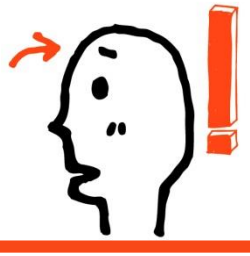
Android Java

2장 자바 프로그래밍을 위한 변수와 데이터형

자바 프로그래밍을 위한 변수와 데이터형



01 변수, 제대로 알고 넘어가자

02 데이터형의 종류와 사용법 익히기

03 char형이 곧 문자형이다

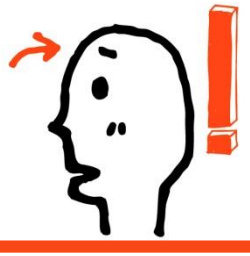
04 부동 소수형 이해하기

05 불리언형과 참조형 변수도 잊지 말자

06 자료형을 기반으로 표현이 되는 상수

07 형 변환으로 데이터 형태를 변경하자

변수, 제대로 알고 넘어가자



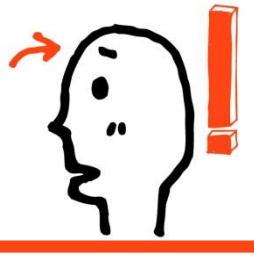
변수의 선언과 사용

프로그래밍이란 무엇일까?

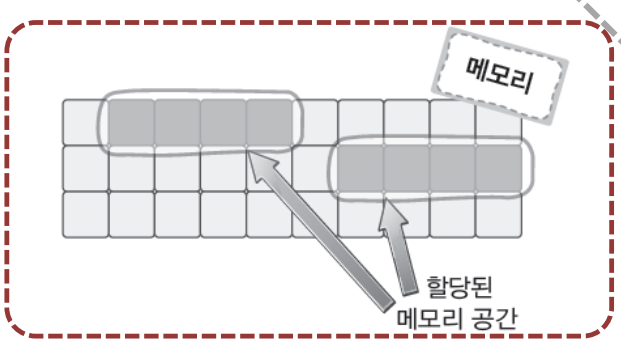
애플리케이션의 동작 도중 발생한 데이터나 외부에서 가져온 데이터들을 메모리와 같은 저장 매체에 넣고 다루는 것

→ 프로그래밍에서 데이터를 다루는 가장 작고 기본적인 작업 단위는 **변수를 선언하고 사용하는 것**

변수, 제대로 알고 넘어가자

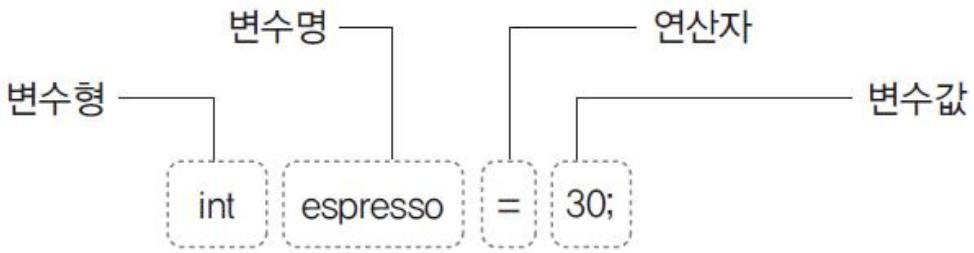


변수의 선언과 사용



- **변수란?** 메모리 영역에 데이터를 저장할 공간을 의미

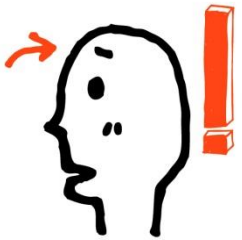
- 개발자는 변수를 선언함으로써 사용할 공간 크기를 정하고 변수명과 연산자를 이용해서 데이터를 가져오거나 변경할 수 있음



△ 그림 2-1 변수 선언 방법의 분해

- espresso라는 변수명에는 30이라는 값이 설정되고, 30이라는 값은 JVM 메모리에 할당됨을 의미

변수, 제대로 알고 넘어가자



변수에 대한 간단한 이해

"난 10진수 정수의 저장을 위한

메모리 공간을 할당하겠다."

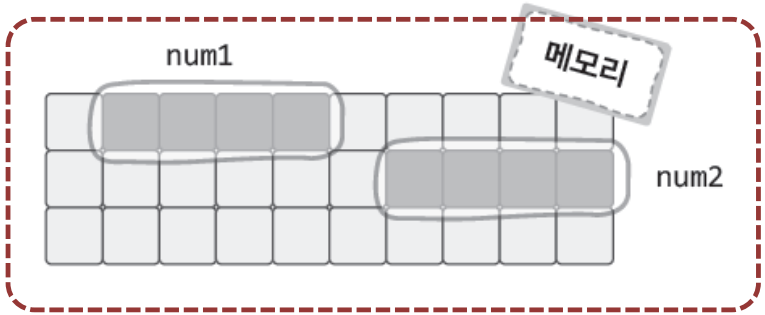
"그리고 그 메모리 공간의 이름을 num이라 하겠다."

```
int num;
```

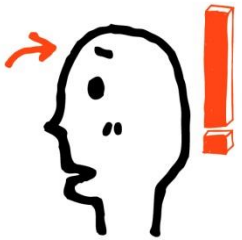
변수의 선언

int와 같이 변수의 특성을 결정짓는 키워드를 가리켜 자료형이라 한다.

```
int num1;
int num2;
```



변수, 제대로 알고 넘어가자



모두를 위한 변수명 규칙

자바에서 변수명(Variable name)을 선언할 때는 지켜야 할 공통 규칙이 있으며 이를 명명규칙(Naming convention)이라 함

반드시 지켜야 하는 명명 규칙

- 대소문자는 서로 다르게 인식하며 이름의 길이에는 제한이 없다.

예 : int espresso = 1;과 int Espresso = 1;에서 espresso와 Espresso는 서로 다른 변수명

- 미리 정해진 예약어를 변수 이름으로 사용할 수 없다.

예 : int static = 0;은 사용할 수 없음 static이라는 이름은 이미 자바 문법에서 사용되고 있는 예약어 (Abstract, Class, extends, New, while, Assert, continue, final, native, Strickfp 등)

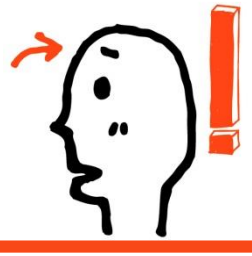
- 변수명은 숫자로 시작할 수 없다

예 : int 2coffee = 1;은 사용할 수 없음, 하지만 int coffee1 = 1;은 사용 가능

- 특수문자는 '_'과 '$'만 허용한다

예 : int coffee_size = 1;은 사용 가능하지만 int coffee&sugar = 1;은 불가능

변수, 제대로 알고 넘어가자



모두를 위한 변수명 규칙

다음의 규칙은 어긴다고 하여도 컴파일과 애플리케이션 실행에는 아무런 문제가 없지만
코드의 통일성이나 추후 유지 보수, 타인과의 협업을 위해서 규칙을 준수하는 것을 강력히 권고

권장 명명 규칙

- 변수와 메소드 이름의 첫 글자는 소문자로 선언

예 : int espresso = 1;(권장), int Expresso = 1;(비권장)

- 여러 단어일 때 첫 번째 단어의 첫 글자는 소문자로 선언, 다음 단어의 첫 글자는 대문자로 선언

예 : int caffeLatte = 1;(권장), int caffe_latte = 1;(비권장)

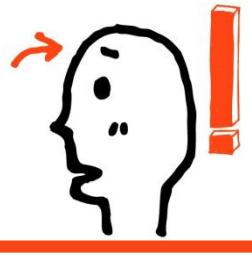
- 상수일 때는 모든 글자를 대문자로 선언하고 여러 단어로 된 상수일 때는 '_' 로 구분

예 : final int COFFEE_DEFAULT_SIZE = 1; (권장)

- 변수명은 반드시 사용 목적을 내포

예 : int age=33;(권장), int abcd = 33; (비권장)

변수, 제대로 알고 넘어가자



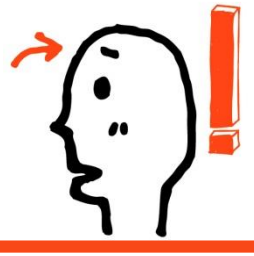
예약어를 변수로 선언하면 안돼요!

예약어란 이미 프로그래밍 언어에서 문법으로 사용하고 있는 단어를 의미한다. 이클립스에서 예약어를 변수명으로 선언하면 빨간색 밑줄로 에러를 자동 표기해준다. 물론 컴파일도 되지 않는다.

