



# 객체의 생성 및 클래스의 정적 구성 요소

---

Java Programming  
강 성 관



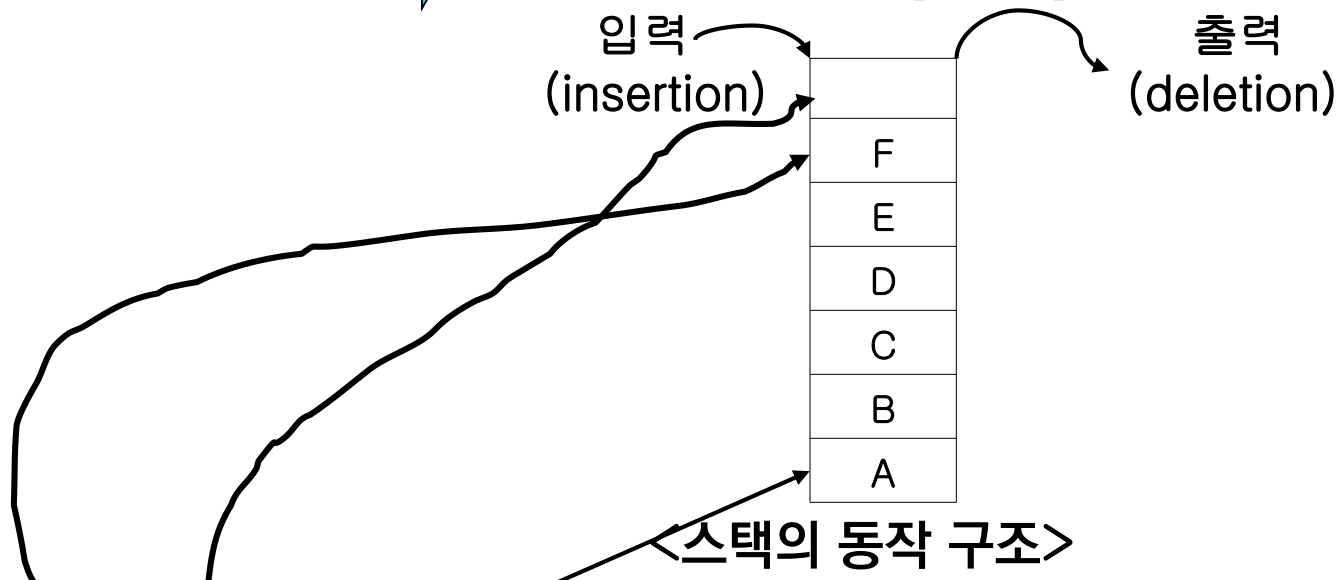
# 목차

---

- Stack의 정의
- 생성자 메서드
- 생성자 오버로딩
- 레퍼런스 this
- 생성자 this( )
- 정적 기억 공간 & 정적 변수
- 정적 멤버변수를 갖는 클래스
- 정적 메서드(Static Method)
- Math 클래스 내부의 정적 멤버변수와 정적 멤버함수

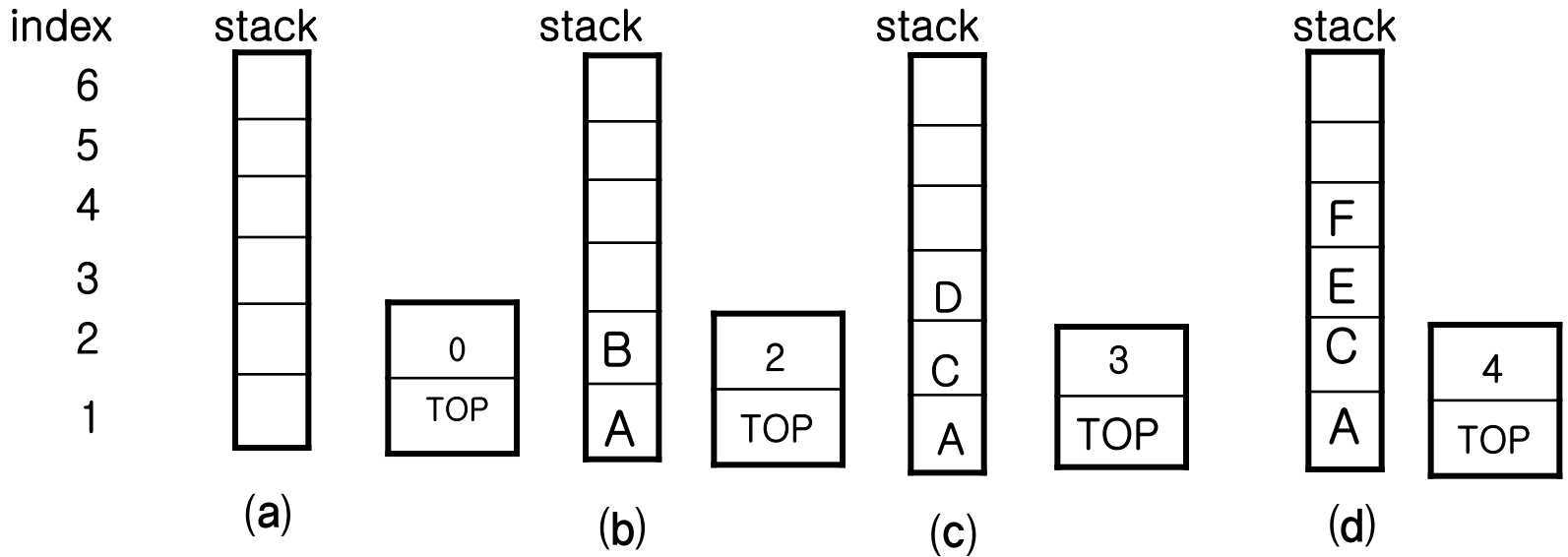
# Stack(스택)의 정의 (1)

