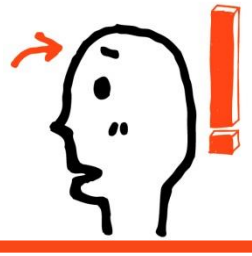
Android Java

19장 자바의 다양한 기본 클래스

자바의 다양한 기본 클래스

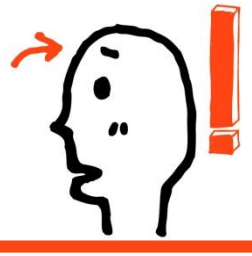


01 Wrapper 클래스

02 BigInteger 클래스와 BigDecimal 클래스

03 Math 클래스와 난수의 생성, 그리고 문자열 토큰의 구분

기본 자료형 데이터를 감싸는 Wrapper 클래스



```
public static void showData(Object obj)
{
    System.out.println(obj);
}
```

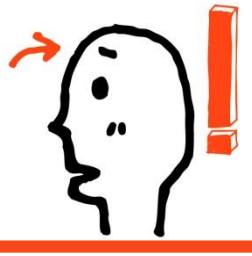
왼쪽의 메소드에서 보이듯이 기본 자료형 데이터를 인스턴스화 해야 하는 상황이 발생할 수 있다. 이러한 상황에 사용할 수 있는 클래스를 가리켜 **Wrapper 클래스**라 한다.

```
class IntWrapper
{
    private int num;
    public IntWrapper(int data)
    {
        num=data;
    }
    public String toString()
    {
        return ""+num;
    }
}
```

프로그래머가 정의한 int형 기본 자료형에 대한 Wrapper 클래스!

이렇듯 Wrapper 클래스는 기본 자료형 데이터를 저장 및 참조할 수 있는 구조로 정의된다.

자바에서 제공하는 Wrapper 클래스



- Boolean — Boolean(boolean value)
- Character — Character(char value)
- Byte — Byte(byte value)
- Short — Short(short value)
- Integer — Integer(int value)
- Long — Long(long value)
- Float — Float(float value), Float(double value)

순전히 기본 자료형 데이터의 표현이 목적이라면, 별도의 클래스 정의 없이 제공되는 Wrapper 클래스를 사용하면 된다!

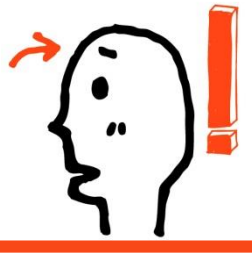
위의 클래스 이외에도 문자열 기반으로 정의된 Wrapper 클래스도 존재하기 때문에 다음과 같이 인스턴스 생성도 가능. 단, Character 클래스 제외!

```
Integer num1=new Integer("240")
Double num2=new Double("12.257");
```

자바제공 Wrapper 클래 사용의 예!

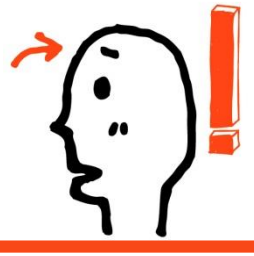
```
public static void main(String[] args)
{
    Integer intInst=new Integer(3);
    showData(intInst);
    showData(new Integer(7));
}
```

주요 메소드



메소드	설명
static int bitCount(int i)	인자 i의 이진수 표현에서 1의 개수를 리턴
float floatValue()	float 타입으로 변환된 값 리턴
int intValue()	int 타입으로 변환된 값 리턴
long longValue()	long 타입으로 변환된 값 리턴
short shortValue()	short 타입으로 변환된 값 리턴
static int parseInt(String s)	스트링 s를 10진 정수로 변환된 값 리턴
static int parseInt(String s, int radix)	스트링 s를 지정된 진법의 정수로 변환된 값 리턴
static String toBinaryString(int i)	인자 i를 이진수 표현으로 변환된 스트링 리턴
static String toHexString(int i)	인자 i를 16진수 표현으로 변환된 스트링 리턴
static String toOctalString(int i)	인자 i를 8진수 표현으로 변환된 스트링 리턴
static String toString(int i)	인자 i를 스트링으로 변환하여 리턴

Wrapper 활용



Wrapper 객체로부터 기본 데이터 타입 알아내기

```
Integer i = new Integer(10);
int ii = i.intValue(); // ii = 10

Character c = new Character('c' );
char cc = c.charValue(); // cc = 'c'
```

```
Float f = new Float(3.14);
float ff = f.floatValue(); // ff = 3.14

Boolean b = new Boolean(true);
boolean bb = b.booleanValue(); // bb = true
```

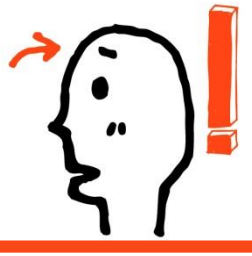
문자열을 기본 데이터 타입으로 변환

```
int i = Integer.parseInt("123"); // i = 123
boolean b = Boolean.parseBoolean("true"); // b = true
float f = Float.parseFloat("3.141592" ); // f = 3.141592
```

기본 데이터 타입을 문자열로 변환

```
String s1 = Integer.toString(123); // 정수 123을 문자열 "123" 으로 변환
String s2 = Integer.toHexString(123); // 정수 123을 16진수의 문자열 "7b"로 변환
String s3 = Float.toString(3.141592f); // 실수 3.141592를 문자열 "3.141592"로 변환
String s4 = Charater.toString('a'); // 문자 'a'를 문자열 "a"로 변환
String s5 = Boolean.toString(true); // 불린 값 true를 문자열 "true"로 변환
```

Wrapper 클래스의 두 가지 기능

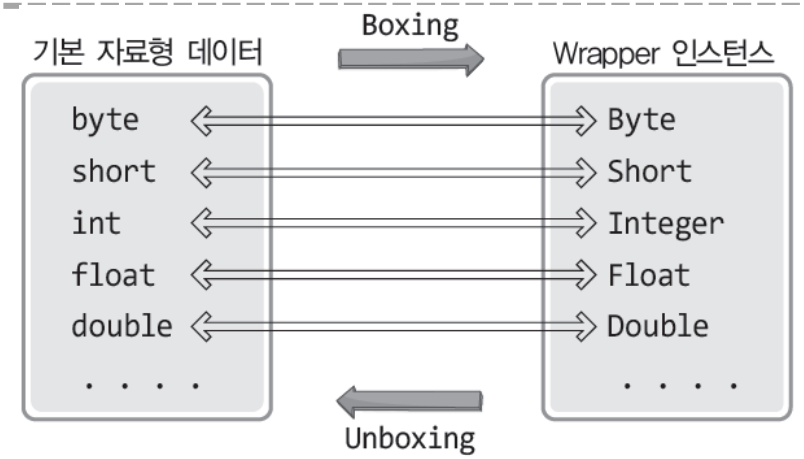


Boxing

✈ 기본 자료형 데이터를 Wrapper 인스턴스로 감싸는 것!

UnBoxing

✈ Wrapper 인스턴스에 저장된 데이터를 꺼내는 것!




```
class BoxingUnboxing
{
    public static void main(String[] args)
    {
        Integer iValue=new Integer(10);
        Double dValue=new Double(3.14);
        System.out.println(iValue);
        System.out.println(dValue);
        iValue=new Integer(iValue.intValue()+10);
        dValue=new Double(dValue.doubleValue()+1.2);
        System.out.println(iValue);
        System.out.println(dValue);
    }
}
```

실행 결과

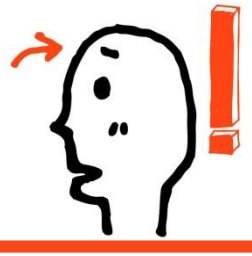
```
10
3.14
20
4.34
```

본래 Wrapper 클래스는 산술연산을 고려해서 정의되는 클래스가 아니다. 따라서 Wrapper 인스턴스를 대상으로 산술연산을 할 경우에는 왼쪽과 같이 코드가 복잡해진다.