자바의 예약어

boolean	if	interface	class	true
char	else	package	volatile	false
byte	final	switch	while	trhows
float	private	case	return	native
void	protected	break	throw	implements
short	public	default	try	import
double	static	for	catch	synchronized
int	new	continue	finally	const
long	this	do	transient	enum
abstract	super	extends	instanceof	null

변수, 제대로 알고 넘어가자



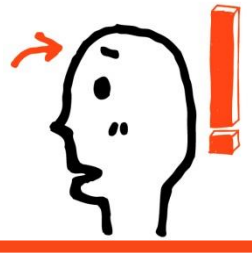
변수의 선언 및 활용 예

```
class UseVariable
{
    public static void main(String[] args)
    {
        int num1;
        num1=10;

        int num2=20;
        int num3=num1+num2;
        System.out.println(num1+"+"+num2+"="+num3);
    }
}
```

실행결과

```
10+20=30
```

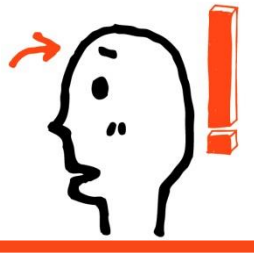
데이터형의 종류와 사용법 익히기

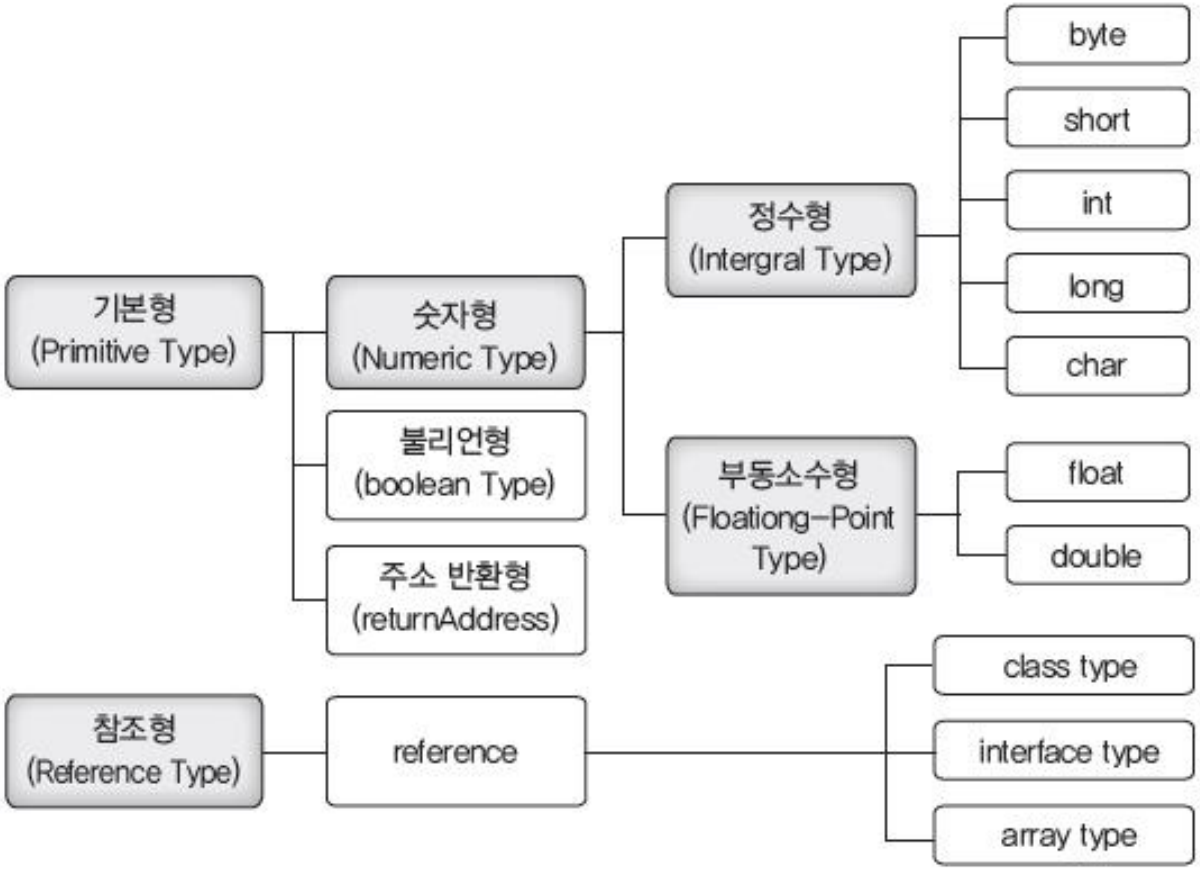


데이터형의 종류와 사용법 익히기

- 자바는 엄격한 정적 타입 지정 언어이므로 반드시 데이터형(Date type)을 선언
- 데이터형은 각각 메모리 크기가 다르며 저장할 수 있는 데이터의 종류도 다름
- 그러므로 변수 데이터의 목적에 맞는 데이터형 선택이 필요
- 개발자는 자신이 사용하고자 하는 데이터에 따라서 적절한 데이터형을 선택할 수 있어야 함

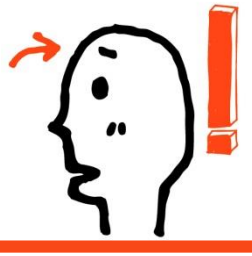
데이터형의 종류와 사용법 익히기





△ 그림 2-2 데이터형의 종류

데이터형의 종류와 사용법 익히기



비트

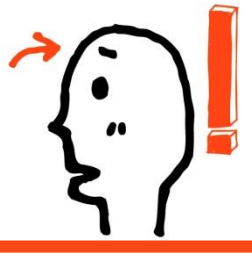
- 0(OFF)과 1(ON)만 존재

전기 스위치	의미	2진수	10진수
	꺼짐, 꺼짐	00	0
	꺼짐, 켜짐	01	1
	켜짐, 꺼짐	10	2
	켜짐, 켜짐	11	3

그림 3-26 2개의 전기 스위치와 2진수, 10진수의 비교

- n개의 전기 스위치로 표현할 수 있는 가짓 수= 2^n
- 3비트로 표현할 수 있는 가짓수는 2^3=8개, 4비트로 표현할 수 있는 가짓수는 2^4=16개

데이터형의 종류와 사용법 익히기



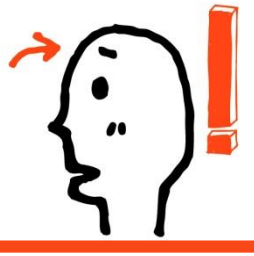
■ 진수

표 3-3 10진수, 2진수, 16진수의 변환

10진수(0~9)	2진수(0, 1)	16진수(0~F)
00	0000	0
01	0001	1
02	0010	2
03	0011	3
04	0100	4
05	0101	5
06	0110	6
07	0111	7
08	1000	8
09	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

TIP/ 보기 좋게 하기 위해 10진수 0은 00으로, 2진수 0은 0000으로 자릿수를 맞춰서 나타냈다. 또한 8진수도 사용할 수 있는데 활용도가 비교적 낮다.

데이터형의 종류와 사용법 익히기



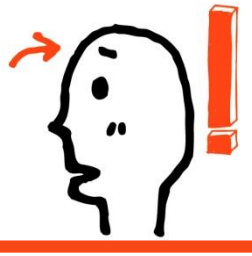
■ 바이트

- JAVA에서 가장 많이 사용되는 단위. 바이트는 8개의 비트가 합쳐진 것

표 3-4 비트와 바이트의 크기에 따른 숫자의 범위

비트 수	바이트 수	표현 개수	2진수	10진수	16진수
1		$2^1=2$	0~1	0~1	0~1
2		$2^2=4$	0~11	0~3	0~3
4		$2^4=16$	0~1111	0~15	0~F
8	1	$2^8=256$	0~11111111	0~255	0~FF
16	2	$2^{16}=65536$	0~11111111 11111111	0~65535	0~FFFF
32	4	$2^{32}=$약 42억	0~…	0~약 42억	0~FFFF FFFF
64	8	$2^{64}=$약 1800경	0~…	0~약 1800경	0~…

데이터형의 종류와 사용법 익히기

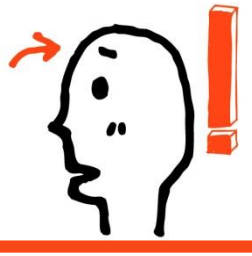


▷ 데이터의 크기와 성질에 따라서
반드시 변수형도 적절한 형태의 모양과 크기가 되어야 함

자료형	데이터	메모리 크기	표현 가능 범위
boolean	참과 거짓	1 바이트	true, false
char	문자	2 바이트	모든 유니코드 문자
byte	정수	1 바이트	−128 ~ 127
short		2 바이트	−32768 ~ 32767
int		4 바이트	−2147483648 ~ 2147483647
long		8 바이트	−9223372036854775808 ~ 9223372036854775807
float	실수	4 바이트	$\pm(1.40\times10^{-45} \sim 3.40\times10^{38})$
double		8 바이트	$\pm(4.94\times10^{-324} \sim 1.79\times10^{308})$

△ 표 2-2 숫자형의 종류와 메모리 크기 그리고 그에 따른 저장 범위

데이터형의 종류와 사용법 익히기



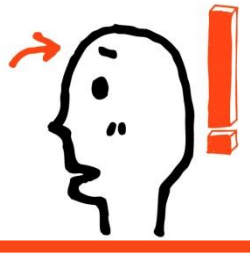
정수를 담기 위한 정수형

- **정수란?** 양의 정수와 음의 정수를 포함하는 것으로 부호가 있는 자연수를 의미
- 자바의 정수형(Integral types) : **byte, short, int, long**
- 자바에서는 unsigned 타입을 제공하지 않음. 즉, 숫자형은 무조건 부호를 갖고 있음

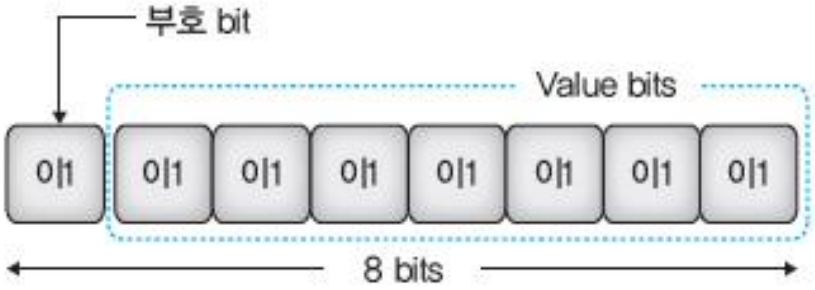
데이터형	크기	기본값	표현 범위
byte	8bit	0	–128 ~ 127
short	16bit	0	–32,768 ~ 32,767
int	32bit	0	–2,147,483,648 ~ 2,147,483,647
long	64bit	0	– 9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807

△ 표 2-3 정수형의 크기와 표현 범위

데이터형의 종류와 사용법 익히기

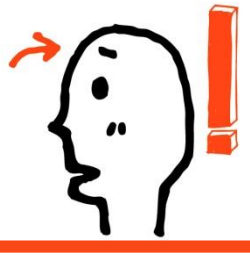


정수를 담기 위한 정수형



- 컴퓨터에서의 기본 데이터 단위는 0 혹은 1을 저장할 수 있는 비트(bit)
- byte 변수형의 크기는 8bit이므로 저장 가능한 데이터는 2^8(=256)개 만큼 표현 가능
- byte 변수형의 크기인 8bit 중 첫 번째 bit는 부호(음수, 양수) 정보를 저장
  나머지 7개의 bit에 정수 데이터를 저장
- 부호와 함께 숫자를 표현할 수 있는 범위는 음수로는 -128 ~ -1까지, 양수로는 0 ~ 127(2^7)

데이터형의 종류와 사용법 익히기



정수 자료형에 대한 논의

- byte, short, int, long
- 정수를 표현하는데 사용되는 바이트 크기에 따라서 구분이 됨

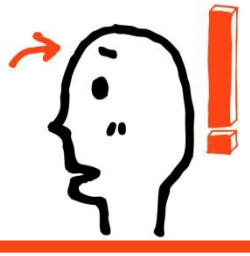
✓ 작은 크기의 정수 저장에는 short? 아니면 int형 변수?