- ① 프로그램에서 각 함수에 의해 필요로 되는 자료들을 보관하기 위해 할당된 메모리의 특수한 영역
- ② 입·출력(Insertion/Deletion)이 리스트의 한쪽 끝으로만 제한된 리스트 → last-in-first-out [LIFO] 구조



- ③ limit : 입·출력이 허용되는 리스트의 끝  
BOTTOM : TOP의 반대쪽 끝
- ④ Stack pointer : 가장 최근에 입력된 value의 위치를 가리키는 포인터  
(출력 우선 순위가 가장 높은 value 를 가리키는 포인터)
- ⑤ 컴파일러, 서브루틴의 복귀주소(return address)기억에 사용

# Stack (2)



# Stack (3)

## (1) 입력과 출력의 개념

- ㉠ 초기상태  $TOP = nil$
- ㉡ A를 입력하고 ( $TOP=1$ ) 계속해서 B를 입력한 상태
- ㉢ B를 출력하고 ( $TOP=1$ ) C를 입력( $TOP=2$ ) 한 다음 이어서 D를 입력한 상태
- ㉣ D를 출력하고 ( $TOP=2$ ) E를 입력( $TOP=3$ ) 한 다음 이어서 F를 입력한 상태

## (2) 스택에서 입력과 출력의 알고리즘

- ① 입력(insert, push, put) : 최대 인덱스값(stack length)을 N이라 하면
  - ㉠ IF  $TOP = nil$ 이면  $TOP=0$ ;
  - ㉡  $TOP \leftarrow TOP+1$ ;
  - ㉢  $TOP > N$  이면 overflow 이므로 exit;
  - $TOP \leq N$  이면  $STACK[TOP] \leftarrow new\ atom$ ;
- ② 출력(delete, pop, pull)
  - ㉠ IF  $TOP = nil$ 이면 underflow 이므로 exit;
  - ㉡ remove  $STACK[TOP]$ ;
  - ㉢  $TOP \leftarrow TOP - 1$ ;
  - ㉣ IF  $TOP = 0$ 이면  $TOP \leftarrow nil$ ;



# 생성자 메서드

## □ 클래스의 생성자 (Constructor) 메서드

- 새로운 객체에 대해서 기억 공간(힙 영역)이 할당
- 객체가 선언될 때 마다 자동적으로 호출되는 메서드 .
- 클래스와 같은 이름 사용
- 반환 값(타입)은 없다.(void형 조차도 가질 수 없다.)
- 파라미터는 가질 수 있다.
- 기능
  - 클래스 내의 필드들에게 초기값 부여

## ★ 생성자의 기본 형식

```
접근_지정자    클래스_이름(인수1, 인수2,...){  
    문장1;  
}
```

# <예제> 생성자 정의하기

```
001: class MyDate{
002:     private int year;
003:     private int month;
004:     private int day;
005:     public MyDate(){
006:         System.out.println("[생성자] : 객체가 생성될 때 자동 호출됩니다.");
007:     }
008:     public void print(){
009:         System.out.println(year+ "/" +month+ "/" +day);
010:     }
011:
012: }
013: public class ConstructorTest02 {
014:     public static void main(String[] args) {
015:         MyDate d = new MyDate();
016:         d.print();
017:     }
018: }
```



## <예제> 생성자의 정의와 호출

```
001: class MyDate{
002:     private int year;
003:     private int month;
004:     private int day;
005:     public MyDate(){
006:         year=2006;
007:         month=4;
008:         day=1;
009:     }
010:     public void print(){
011:         System.out.println(year+ "/" +month+ "/" +day);
012:     }
013: }
014: public class ConstructorTest03 {
015:     public static void main(String[] args) {
016:         MyDate d=new MyDate();
017:         d.print();
018:     }
019: }
```





# 생성자 오버로딩(Constructor Overloading)

<예제> - 전달인자를 갖는 생성자 정의

---

```
001: class MyDate{
002:     private int year;
003:     private int month;
004:     private int day;
005:     //생성자는 속성(멤버변수)들의 초기화 작업을 목적으로 한다.
006:
007:     //[1] 전달인자 없는 생성자 정의
008:     public MyDate(){
009:         year=2006;    month=4;    day=1;
010:     }
011:     //[2] 전달인자 있는 생성자 정의
012:     public MyDate(int new_year, int new_month, int new_day){
013:         year=new_year;
014:         month=new_month;
015:         day=new_day;
016:     }
```

# 생성자 오버로딩(Constructor Overloading)

<예제> - 전달인자를 갖는 생성자 정의

---

```
017:
018: public void print(){
019: System.out.println(year+ "/" +month+ "/" +day);
020: }
021:}
022:
023:public class ConstructorTest04 {
024: public static void main(String[] args) {
025:     MyDate d=new MyDate();
026:     d.print();
027:
028:     MyDate d2=new MyDate(2007, 7, 19);
029:     d2.print();
030: }
031:}
```