Auto Boxing & Auto Unboxing

을 이용하면 다소 편해진다!

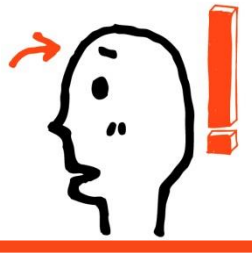
주요 메소드



- 언박싱 코드
  - 각 포장 클래스마다 가지고 있는 클래스 호출
  - 기본 타입명 + Value()

기본 타입의 값을 이용		
byte	num	= obj.byteValue();
char	ch	= obj.charValue();
short	num	= obj.shortValue();
int	num	= obj.intValue();
long	num	= obj.longValue();
float	num	= obj.floatValue();
double	num	= obj.doubleValue();
boolean	bool	= obj.booleanValue();

Auto Boxing & Auto Unboxing



자바제공 Wrapper 클래스를 사용하는 것이 좋은 이유!

Auto Boxing

✈ 기본 자료형 데이터가 자동으로 Wrapper 인스턴스로 감싸지는 것!

Auto UnBoxing

✈ Wrapper 인스턴스에 저장된 데이터가 자동으로 꺼내지는 것!

기본 자료형 데이터와 와야 하는데, Wrapper 인스턴스가 있다면, Auto Unboxing!

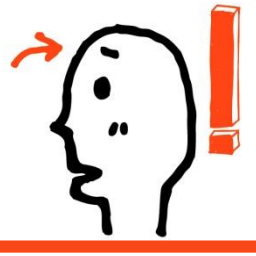
인스턴스가 와야 하는데, 기본 자료형 데이터가 있다면, Auto Boxing!

```
public static void main(String[] args)
{
    Integer iValue=10;      // auto boxing
    Double dValue=3.14;     // auto boxing

    System.out.println(iValue);
    System.out.println(dValue);

    int num1=iValue;        // auto unboxing
    double num2=dValue;     // auto unboxing
    System.out.println(num1);
    System.out.println(num2);
}
```

실행결과

```
10
3.14
10
3.14
```

Auto Boxing & Auto Unboxing 어떻게?



```
public static void main(String[] args)
{
    Integer num1=10;
    Integer num2=20;

    num1++;
    System.out.println(num1);

    num2+=3;
    System.out.println(num2);

    int addResult=num1+num2;
    System.out.println(addResult);

    int minResult=num1-num2;
    System.out.println(minResult);
}
```

Auto Boxing, Auto Unboxing 동시 발생

```
num1=new Integer(num1.intValue()+1);
```

Auto Boxing, Auto Unboxing 동시 발생

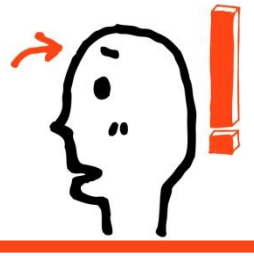
```
num2=new Integer(num2.intValue()+3);
```

```
11
23
34
-12
```

실행결과

예제에서 보이듯이 Auto Boxing과 Unboxing은 다양한 형태로 진행된다. 특히 산술연산의 과정에서도 발생을 한다는 사실에 주목할 필요가 있다.

Auto Boxing & Auto Unboxing 어떻게?



❖ 문자열을 기본 타입 값으로 변환

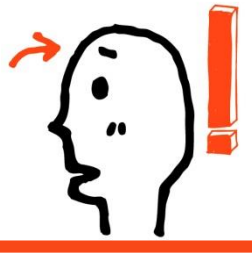
- parse + 기본타입 명 → 정적 메소드

기본 타입의 값을 이용		
byte	num	= Byte.parseByte("10");
short	num	= Short.parseShort("100");
int	num	= Integer.parseInt("1000");
long	num	= Long.parseLong("10000");
float	num	= Float.parseFloat("2.5F");
double	num	= Double.parseDouble("3.5");
boolean	bool	= Boolean.parseBoolean("true");

❖ 포장값 비교

- 포장 객체는 내부 값을 비교하기 위해 ==와 != 연산자 사용 불가
- 값을 언박싱해 비교하거나, equals() 메소드로 내부 값 비교할 것

매우 큰 정수의 표현을 위한 BigInteger 클래스



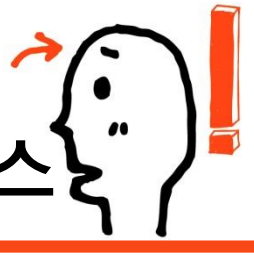
```
class SoBigInteger
{
    public static void main(String[] args)
    {
        System.out.println("최대 정수 : " + Long.MAX_VALUE);
        System.out.println("최소 정수 : " + Long.MIN_VALUE);

        BigInteger bigValue1=new BigInteger("10000000000000000000");
        BigInteger bigValue2=new BigInteger("-99999999999999999999");

        BigInteger addResult=bigValue1.add(bigValue2);
        BigInteger mulResult=bigValue1.multiply(bigValue2);

        System.out.println("큰 수의 덧셈결과 : "+addResult);
        System.out.println("큰 수의 곱셈결과 : "+mulResult);
    }
}
```

실행결과

```
최대 정수 : 9223372036854775807
최소 정수 : -9223372036854775808
큰 수의 덧셈결과 : 1
큰 수의 곱셈결과 : -999999999999999999990000000000000000000
```

큰 정수를 문자열로 표현한 이유는 숫자로 표현이 불가능하기 때문이다! 기본 자료형의 범위를 넘어서는 크기의 정수는 숫자로 표현 불가능하다!

오차 없는 실수의 표현을 위한 BigDecimal 클래스



```
public static void main(String[] args)
{
    BigDecimal e1=new BigDecimal(1.6);
    BigDecimal e2=new BigDecimal(0.1);

    System.out.println("두 실수의 덧셈결과 : "+ e1.add(e2));
    System.out.println("두 실수의 곱셈결과 : "+ e1.multiply(e2));
}
```

실수 1.6과 0.1을 숫자로 표현하는 순간 이미 오차가 포함되어 버렸다.

실행결과

```
1.899999999999999966693309261245303 78
0.340000000000000009992007221626408 37
```

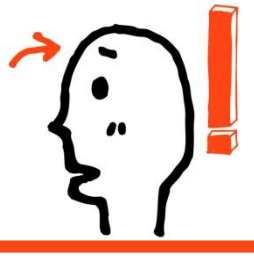
```
public static void main(String[] args)
{
    BigDecimal e1=new BigDecimal("1.6");
    BigDecimal e2=new BigDecimal("0.1");

    System.out.println("두 실수의 덧셈결과 : "+ e1.add(e2));
    System.out.println("두 실수의 곱셈결과 : "+ e1.multiply(e2));
}
```

실수도 문자열로 표현을 해서, BigDecimal 클래스에 오차 없는 값을 전달해야 한다!

실행결과

```
두 실수의 덧셈결과 : 1.7
두 실수의 곱셈결과 : 0.16
```

수학관련 기능을 제공하는 Math 클래스



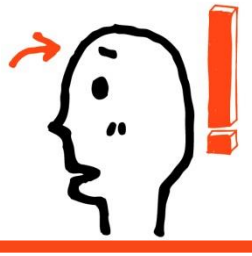
```
public static void main(String[] args)
{
    System.out.println("원주율 : " + Math.PI);
    System.out.println("2의 제곱근 : " + Math.sqrt(2));

    System.out.println(
        "파이에 대한 Degree : " + Math.toDegrees(Math.PI));
    System.out.println(
        "2파이에 대한 Degree : " + Math.toDegrees(2.0*Math.PI));

    double radian45=Math.toRadians(45);    // 라디안으로의 변환!
    System.out.println("싸인 45 : " + Math.sin(radian45));
    System.out.println("코싸인 45 : " + Math.cos(radian45));
    System.out.println("탄젠트 45 : " + Math.tan(radian45));

    System.out.println("로그 25 : " + Math.log(25));
    System.out.println("2의 4승 : "+ Math.pow(2, 4));
}
```

실행결과

```
원주율 : 3.141592653589793
2의 제곱근 : 1.4142135623730951
파이에 대한 Degree : 180.0
2파이에 대한 Degree : 360.0
싸인 45 : 0.7071067811865475
코싸인 45 : 0.7071067811865476
탄젠트 45 : 0.9999999999999999
로그 25 : 3.2188758248682006
2의 4승 : 16.0
```

Math 클래스에는 수학관련 메소드가 static으로 정의되어 있다는 사실을 기억하는 것이 중요하다! 필요한 메소드는 API 문자를 통해 참조하면 된다. 단 대부분의 메소드가 라디안 단위로 정의되어 있음은 기억하기 바란다!