- CPU는 int형 데이터의 크기만 연산 가능
- 때문에 연산직전에 short형 데이터는 int형 데이터로 자동변환
- 변환의 과정을 생략할 수 있도록 int를 선택한다!

✓ 그럼 short와 byte는 왜 필요한가?

- 연산보다 데이터의 양이 중요시 되는 상황도 존재!
- MP3 파일, 동영상 파일
- 데이터의 성격이 강하다면 short와 byte를 활용!

데이터형의 종류와 사용법 익히기



실수 자료형에 대한 논의

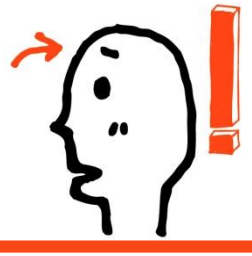
✓ 자바의 2가지 실수 자료형

- float, double
- float는 소수점 이하 6자리 double은 12자리 정밀도

✓ 실수 자료형의 선택기준

- float와 double 모두 매우 충분한 표현의 범위를 자랑한다!
- 이 둘의 가장 큰 차이점은 정밀도!
- 따라서 필요한 정밀도를 바탕으로 자료형을 결정한다.
- 일반적으로 double의 선택이 선호됨.

데이터형의 종류와 사용법 익히기

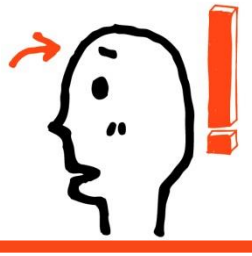


명시적 선언을 위한 정수 리터럴

```
byte value1 = 120;

short value2 = 250;

int value3 = 300;

long value4 = 25000L; 또는 long value4 = 25000l; 또는 long value4 = 25000;
```

- value4 변수 이름처럼 숫자 뒤에 대문자 L 혹은 소문자 l을 붙이는 경우가 있음
- 이는 변숫값이 long형이라고 명시적으로 선언하는 것
- 이러한 문자(표기법)를 자바에서는 **리터럴(literal)**이라 함
- 정수형 리터럴 중에서 접미사 L이 붙으면 long 변수형을 나타내며,
  L이 없는 경우에는 int 변수형을 의미

데이터형의 종류와 사용법 익히기



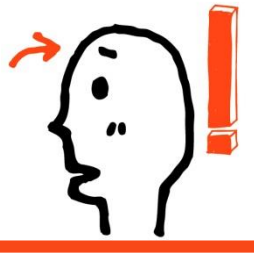
명시적 선언을 위한 정수 리터럴

정수 리터럴은 진수법(10진수, 16진수, 8진수, 2진수)에 따라서 표기 방법이 바뀜

진수법	접두어[1]	표기 숫자
10진법	–	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
16진법	0x 또는 0X	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
8진법	0	0, 1, 2, 3, 4, 5, 6, 7
2진법	0b 또는 0B	0, 1

"16진법과 2진법의 리터럴 중에서 0X, 0B와 같이 **대문자**를 이용하기를 추천한다.
소문자보다 명확하게 표현되기 때문이다."

데이터형의 종류와 사용법 익히기



명시적 선언을 위한 정수 리터럴

코드 2-2 package com.gilbut.chapter2;

```
1   public class ScaleExample
2   {
3      public static void main(String[] args)
4      {
5         int decimal = 12;            //10진수
6         int hexadecimal = 0XC;       //16진수
7         int octet = 014;             //8진수
8         int binary = 0B1100;         //2진수
9
10        System.out.println(decimal);
11        System.out.println(hexadecimal);
12        System.out.println(octet);
13        System.out.println(binary);
14     }
15  }
```

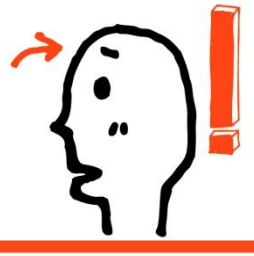
진수 표기법은 10진수와 쉽게 구별하기 위해서 사용합니다.

실행결과

```
12
12
12
12
```

- ScaleExample 클래스는 정수 12를 각 진수법에 따라서 표현하고 그것을 콘솔 화면에 출력하는 예제

char형이 곧 문자형이다



- '유니코드의 코드는 정수이며 유니코드에 맵핑된 것이 문자'
- 유니코드는 16bit로 구성되어 있어서 65,536개의 문자들을 표현할 수 있으며,
  자바의 Char형은 유니코드를 표현하므로 그 크기가 16bit
- char형은 유니코드를 의미

```
char firstChar = 'A';
char firstChar = '\u0041';
```

```
char ch1='A';
char ch2='한';
```

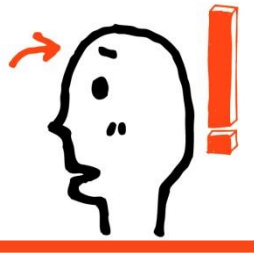
오른쪽의 두 코드는 사실상 동일한 코드

변환 발생

```
char ch1=65;
char ch2=54620;
```

- **char형 리터럴**은 문자를 표현할 때 **홑따옴표(' ')**를 사용
  문자열 리터럴은 **쌍따옴표(" ")**를 사용
- 유니코드를 사용할 때는 **'₩u'**를 사용해서 유니코드임을 명시적으로 보여줘야 함

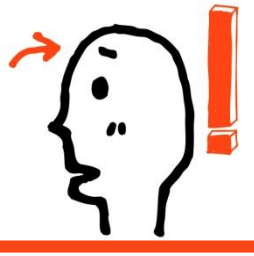
char형이 곧 문자형이다



특수 문자	특수 문자 표기법	유니코드
backspace(백스페이스)	₩b	₩u0008
tab(탭)	₩t	₩u0009
line feed(개행문자) = LF	₩n	₩u000a
carriage return(복귀문자) = CR	₩r	₩u000d
double quote(쌍따옴표)	₩"	₩u0022
single quote(홑따옴표)	₩'	₩u0027
backslash(역슬래시)	₩₩	₩u005c

△ 표 2-5 자바에서 사용할 수 있는 특수 문자와 표기법

char형이 곧 문자형이다



코드 2-3 package com.gilbut.chapter2;

```
1   public class CharUsage
2   {
3       public static void main(String[] args)
4       {
5           char tab = '\t';
6           char linefeed = '\n';
7           char cReturn = '\r';
8           char dQuote = '\"';
9           char sQuote = '\'';
10          char bSlash = '\\';
11
12          System.out.println("1> ABCD" + tab       + "EFG");
13          System.out.println("2> ABCD" + linefeed  + "EFG");
14          System.out.println("3> ABCD" + cReturn   + "EFG");
15          System.out.println("4> ABCD" + dQuote    + "EFG");
16          System.out.println("5> ABCD" + sQuote    + "EFG");
17          System.out.println("6> ABCD" + bSlash    + "EFG");
18      }
```

문자(char형)를 의미하는 홑따옴표

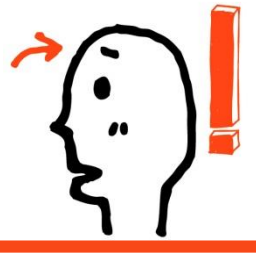
특수 문자를 의미하는 역슬래시

문자열을 의미하는 쌍 따옴표

특수 문자가 화면에 어떻게 출력되는지 실행 결과를 확인하세요!

- CharUsage 클래스는 특수 문자를 사용하는 방법에 대해서 보여줌

char형이 곧 문자형이다

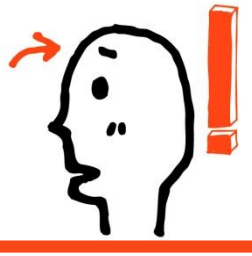


실행결과

```
1> ABCD EFG
2> ABCD
EFG
3> ABCD
EFG
4> ABCD"EFG
5> ABCD'EFG
6> ABCD\EFG
```

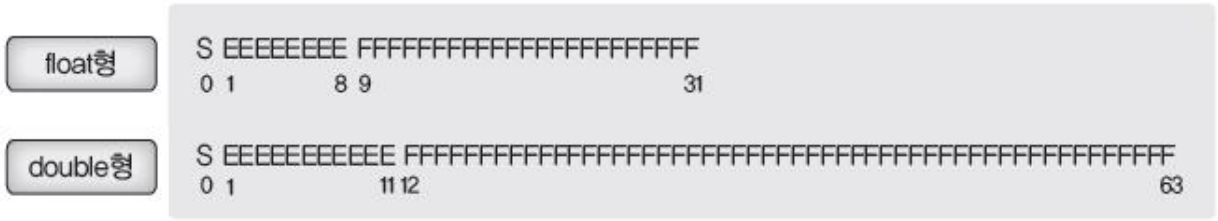
- 실행 결과를 보면 'ABCD' 문자열과 'EFG' 문자열 사이에 특수 문자들이 위치
- 코드 13라인의 linefeed(라인 피트)와 14라인의 cReturn(캐리지 리턴)의 출력된 결과가 동일