# 디폴트 생성자(Default Constructor) -파라미터가 없는 생성자

---

<예제> 자바가 제공하는 디폴트 생성자 호출

```
001: class MyDate{
002:   private int year;
003:   private int month;
004:   private int day;
005:   public void print(){
006:     System.out.println(year+ "/" +month+ "/" +day);
007:   }
008: }
009: public class ConstructorTest01 {
010:   public static void main(String[] args) {
011:     MyDate d=new MyDate(); //디폴트 생성자 호출
012:     d.print();
013:   }
014: }
```

## <예제>디폴트 생성자 부재로 인한 에러 발생



```
001: class MyDate{
002:   private int year;
003:   private int month;
004:   private int day;
005:   public MyDate(int new_year, int new_month, int new_day){
006:     year=new_year;
007:     month=new_month;
008:     day=new_day;
009:   }
010:   public void print(){
011:     System.out.println(year+ "/" +month+ "/" +day);
012:   }
013: }
014:
015: public class ConstructorTest05 {
016:   public static void main(String[] args) {
017:     MyDate d=new MyDate();
018:     d.print();
019:
020:     MyDate d2=new MyDate(2007, 7, 19);
021:     d2.print();
022:   }
023: }
```

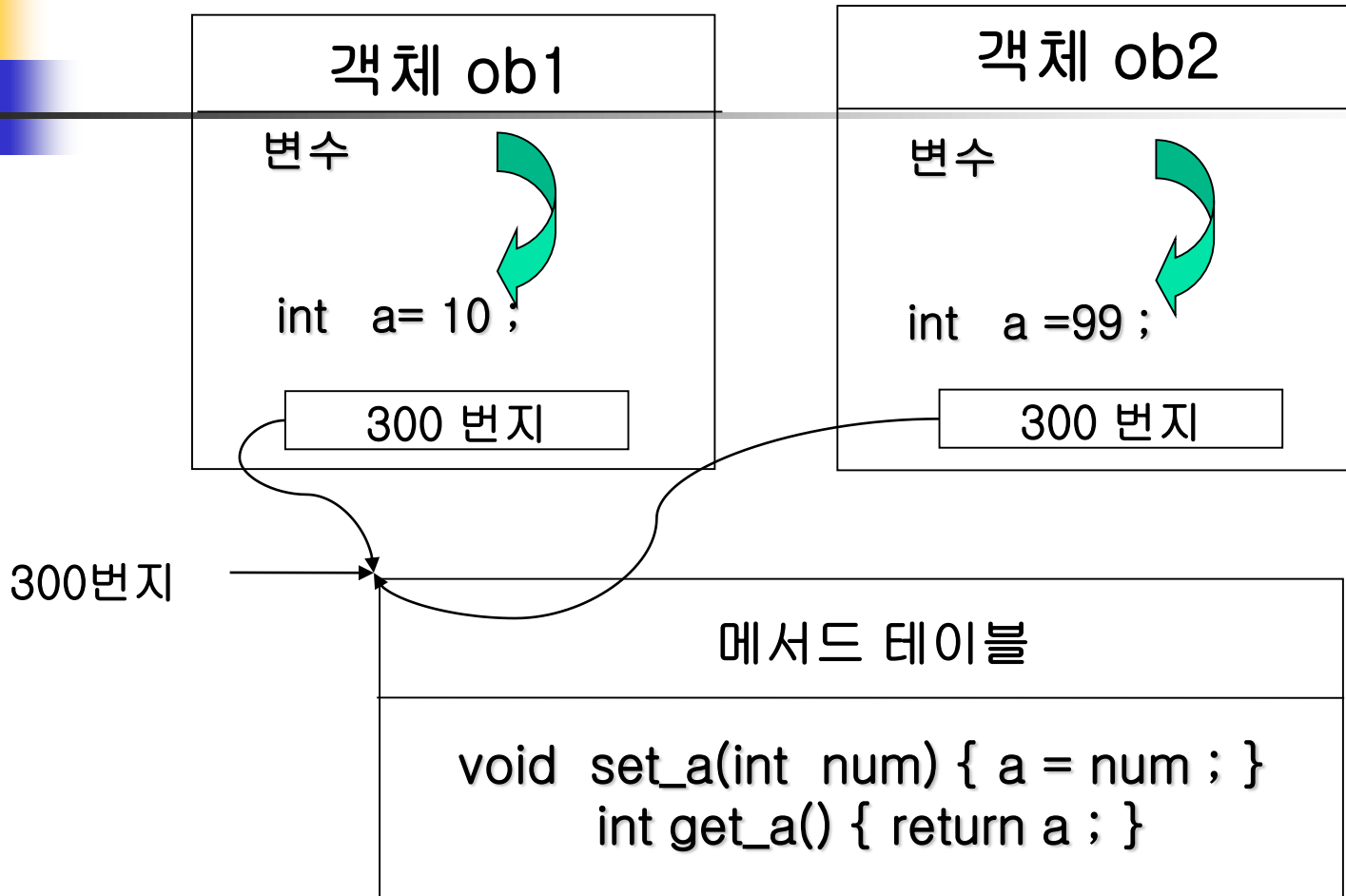


# 레퍼런스 this(1)

---

- 클래스를 구성하는 멤버변수는 객체 생성할 때마다 새롭게 메모리 할당을 하기 때문에 객체 단위로 따로 관리됨.
- 하지만 메서드는 모든 객체가 이 메서드를 공유해서 사용하기 때문에 어떤 객체에 의해서 사용되는 메서드인지 구분해야 할 목적으로 레퍼런스 this가 존재해야 함.

