.
- 생성자는 리턴값이 없다. (하지만 void를 쓰지 않는다.)

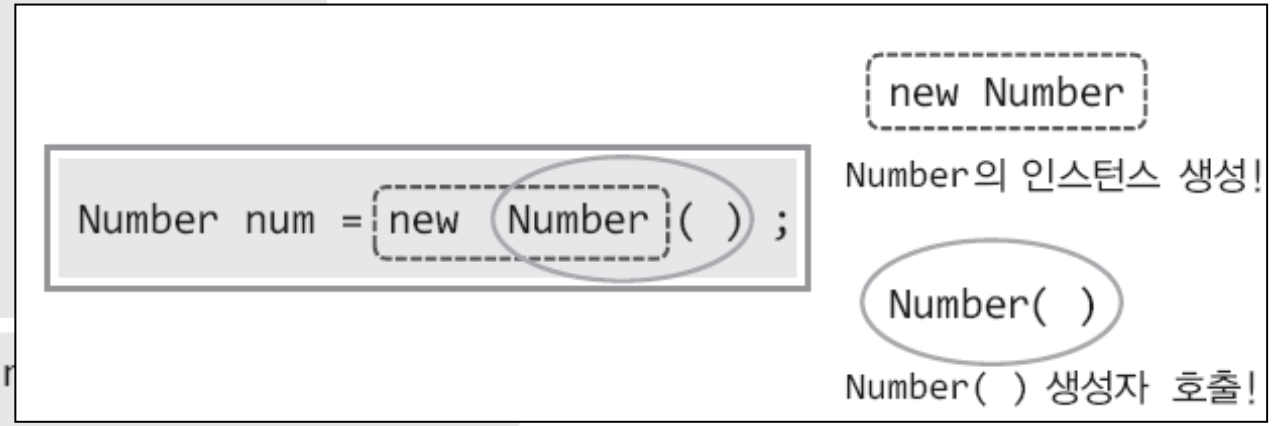
```
클래스이름(타입 변수명, 타입 변수명, ... ) {
    // 인스턴스 생성시 수행될 코드
    // 주로 인스턴스 변수의 초기화 코드를 적는다.
}
```

```
class Card {
    ...
    Card() {  // 매개변수가 없는 생성자.
        // 인스턴스 초기화 작업
    }

    Card(String kind, int number) { // 매개변수가 있는 생성자
        // 인스턴스 초기화 작업
    }
}
```

딱 한번만 호출되는 메소드! 생성자!

```
class Number
{
    int num;
    public Number()     // 생성자!
    {
        num=10;
        System.out.println("생성자 호출!");
    }
    public int getNumber()
    {
        return num;
    }
}
```

자바 인스턴스 생성시 생성자는 반드시 호출된다!




```
public static void main
{
    Number num1=new Number();
    System.out.println(num1.getNumber());

    Number num2=new Number();
    System.out.println(num2.getNumber());
}
```

```
생성자 호출!
10
생성자 호출!
10
```

딱 한번만 호출되는 메소드! 생성자!

```
class Number
{
	int num;
	public Number()
	{
		num=10;
		System.out.println("생성자 호출!");
	}
	public int getNumber()  // 인스턴스 메서드
	{
		return num;
	}
}
class Constructor1
{
	public static void main(String[] args)
	{
		Number num1=new Number();
		System.out.println(num1.getNumber());

		Number num2=new Number();
		System.out.println(num2.getNumber());
	}
}
```

```
생성자 호출!
10
생성자 호출!
10
```

디폴트 생성자(Default Constructor)

```
FruitSeller seller = new FruitSeller();
FruitBuyer buyer = new FruitBuyer();
```

호출되는 디폴트 생성자

```
public FruitSeller()
{
    // 텅 비어있다!
}
```

생성자를 정의하지 않았을 때에만 삽입!

생성자가 없어도 인스턴스 생성이 가능한 이유는 자동으로 삽입되는 디폴트 생성자에 있다.

디폴트 생성자(Default Constructor)

기본 생성자(default constructor)

"모든 클래스에는 반드시 하나 이상의 생성자가 있어야 한다."

```
class Data1 {
   int value;
}

class Data2 {
   int value;
   Data2(int x) {    // 매개변수가 있는 생성자.
        value = x;
   }
}

class ConstructorTest {
   public static void main(String[] args) {
        Data1 d1 = new Data1();
        Data2 d2 = new Data2();        // compile error발생
   }
}
```

```
class Data1 {
    int value;
    Data1() {} // 기본생성자
}
```

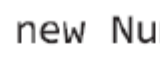
디폴트 생성자(Default Constructor)

```
class Data1 {
        int value;
}

class Data2 {
        int value;
        Data2(int x) {      // 매개변수가 있는 생성자.
                value = x;
        }
}

class ConstructorTest {
        public static void main(String[] args) {
                Data1 d1 = new Data1();
                Data2 d2 = new Data2();              // compile error발생
        }
}
```

값을 전달받는 생성자

호출되는 생성자

```
Number num = new Number(10);
Number num = new Number(30);
```



```
public Number(int n)
{
    . . . .
}
```

```
class FruitSeller
{
    int numOfApple;
    int myMoney;
    final int APPLE_PRICE;
    public FruitSeller(int money, int appleNum, int price)
    {
        myMoney=money;
        numOfApple=appleNum;
        APPLE_PRICE=price;
    }
    . . . . . .
}
```

```
public static void main(String[] args)
{
    FruitSeller seller1 = new FruitSeller(0, 30, 1500);
    FruitSeller seller2 = new FruitSeller(0, 20, 1000);
    . . . . . .
}
```

생성자 내에서는 final 멤버 변수의 초기화가 가능하다!