부동 소수형 이해하기



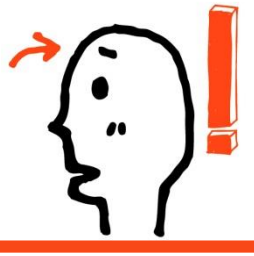
float과 double(자바의 실수형)

- float형의 목적은 single precision 단순 정확도를 위한 것
- double형의 목적은 double precision 두 배의 정확도를 위한 것



- bit 구성도 중 S는 부호를 나타내고 E는 지수 그리고 F는 소수를 저장하는데 사용
- double형이 float형보다 표현 범위가 넓기 때문에 더 정확하게 데이터를 표현 가능
- 하지만 float형을 사용하면 보다 빠른 연산 결과를 얻을 수 있음

부동 소수형 이해하기



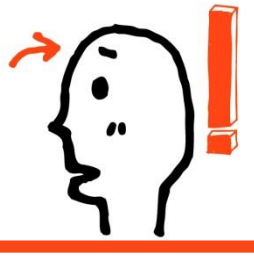
부동 소수형 리터럴

변수형	리터럴	예
float형	접미사 F 또는 f	float price = 2.99F;
double형	접미사 D 또는 d	double pi = 3.14159265358979323 84D;
10진수의 밑수	E or e	double tax = 0.0299E1;

△ 표 2-7 부동 소수점 수 리터럴

▷ long형의 리터럴과 마찬가지로 코드의 가독성을 높이기 위해 접미사는 **대문자**로 표기

부동 소수형 이해하기



부동 소수형 리터럴

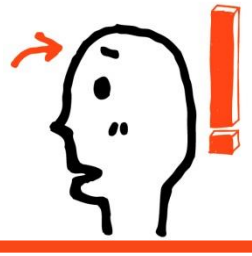
코드 2-4 package com.gilbut.chapter2;

```
1   public class FloatPointNumber
2   {
3       public static void main(String[] args)
4       {
5           char TAB_CHAR = '\t';
6
7           float price = 2.99f;
8           double pi = 3.1415926535897d;
9           double tax = 0.0299e1;
10
11          System.out.println("Variable price" + TAB_CHAR + ": " + price);
12          System.out.println("Variable pi" + TAB_CHAR + ": " + pi);
13          System.out.println("Variable tax" + TAB_CHAR + ": " + tax);
14
15          float maxFloatValue = Float.MAX_VALUE;
16          float minFloatValue = Float.MIN_VALUE;
17          System.out.println("Maximum value" + TAB_CHAR + ": " + maxFloatValue);
18          System.out.println("Minimum value" + TAB_CHAR + ": " + maxFloatValue);
19
20          System.out.println("Overflow" + TAB_CHAR + ": " + maxFloatValue * 10);
21          System.out.println("Underflow" + TAB_CHAR + ": " + minFloatValue / 100);
22
23      }
24  }
```

변수를 선언할 때 명확하게 float형 숫자인지, double형 숫자인지 알려주면 JVM은 정확한 연산을 할 수 있습니다.

오버플로우와 언더플로우 출력 구문

부동 소수형 이해하기



부동 소수형 리터럴

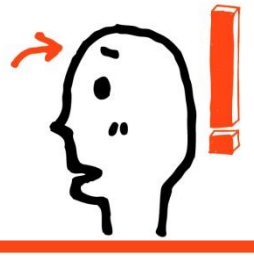
실행결과

```
Variable price : 2.99
Variable pi : 3.1415926535897
Variable tax : 0.299
Maximum value : 3.4028235E38
Minimum value : 1.4E-45
Overflow : Infinity
Underflow : 0.0
```

- float형이 표현할 수 있는 최댓값과 최솟값 범위를 초과
  더 이상 정상적인 값을 출력하지 못하고 Infinity와 0.0을 출력

- 최댓값을 초과하면 Infinity라는 상수값을 반환, 이 값을 float형에 다시 담으려고 한다면
  → 오버플로우(overflow) 발생 ↔ 언더플로우 (underflow)

불리언형과 참조형 변수도 잊지 말자

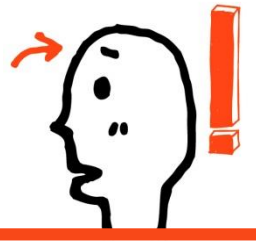


참 또는 거짓, 불리언형

- **Boolean형** 참(true) 또는 거짓(false) 값을 가지는 데이터형
- Boolean형은 1bit의 데이터 크기를 가지므로 오직 두 가지 값만 가질 수 있음
- 흑백 논리를 사용하는 로직에서 많이 사용
- 0 또는 1의 정수를 직접 저장하지 않고 자바에서 제공하는 **true, false와 같은 리터럴을 사용**
- true, false 앞뒤에 작은 따옴표나 큰 따옴표가 보이지 않으므로
  문자형이나 문자열형이 아닌 리터럴

```
boolean isSuccess = true;
boolean isFile = false;
```

불리언형과 참조형 변수도 잊지 말자



키워드 true와 false에 대한 좋은 이해

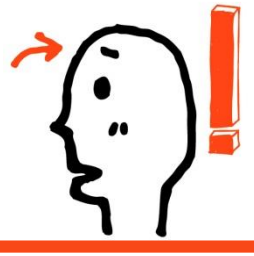
- 숫자의 관점에서 이해하려 들지 말자.
- 자바에서의 true와 false는 그 자체로 저장이 가능한 데이터이다.
- true와 false의 저장을 위한 자료형 boolean!

```
class Boolean
{
    public static void main(String[] args)
    {
        boolean b1=true;
        boolean b2=false;

        System.out.println(b1);
        System.out.println(b2);
        System.out.println(3<4);
        System.out.println(3>4);
    }
}
```

실행결과

```
true
false
true
false
```

자료형을 기반으로 표현이 되는 상수



상수를 언제 사용했었지?

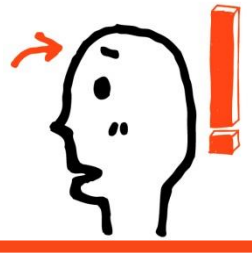
표현되는 데이터는 상수 아니면 변수

- int num = 1 + 5;
- System.out.println(2.4 + 7.5);

상수와 변수의 비교

- 변수와 마찬가지로 상수도 메모리 공간에 저장이 된다.
- 다만 이름이 존재하지 않으니 값의 변경이 불가능하다.
- 상수는 존재 의미가 없어지면 바로 소멸된다.

자료형을 기반으로 표현이 되는 상수



상수도 자료형을 기반으로 저장됩니다.

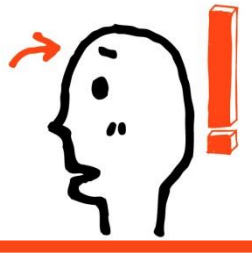
상수의 저장방식의 근거는 자료형

- int, double과 같은 자료형은 데이터 표현의 기준이다.
- 따라서 변수뿐만 아니라 상수의 데이터 저장 및 참조의 기준이다.

정수형 상수와 실수형 상수의 표현 자료형

- 정수형 상수 int형으로 표현
- 실수형 상수 double형으로 표현

자료형을 기반으로 표현이 되는 상수



접미사 이야기

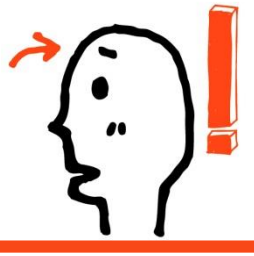
- 다음 세 문장에서 컴파일 오류가 발생하는 이유는?

```
int num1=10000000000;      // num에 저장 불가!
long num2=10000000000;     // 상수의 표현이 먼저이므로!
float num3=12.45;          // 12.45는 double형 상수
```

- 접미사를 이용한 상수표현방식의 변경

```
long num1=10000000000L; // 접미사 L은 long형 상수표현을 의미
float num2=12.45F;      // 접미사 F는 float형 상수표현을 의미
```

형 변환으로 데이터 형태를 변경하자



자료형의 변환이 의미하는 것은?

- 자료형의 변환은 표현방법의 변환

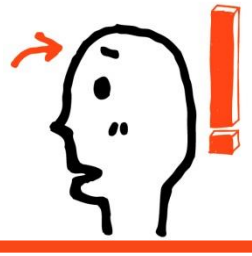
```
int main(String[] args)
{
    short num1=10;
    short num2=20;
    short result = num1 + num2;
    . . . .
}
```

num1(10) → 00000000 00001010
num2(20) → 00000000 00010100

short to int

int형 정수 10 → 00000000 00000000 00000000 00001010
int형 정수 20 → 00000000 00000000 00000000 00010100

형 변환으로 데이터 형태를 변경하자

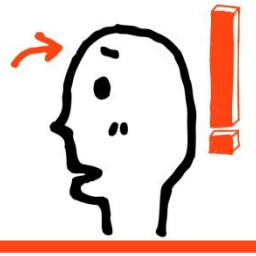


자료형의 변환이 의미하는 것은?

- 자료형의 변환은 표현방법의 변환

```
int형 정수 1 → 00000000 00000000 00000000 00000001
                    short to int
float형 실수 1.0 → 00111111 10000000 00000000 00000000
```

형 변환으로 데이터 형태를 변경하자



자료형을 일치시켜야 하는 이유?

- 사람도 못하는데 컴퓨터가 할 수 있겠는가?