## 레퍼런스 this (2)



< 각각 다른 메모리 공간에 객체가 들어간 모습 >

## 레퍼런스 this (3)

0x10 번지에  
할당 되었다면

객체 ob1

this

0x10번지를  
의미하는 참조자

0x20 번지에  
할당 되었다면

객체 ob2

this

0x20번지를  
의미하는 참조자



## 레퍼런스 this (4)

- JAVA에서 사용하는 특별한 의미의 **this** 참조자
  - 모든 클래스 메서드들은 **this** 라는 참조자를 숨겨진 매개변수로 가진다.
  - 메서드를 호출한 객체(자기 자신의 객체)를 가리키는 참조자
  - **static 메서드**는 **this** 참조자를 갖고 있지 않다.
    - 모든 객체는 키워드인 **this** 참조자를 통해 자기 자신의 주소에 접근한다.
  - **this** 참조자는 객체의 정적메서드(**static 메서드**)가 아닌 메서드가 호출될 때 암시적 인수로 객체에 전달된다.






## this를 사용해야만 하는 경우

---


- 메서드(생성자 포함)의 전달인자와 객체의 필드가 동일한 이름일 경우
  - 전달인자와 필드가 구분되지 않기 때문에 문제가 발생함
    - 이를 구분 짓기 위해서 필드 앞에 레퍼런스 this를 덧붙임.

## <예제> 멤버변수와 속성이 동일해서 생긴 문제점




```
001: class MyDate{
002: private int year;
003: private int month;
004: private int day;
005: //생성자 정의하기
006: public MyDate(){
007: }
008: public MyDate(int new_year, int new_month, int new_day){
009:   year=new_year;   month=new_month;   day=new_day;
010: }
011: //전달인자가 객체 속성의 이름과 동일한 메서드
012: public void SetYear(int year){
013:   //this.year=year;
014:   year=year;
015: }
```

## <예제> 멤버변수와 속성이 동일해서 생긴 문제점



```
016: public void SetMonth(int new_month){
017:     month=new_month;
018: }
019: public void print(){
020:     System.out.println(year+ "/" +month+ "/" +day);
021: }
022:}
023:
024:public class ConstructorTest06 {
025:     public static void main(String[] args) {
026:         MyDate d=new MyDate(2007, 7, 19);
027:         d.print();
028:         d.SetYear(2008); //변경되지 않음
029:         d.print();      //-----
030:         d.SetMonth(8);  //변경됨
031:         d.print();      //-----
032:     }
033:}
```

## <예제> 속성(멤버변수) 앞에 레퍼런스 this 붙이기



```
001: class MyDate{
002:     private int year;
003:     private int month;
004:     private int day;
005:     public MyDate(){
006:     }
007:     //생성자 역시 매개변수의 이름을 속성과 동일하게 줄 수 있다.
008:     public MyDate(int year, int month, int day){
009:         //멤버변수로 속성 값을 초기화하려면 대입연산자 왼쪽에 this를 붙여야 한다.
010:         this.year=year; this.month=month; this.day=day;
011:     }
012:     public void SetYear(int year){ //대입연산자 왼쪽에 this를 붙였기에
013:         his.year=year;           //속성 값이 변경됨
014:     }
015:     public void SetMonth(int month){ //대입연산자 왼쪽에 this를 붙였기에
016:         this.month=month;         //속성 값이 변경됨
017:     }
```



## <예제> 속성(멤버변수) 앞에 레퍼런스 this 붙이기

---

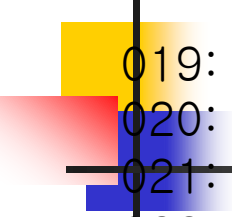
```
018: public void print(){
019:     System.out.println(year+ "/" +month+ "/" +day);
020: }
021:}
022:
023:public class ConstructorTest07 {
024:     public static void main(String[] args) {
025:         MyDate d=new MyDate(2007, 7, 19);
026:         d.print();
027:         d.SetYear(2008); //2008년으로 변경
028:         d.SetMonth(8);   //8월로 변경
029:         d2.print();
030:     }
031:}
```

# 생성자 this( )

<예제> this로 생성자 호출하기

```
001: class MyDate{
002:     private int year;
003:     private int month;
004:     private int day;
005:     public MyDate(){
006:         this(2006, 1, 1);           //14:에 정의된 생성자 호출
007:     }
008:     public MyDate(int new_year){
009:         this(new_year, 1, 1);       //14:에 정의된 생성자 호출
010:     }
011:     public MyDate(int new_year, int new_month){
012:         this(new_year, new_month, 1); //14:에 정의된 생성자 호출
013:     }
014:     public MyDate(int new_year, int new_month, int new_day){
015:         year=new_year;
016:         month=new_month;
017:         day=new_day;
018:     }
```

# 생성자 this( )



```
019:
020: public void print(){
021: System.out.println(year+ "/" +month+ "/" +day);
022: }
023:}
024:
025:public class ConstructorTest10 {
026: public static void main(String[] args) {
027:     MyDate d=new MyDate(2007, 7, 19); //14:에 정의된 생성자 호출
028:     d.print();
029:     MyDate d2=new MyDate(2007, 7);    //11:에 정의된 생성자 호출
030:     d2.print();
031:     MyDate d3=new MyDate(2007);       //8:에 정의된 생성자 호출
032:     d3.print();
033:     MyDate d4=new MyDate();           //5:에 정의된 생성자 호출
034:     d4.print();
035: }
036:}
```



# 정적 기억 공간 & 정적 변수

---

- 정적 기억 공간
  - 프로그램이 실행되는 동안에 지속적으로 존재하는 특별한 공간.
- 정적 변수(static 변수)
  - *static* 키워드를 붙여서 정의.
    - 클래스단위의 멤버를 위한 static 예약어
  - 프로그램이 실행되는 전체 시간 동안 계속 존속한다.
  - 스택(stack)에 저장되지 않는다.
  - 넉넉하게 크기가 정해진 메모리 블록은 프로그램이 실행되는 동안에 크기가 변하지 않는다.