값을 전달받는 생성자

```
class Number
{
	int num;
	public Number(int n)
	{
		num=n;
		System.out.println("인자 전달: "+n);
	}
	public int getNumber()
	{
		return num;
	}
}
class Constructor2
{
	public static void main(String[] args)
	{
		Number num1=new Number(10);
		System.out.println("메소드 반환 값: "+num1.getNumber());

		Number num2=new Number(20);
		System.out.println("메소드 반환 값: "+num2.getNumber());
	}
}
```

개선된 사과장수

```
class FruitSeller
{
        int numOfApple;
        int myMoney;
        final int APPLE_PRICE;
        public FruitSeller(int money, int appleNum, int price)
        {
                myMoney=money;
                numOfApple=appleNum;
                APPLE_PRICE=price;
        }
        public int saleApple(int money)
        {
                int num=money/APPLE_PRICE;
                numOfApple-=num;
                myMoney+=money;
                return num;
        }
        public void showSaleResult()
        {
                System.out.println("남은 사과: " + numOfApple);
                System.out.println("판매 수익: " + myMoney);
        }
}
```

개선된 사과장수

```
class FruitBuyer
{
        int myMoney;
        int numOfApple;

        public FruitBuyer(int money)
        {
                myMoney=money;
                numOfApple=0;
        }

        public void buyApple(FruitSeller seller, int money)
        {
                numOfApple+=seller.saleApple(money);
                myMoney-=money;
        }
        public void showBuyResult()
        {
                System.out.println("현재 잔액: " + myMoney);
                System.out.println("사과 개수: " + numOfApple);
        }
}
```

개선된 사과장수

```
class FruitSalesMain3
{
        public static void main(String[] args)
        {
                FruitSeller seller1 = new FruitSeller(0, 30, 1500);
                FruitSeller seller2 = new FruitSeller(0, 20, 1000);

                FruitBuyer buyer = new FruitBuyer(10000);
                buyer.buyApple(seller1, 4500);
                buyer.buyApple(seller2, 2000);

                System.out.println("과일 판매자1의 현재 상황");
                seller1.showSaleResult();

                System.out.println("과일 판매자2의 현재 상황");
                seller2.showSaleResult();

                System.out.println("과일 구매자의 현재 상황");
                buyer.showBuyResult();
        }
}
```

매개변수가 있는 생성자

```
class Car {
      String color;           // 색상
      String gearType;        // 변속기 종류 - auto(자동), manual(수동)
      int door;               // 문의 개수

      Car() {}  // 생성자
      Car(String c, String g, int d) { // 생성자
            color = c;
            gearType = g;
            door = d;
      }
}
```

```
Car c = new Car();
c.color = "white";
c.gearType = "auto";
c.door = 4;
```

```
Car c = new Car("white","auto",4);
```

매개변수가 있는 생성자

```
class Car {
        String color;                   // 색상
        String gearType;                // 변속기 종류 - auto(자동), manual(수동)
        int door;                       // 문의 개수
        Car() {}
        Car(String c, String g, int d) {
                color = c;
                gearType = g;
                door = d;
        }
}
class CarTest {
        public static void main(String[] args) {
                Car c1 = new Car();
                c1.color = "white";
                c1.gearType = "auto";
                c1.door = 4;
                Car c2 = new Car("white", "auto", 4);
                System.out.println("c1의 color=" + c1.color + ", gearType=" +
c1.gearType+ ", door="+c1.door);
                System.out.println("c2의 color=" + c2.color + ", gearType=" +
c2.gearType+ ", door="+c2.door);
        }
}
```

생성자에서 다른 생성자 호출하기 – this()

- this() – 생성자, 같은 클래스의 다른 생성자를 호출할 때 사용
  다른 생성자 호출은 생성자의 첫 문장에서만 가능

```
1  class Car {
2      String color;
3      String gearType;
4      int door;
5
6      Car() {
7          color = "white";
8          gearType = "auto";
9          door = 4;
10     }
11
12     Car(String c, String g, int d) {
13         color = c;
14         gearType = g;
15         door = d;
16     }
17
18 }
19
```

* 코드의 재사용성을 높인 코드

```
Car() {
    //Car("white","auto",4);
    this("white","auto",4);
}
```

```
Car() {
    door = 5;
    this("white","auto",4);
}
```

참조변수 this

- this – 인스턴스 자신을 가리키는 참조변수. 인스턴스의 주소가 저장되어있음
  모든 인스턴스 메서드에 지역변수로 숨겨진 채로 존재

```
1  class Car {
2      String color;
3      String gearType;
4      int door;
5
6      Car() {
7          //Card("white","auto",4);
8          this("white","auto",4);
9      }
10
11     Car(String c, String g, int d){
12         color = c;
13         gearType = g;
14         door = d;
15     }
16 }
17
```

* 인스턴스변수와 지역변수를 구별하기 위해 참조변수 this사용

```
Car(String color, String gearType, int door){
    this.color = color;
    this.gearType = gearType;
    this.door = door;
}
```

this()

```
class Car {
        String color;                   // 색상
        String gearType;                // 변속기 종류 - auto(자동), manual(수동)
        int door;                       // 문의 개수

        Car() {
                this("white", "auto", 4);
        }

        Car(String color) {
                this(color, "auto", 4);
        }

        Car(String color, String gearType, int door) {
                this.color = color;
                this.gearType = gearType;
                this.door = door;
        }
}
```

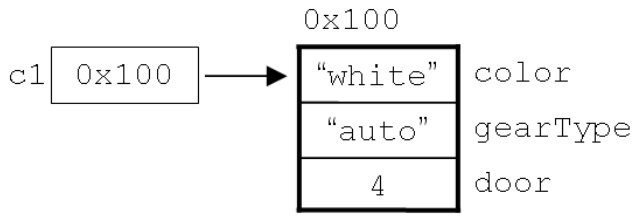
this()

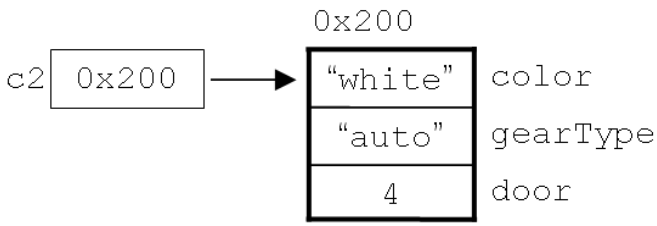
```
class CarTest2 {
	public static void main(String[] args) {
		Car c1 = new Car();
		Car c2 = new Car("blue");

		System.out.println("c1의 color=" + c1.color + ", gearType=" +
c1.gearType+ ", door="+c1.door);
		System.out.println("c2의 color=" + c2.color + ", gearType=" +
c2.gearType+ ", door="+c2.door);
	}
}
```