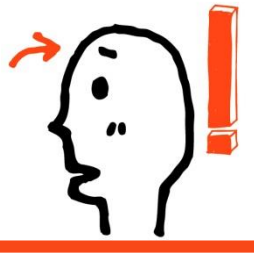
```
  00000000 00000000 00000000 00000001
+ 00000000 00000000 00000000 00000010
-----------------------------------
  00000000 00000000 00000000 00000011
```

[그림 3-1 : 1 더하기 2는 3]

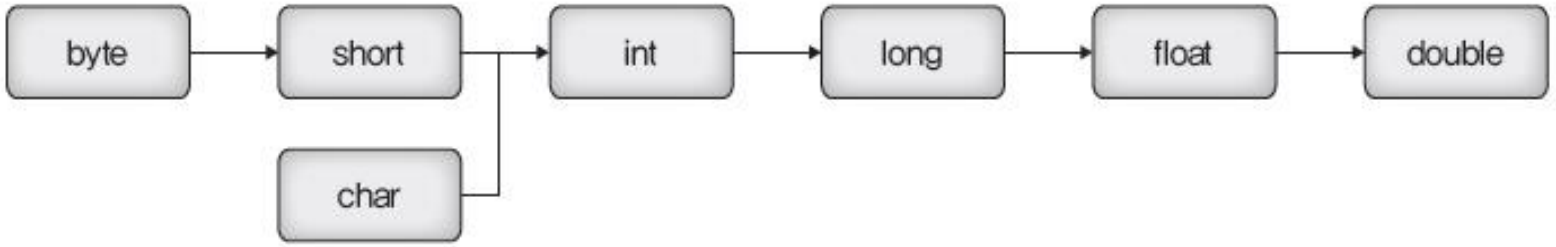
```
  00000000 00000000 00000000 00000001
+ 00111111 10000000 00000000 00000000
-----------------------------------
  ???????? ???????? ???????? ????????
```

[그림 3-2 : 1 더하기 1.0은?]

형 변환으로 데이터 형태를 변경하자



- boolean형을 제외하면 모두 형 변환이 가능 (boolean형은 숫자형이 아닌 논리형)



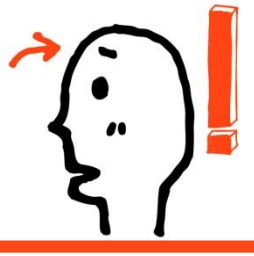
△ 그림 2-5 안정적인 자동 형 변환 순서

- 오른쪽에서 왼쪽 방향으로 형 변환을 할 경우에는 반드시 명시적 형 변환을 해주어야 함
- 왼쪽 방향으로 향할 수록 각 데이터형의 메모리 크기가 줄어들기 때문에 데이터 손실이 발생할 수 있음

```
double num2=3.5f+12;
```

12가 12f로 자동 형 변환

형 변환으로 데이터 형태를 변경하자



명시적 형 변환

명시적 형 변환을 하는 이유

- 자동 형 변환 발생지점의 표시를 위해서 case 1
- 자동 형 변환의 규칙에 위배되지만 변환이 필요한 상황 case 2

case 2

```
long num1 = 2147483648L;
int num2 = (int)num1;
```

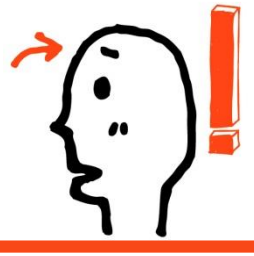
case 1

```
int num3 = 100;
long num4 = (long)num3;
```

Android Java

3장 자바에서 사용하는 연산자 & 연산

자바에서 사용하는 연산자 & 연산



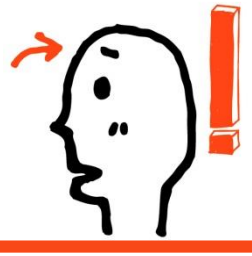
01 기본 연산자 익히기

02 논리 연산자와 비트 연산자 익히기

03 알아두면 편리한 시프트 연산자와 3항 연산자

04 String을 사용한 기본 문자열 연산 익히기

기본 연산자 익히기



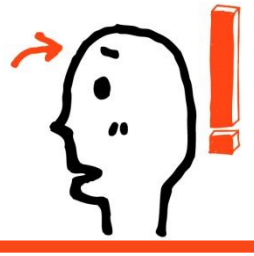
대입 연산자와 산술연산자

- **대입 연산자는?** 변수에 값이나 연산 결과를 저장하는 연산자
- **산술 연산자는?** 연산자들 중 가장 기본이 되는 연산자로서 사칙 연산자들이 이에 속함

표 4-1 산술 연산자의 종류

산술 연산자	설명	사용 예	
=	대입	a=3	정수 3을 a에 대입한다.
+	더하기	a=5+3	5와 3을 더한 값을 a에 대입한다.
–	빼기	a=5–3	5에서 3을 뺀 값을 a에 대입한다.
*	곱하기	a=5*3	5와 3을 곱한 값을 a에 대입한다.
/	나누기	a=5/3	5를 3으로 나눈 값을 a에 대입한다.
%	나머지 값	a=5%3	5를 3으로 나눈 뒤 나머지 값을 a에 대입한다.

기본 연산자 익히기

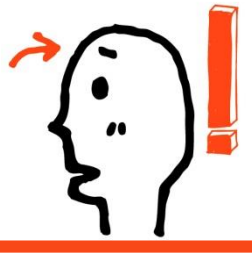


```
class ArithOp
{
    public static void main(String[] args)
    {
        int n1=7;
        int n2=3;

        int result=n1+n2;
        System.out.println("덧셈 결과 : " + result);

        result=n1-n2;
        System.out.println("뺄셈 결과 : " + result);
        System.out.println("곱셈 결과 : " + n1*n2);
        System.out.println("나눗셈 결과 : " + n1/n2);
        System.out.println("나머지 결과 : " + n1%n2);
    }
}
```

실행 결과

```
덧셈 결과 : 10
뺄셈 결과 : 4
곱셈 결과 : 21
나눗셈 결과 : 2
나머지 결과 : 1
```

기본 연산자 익히기



- 피연산자가 정수면 정수형 연산진행
- 피연산자가 실수면 실수형 연산진행, 단 % 연산자 제외!

```
class DivOpnd
{
    public static void main(String[] args)
    {
        System.out.println("정수형 나눗셈 : " + 7/3);
        System.out.println("실수형 나눗셈 : " + 7.0f/3.0f);
        System.out.println("형 변환 나눗셈 : " + (float)7/3);
    }
}
```

실행결과

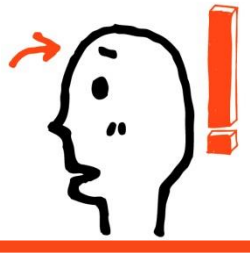
```
정수형 나눗셈 : 2
실수형 나눗셈 : 2.3333333
형 변환 나눗셈 : 2.3333333
```

```
class AmpOpnd
{
    public static void main(String[] args)
    {
        System.out.println("정수형 나머지 : " + 7%3);
        System.out.println("실수형 나머지 : " + 7.2 % 2.0);
    }
}
```

실행결과

```
정수형 나머지 : 1
실수형 나머지 : 1.2000000000000002
```

기본 연산자 익히기



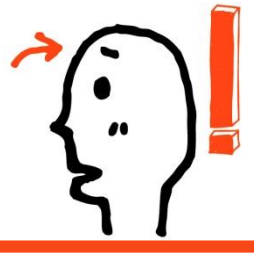
복합 대입 연산자

'=' 연산자 외에 '+ ='나 '- =' 등과 같은 연산자들을 사용하면 실제로 풀어 쓴 연산보다 좀더 빠른 연산이 이뤄지는 장점이 있음

기호	연산자 사용법	설명
=	a = 3	변수명 a에 변숫값 3을 대입한다.
+=	a += b	a = a + b 의미와 같으며 변수 a에 b 값을 더한 후 그 값을 변수 a에 대입한다.
-=	a -= b	a = a - b 의미와 같으며 변수 a에 변수 b 값을 뺀 후 그 값을 변수 a에 대입한다.
*=	a *= b	a = a * b 의미와 같으며 변수 a에 변수 b 값을 곱한 후 그 값을 변수 a에 대입한다.
/=	a /= b	a = a / b 의미와 같으며 변수 a에 변수 b 값을 나눈 후 그 값을 변수 a에 대입한다.

△ 표 3-1 대입 연산자의 종류와 사용법

기본 연산자 익히기



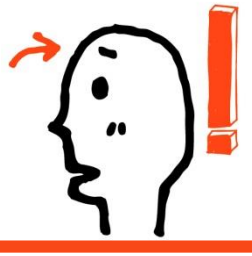
복합 대입 연산자

```
class Comp
{
    public static void main(String[] args)
    {
        double e=3.1;
        e+=2.1;
        e*=2;
        int n=5;
        n*=2.2;
        System.out.println(e);
        System.out.println(n);
    }
}
```

실행 결과

```
10.4
11
```

기본 연산자 익히기



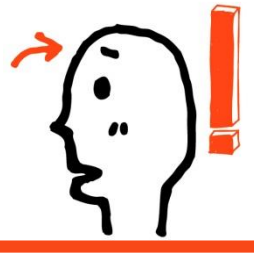
비교(관계) 연산자

- **비교 연산자는?** 피연산자를 비교하기 위해서 사용 결과값은 **true/false**
- 비교 연산자는 조건문과 반복문에서 많이 사용

수식	예시	설명
〉	a〉b	a가 b보다 크면 true, 아니면 false
〈	a〈b	a가 b보다 작으면 true, 아니면 false
〉=	a〉=b	a가 b보다 크거나 같으면 true, 아니면 false
〈=	a〈=b	a가 b보다 작거나 같으면 true, 아니면 false
==	a==b	a와 b가 같으면 true, 아니면 false
!=	a!=b	a와 b가 다르면 true, 아니면 false

연산의 결과로 true or false 반환

기본 연산자 익히기



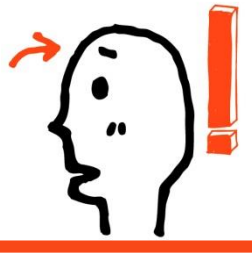
비교 연산자의 예

```
class CmpOp
{
    public static void main(String[] args)
    {
        int A=10, B=20;

        if(true)
            System.out.println("참 입니다!");
        else
            System.out.println("거짓 입니다!");

        if(A>B)
            System.out.println("A가 더 크다!");
        else
            System.out.println("A가 더 크지 않다!");

        if(A!=B)
            System.out.println("A와 B는 다르다!");
        else
            System.out.println("A와 B는 같다!");
    }
}
```