Ex) `static double fee = 56.50 ;      static int fee = 56 ;`





# 정적 멤버변수를 갖는 클래스

- 모든 객체 인스턴스들이 하나의 멤버변수를 공유할 필요성이 있을 경우 **static**이란 예약어를 사용.
- static 필드
  - 데이터 값의 공유를 위해 선언하는 공간 (멤버필드로만 가능)
    - 클래스 이름으로 접근 가능
    - 객체 발생 전 메모리 할당
- **static 초기화 영역** : static 멤버 필드의 값을 초기화 하기 위한 영역  
형식) `static { 초기화 구문... }`

# 정적 멤버변수를 갖는 클래스

## ■ 정적 초기화 블록

- 정적 초기화 블록 : static 키워드가 붙은 블록
  - 정적 필드의 초기값 설정에 주로 사용됨
  - 클래스가 사용되기 전에 자바 가상 기계에 의해 단 한 번 호출됨
  - 정적 필드는 특정 객체에 속하지 않기 때문에, 생성자에서 초기값을 대입하면 안됨.
  - 정적 필드의 초기값을 선언문에서 대입할 수 없는 경우.
    - 정적 초기화 블록(static initialization block)을 사용해야 함.



## 정적 초기화 블록

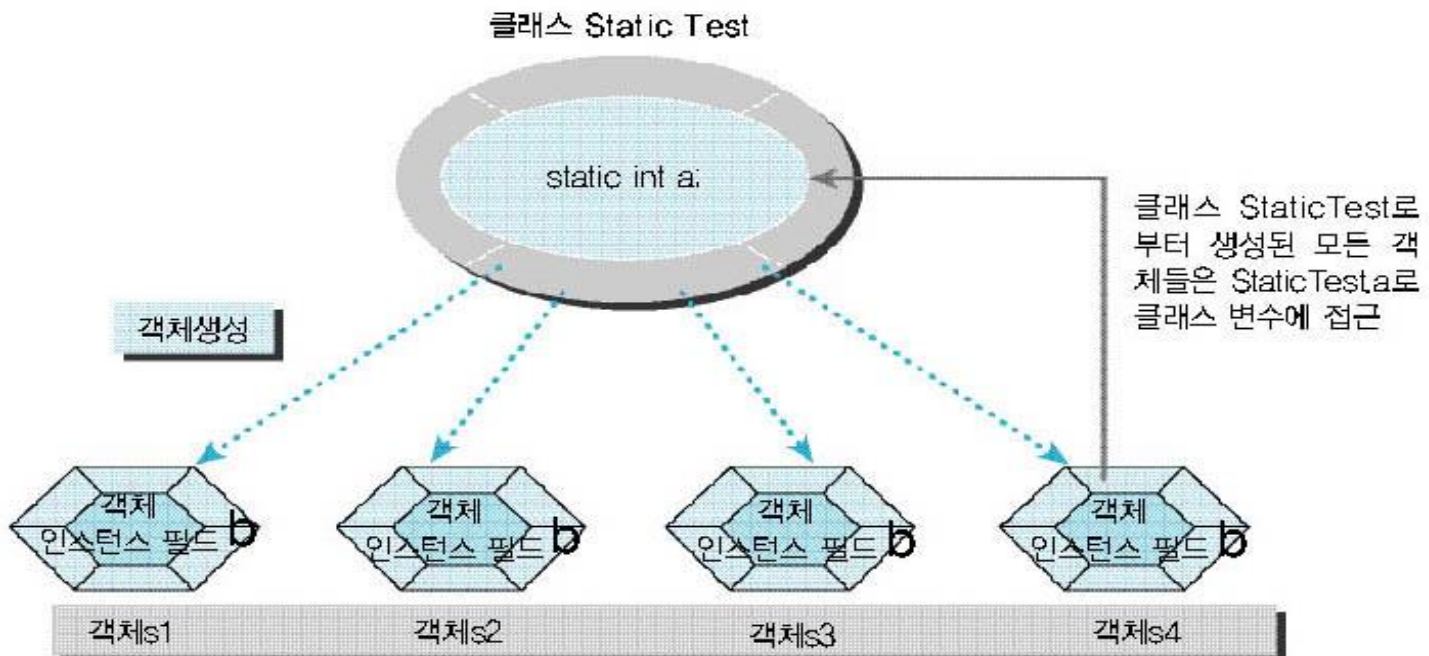
- [예제 ] 정적 초기화 블록을 포함하는 클래스의 예

```
class HundredNumbers {  
    static int arr[];  
    static {  
        arr = new int[100];  
        for (int cnt = 0; cnt < 100; cnt++)  
            arr[cnt] = cnt;  
    }  
}
```

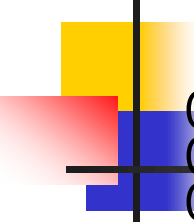
정적 초기화 블록

# 정적 멤버변수를 갖는 클래스

```
class StaticTest{  
    static int a=10;  
    int b=20;  
}  
  
StaticTest s1 = new StaticTest();  
StaticTest s2 = new StaticTest();  
StaticTest s3 = new StaticTest();  
StaticTest s4 = new StaticTest();
```