수학관련 기능을 제공하는 Math 클래스

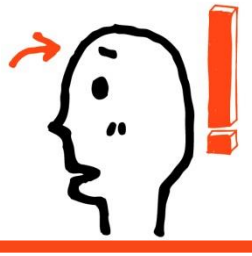


❖ Math 클래스

- 수학 계산에 사용할 수 있는 정적 메소드 제공

메소드	설명	예제 코드	리턴값
int abs(int a) double abs(double a)	절대값	int v1 = Math.abs(-5); double v2 = Math.abs(-3.14);	v1 = 5 v2 = 3.14
double ceil(double a)	올림값	double v3 = Math.ceil(5.3); double v4 = Math.ceil(-5.3);	v3 = 6.0 v4 = -5.0
double floor(double a)	버림값	double v5 = Math.floor(5.3); double v6 = Math.floor(-5.3);	v5 = 5.0 v6 = -6.0
int max(int a, int b) double max(double a, double b)	최대값	int v7 = Math.max(5, 9); double v8 = Math.max(5.3, 2.5);	v7 = 9 v8 = 5.3
int min(int a, int b) double min(double a, double b)	최소값	int v9 = Math.min(5, 9); double v10 = Math.min(5.3, 2.5);	v9 = 5 v10 = 2.5
double random()	랜덤값	double v11 = Math.random();	0.0<= v11<1.0
double rint(double a)	가까운 정수의 실수값	double v12 = Math.rint(5.3); double v13 = Math.rint(5.7);	v12 = 5.0 v13 = 6.0
long round(double a)	반올림값	long v14 = Math.round(5.3); long v15 = Math.round(5.7);	v14 = 5 v15 = 6

수학관련 기능을 제공하는 Random 클래스



❖ Random 클래스

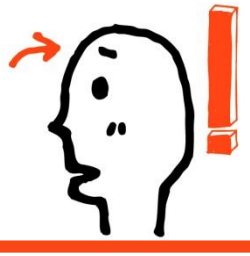
- boolean, int, long, float, double 난수 입수 가능
- 난수를 만드는 알고리즘에 사용되는 종자값(seed) 설정 가능
  - 종자값이 같으면 같은 난수
- Random 클래스로 부터 Random객체 생성하는 방법

생성자	설명
Random()	호출시 마다 다른 종자값(현재시간 이용)이 자동 설정된다.
Random(long seed)	매개값으로 주어진 종자값이 설정된다.

- Random 클래스가 제공하는 메소드

리턴값	메소드(매개변수)	설명
boolean	nextBoolean()	boolean 타입의 난수를 리턴
double	nextDouble()	double 타입의 난수를 리턴(0.0<= ~ < 1.0)
int	nextInt()	int 타입의 난수를 리턴(-2^{32}<= ~ <=2^{32}-1);
int	nextInt(int n)	int 타입의 난수를 리턴(0<= ~ <n)

씨드(Seed) 기반의 난수 생성



```
public static void main(String[] args)
{
    Random rand=new Random(12);

    for(int i=0; i<100; i++)
        System.out.println(rand.nextInt(1000));
}
```

12라는 씨앗이 전달되었으니, 12를 심어서 만들어지는 난수가 총 100개 생성된다.

씨앗! 이 동일하면 생성되는 난수의 패턴은 100% 동일!

이렇듯 컴퓨터의 난수는 씨앗을 기반으로 생성되기 때문에 가짜 난수(Pseudo-random number)라 불린다.

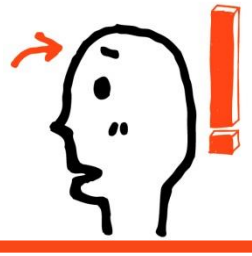
```
public static void main(String[] args)
{
    Random rand=new Random(12);
    rand.setSeed(System.currentTimeMillis());

    for(int i=0; i<100; i++)
        System.out.println(rand.nextInt(1000));
}
```

현재 시간을 밀리 초 단위로 반환

Random 클래스의 디폴트 생성자 내에서는 왼편의 코드와 같이 씨드를 설정한다.

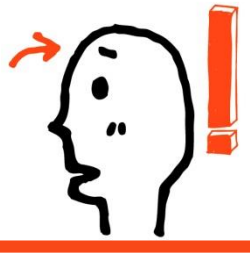
매 실행 시마다 다른 유형의 난수를 발생시키는 방법

난수의 발생의 분포도 확인



```
import java.util.Random;

public class RandomExample
{
		private final int randomRange = 10;
		private int loopCount = 1000000;
		private int[] randomTable = new int[randomRange];
		private Random random = new Random(2);

		public static void main(String[] args)
		{
				RandomExample example = new RandomExample();
				example.init();
				example.generate();
				example.printStatus();
		}

		public void init()
		{
				for (int i = 0; i < randomTable.length; i++)
				{
						randomTable[i] = 0;
				}
		}
```

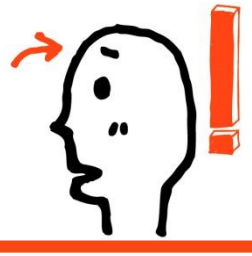
난수의 발생의 분포도 확인



```
public void generate()
		{
			for (int i = 0; i < loopCount; i++)
			{
				int randomVlu = random.nextInt(randomRange);
				randomTable[randomVlu]++;
			}
		}

		public void printStatus()
		{
			for (int i = 0; i < randomTable.length; i++)
			{
				System.out.println("Random value (" + i + ") =  " + randomTable[i]);
			}
		}
}
```

```
Random value (0) = 99628
Random value (1) = 100097
Random value (2) = 99847
Random value (3) = 99879
Random value (4) = 100111
Random value (5) = 100021
Random value (6) = 100075
Random value (7) = 100166
Random value (8) = 99891
```

StringTokenizer 클래스 생성자와 주요메소드



❖ 문자열 분리 방법

- String의 split() 메소드 이용
- java.util.StringTokenizer 클래스 이용

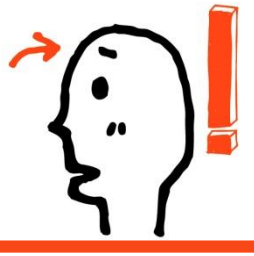
❖ String의 split()

- 정규표현식을 구분자로 해서 부분 문자열 분리
- 배열에 저장하고 리턴

```
홍길동&이수홍,박연수,김자바-최명호
```

```
String[] names = text.split("&|,|-");
```

StringTokenizer 클래스 생성자와 주요메소드



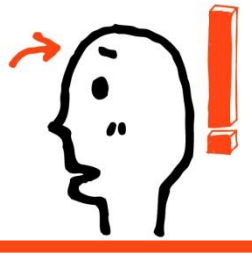
생성자

생성자	설명
StringTokenizer(String str)	str 스트링으로 파싱한 스트링 토크나이저 생성
StringTokenizer(String str, String delim)	str 스트링과 delim 구분 문자로 파싱한 스트링 토크나이저 생성
StringTokenizer(String str, String delim, boolean returnDelims)	str 스트링과 delim 구분 문자로 파싱한 스트링 토크나이저 생성. returnDelims가 true이면 delim이 포함된 문자도 토큰에 포함된다.