생성자를 이용한 인스턴스의 복사

- 인스턴스간의 차이는 인스턴스변수의 값 뿐 나머지는 동일하다.
- 생성자에서 참조변수를 매개변수로 받아서 인스턴스변수들의 값을 복사한다.
- 똑같은 속성값을 갖는 독립적인 인스턴스가 하나 더 만들어진다.

```
1  class Car {
2      String color;        // 색상
3      String gearType;     // 변속기 종류 - auto(자동), manual(수동)
4      int door;            // 문의 개수
5
6      Car() {
7          this("white", "auto", 4);
8      }
9
10     Car(String color, String gearType, int door) {
11         this.color = color;
12         this.gearType = gearType;
13         this.door = door;
14     }
15
16     Car(Car c) {    // 인스턴스의 복사를 위한 생성자.
17         color = c.color;
18         gearType = c.gearType;
19         door = c.door;
20     }
21 }
22
23 class CarTest3 {
24     public static void main(String[] args) {
25         Car c1 = new Car();
26         Car c2 = new Car(c1); // Car(Car c)를 호출
27     }
28 }
```






```
Car(Car c) {
    this(c.color, c.gearType, c.door);
}
```

this

```
class Car {
        String color;                   // 색상
        String gearType;                // 변속기 종류 - auto(자동), manual(수동)
        int door;                       // 문의 개수

        Car() {
                this("white", "auto", 4);
        }

        Car(Car c) {            // 인스턴스의 복사를 위한 생성자.
                color = c.color;
                gearType = c.gearType;
                door = c.door;
        }

        Car(String color, String gearType, int door) {
                this.color = color;
                this.gearType = gearType;
                this.door = door;
        }
}
```

this

```
class CarTest3 {
	public static void main(String[] args) {
		Car c1 = new Car();
		Car c2 = new Car(c1);	// c1의 복사본 c2를 생성한다.

		System.out.println("c1의 color=" + c1.color + ", gearType=" +
c1.gearType+ ", door="+c1.door);
		System.out.println("c2의 color=" + c2.color + ", gearType=" +
c2.gearType+ ", door="+c2.door);

		c1.door=100;	// c1의 인스턴스변수 door의 값을 변경한다.
		System.out.println("c1.door=100; 수행 후");
		System.out.println("c1의 color=" + c1.color + ", gearType=" +
c1.gearType+ ", door="+c1.door);
		System.out.println("c2의 color=" + c2.color + ", gearType=" +
c2.gearType+ ", door="+c2.door);
	}
}
```

변수의 초기화

변수의 초기화

- 변수를 선언하고 처음으로 값을 저장하는 것
- 멤버변수(인스턴스변수,클래스변수)와 배열은 각 타입의 기본값으로 자동초기화되므로 초기화를 생략할 수 있다.

- 지역변수는 사용전에 꼭!!! 초기화를 해주어야한다.

자료형	기본값
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형 변수	null

```
class InitTest {
	int x;		// 인스턴스변수
	int y = x;	// 인스턴스변수

	void method1() {
		int i;	// 지역변수
		int j = i;	// 컴파일 에러!!! 지역변수를 초기화하지 않고 사용했음.
	}
}
```

변수의 초기화 – 예시(examples)

선언예	설 명
int i=10; int j=10;	int형 변수 i를 선언하고 10으로 초기화 한다. int형 변수 j를 선언하고 10으로 초기화 한다.
int i=10, j=10;	같은 타입의 변수는 콤마(,)를 사용해서 함께 선언하거나 초기화 할 수 있다.
int i=10, long j=0;	타입이 다른 변수는 함께 선언하거나 초기화할 수 없다.
int i=10; int j=i;	변수 i에 저장된 값으로 변수 j를 초기화 한다. 변수 j는 i의 값인 10으로 초기화 된다.
int j=i; int i=10;	변수 i가 선언되기 전에 i를 사용할 수 없다.

```
class Test
{
    int j = i;
    int i = 10; // 에러!!!
}
```

```
class Test
{
    int i = 10;
    int j = i;  // OK
}
```

멤버변수의 초기화

▶ 멤버변수의 초기화 방법

1. 명시적 초기화(explicit initialization)

```
class Car {
      int door = 4;                 // 기본형(primitive type) 변수의 초기화
      Engine e = new Engine();      // 참조형(reference type) 변수의 초기화

      //...
}
```

2. 생성자(constructor)

```
Car(String color, String gearType, int door){
    this.color = color;
    this.gearType = gearType;
    this.door = door;
}
```

3. 초기화 블럭(initialization block)

- 인스턴스 초기화 블럭 : { }
- 클래스 초기화 블럭 : static { }

초기화 블럭(initialization block)

- 클래스 초기화 블럭 – 클래스변수의 복잡한 초기화에 사용되며 클래스가 로딩될 때 실행된다.