실행 결과

```
참 입니다!
A가 더 크지 않다!
A와 B는 다르다!
```

논리 연산자와 비트 연산자 익히기

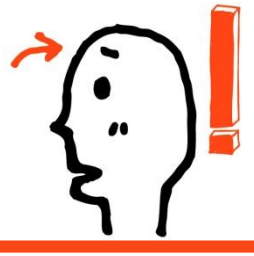


true/false 판단을 위한 논리 연산자

- **논리 연산자는?** true 또는 false값을 갖는 boolean형의 피연산자들을 판단해서 그 결과값을 true/false 로 반환하는 것
- 비교 연산자와 마찬가지로 논리 연산자들도 조건문과 반복문에서 많이 사용

연산자	연산자의 기능	결합방향
&&	예) A && B A와 B 모두 true이면 연산결과는 true (논리 AND)	➡
||	예) A || B A와 B 둘 중 하나라도 true이면 연산결과는 true (논리 OR)	➡
!	예) !A 연산결과는 A가 true이면 false, A가 false이면 true (논리 NOT)	⬅

논리 연산자와 비트 연산자 익히기



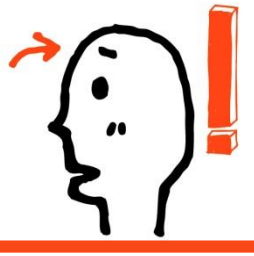
true/false 판단을 위한 논리 연산자

```
class LogicOp
{
    public static void main(String[] args)
    {
        int num1=10, num2=20;
        boolean result1=(num1==10 && num2==20);
        boolean result2=(num1<=12 || num2>=30);
        System.out.println("num1==10 그리고 num2==20 : " + result1);
        System.out.println("num1<=12 또는 num2>=30 : " + result2);
        if(!(num1==num2))
            System.out.println("num1과 num2는 같지 않다.");
        else
            System.out.println("num1과 num2는 같다.");
    }
}
```

실행 결과

```
num1==10 그리고 num2==20 : true
num1<=12 또는 num2>=30 : true
num1과 num2는 같지 않다.
```

논리 연산자와 비트 연산자 익히기



true/false 판단을 위한 논리 연산자 (NOT 연산자)

코드 3-3 package com.gilbut.chapter3;

```
1   public class NotOperation
2   {
3      public static void main(String[] args)
4      {
5         boolean isWorking = false;
6
7         isWorking = !isWorking;
8         System.out.println("isWorking : " + isWorking);
9         isWorking = !isWorking;
10        System.out.println("isWorking : " + isWorking);
11     }
12  }
```

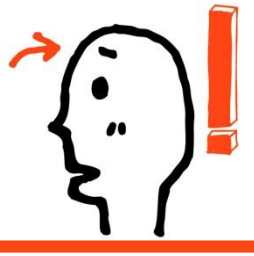
false값을 가진 isWorking 변수에 NOT 연산을 한 후 다시 isWorking 변수에 대입

실행결과

```
isWorking : true
isWorking : false
```

- false로 선언된 isWorking 변수에 NOT 연산자를 붙이면 false에서 true로 값이 바뀌는 것을 알 수 있음

논리 연산자와 비트 연산자 익히기

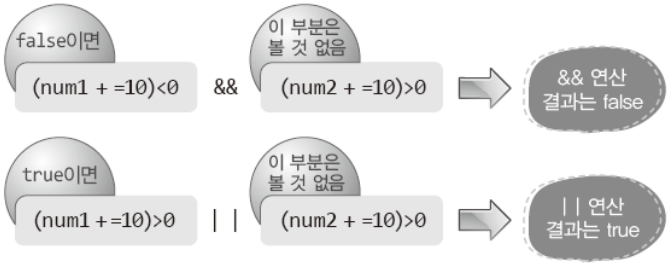


true/false 판단을 위한 논리 연산자 (NOT 연산자)

```
class SCE
{
    public static void main(String[] args)
    {
        int num1=0, num2=0;
        boolean result;

        result = (num1+=10)<0 && (num2+=10)>0;
        System.out.println("result="+result);
        System.out.println("num1="+num1+", num2="+num2);

        result = (num1+=10)>0 || (num2+=10)>0;
        System.out.println("result="+result);
        System.out.println("num1="+num1+", num2="+num2);
    }
}
```

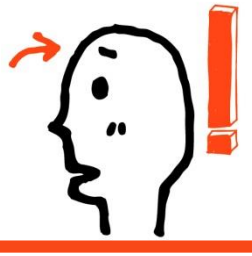
실행결과

```
result=false
num1=10, num2=0
result=true
num1=20, num2=0
```



Short-Circuit Evaluation

기본 연산자 익히기



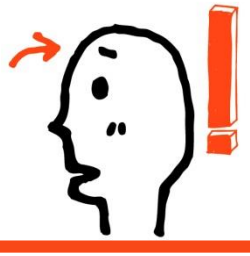
간결한 코드 작성을 위한 단항 연산자

- **단항 연산자(unary operator)는?** 피연산자가 하나인 연산자
- 피연산자가 2개이면 이항 연산자, 3개이면 삼항 연산자
- **피연산자는?** 연산의 대상이 되는 변수

	연산자	설명
증감 연산자	++	피연산자의 변숫값을 1 증가시킨다. 사용법 : i++, ++i
	--	피연산자의 변숫값을 1 감소시킨다. 사용법 : i--, --i

△ 표 3-2 단항 연산자의 종류와 설명

기본 연산자 익히기



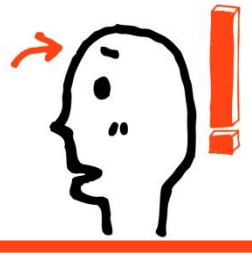
간결한 코드 작성을 위한 단항 연산자

- 단항 연산자는 연산자와 변수를 붙여서 사용
- **전위형** 단항 연산자가 피연산자의 앞에 위치
- **후위형** 연산자가 피연산자의 뒤에 위치
- 전위형이냐 후위형 이냐는 연산자의 종류에 따라서 다르며, 둘 다 지원하는 연산자도 있음

• 전위형 : (연산자) (피연산자)

• 후위형 : (피연산자) (연산자)

기본 연산자 익히기



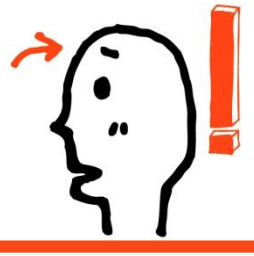
간결한 코드 작성을 위한 단항 연산자

- 사용 가능한 변수형 : **byte, short, char, int, long, float, double형**
- **전위형** 변수가 사용되기 이전에 증감 연산
- **후위형** 변수가 사용된 이후 증감 연산을 실시

```
i = i + 1;   //변수 i는 int형의 변수라고 가정한다.
i = i++;
```

- 증감 연산자를 사용하는 이유
  1. 증감 연산자를 사용하는 것이 이항 연산자를 사용하는 것보다 속도가 빠르기 때문
  2. 코드를 보다 간단하게 구성할 수 있기 때문

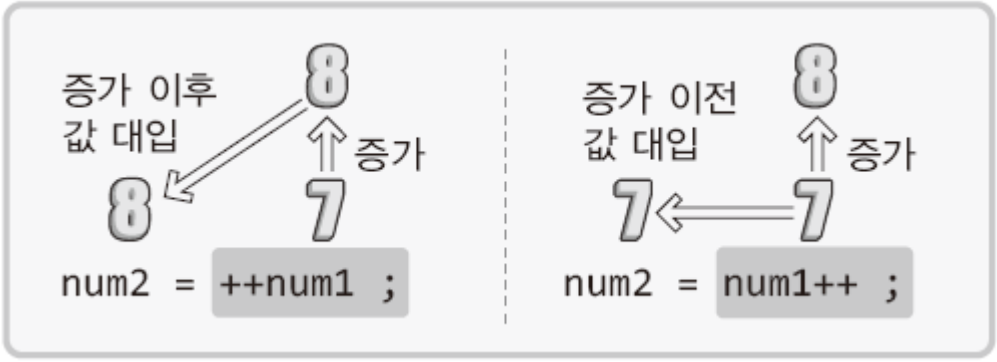
기본 연산자 익히기



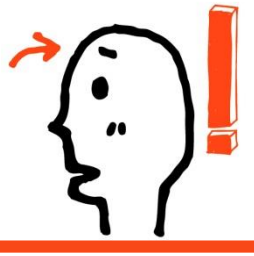
간결한 코드 작성을 위한 단항 연산자

연산자	연산자의 기능	결합방향
++ (prefix)	피연산자에 저장된 값을 1 증가 예) val = ++n;	⬅
-- (prefix)	피연산자에 저장된 값을 1 감소 예) val = --n;	⬅

연산자	연산자의 기능	결합방향
++ (postfix)	피연산자에 저장된 값을 1 증가 예) val = n++;	⬅
-- (postfix)	피연산자에 저장된 값을 1 감소 예) val = n--;	



기본 연산자 익히기



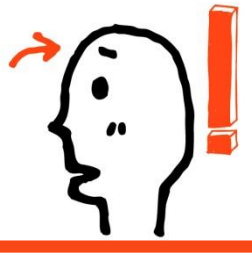
간결한 코드 작성을 위한 단항 연산자

코드 3-2 package com.gilbut.chapter3;

```
1   public class UnaryOperator
2   {
3       public static void main(String[] args)
4       {
5           int vluAfter = 0;
6           long vluBefore = 0;
7           char chrAfter = 'A';
8
9           System.out.println("First reference : " + vluAfter++ );
10          System.out.println("First reference : " + --vluBefore);
11          System.out.println("First reference : " + chrAfter++);
12
13          System.out.println("second reference : " + vluAfter);
14          System.out.println("second reference : " + vluBefore);
15          System.out.println("second reference : " + chrAfter);
16      }
17  }
```

vluAfter과 vluBefore 변수는 둘다 0 값을 갖고 있습니다. 각각 후위형과 전위형 연산자를 사용한 실행 결과를 비교해보세요.