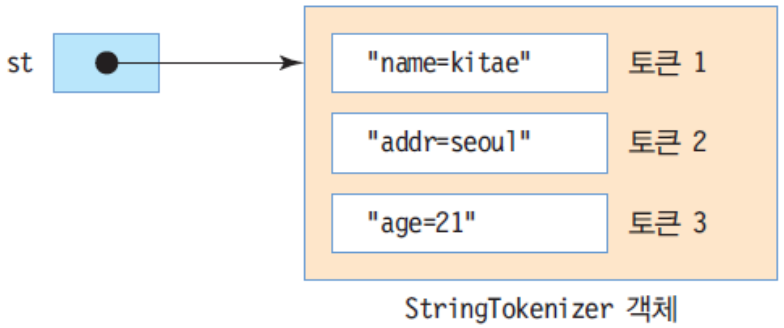
주요 메소드

생성자	설명
int countTokens()	스트링 토크나이저에 포함된 토큰의 개수 리턴
boolean hasMoreTokens()	스트링 토크나이저에 다음 토큰이 있으면 true 리턴
String nextToken()	다음 토큰 리턴

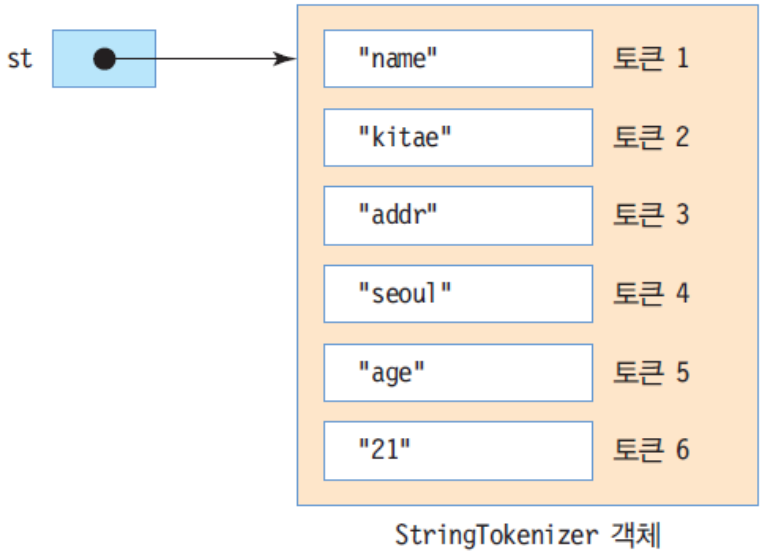
StringTokenizer 객체 생성과 문자열 분리



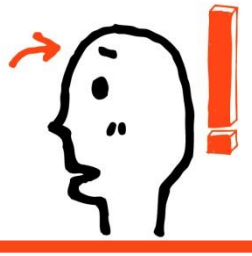
```
String query = "name=kitae&addr=seoul&age=21";
StringTokenizer st = new StringTokenizer(query, "&");
```




```
StringTokenizer st = new StringTokenizer(query, "&=");
```



문자열 토큰(Token)의 구분



"08 : 45"

"11 : 24"

콜론 : 을 기준으로 문자열을 나눈다고 할 때,

✈ 08, 45, 11, 24는 토큰(token)

✈ 콜론 : 는 구분자(delimiter)

```
public static void main(String[] args)
{
    String strData="11:22:33:44:55";
    StringTokenizer st=new StringTokenizer(strData, ":");

    while(st.hasMoreTokens())
        System.out.println(st.nextToken());
}
```

토큰을 나눌 대상 문자열 정보

구분자 정보

반환할 토큰이 있는가?

다음 번 토큰 반환!

실행결과

```
11
22
33
44
55
```

구분자 정보는 둘 이상 담을 수 있다. 하나의 문자열 안에 둘 이상을 담을 수 있다.

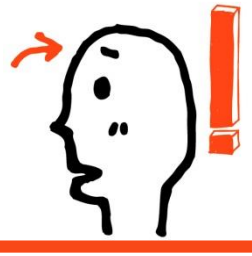
THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술

Android Java

19-1장 Arrays 클래스

Arrays 클래스의 배열 복사 메소드



```
public static int[] copyOf(int[] original, int newLength)
```

→ original에 전달된 배열을 첫 번째 요소부터 newLength의 길이만큼 복사

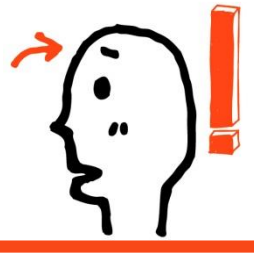
```
public static int[] copyOfRange(int[] original, int from, int to)
```

→ original에 전달된 배열을 인덱스 from부터 to 이전 요소까지 복사

```
public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

→ 배열 src의 srcPos에서 배열 dest의 destPos로 length 길이만큼 복사

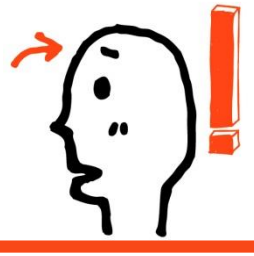
copyOf 메소드 호출의 예



```
public static void main(String[] args) {
   double[] arOrg = {1.1, 2.2, 3.3, 4.4, 5.5};

   // 배열 전체를 복사
   double[] arCpy1 = Arrays.copyOf(arOrg, arOrg.length);

   // 세번째 요소까지만 복사
   double[] arCpy2 = Arrays.copyOf(arOrg, 3);

   for(double d : arCpy1)
      System.out.print(d + "\t");
   System.out.println();

   for(double d : arCpy2)
      System.out.print(d + "\t");
   System.out.println();
 }
```

```
명령 프롬프트
C:₩JavaStudy>java CopyOfArrays
1.1     2.2     3.3     4.4     5.5
1.1     2.2     3.3

C:₩JavaStudy>
```

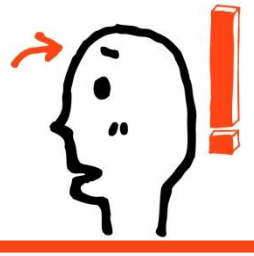
arraycopy 메소드 호출의 예



```
public static void main(String[] args) {
   double[] org = {1.1, 2.2, 3.3, 4.4, 5.5};
   double[] cpy = new double[3];

   // 배열 org의 인덱스 1에서 배열 cpy 인덱스 0으로 세 개의 요소 복사
   System.arraycopy(org, 1, cpy, 0, 3);

   for(double d : cpy)
      System.out.print(d + "\t");
   System.out.println();
}
```

```
명령 프롬프트
C:₩JavaStudy>java CopyOfSystem
2.2     3.3     4.4

C:₩JavaStudy>_
```

두 배열의 내용 비교



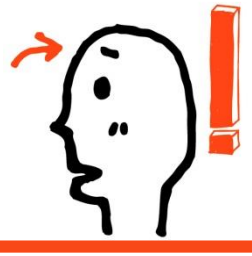
```
public static boolean equals(int[] a, int[] a2)
```

→ 매개변수 a와 a2로 전달된 배열의 내용을 비교하여 true 또는 false 반환

```
public static void main(String[] args) {
    int[] ar1 = {1, 2, 3, 4, 5};
    int[] ar2 = Arrays.copyOf(ar1, ar1.length);
    System.out.println(Arrays.equals(ar1, ar2));
}
```

```
명령 프롬프트
C:₩JavaStudy>java ArrayEquals
true

C:₩JavaStudy>_
```

인스턴스 저장 배열의 비교 예

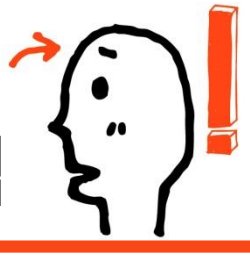


```
class INum {
   private int num;
   public INum(int num) {
      this.num = num;
   }
}

class ArrayObjEquals {
   public static void main(String[] args) {
      INum[] ar1 = new INum[3];
      INum[] ar2 = new INum[3];
      ar1[0] = new INum(1); ar2[0] = new INum(1);
      ar1[1] = new INum(2); ar2[1] = new INum(2);
      ar1[2] = new INum(3); ar2[2] = new INum(3);
      System.out.println(Arrays.equals(ar1, ar2));
   }
}
```

결과가 의미하는 바는? 어떤 식으로 비교?