- 인스턴스 초기화 블럭 – 생성자에서 공통적으로 수행되는 작업에 사용되며 인스턴스가 생성될 때 마다 (생성자보다 먼저) 실행된다.

```
class InitBlock {
    static { /* 클래스 초기화블럭 입니다. */ }

    { /* 인스턴스 초기화블럭 입니다. */ }

    // ...
}
```

```
1  class StaticBlockTest {
2      static int[] arr = new int[10]; // 명시적 초기화
3
4      static { // 배열 arr을 1~10사이의 값으로 채운다.
5          for(int i=0;i<arr.length;i++) {
6              arr[i] = (int)(Math.random()*10) + 1;
7          }
8      }
9      //...
10 }
```

변수의 초기화

```
class BlockTest {

        static {
                System.out.println("static { }");
        }

        {
                System.out.println("{ }");
        }

        public BlockTest() {
                System.out.println("생성자");
        }

        public static void main(String args[]) {
                System.out.println("BlockTest bt = new BlockTest(); ");
                BlockTest bt = new BlockTest();

                System.out.println("BlockTest bt2 = new BlockTest(); ");
                BlockTest bt2 = new BlockTest();
        }
}
```

변수의 초기화

```
class StaticBlockTest
{
	static int[] arr = new int[10];

	static {
		for(int i=0;i<arr.length;i++) {
			// 1과 10사이의 임의의 값을 배열 arr에 저장한다.
			arr[i] = (int)(Math.random()*10) + 1;
		}
	}

	public static void main(String args[]) {
		for(int i=0; i<arr.length;i++)
			System.out.println("arr["+i+"] :" + arr[i]);
	}
}
```

멤버변수의 초기화 시기와 순서

- 클래스변수 초기화 시점 : 클래스가 처음 로딩될 때 단 한번
- 인스턴스변수 초기화 시점 : 인스턴스가 생성될 때 마다

```
1  class InitTest {
2      static int cv = 1;  // 명시적 초기화
3      int iv = 1;         // 명시적 초기화
4
5      static {    cv = 2; }   // 클래스 초기화 블럭
6      {   iv = 2; }           // 인스턴스 초기화 블럭
7
8      InitTest() { // 생성자
9          iv = 3;
10     }
11 }
```

```
InitTest it = new InitTest();
```

클래스 초기화			인스턴스 초기화			
기본값	명시적 초기화	클래스 초기화블럭	기본값	명시적 초기화	인스턴스 초기화블럭	생성자
cv 0	cv 1	cv 2	cv 2 iv 0	cv 2 iv 1	cv 2 iv 2	cv 2 iv 3
1	2	3	4	5	6	7

변수의 초기화

```
class Product {
        static int count = 0;    // 생성된 인스턴스의 수를 저장하기 위한 변수
        int serialNo;                // 인스턴스 고유의 번호

        {
                ++count;
                serialNo = count;
        }

        public Product() {}
}

class ProductTest {
        public static void main(String args[]) {
                Product p1 = new Product();
                Product p2 = new Product();
                Product p3 = new Product();

                System.out.println("p1의 제품번호(serial no)는 " + p1.serialNo);
                System.out.println("p2의 제품번호(serial no)는 " + p2.serialNo);
                System.out.println("p3의 제품번호(serial no)는 " + p3.serialNo);
                System.out.println("생산된 제품의 수는 모두 "+Product.count+"개 입니다.");
        }
}
```

변수의 초기화

```
class Document {
        static int count = 0;
        String name;                    // 문서명(Document name)

        public Document() {     // 문서 생성 시 문서명을 지정하지 않았을 때
                this("제목없음" + ++count);
        }

        public Document(String name) {
                this.name = name;
                System.out.println("문서 " + this.name + "가 생성되었습니다.");
        }
}

class DocumentTest {
        public static void main(String args[]) {
                Document d1 = new Document();
                Document d2 = new Document("자바.txt");
                Document d3 = new Document();
                Document d4 = new Document();
        }
}
```

메서드 오버로딩(Overloading)

매개변수 형(type)이 다르거나 개수가 다르거나

- 메소드 오버로딩이란 동일한 이름의 메소드를 둘 이상 동시에 정의하는 것을 뜻한다.
- 메소드의 매개변수 선언(개수 또는 자료형)이 다르면 메소드 오버로딩 성립
- 오버로딩 된 메소드는 호출 시 전달하는 인자를 통해서 구분된다.

매개변수 형(type)이 다르거나 개수가 다르거나

메서드 오버로딩(method overloading)이란?

"하나의 클래스에 같은 이름의 메서드를 여러 개 정의하는 것을 메서드 오버로딩, 간단히 오버로딩이라고 한다."

* overload - vt. 과적하다. 부담을 많이 지우다.

오버로딩의 조건

- 메서드의 이름이 같아야 한다.
- 매개변수의 개수 또는 타입이 달라야 한다.
- 매개변수는 같고 리턴타입이 다른 경우는 오버로딩이 성립되지 않는다.
  (리턴타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다.)

매개변수 형(type)이 다르거나 개수가 다르거나

```
class AAA
{
    void isYourFunc(int n) { . . . }
    void isYourFunc(int n1, int n2) { . . . }
    void isYourFunc(int n1, double n2) { . . . }
     . . . .
}
```

오버로딩 된 메소드

```
AAA inst = new AAA();
inst.isYourFunc(10);
inst.isYourFunc(10, 20);
inst.isYourFunc(12, 3.15);
```

전달되는 인자의 유형을 통해서 호출되는 함수가 결정!

오버로딩의 예

- System.out.println메서드
  - 다양하게 오버로딩된 메서드를 제공함으로써 모든 변수를 출력할 수 있도록 설계

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

매개변수 형(type)이 다르거나 개수가 다르거나

오버로딩의 예

- 매개변수의 이름이 다른 것은 오버로딩이 아니다.

```
[보기1]
int add(int a, int b) { return a+b; }
int add(int x, int y) { return x+y; }
```

- 리턴타입은 오버로딩의 성립조건이 아니다.

```
[보기2]
int add(int a, int b) { return a+b; }
long add(int a, int b) { return (long)(a + b); }
```

매개변수 형(type)이 다르거나 개수가 다르거나

오버로딩의 예

- 매개변수의 타입이 다르므로 오버로딩이 성립한다.

```
[보기3]
long add(int a, long b) { return a+b; }
long add(long a, int b) { return a+b; }
```

- 오버로딩의 올바른 예 – 매개변수는 다르지만 같은 의미의 기능수행

```
[보기4]
int add(int a, int b) { return a+b; }
long add(long a, long b) { return a+b; }
int add(int[] a) {
      int result =0;

      for(int i=0; i < a.length; i++) {
            result += a[i];
     }
      return result;
}
```