## <예제> 정적 멤버변수와 인스턴스 멤버변수의 차이점



```
001: class StaticTest{
002:   static int a=10;
003:   int b=20;
004: }
005: class StaticTest01 {
006:   public static void main(String[] args){
007:     System.out.println("StaticTest.a->" + StaticTest.a);
008:     StaticTest s1 = new StaticTest();
009:     StaticTest s2 = new StaticTest();
010:
011:     System.out.println("s1.a->" + s1.a + "\t s2.a->" + s2.a);
012:     System.out.println("s1.b->" + s1.b + "\t s2.b->" + s2.b);
013:
014:     s1.a=100;
015:     System.out.print("s1.a->" + s1.a );
016:     System.out.println("\t s2.a->" + s2.a);
017:
018:     s1.b=200;
019:     System.out.print("s1.b->" + s1.b);
020:     System.out.println("\t s2.b->" + s2.b);
021:   }
022: }
```



# 정적 메서드(Static Method)

- 인스턴스 차원이 아닌 클래스 차원에서 사용하도록 설계하기 위한 정적 메소드

```
public static int getA( ){  
    return a;  
}
```

- 클래스 차원에서 사용할 수 있는 메서드란 객체(인스턴스)의 생성 없이 클래스명으로 호출 가능하다는 의미

```
StaticTest.getA();
```



## <예제> 정적 메서드 정의하기

---

```
001: class StaticTest{
002:   private static int a=10;
003:   private int b=20;
004:
005:   public static void setA(int new_a){
006:     a = new_a;
007:   }
008:   public static int getA(){
009:     return a;
010:   }
011: }
```



## <예제> 정적 메서드 정의하기

---

```
012:public class StaticTest02 {  
013:  public static void main(String[] args) {  
014:  
015:    System.out.println(StaticTest.getA());  
016:  
017:    StaticTest s1=new StaticTest();  
018:    StaticTest s2=new StaticTest();  
019:  
020:    s1.setA(10000);  
021:    int res1=s1.getA();  
022:    System.out.println(res1);  
023:    System.out.println(s2.getA());  
024:  }  
025:}
```





# 정적 메서드와 인스턴스 메서드의 차이점


---

1. 정적 메서드에서는 this 레퍼런스를 사용할 수 없음.
2. 정적 메서드에서는 인스턴스 변수를 사용할 수 없음.
3. 정적 메서드는 오버라이딩(overriding)되지 않음.

## <예제>-정적 메서드에서 this 사용 불가능

```
001: class StaticTest{
002:   private static int a=10;
003:   private int b=20;
004:   public static void printA(){ //정적 메서드에서는 this를 사용하지 못함
005:     System.out.println(a);
006:     System.out.println(this.a); //컴파일 에러 발생
007:   }
008:
009:   public void printB(){ //this는 인스턴스 메서드에서 여러 객체에 의해서
010:     System.out.println(this.b); //메서드가 호출될 때 이를 구분하기 위해서 사용
011:   }
012: }
013: public class StaticTest03 {
014:   public static void main(String[] args) {
015:     StaticTest.printA();
015:     StaticTest s1 = new StaticTest();
016:     StaticTest s2 = new StaticTest();
017:     s1.printB();
018:     s2.printB();
019:   }
020: }
```

# <예제>-정적 메서드에서 인스턴스 멤버 사용 불가능



```
001: class StaticTest{
002:     private static int a=10;
003:     private int b=20;
004:     public static void printA(){
005:         System.out.println(a);
006:         System.out.println(b); //컴파일 에러 발생
007:     }
008:
009:     public void printB(){
010:         System.out.println(b);
011:     }
012: }
013: public class StaticTest04 {
014:     public static void main(String[] args) {
015:         StaticTest.printA();
015:         StaticTest s1 = new StaticTest();
016:         StaticTest s2 = new StaticTest();
017:         s1.printB();
018:         s2.printB();
019:     }
020: }
```