```
명령 프롬프트
C:₩JavaStudy>java ArrayObjEquals
false

C:₩JavaStudy>
```

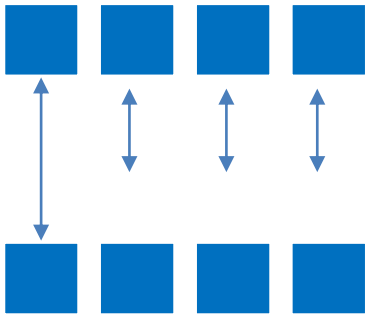
Arrays의 equals 메소드가 내용을 비교하는 방식



```
public static boolean equals(Object[] a, Object[] a2)
```

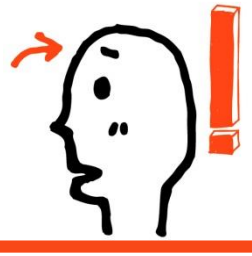
다음은 실제 Java의 Arrays.equals 메소드의 일부!

```
for (int i=0; i<length; i++) {
   Object o1 = a[i];
   Object o2 = a2[i];

   if (!(o1==null ? o2==null : o1.equals(o2)))
       return false;
}
```



각 요소별로 Object 클래스의 equals 메소드로 비교

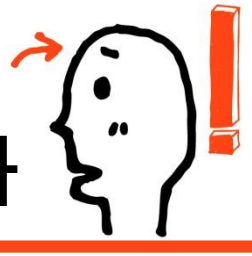
Object 클래스의 equals 메소드는?



```
public boolean equals(Object obj) {
   if(this == obj) // 두 인스턴스가 동일 인스턴스이면
      return true;
   else
      return false;
}    // 이렇듯 Object 클래스에 정의된 equals 메소드는 참조 값 비교를 한다.
```

따라서 Arrays 클래스의 equals 메소드가 두 배열의 내용 비교를 하도록 하려면 비교 대상의 equals 메소드를 내용 비교의 형태로 오버라이딩 해야 한다.

Object 클래스의 equals 메소드 오버라이딩 결과

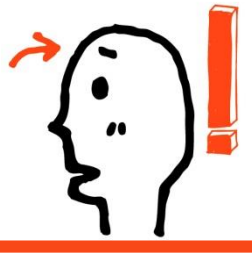


```
class INum {
   private int num;
   public INum(int num) {
      this.num = num;
   }

   @Override
   public boolean equals(Object obj) {
      if(this.num == ((INum)obj).num)
         return true;
      else
         return false;
   }
}
```

```
public static void main(String[] args) {
   INum[] ar1 = new INum[3];
   INum[] ar2 = new INum[3];
   ar1[0] = new INum(1); ar2[0] = new INum(1);
   ar1[1] = new INum(2); ar2[1] = new INum(2);
   ar1[2] = new INum(3); ar2[2] = new INum(3);
   System.out.println(Arrays.equals(ar1, ar2));
}
```

```
명령 프롬프트
C:₩JavaStudy>java ArrayObjEquals2
true

C:₩JavaStudy>_
```

배열의 정렬: Arrays 클래스의 sort 메소드



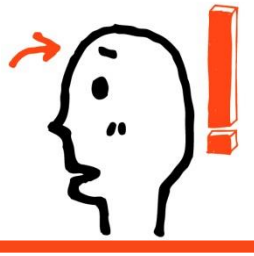
```
public static void sort(int[] a)
```

→ 매개변수 a로 전달된 배열을 오름차순(Ascending Numerical Order)으로 정렬

```
public static void main(String[] args) {
   int[] ar1 = {1, 5, 3, 2, 4};
   double[] ar2 = {3.3, 2.2, 5.5, 1.1, 4.4};
   Arrays.sort(ar1);
   Arrays.sort(ar2);

   for(int n : ar1)
      System.out.print(n + "\t");
   System.out.println();

   for(double d : ar2)
      System.out.print(d + "\t");
   System.out.println();
}
```

```
명령 프롬프트
C:₩JavaStudy>java ArraySort
1       2       3       4       5
1.1     2.2     3.3     4.4     5.5

C:₩JavaStudy>_
```

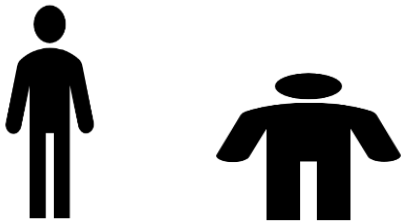
오름차순 정렬이란?



값이 작은 순에서 큰 순으로 세워 나가는 정렬

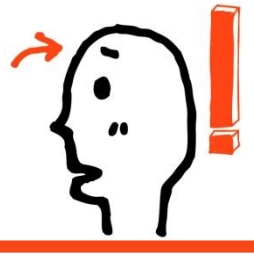
ex) 1, 2, 3, 4, 5, 6, 7

수의 경우는 오름차순에 대한 기준이 명확하다. 그렇다면 다음 두 사람을 오름차순으로 정렬해서 줄을 세운다고 하면? 한 사람은 키가 크고 다른 한 사람은 무게가 많이 나간다.



이 때에는 기준이 필요하다 오름차순 순서상 크고 작음에 대한 기준이 필요하다.
그래서 클래스를 정의할 때 오름차순 순서상 크고 작음에 대한 기준을 정의해야 한다.

compareTo 메소드 정의 기준



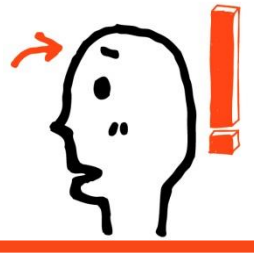
```
interface Comparable

   → int compareTo(Object o)
```

인자로 전달된 o가 작다면 양의 정수 반환

인자로 전달된 o가 크다면 음의 정수 반환

인자로 전달된 o와 같다면 0을 반환

클래스에 정의하는 오름차순 기준

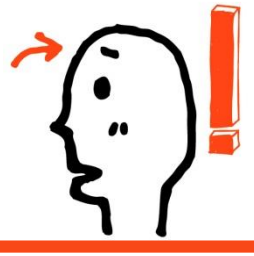


```
class Person implements Comparable {
   private String name;
   Private int age;

   . . . .

   @Override
   public int compareTo(Object o) {
      Person p = (Person)o;
      if(this.age > p.age)
         return 1;    // 인자로 전달된 o가 작다면 양의 정수 반환
      else if(this.age < p.age)
         return -1;    // 인자로 전달된 o가 크다면 음의 정수 반환
      else
         return 0;    // 인자로 전달된 o와 같다면 0을 반환
   }
   . . . .
}
```

Comparable 인터페이스를 구현한다는 것은
오름차순 순서상 크고 작음에 대한 기준을 제공한다는 의미

나이가 어릴수록 오름차순 순서상 작은 것으로 정의됨

다음과 같이 구현도 가능!

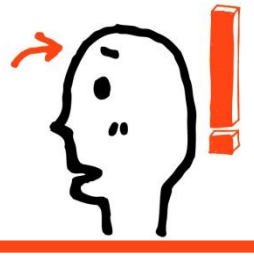


```
public int compareTo(Object o) {
   Person p = (Person)o;

   if(this.age > p.age)
      return 1;    // 인자로 전달된 o가 작다면 양의 정수 반환
   else if(this.age < p.age)
      return -1;    // 인자로 전달된 o가 크다면 음의 정수 반환
   else
      return 0;    // 인자로 전달된 o와 같다면 0을 반환
}
```

대신