매개변수 형(type)이 다르거나 개수가 다르거나

```
class OverloadingTest {
        public static void main(String args[]) {
                MyMath3 mm = new MyMath3();
                System.out.println("mm.add(3, 3) 결과:" + mm.add(3,3));
                System.out.println("mm.add(3L, 3) 결과: " + mm.add(3L,3));
                System.out.println("mm.add(3, 3L) 결과: " + mm.add(3,3L));
                System.out.println("mm.add(3L, 3L) 결과: " + mm.add(3L,3L));

                int[] a = {100, 200, 300};
                System.out.println("mm.add(a) 결과: " + mm.add(a));
    }
}
```

매개변수 형(type)이 다르거나 개수가 다르거나

```
class MyMath3 {
	int add(int a, int b) {
		System.out.print("int add(int a, int b) - ");
		return a+b;
	}
	long add(int a, long b) {
		System.out.print("long add(int a, long b) - ");
		return a+b;
	}
	long add(long a, int b) {
		System.out.print("long add(long a, int b) - ");
		return a+b;
	}
	long add(long a, long b) {
		System.out.print("long add(long a, long b) - ");
		return a+b;
	}
	int add(int[] a) {			// 배열의 모든 요소의 합을 결과로 돌려준다.
		System.out.print("int add(int[] a) - ");
		int result = 0;
		for(int i=0; i < a.length;i++) {
			result += a[i];
		}
		return result;
	}		}
```

생성자도 오버로딩의 대상이 됩니다.

생성자의 오버로딩은 하나의 클래스를 기반으로 다양한 형태의 인스턴스 생성을 가능하게 한다.

생성자도 오버로딩의 대상이 됩니다.

```
class Person{
        private int perID;
        private int milID;
        public Person(int pID, int mID){
                perID=pID;
                milID=mID;
        }
        public Person(int pID){
                perID=pID;
                milID=0;
        }
        public void showInfo(){

                System.out.println("민번: "+perID);
                if(milID!=0)
                        System.out.println("군번: "+milID+'₩n');
                else
                        System.out.println("군과 관계 없음 ₩n");
        }
}
```

생성자도 오버로딩의 대상이 됩니다.

```
class Overloading
{
	public static void main(String[] args)
	{
		Person man=new Person(950123, 880102);
		Person woman=new Person(941125);
		man.showInfo();
		woman.showInfo();
	}
}
```

키워드 this를 이용한 다른 생성자의 호출

- 키워드 this를 이용하면 생성자 내에서 다른 생성자를 호출할 수 있다.
- 이는 생성자의 추가 정의에 대한 편의를 제공한다.
- 생성자마다 중복되는 초기화 과정의 중복을 피할 수 있다.

키워드 this를 이용한 다른 생성자의 호출

```
class Person
{
    private int perID;
    private int milID;
    private int birthYear;
    private int birthMonth;
    private int birthDay;

    public Person(int perID, int milID, int bYear, int bMonth, int bDay)
    {
        this.perID=perID;
        this.milID=milID;
        birthYear=bYear;
        birthMonth=bMonth;
        birthDay=bDay;
    }
    public Person(int pID, int bYear, int bMonth, int bDay)
    {
        this(pID, 0, bYear, bMonth, bDay);
    }
```

생성자의 재호출을 위한 키워드 this가 존재하지 않았다고 생각해 보자. 이 클래스의 생성자 정의에 어떠한 변화가 있어야 하는가?

인자로 pID, 0, bYear, bMonth, bDay를 전달받는 생성자의 호출문장

키워드 this를 이용한 다른 생성자의 호출

```
class Person{
        private int perID;
        private int milID;
        private int birthYear;
        private int birthMonth;
        private int birthDay;

        public Person(int perID, int milID, int bYear, int bMonth, int bDay){
                this.perID=perID;
                this.milID=milID;
                birthYear=bYear;
                birthMonth=bMonth;
                birthDay=bDay;
        }
        public Person(int pID, int bYear, int bMonth, int bDay)          {
                this(pID, 0, bYear, bMonth, bDay);
        }
        public void showInfo()      {
                System.out.println("민번: "+perID);
                System.out.println("생년월일: " + birthYear + "/" + birthMonth +
                                                                "/" + birthDay);
```

키워드 this를 이용한 다른 생성자의 호출

```
                if(milID!=0)
                        System.out.println("군번: "+milID+'₩n');
                else
                        System.out.println("군과 관계 없음 ₩n");
        }
}

class CstOverloading
{
        public static void main(String[] args)
        {
                Person man=new Person(950123, 880102, 1995, 12, 3);
                Person woman=new Person(990117, 1999, 11, 7);
                man.showInfo();
                woman.showInfo();
        }
}
```

System.out.println & public static void main

System하고 out이 무엇이냐?

- System : java.lang 패키지에 묶여있는 클래스의 이름
  - import java.lang.*; 자동 삽입되므로 System이란 이름을 직접쓸 수 있음.
- out : staitc 변수이되 인스턴스를 참조하는 참조변수
  - PrintStream이라는 클래스의 참조변수

```
public class System
{
        public static final PrintStream out;
        . . . .
}
```

static final로 선언되었으니, 인스턴스의 생성 없이 system.out 이라는 이름으로 접근 가능하다!

System.out.println()은 Sytem 클래스의 멤버 out이 참조하는 인스턴스의 println 메소드를 호출하는 문장이다!

public static void main

```
class Employer      /* 고용주 */
{
    private int myMoney;
    public Employer(int money)
    {
        myMoney=money;
    }
    public void payForWork(Employee emp, int money)
    {
        if(myMoney<money)
            return;
        emp.earnMoney(money);
        myMoney-=money;
    }
    public void showMyMoney()
    {
        System.out.println(myMoney);
    }
}
```

```
class Employee      /* 고용인 */
{
    private int myMoney;
    public Employee(int money)
    {
        myMoney=money;
    }
    public void earnMoney(int money)
    {
        myMoney+=money;
    }
    public void showMyMoney()
    {
        System.out.println(myMoney);
    }
}
```

```
Employer emr=new Employer(3000);
Employee eme=new Employee(0);

emr.payForWork(eme, 1000);
emr.showMyMoney();
eme.showMyMoney();
```

이 문장들을 main 메소드로 묶어서 어디에 넣겠는가?

main 메소드는 static의 형태로 정의하기로 약속했으므로, 어디에 존재하든 상관없다!