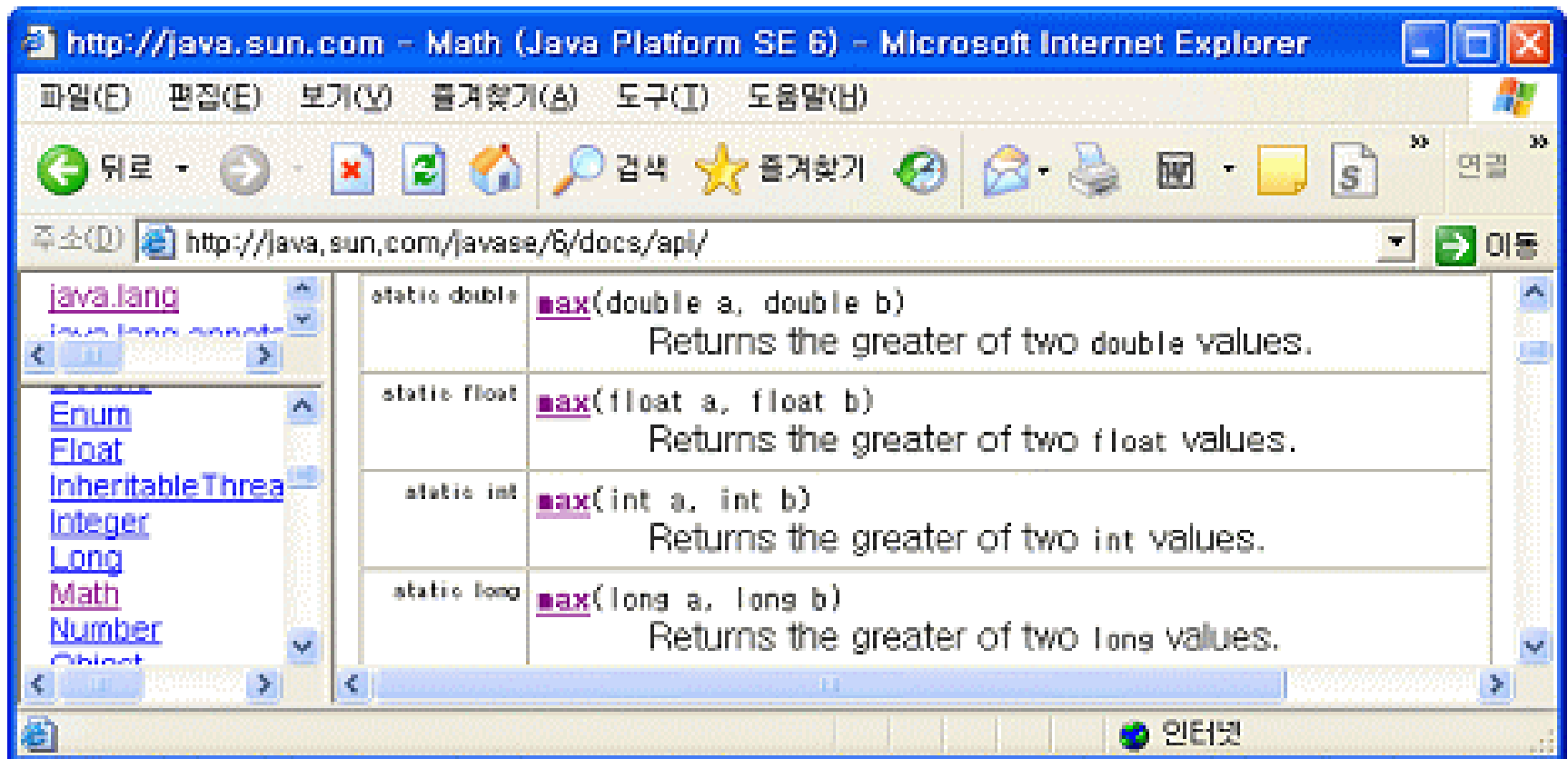


## <예제>-인스턴스 멤버변수와 정적 멤버변수의 메모리 할당 순서

---

```
001:public class StaticTest05 {  
002:  int b= check(2);  
003:  static int a=check(1);  
004:  public static int check(int i){  
005:    System.out.println("call "+i);  
006:    return 0;  
007:  }  
008:  public static void main(String[] args) {  
009:    System.out.println("메인");  
010:    StaticTest05 s2=new StaticTest05( );  
011:  }  
012:  static int c=check(3);  
013:}
```