```
public int compareTo(Object o) {
   Person p = (Person)o;
   return this.age - p.age;
}
```

배열에 저장된 인스턴스들의 정렬의 예

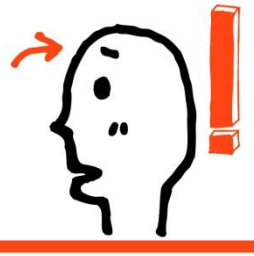


```
class Person implements Comparable {
   private String name;
   Private int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }
   public int compareTo(Object o) {
      Person p = (Person)o;
      return this.age - p.age;
   }
   public String toString() {
      return name + ": " + age;
   }
}
```

```
public static void main(String[] args) {
   Person[] ar = new Person[3];
   ar[0] = new Person("Lee", 29);
   ar[1] = new Person("Goo", 15);
   ar[2] = new Person("Soo", 37);

   Arrays.sort(ar);
   for(Person p : ar)
      System.out.println(p);
}
```

```
명령 프롬프트
C:₩JavaStudy>java ArrayObjSort
Goo: 15
Lee: 29
Soo: 37

C:₩JavaStudy>
```

배열의 탐색: 기본 자료형 값 대상



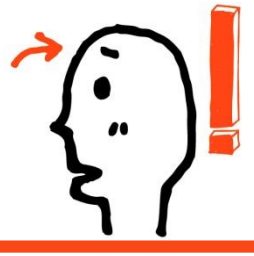
public static int binarySearch(int[] a, int key)

→ 배열 a에서 key를 찾아서 있으면 key의 인덱스 값, 없으면 0보다 작은 수 반환

binarySearch는 이진 탐색을 진행!

그리고 이진 탐색을 위해서는 탐색 이전에 데이터들이 오름차순으로 정렬되어 있어야 한다.

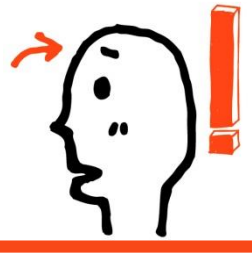
배열의 탐색: 기본 자료형 값 대상의 예



```
class ArraySearch {
   public static void main(String[] args) {
      int[] ar = {33, 55, 11, 44, 22};
      Arrays.sort(ar);   // 탐색 이전에 정렬이 선행되어야 한다.

      for(int n : ar)
         System.out.print(n + "\t");
      System.out.println();

      int idx = Arrays.binarySearch(ar, 33); // 배열 ar에서 33을 찾아라.
      System.out.println("Index of 33: " + idx);
   }
}
```

```
명령 프롬프트
C:₩JavaStudy>java ArraySearch
11      22      33      44      55
Index of 33: 2

C:₩JavaStudy>
```

배열의 탐색: 인스턴스 대상



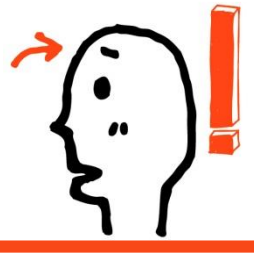
```
public static int binarySearch(Object[] a, Object key)
```

마찬가지로 탐색 대상들은 오름차순으로 정렬되어 있어야 한다.

그리고 탐색 대상의 확인 여부는 compareTo 메소드의 호출 결과를 근거로 한다.

즉, 탐색 방식은 인스턴스의 내용 비교이다!

배열의 탐색: 인스턴스 대상의 예



```
class Person implements Comparable {
   private String name;
   private int age;
   . . .
   @Override
   public int compareTo(Object o) {
      Person p = (Person)o;
      return this.age - p.age;   // 나이가 같으면 0을 반환
   }
   . . .
}
```

```
public static void main(String[] args) {
   Person[] ar = new Person[3];
   ar[0] = new Person("Lee", 29);
   ar[1] = new Person("Goo", 15);
   ar[2] = new Person("Soo", 37);
   Arrays.sort(ar);   // 탐색에 앞서 정렬을 진행
   int idx = Arrays.binarySearch(ar, new Person("Who are you?", 37));
   System.out.println(ar[idx]);
}
```

```
명령 프롬프트
C:₩JavaStudy>java ArrayObjSearch
Soo: 37

C:₩JavaStudy>
```

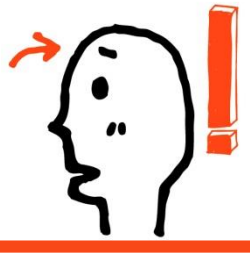
THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술

Android Java

20장 제네릭

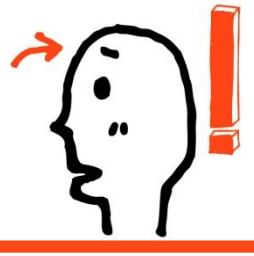
제네릭



01 제네릭 클래스의 이해와 설계

02 제네릭을 구성하는 다양한 문법적 요소

AppleBox와 OrangeBox 클래스의 설계



```
class AppleBox
{
    Apple item;
    public void store(Apple item) { this.item=item; }
    public Apple pullOut() { return item; }
}

class OrangeBox
{
    Orange item;
    public void store(Orange item) { this.item=item; }
    public Orange pullOut() { return item; }
}
```

Apple 상자와 Orange 상자를 구분한 모델

구현의 편의만 놓고 보면, FruitBox 클래스가 더 좋아 보인다. 하지만 FruitBox 클래스는 자료형에 안전하지 못하다는 단점이 있다.

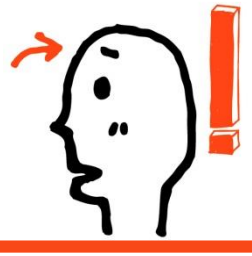
물론 AppleBox와 OrangeBox는 구현의 불편함이 따르는 단점이 있다. 그러나 자료형에 안전하다는 장점이 있다.

```
Class FruitBox
{
    Object item;
    public void store(Object item) { this.item=item; }
    public Object pullOut() { return item; }
}
```

무엇이든 담을 수 있는 상자 클래스

AppleBox, OrangeBox의 장점인 자료형의 안전성과 FruitBox의 장점인 구현의 편의성을 한데 모은것이 제네릭이다!

자료형의 안전성에 대한 논의



```
public static void main(String[] args)
{
    FruitBox fBox1=new FruitBox();
    fBox1.store(new Orange(10));
    Orange org1=(Orange)fBox1.pullOut();
    org1.showSugarContent();

    FruitBox fBox2=new FruitBox();
    fBox2.store("오렌지");
    Orange org2=(Orange)fBox2.pullOut();
    org2.showSugarContent();
}
```

예외 발생지점

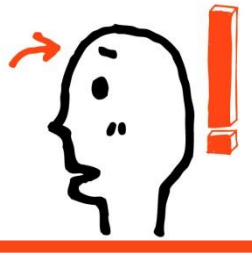
실행중간에 Class Casting Exception이 발생한다!
그런데 실행중간에 발생하는 예외는 컴파일 과정에서 발견되는 오류상황보다 발견 및 정정이 어렵다!
즉, 이는 자료형에 안전한 형태가 아니다.