다만 실행하는 방식에만 차이가 있을 뿐이다!

예제

```
class Employer{      /* 고용주 */
        private int myMoney;
        public Employer(int money) {
                myMoney=money;
        }
        public void payForWork(Employee emp, int money)  {
                if(myMoney<money)        return;
                emp.earnMoney(money);
                myMoney-=money;
        }
        public void showMyMoney() {
                System.out.println(myMoney);
        }
}
class Employee    /* 고용인 */
{
        private int myMoney;
        public Employee(int money) {        myMoney=money;          }
        public void earnMoney(int money) {          myMoney+=money;        }
        public void showMyMoney() {        System.out.println(myMoney);        }

}
```

String 클래스

String 클래스의 인스턴스 생성

- JAVA는 큰 따옴표로 묶여서 표현되는 문자열을 모두 인스턴스화 한다.
- 문자열은 String 이라는 이름의 클래스로 표현된다.

```
String str1 = "String Instance";
String str2 = "My String";
```

두 개의 String 인스턴스 생성,
그리고 참조변수 str1과 str2로 참조

```
System.out.println("Hello JAVA!");
System.out.println("My Coffee");
```

println 메소드의 매개변수형이 String이
기 때문에 이러한 문장의 구성이 가능하다.

String 클래스의 인스턴스 생성

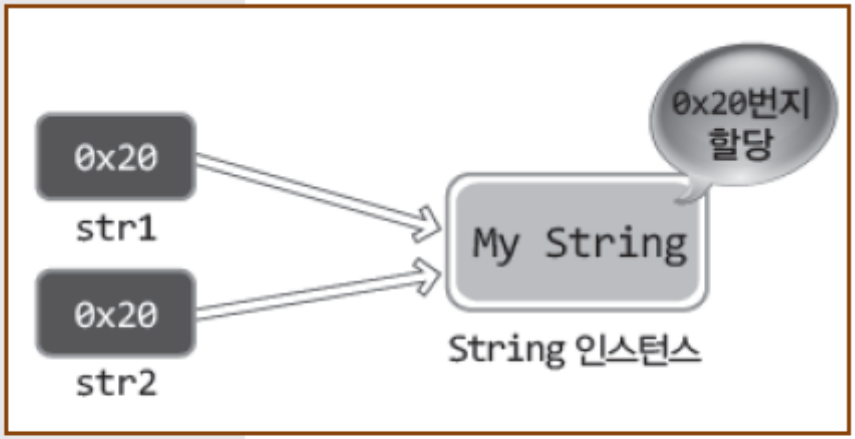
```
class StringInstance
{
	public static void main(String[] args)
	{
		java.lang.String str="My name is Sunny";

		int strLen1=str.length();
		System.out.println("길이 1: "+strLen1);

		int strLen2="한글의 길이는 어떻게?".length();
		//문자열의 선언 → 인스턴스 생성

		System.out.println("길이 2: "+strLen2);
	}
}
```

String 인스턴스는 상수형태의 인스턴스이다.

- String 인스턴스에 저장된 문자열의 내용은 변경이 불가능하다.
- 이는 동일한 문자열의 인스턴스를 하나만 생성해서 공유하기 위함이다.

String 인스턴스는 상수형태의 인스턴스이다.

```
public static void main(String[] args)
{
    String str1="My String";
    String str2="My String";
    String str3="Your String";

    if(str1==str2)
        System.out.println("동일 인스턴스 참조");
    else
        System.out.println("다른 인스턴스 참조");
```



String 인스턴스의 문자열 변경이 불가능하기 때문에 둘 이상의 참조변수가 동시에 참조를 해도 문제가 발생하지 않는다!