# Math 클래스 내부의 정적 멤버변수와 정적 멤버함수



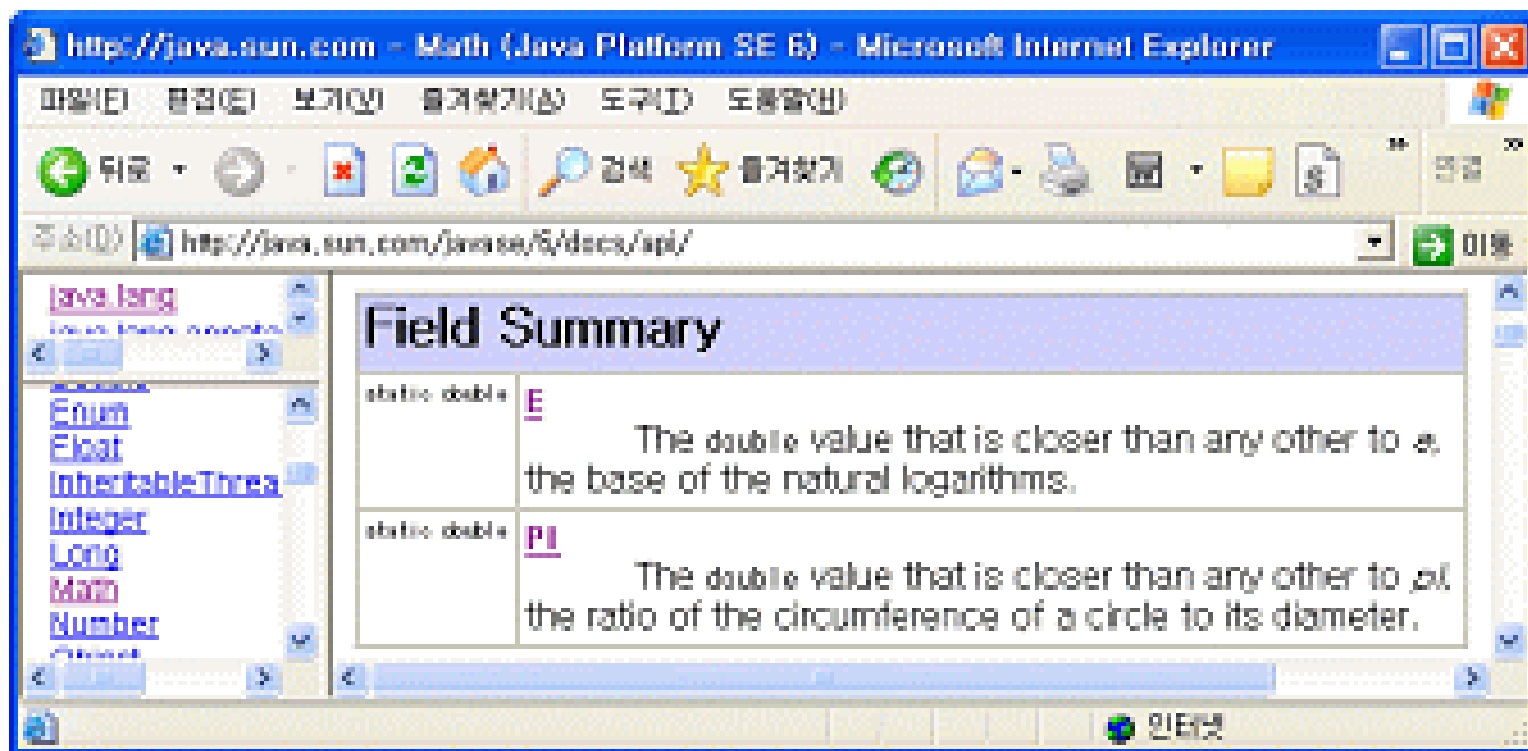


## <예제> Math 클래스의 정적 메서드의 사용 예

---

```
001: class StaticTest06 {  
002:     public static void main(String[] args) {  
003:         int a=40, b=30, c=10;  
004:  
005:         int res;  
006:         res=Math.max(a, b);  
007:         System.out.println(a + "와 " + b + " 중 최대값: "+res);  
008:  
009:         res=Math.max(b, c);  
010:         System.out.println(b + "와 " + c + " 중 최대값: "+res);  
011:     }  
012: }
```

# Math 클래스 내부의 정적 멤버변수와 정적 멤버함수





## <예제> Math 클래스의 정적 멤버변수 의 사용 예

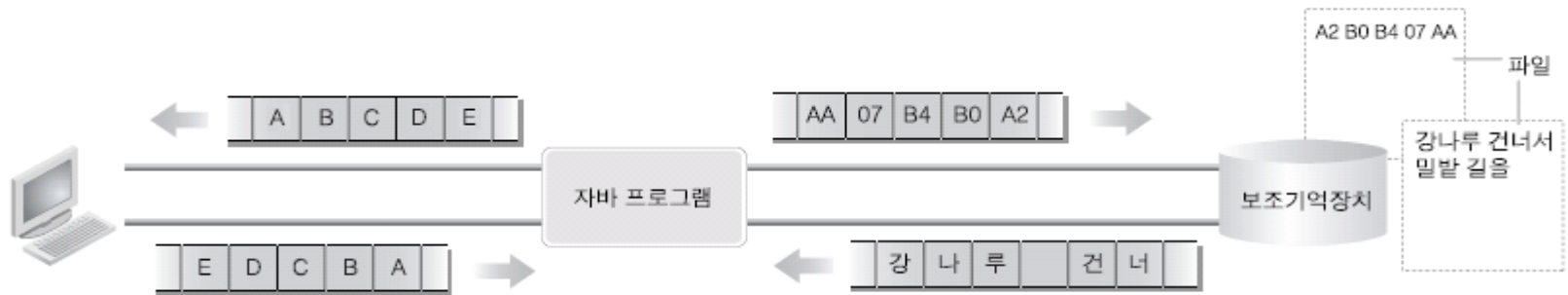
---

```
001: class StaticTest07 {  
002:     public static void main(String[] args) {  
003:         System.out.println(Math.PI);  
004:         int r=5;  
005:         double area;  
006:         area=r*r*Math.PI;  
007:         System.out.println("반지름이 "+r+"인 원의 면적 "+ area);  
008:     }  
009: }
```



# • 스트림(Stream)이란?

- 일차원적인 데이터의 연속적인 흐름



## • 흐름의 방향에 따른 분류

- 입력 스트림(input stream)
- 출력 스트림(output stream)

## • 데이터의 형태에 따른 분류

- 문자 스트림(character stream)
- 바이트 스트림(byte stream)

# 입출력 기능/성능을 향상시키는 클래스들

## • 버퍼를 이용해서 입출력 성능을 향상시키는 클래스들

- 스트림의 종류에 따라 4개의 클래스가 있음

클래스 이름	설명
BufferedInputStream	바이트 입력 스트림을 버퍼링하는 클래스
BufferedOutputStream	바이트 출력 스트림을 버퍼링하는 클래스
BufferedReader	문자 입력 스트림을 버퍼링하는 클래스
BufferedWriter	문자 출력 스트림을 버퍼링하는 클래스



## JAVA에서 equals()와 ==연산자의 차이(1)

- equals() 메서드
  - 클래스 인스턴스를 비교해서, 인스턴스의 내용이 같은지를 비교하는 메서드.
  - 클래스 객체의 내용을 비교하여 같은 내용의 객체 인가를 확인할 때 사용.
- == 연산자
  - 레퍼런스의 값이 같은지를 비교할 때 사용.
  - 객체 안의 내용물(객체 안에 들어있는 변수들의 값)의 값들이 같은지를 비교하는 데는 사용할 수 없음.

## JAVA에서 equals()와 ==연산자의 차이(2)