```
public static void main(String[] args)
{
    OrangeBox fBox1=new OrangeBox();
    fBox1.store(new Orange(10));
    Orange org1=fBox1.pullOut();
    org1.showSugarContent();

    OrangeBox fBox2=new OrangeBox();
    fBox2.store("오렌지");
    Orange org2=fBox2.pullOut();
    org2.showSugarContent();
}
```

에러 발생지점

컴파일 과정에서, 메소드 store에 문자열 인스턴스가 전달될 수 없음이 발견된다. 이렇듯 컴파일 과정에서 발견된 자료형 관련 문제는 발견 및 정정이 간단하다.
즉, 이는 자료형에 안전한 형태이다.

제네릭(Generics) 클래스 설계



```
class FruitBox
{
   Object item;
   public void store(Object item){this.item=item;}
   public Object pullOut(){return item;}
}
```

제네릭 정의의 결과

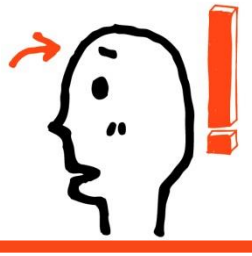
T라는 이름이 매개변수화 된 자료형임을 나타냄

```
class FruitBox <T>
{
   T item;
   public void store( T item ) {this.item=item;}
   public T pullOut() {return item;}
}
```

T에 해당하는 자료형의 이름은 인스턴스를 생성하는 순간에 결정이 된다!

제네릭 클래스가 자료형의 안전과 정의의 편의라는 두 마리 토끼를 실제로 잡았는지 생각해보자!

제네릭 클래스 기반 인스턴스 생성



```
FruitBox<Orange> orBox=new FruitBox<Orange>();
```

T를 Orange로 결정해서 FruitBox의 인스턴스를 생성하고 이를 참조할 수 있는 참조변수를 선언해서 참조 값을 저장한다!

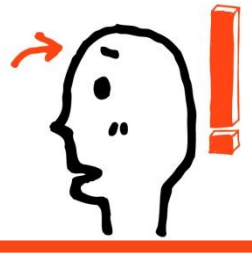
```
FruitBox<Apple> apBox=new FruitBox<Apple>();
```

T를 Apple로 결정해서 FruitBox의 인스턴스를 생성하고 이를 참조할 수 있는 참조변수를 선언해서 참조 값을 저장한다!

```
public static void main(String[] args)
{
    FruitBox<Orange> orBox=new FruitBox<Orange>();
    orBox.store(new Orange(10));
    Orange org=orBox.pullOut();
    org.showSugarContent();

    FruitBox<Apple> apBox=new FruitBox<Apple>();
    apBox.store(new Apple(20));
    Apple app=apBox.pullOut();
    app.showAppleWeight();
}
```

인스턴스 생성시 결정된 T의 자료형에 일치하지 않으면 컴파일 에러가 발생하므로 자료형에 안전한 구조임!

제네릭 클래스 기반 인스턴스 생성



❖ 제네릭 타입 사용 여부에 따른 비교

- 제네릭 타입 사용한 경우
  - 클래스 선언할 때 타입 파라미터 사용
  - 컴파일 시 타입 파라미터가 구체적인 클래스로 변경

```
public class Box<T> {
   private T t;
   public T get() { return t; }
   public void set(T t) { this.t = t; }
}
```

```
Box<String> box = new Box<String>();
```

```
public class Box<String> {
   private String t;
   public void set(String t) { this.t = t; }
   public String get() { return t; }
}
```

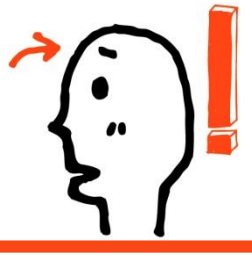
```
Box<String> box = new Box<String>();
box.set("hello");
String str = box.get();
```

```
Box<Integer> box = new Box<Integer>();
```

```
public class Box<Integer> {
   private Integer t;
   public void set(Integer t) { this.t = t; }
   public Integer get() { return t; }
}
```

```
Box<Integer> box = new Box<Integer>();
box.set(6);
int value = box.get();
```

제네릭 클래스 기반 인스턴스 생성



❖ 제네릭 타입은 두 개 이상의 타입 파라미터 사용 가능

- 각 타입 파라미터는 콤마로 구분
  - Ex) class<K, V, …> { … }
  - interface<K, V, …> { … }

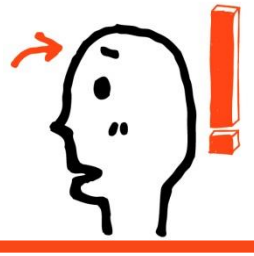
```
public class Product<T, M> {
    private T kind;
    private M model;

    public T getKind() { return this.kind; }
    public M getModel() { return this.model; }

    public void setKind(T kind) { this.kind = kind; }
    public void setModel(M model) { this.model = model; }
}
```

```
Product<Tv, String> product = new Product<Tv, String>();
```

- 자바 7부터는 다이아몬드 연산자 사용해 간단히 작성과 사용 가능

```
Product<Tv, String> product = new Product<>();
```

제네릭 메소드의 정의와 호출



❖ 제네릭 메소드

- 매개변수 타입과 리턴 타입으로 타입 파라미터를 갖는 메소드
- 제네릭 메소드 선언 방법
  - 리턴 타입 앞에 “<>” 기호를 추가하고 타입 파라미터 기술
  - 타입 파라미터를 리턴 타입과 매개변수에 사용

```
public <타입파라미터,...> 리턴타입 메소드명(매개변수,...) { ... }
```

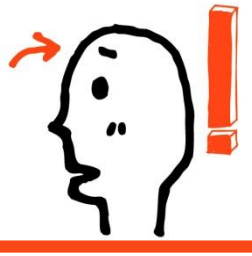
```
public <T> Box<T> boxing(T t) { ... }
```

- 제네릭 메소드 호출하는 두 가지 방법

```
리턴타입 변수 = <구체적타입> 메소드명(매개값);      //명시적으로 구체적 타입 지정
리턴타입 변수 = 메소드명(매개값);                  //매개값을 보고 구체적 타입을 추정
```

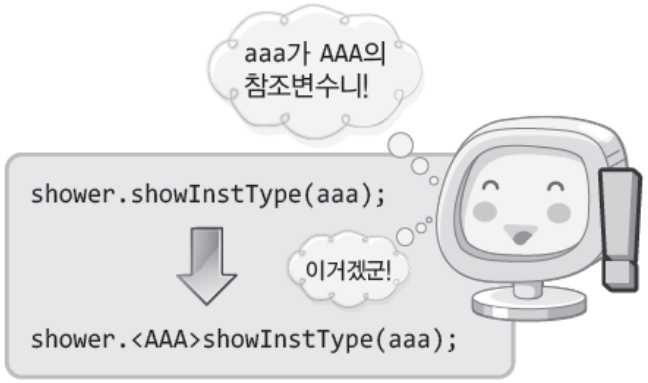
```
Box<Integer> box = <Integer>boxing(100);      //타입 파라미터를 명시적으로 Integer 로 지정
Box<Integer> box = boxing(100);               //타입 파라미터를 Integer 으로 추정
```

제네릭 메소드의 정의와 호출



```
class InstanceTypeShower
{
    int showCnt=0;

    public <T> void showInstType(T inst)
    {
        System.out.println(inst);
        showCnt++;
    }

    void showPrintCnt() { · · · · · }
}
```

클래스의 메소드만 부분적으로 제네릭화 할 수 있다!



제네릭 메소드의 호출과정에서 전달되는 인자를 통해서 제네릭 자료형을 결정할 수 있으므로 자료형의 표현은 생략 가능하다!

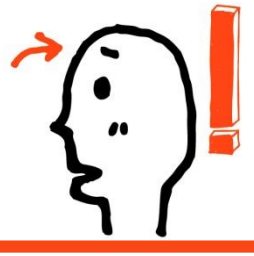
```
public static void main(String[] args)
{
    AAA aaa=new AAA();
    BBB bbb=new BBB();

    InstanceTypeShower shower=new InstanceTypeShower();
    shower.<AAA>showInstType(aaa);
    shower.<BBB>showInstType(bbb);
    shower.showPrintCnt();
}
```

제네릭 메소드의 호출방법! T를 AAA로, BBB로 결정하여 호출하고 있다!

일반적인 메소드 호출방법!

```
shower.showInstType(aaa);
shower.showInstType(bbb);
```

제네릭 메소드와 둘 이상의 자료형



```
class InstanceTypeShower2
{
    public <T, U> void showInstType(T inst1, U inst2)
    {
        System.out.println(inst1);
        System.out.println(inst2);
    }
}
```

T, U와 같은 문자는 상징적이다.

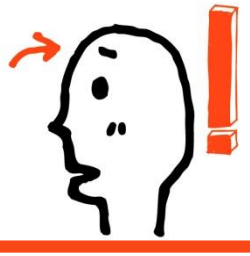
따라서 타 문자로도 대체 가능!