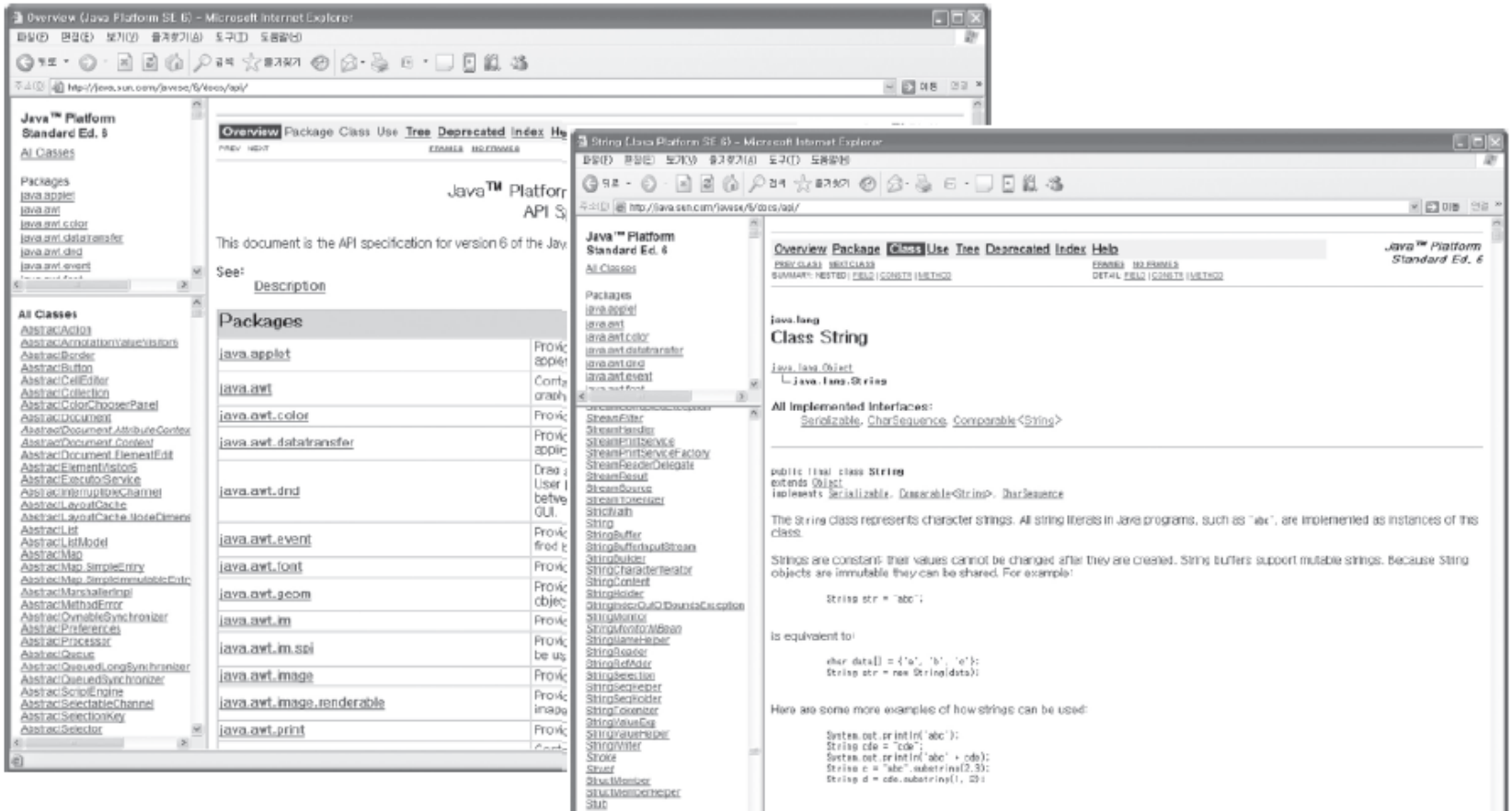
String 인스턴스는 상수형태의 인스턴스이다.

```
class ImmutableString
{
	public static void main(String[] args)
	{
		String str1="My String";
		String str2="My String";
		String str3="Your String";

		if(str1==str2)
			System.out.println("동일 인스턴스 참조");
		else
			System.out.println("다른 인스턴스 참조");

		if(str2==str3)
			System.out.println("동일 인스턴스 참조");
		else
			System.out.println("다른 인스턴스 참조");

	}
}
```

API Document의 참조를 통한 String 클래스의 인스턴스 메소드 관찰

생성자를 뺀 메소드의 수만 50개가 넘습니다.

- 책에서 메소드의 기능을 찾는 습관은 조금씩 버려야 한다.
- API 문서를 볼 줄 모르는 자바 개발자는 있을 수 없다.
- API 문서를 참조하지 않고 개발하는 자바 개발자도 있을 수 없다.



String 클래스가 제공하는 유용한 메소드들

• 문자열의 길이 반환	public int length()
• 두 문자열의 결합	public String concat(String str)
• 두 문자열의 비교	public int compareTo(String anotherString)

String 클래스가 제공하는 유용한 메소드들

```
class StringMethod
{
        public static void main(String[] args)
        {
                String str1="Smart";
                String str2=" and ";
                String str3="Simple";
                String str4=str1.concat(str2).concat(str3);

                System.out.println(str4);
                System.out.println("문자열 길이: "+str4.length());

                if(str1.compareTo(str3)<0)
                        System.out.println("str1이 앞선다");
                else
                        System.out.println("str3이 앞선다");
        }
}
```

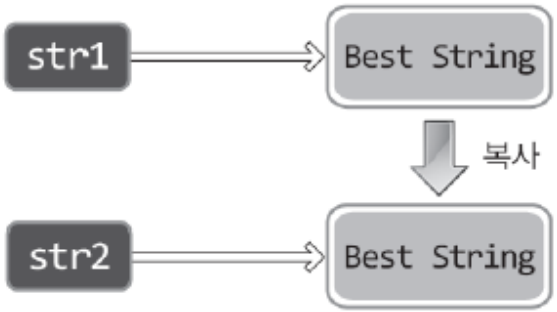
자바에서의 문자열 복사!

- 자바에서는 문자열을 상수의 형태로 관리하고, 또 동일한 유형의 문자열을 둘 이상 유지하지 않으므로 문자열의 복사라는 표현이 흔하지 않다.
- 무엇보다도 자바에서는 문자열을 복사가 필요 없다.

자바에서의 문자열 복사!

그러나 원하는 것이 인스턴스를 새로 생성해서 문자열의 내용을 그대로 복사하는 것이라면 다음과 같이 코드를 구성하면 된다.

```
String str1="Best String";
String str2=new String(str1);
```




```
public String(String original)
```

새로운 문자열 인스턴스의 생성에 사용되는 생성자

```
class StringCopy
{
    public static void main(String[] args)
    {
        String str1="Lemon";
        String str2="Lemon";
        String str3=new String(str2);

        if(str1==str2)
            System.out.println(" 실행 ");
        else
            System.out.println(" · · · ");

        if(str2==str3)
            System.out.println(" · · · ");
        else
            System.out.println(" 실행 ");
    }
}
```

비교 연산자는 참조 값 비교!

자바에서의 문자열 복사!

```
class StringCopy
{
	public static void main(String[] args)
	{
		String str1="Lemon";
		String str2="Lemon";
		String str3=new String(str2);

		if(str1==str2)
			System.out.println("str1과 str2는 동일 인스턴스 참조");
		else
			System.out.println("str1과 str2는 다른  인스턴스 참조");


		if(str2==str3)
			System.out.println("str2와 str3는 동일 인스턴스 참조");
		else
			System.out.println("str2와 str3는 다른  인스턴스 참조");

	}
}
```

+ 연산과 += 연산의 진실

```
public static void main(String[] args)
{
    String str1="Lemon"+"ade";
    String str2="Lemon"+'A';
    String str3="Lemon"+3;
    String str4=1+"Lemon"+2;
    Str4+='!';

    System.out.println(str1);
    System.out.println(str2);
    System.out.println(str3);
    System.out.println(str4);
}
```

```
String str1="Lemon".concat("ade");
String str2="Lemon".concat(String.valueOf('A'));
String str3="Lemon".concat(String.valueOf(3));
```

위 예제의 str4의 선언이 다음과 같이 처리된다면?

```
String str4=Strimg.valueOf(1).concat("Lemon").concat(String.valueOf(2));
```

아무리 많은 + 연산을 취하더라도 딱 두 개의 인스턴스만 생성된다. StringBuilder 클래스의 도움으로...