기본 연산자 익히기



간결한 코드 작성을 위한 단항 연산자

실행결과

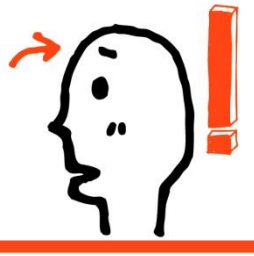
```
First reference : 0
First reference : -1
First reference : A
second reference : 1
second reference : -1
second reference : B
```

vluBefore 변수의 경우 전위형 연산자의 영향으로 첫 번째 참조부터 값이 1만큼 감소합니다.

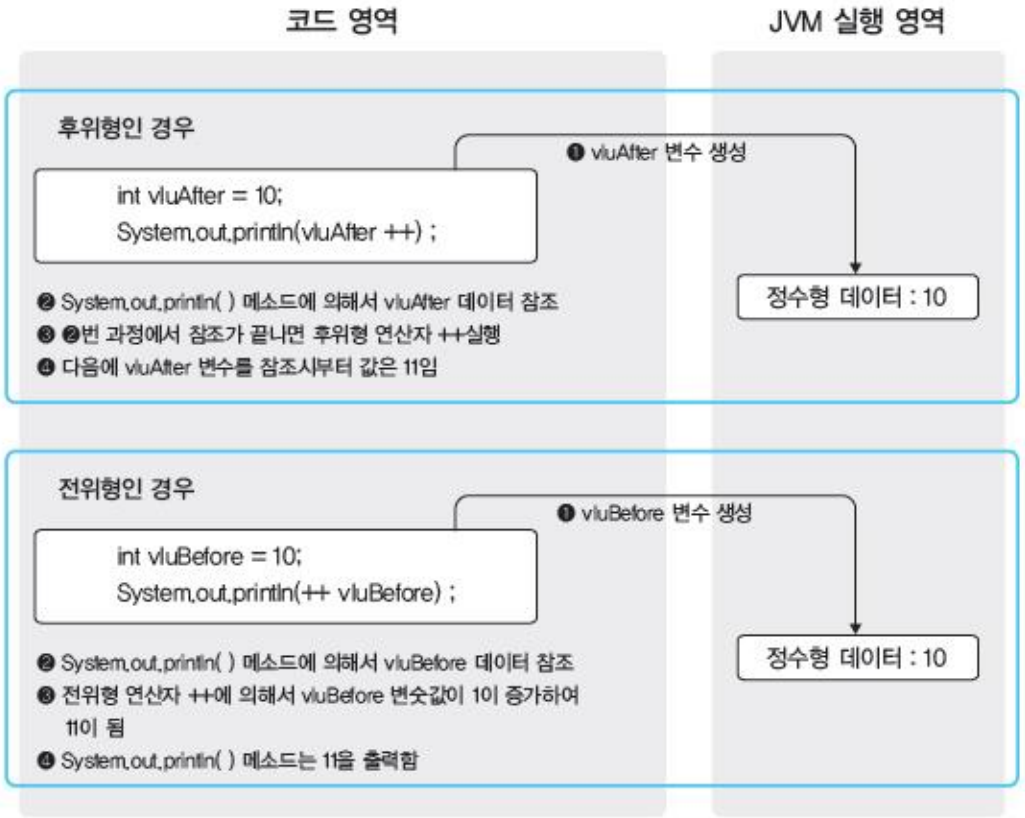
vluAfter 변수의 경우 후위형 연산자의 영향으로 첫 번째 참조될 때는 원래 값을 반환하지만 두 번째 참조될 때는 값이 1만큼 증가한 것을 확인할 수 있습니다.

- UnaryOperator 클래스는 증감 연산자의 위치에 따른 연산 결과의 차이를 확인할 수 있는 예제

기본 연산자 익히기



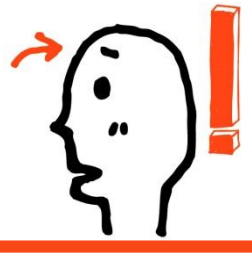
간결한 코드 작성을 위한 단항 연산자



- 전위형과 후위형의 차이는 연산자의 실행 시점에 따라 결정
- 해당 연산의 실행 시점이 프로그램에서 변수가 참조되고 난 후냐 혹은 참조되기 전이냐에 따라 결정

그림 3-1 전위형과 후위형 연산자의 동작 순서

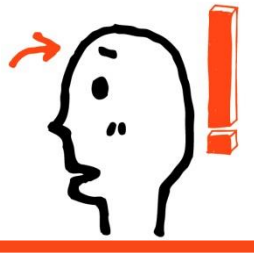
논리 연산자와 비트 연산자 익히기



2진수 연산을 위한 비트 연산자

- **비트 연산자는?** byte, short, int, char 변수형에 대해서 사용 가능한 연산자
- 비트 연산자와 관련된 모든 연산은 2진수 체계에서 처리
- 비트 연산자를 사용하는 이유 : 컴퓨터 내부의 프로세스는 2진수로 처리되고 있기 때문에 보다 빠른 연산이 가능하기 때문
- 요즘에는 하드웨어 성능이 좋아졌기 때문에 속도보다는
  이미지 프로세싱 혹은 네트워크 패킷 프로세싱과 같이
  비트(bit)로 이루어진 하위 레벨(low-level)의 데이터를 다루기 위해서 많이 사용

논리 연산자와 비트 연산자 익히기



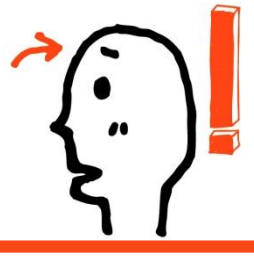
2진수 연산을 위한 비트 연산자

연산자	기호	설명
비트 NOT 연산자	~	2진수의 각 자릿수에 대해서 NOT 연산을 한다. 피연산자가 하나이기 때문에 단항 연산자처럼 사용하면 된다. • 사용 방법 : ~0B001
비트 AND 연산자	&	2진법으로 표시된 두 개의 피연산자가 필요하고 2진법 각 자릿수에 대해서 AND 연산을 실시한다. • 사용 방법 : 0B001 & 0B010
비트 OR 연산자	|	2진법으로 표시된 두 개의 피연산자가 필요하며 2진법 각 자릿수에 대해서 OR 연산을 실시한다. • 사용 방법 : 0B001 | 0B010
비트 XOR 연산자	^	2진법으로 표시된 두 개의 피연산자가 필요하며 2진법 각 자릿수에 대해서 XOR 연산을 실시한다. XOR 연산은 피연산자의 두 값이 서로 다를 때만 true를 반환한다. • 사용 방법 : 0B001 ^ 0B010

△ 표 3-8 비트 연산자의 소개와 설명 표

자바에서 사용하는 연산자 & 연산

논리 연산자와 비트 연산자 익히기



비트연산 진리표

비트 A	비트 B	비트 A & 비트 B
1	1	1
1	0	0
0	1	0
0	0	0

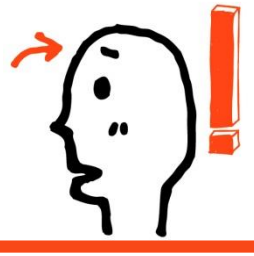
& (and)

비트 A	비트 B	비트 A | 비트 B
1	1	1
1	0	1
0	1	1
0	0	0

| (OR)

자바에서 사용하는 연산자 & 연산

논리 연산자와 비트 연산자 익히기



비트연산 진리표

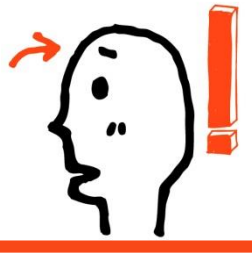
비트 A	비트 B	비트 A ^ 비트 B
1	1	0
1	0	1
0	1	1
0	0	0

^(XOR)

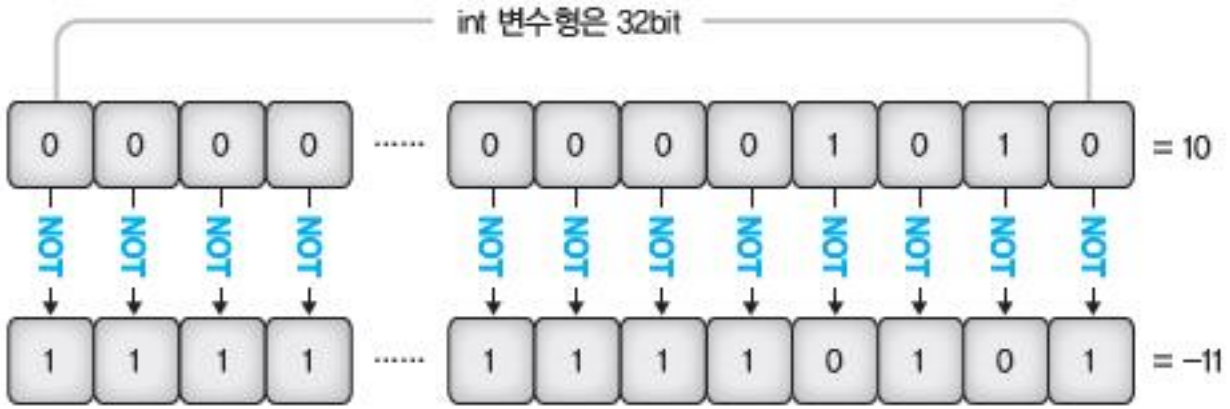
비트	~비트
1	0
0	1

~(NOT)

논리 연산자와 비트 연산자 익히기



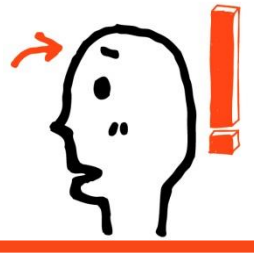
2진수 연산을 위한 비트 연산자 (비트 NOT 연산자)