```
public static void main(String[] args)
{
    AAA aaa=new AAA();
    BBB bbb=new BBB();

    InstanceTypeShower2 shower=new InstanceTypeShower2();
    shower.<AAA, BBB>showInstType(aaa, bbb);
    shower.showInstType(aaa, bbb);
}
```

마찬가지로 메소드 호출 시,

자료형의 정보는 생략이 가능하다!

매개변수의 자료형 제한



public static <T extends AAA> void myMethod(T param) { . . . . . }

T가 AAA를 상속(AAA가 클래스인 경우) 또는 구현(AAA가 인터페이스인 경우)하는 클래스의 자료형이 되어야 함을 명시함

인터페이스 이름

```
public static <T extends SimpleInterface> void showInstanceAncestor(T param)
{
    param.showYourName();
}
```

인자는 SimpleInterface를 구현하는 클래스의 인스턴스이어야 함!

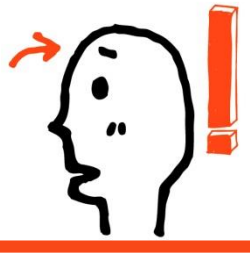
클래스 이름

```
public static <T extends UpperClass> void showInstanceName(T param)
{
    param.showYourAncestor();
}
```

UpperClass를 상속하는 클래스의 인스턴스이어야 함!

키워드 extends는 매개변수의 자료형을 제한하는 용도로도 사용된다!

제네릭 메소드와 배열



제네릭 메소드로의 배열 전달

배열도 인스턴스이므로 제네릭 매개변수에 전달이 가능하다. 하지만 이렇게 전달을 하면 다음과 같은 문장을 쓸 수 없다!

```
System.out.println(arr[i]);
```

다음과 같이 매개변수를 선언하면, 매개변수에 전달되는 참조 값을 배열 인스턴스의 참조 값으로 제한할 수 있다.

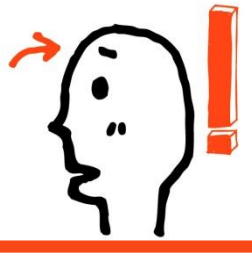
```
T[ ] arr
```

그리고 이렇게 되면 참조 값은 배열 인스턴스의 참조 값임이 100% 보장 되므로 [ ] 연산을 허용한다.

```
public static <T> void showArrayData(T[] arr)
{
    for(int i=0; i<arr.length; i++)
        System.out.println(arr[i]);
}
```

이렇듯 [ ] 연산이 필요하다면 매개변수의 선언을 통해서 전달 되는 참조 값을 배열의 참조 값으로 제한해야 한다.

제네릭 변수의 참조와 상속의 관계

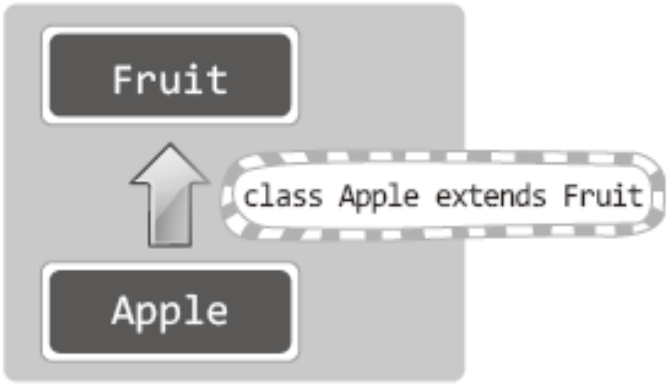


public void ohMethod (FruitBox<Fruit> param) { . . . . }

ohMehod의 인자로 전달될 수 있는 참조 값의 자료형은

✈ FruitBox<Fruit>의 인스턴스 참조 값

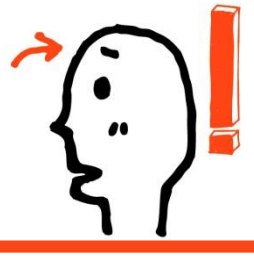
✈ FruitBox<Fruit>을 상속하는 인스턴스의 참조 값



왼쪽과 같은 상속 구조에서
FruitBox<Apple> 클래스는 ohMethod의
인자가 될수 있을까? NO!
반드시 키워드 extends를 이용해서 상속이
명시된 대상만 인자로 전달될 수 있다.

... extends FruitBox<Fruit>

와일드카드와 제네릭 변수의 선언



<? extends Fruit>

<? extends Fruit>가 의미하는 바는 "Fruit을 상속하는 모든 클래스"이다

- FruitBox<? extends Fruit> box1 = new FruitBox<Fruit>();
- FruitBox<? extends Fruit> box2 = new FruitBox<Apple>();

FruitBox<Fruit> 인스턴스의 참조 값도,
FruitBox<Apple> 인스턴스의 참조 값도
인자로 전달받을 수 있는 매개변수의 선언에는 와일드카드 문자 ?가 사용된다!

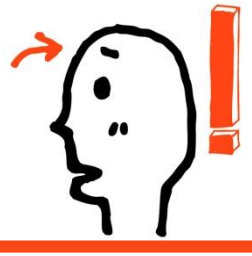
FruitBox<?> box;

자료형에 상관없이 FruitBox<T>의 인스턴를 참조에 사용되는 참조변수,

다음 선언과 동일하다.

- FruitBox<? extends Object> box;

하위 클래스를 제한하는 용도의 와일드 카드



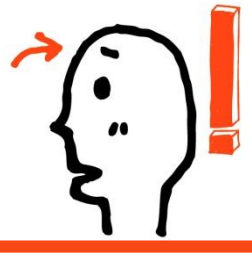
```
FruitBox<? extends Apple> boundedBox;
```

- ~을 상속하는 클래스라면 무엇이든지
- Apple을 상속하는 클래스의 인스턴스라면 무엇이든지 참조 가능한 참조변수 선언

```
FruitBox<? super Apple> boundedBox;
```

- ~이 상속하는 클래스라면 무엇이든지
- Apple이 상속하는 클래스의 인스턴스라면 무엇이든지 참조 가능한 참조변수 선언

제네릭 클래스의 다양한 상속방법



```
class AAA<T>
{
    T itemAAA;
}

class BBB<T> extends AAA<T>
{
    T itemBBB;
}
```

왼쪽과 같이 제네릭 클래스도 상속이 가능하다. 그리고 이 경우 다음과 같이 인스턴스를 생성하게 된다.

```
BBB<String> myString=new BBB<String>( );
BBB<Integer> myInteger=new BBB<Integer>( );
```

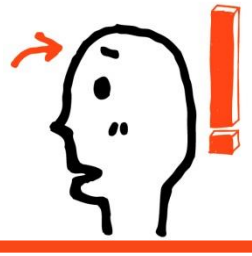
이 경우 T는 각각 String과 Integer로 대체되어 인스턴스가 생성된다.

```
class AAA<T>
{
    T itemAAA;
}

class BBB extends AAA<String>
{
    int itemBBB;
}
```

왼쪽과 같이 제네릭 클래스의 자료형을 결정해서 상속하는 것도 가능하다.

이 경우, BBB 클래스는 제네릭과 관련 없는 클래스가 되어, 일반적인 방법으로 인스턴스를 생성하면 된다.

기본자료형의 이름은 제네릭에 사용 불가!



컴파일 불가능한 문장들!

```
FruitBox<int> fb1=new FruitBox<int>();

FruitBox<double> fb1=new FruitBox<double>();
```

기본 자료형 정보를 이용해서는 제네릭 클래스의 인스턴스 생성이 불가능하다!

제네릭은 클래스와 인스턴스에 관한 이야기이다!

그럼 기본 자료형 정보를 대상으로는 제네릭 인스턴스의 생성이 필요한 상황에서는 어떻게 하죠?

위 질문에 Wrapper 클래스를 떠올리기 바란다! 이와 관련된 이야기는 이후에 계속된다!

THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술