+ 연산과 += 연산의 진실

```
class StringAdd
{
	public static void main(String[] args)
	{
		String str1="Lemon"+"ade";
		String str2="Lemon"+'A';
		String str3="Lemon"+3;
		String str4=1+"Lemon"+2;
		str4+='!';

		System.out.println(str1);
		System.out.println(str2);
		System.out.println(str3);
		System.out.println(str4);
	}
}
```

StringBuilder & StringBuffer 클래스

StringBuilder

- StringBuilder는 문자열의 저장 및 변경을 위한 메모리 공간을 지니는 클래스
- 문자열 데이터의 추가를 위한 append와 삽입을 위한 insert 메소드 제공

StringBuilder

```
class BuilderString
{
    public static void main(String[] args)
    {
        StringBuilder strBuf=new StringBuilder("AB");
        strBuf.append(25);
        strBuf.append('Y').append(true);
        System.out.println(strBuf);
        strBuf.insert(2, false);
        strBuf.insert(strBuf.length(), 'Z');
        System.out.println(strBuf);
    }
}
```

buf: AB

buf: AB25

buf: AB25Ytrue

buf: ABfalse25Ytrue

buf: ABfalse25YtrueZ

실행결과

```
AB25Ytrue
ABfalse25YtrueZ
```

연속해서 함수호출이 가능한 이유는

append 메소드가 strBuf의 참조 값을 반환하기 때문이다.

StringBuilder

```
class BuilderString
{
	public static void main(String[] args)
	{
		StringBuilder strBuf=new StringBuilder("AB");
		strBuf.append(25);
		strBuf.append('Y').append(true);
		System.out.println(strBuf);

		strBuf.insert(2, false);
		strBuf.insert(strBuf.length(), 'Z');
		System.out.println(strBuf);
	}
}
```

참조를 반환하는 메소드

- this의 반환은 인스턴스 자신의 참조 값 반환을 의미한다.
- 그리고 이렇게 반환되는 참조 값을 대상으로 연이은 함수호출이 가능하다.

참조를 반환하는 메소드

```
class SimpleAdder
{
    private int num;
    public SimpleAdder() {num=0;}
    public SimpleAdder add(int num)
    {
        this.num+=num;
        return this;
    }
    public void showResult()
    {
        System.out.println("add result : "+num);
    }
}
```

실행결과

```
add result : 9
```

```
public static void main(String[] args)
{
    SimpleAdder adder=new SimpleAdder();
    adder.add(1).add(3).add(5).showResult();
}
```

add 함수는 adder의 참조 값을 반환한다.

참조를 반환하는 메소드

```
class SimpleAdder
{
        private int num;
        public SimpleAdder(){num=0;}

        public SimpleAdder add(int num){
                this.num+=num;
                return this;
        }
        public void showResult(){
                System.out.println("add result: "+num);
        }
}

class SelfReference
{
        public static void main(String[] args)
        {
                SimpleAdder adder=new SimpleAdder();
                adder.add(1).add(3).add(5).showResult();
        }
}
```

StringBuilder의 버퍼와 문자열 조합

- 추가되는 데이터 크기에 따라서 버퍼의 크기가 자동으로 확장된다.
- 생성자를 통해서 초기 버퍼의 크기를 지정할 수 있다.

- `public StringBuilder()` 기본 16개의 문자저장 버퍼 생성
- `public StringBuilder(int capacity)` capacity개의 문자저장 버퍼 생성
- `public StringBuilder(String str)` str.length()+16 크기의 버퍼 생성

StringBuilder의 버퍼와 문자열 조합

문자열의 복잡한 조합의 과정에서는 StringBuilder의 인스턴스가 활용된다.
때문에 추가로 생성되는 인스턴스의 수는 최대 두 개이다!

```
String str4=1+"Lemon"+2;
```

```
new StringBuilder().append(1).append("Lemon").append(2).toString();
```

StringBuilder 인스턴스의 생성에서 한 개

toString 메소드의 호출에 의해서 한 개

StringBuffer 클래스

- StringBuffer 클래스와 StringBuilder 클래스의 공통점 세 가지
  - 메소드의 수(생성자 포함)
  - 메소드의 기능
  - 메소드의 이름과 매개변수형
- StringBuffer 클래스와 StringBuilder 클래스의 차이점
  - StringBuffer는 쓰레드에 안전, StringBuilder는 쓰레드에 불안전

예제 StringBuilder.java에서 StringBuilder 클래스를
StringBuffer 클래스로 바꿔도 컴파일 및 실행이 된다.

문제

문제 1. 다음 형태로 String 인스턴스를 생성.

String str = “ABCDEFGHIJKLMN”;

그리고 이 문자열을 역순으로 다시 출력하는 프로그램을 작성.

문제 2. 다음 형태로 주민번호를 담고 있는 String 인스턴스를 하나 생성

String str = “990929-1010123”

이 문자열을 이용하여 중간에 삽입된 – 를 뺀 String 인스턴스를 생성.

Project (전화번호 관리 프로그램)

Version 0.1

전화번호 관리 프로그램.

PhoneInfor 라는 이름의 클래스를 정의해 보자.
클래스는 다음의 데이터들의 문자열 형태로 저장이 가능 해야 하며,
저장된 데이터의 적절한 출력이 가능하도록 메소드 정의

- **이름** **name** **String**
- **전화번호** **phoneNumber** **String**
- **생년월일** **birthday** **String**

단, 생년월일 정보는 저장할 수도 있고, 저장 않을 수도 있게끔 생성자 생성.