△ 그림 3-2 NOT 연산자 처리 과정

▷ 위와 같은 과정을 거치면 ~(NOT) 연산의 결과는 10진수로 -11이 됨

논리 연산자와 비트 연산자 익히기



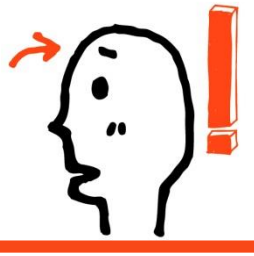
비트연산의 예

```
class BitOperator
{
    public static void main(String[] args)
    {
        int num1=5;     /* 00000000 00000000 00000000 00000101 */
        int num2=3;     /* 00000000 00000000 00000000 00000011 */
        int num3=-1;    /* 11111111 11111111 11111111 11111111 */

        System.out.println(num1 & num2);
        System.out.println(num1 | num2);
        System.out.println(num1 ^ num2);
        System.out.println(~num3);
    }
}
```

실행 결과

```
1
7
6
0
```

논리 연산자와 비트 연산자 익히기



2진수 연산을 위한 비트 연산자 (비트 AND, OR, XOR 연산자)

코드 3-6 package com.gilbut.chapter3;

```
1   public class bitOperation
2   {
3       public static void main(String[] args)
4       {
5           int b1 = 0B0010;
6           int b2 = 0B0101;
7           int b3 = 0B1111;
8
9           int rtAndOp = b1 & b3;
10
11          int rtOrOp = b1 | b2;
12
13          int rtXorOp = b1 ^ b3;
14
15          System.out.println("b1 AND b3 : " + Integer.toBinaryString(rtAndOp));
16          System.out.println("b1 OR  b2 : " + Integer.toBinaryString(rtOrOp));
17          System.out.println("b1 XOR b3 : " + Integer.toBinaryString(rtXorOp));
18      }
19  }
```

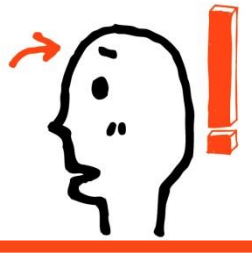
비트 AND 연산자

비트 OR 연산자

비트 XOR 연산자

자바에서 사용하는 연산자 & 연산

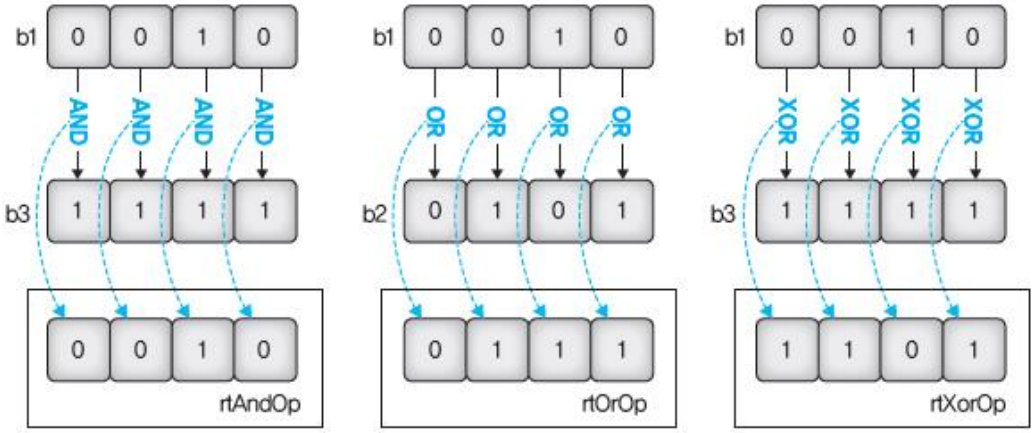
논리 연산자와 비트 연산자 익히기



2진수 연산을 위한 비트 연산자 (비트 AND, OR, XOR 연산자)

실행결과

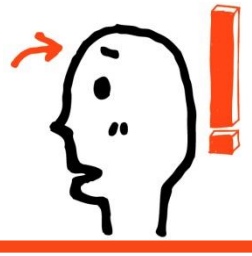
```
b1 AND b3 : 10
b1 OR  b2 : 111
b1 XOR b3 : 1101
```



NOT 연산자의 방식과 같이 각 비트의 자릿수마다 연산이 어떻게 이루어지는지를 그림으로 표현한 것

그림 3-3 AND, OR, XOR 연산자의 처리 과정

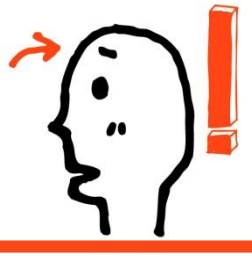
알아두면 편리한 시프트 연산자와 3항 연산자



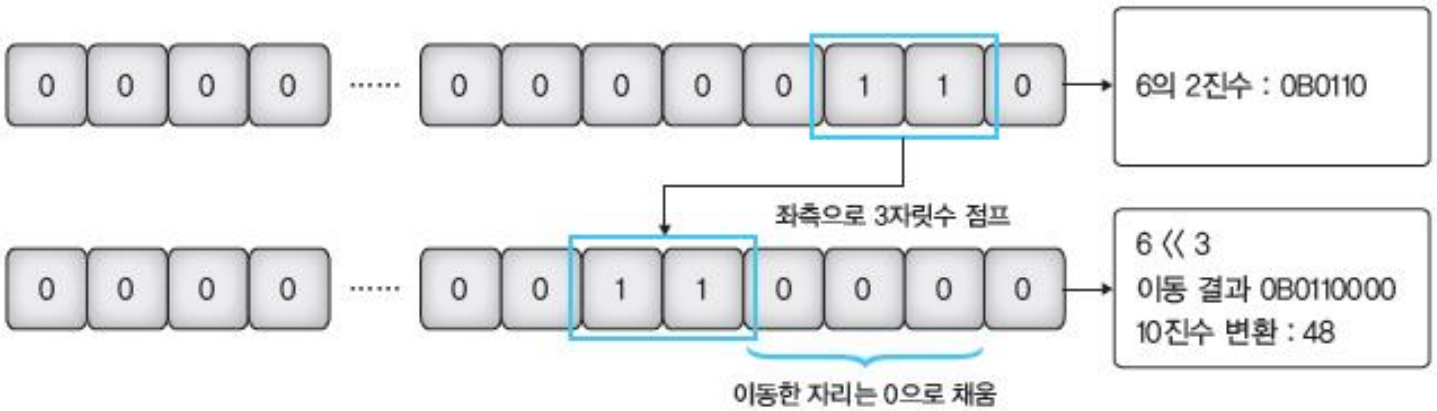
시프트 연산자

- **시프트 연산자는?** 정수형 변수에서만 사용 가능한 연산자
- 이진법으로 표기된 데이터의 자리를 이동시키는 역할을 하며
  이동시키는 방향에 따라서 '<< ' 혹은 ' >>'으로 표기
- 시프트 연산자의 사용 방법은 사칙 연산자와 같이 a >> 2 형태로 사용하며
  a는 피연산자(대상 데이터) 이며, 2는 시프트(이동)하는 횟수를 의미

알아두면 편리한 시프트 연산자와 3항 연산자

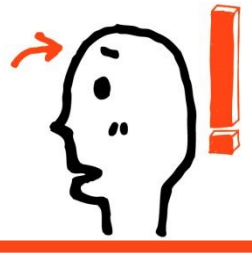


시프트 연산자

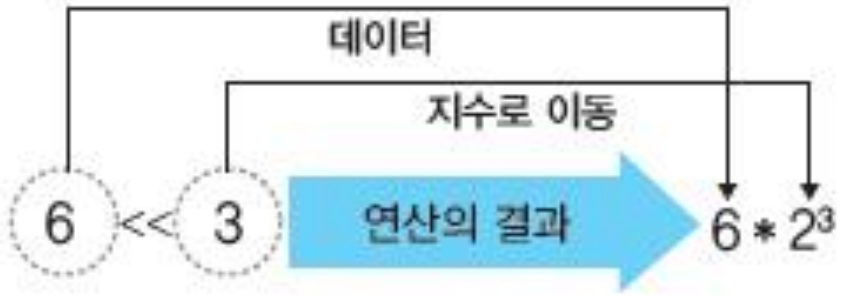


- **6 << 3 연산**이 어떻게 이루어지는지 확인할 수 있음
- 양수의 데이터를 의미하는 bit 1을 시프트 연산자의 방향으로 3칸 점프
- 이동하고 남은 칸은 bit 0으로 채움
- 6 << 3의 결과값을 10진수로 표현하면 48

알아두면 편리한 시프트 연산자와 3항 연산자

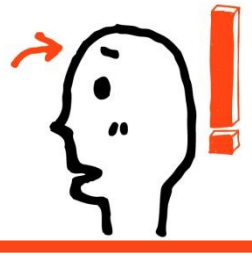


시프트 연산자



- 시프트 연산의 결과를 쉽게 암산하기 위한 수식 이해
- 그림 3-5와 같은 연산의 결과로써 **6 << 3** 의 결과값은 6 * 8 즉, 48
- 연산자에 따라서 시프트 하는 방식이 약간씩 다름

알아두면 편리한 시프트 연산자와 3항 연산자

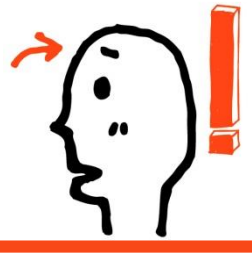


시프트 연산자

연산자	설명
<< 연산자	• 데이터를 왼쪽으로 지정한 숫자만큼 자릿수를 점프한다. • 점프하고 남은 자릿수는 피연산자가 양수인 경우에는 0으로 채우고 피연산자가 음수인 경우에는 1로 채운다. • 음수, 양수를 의미하는 맨 앞의 비트는 연산에 관여하지 않는다. • 연산 결과를 10진수로 바꾸면 피연산자 * $2^{(점프한\ 수)}$와 같다.
>> 연산자	• 데이터를 오른쪽으로 지정한 숫자만큼 자릿수를 점프한다. • 점프하고 남은 자릿수는 피연산자가 양수인 경우에는 0으로 채우고 피연산자가 음수인 경우에는 1로 채운다. • 음수, 양수를 의미하는 맨 앞의 비트는 연산에 관여하지 않는다. • 연산 결과를 10진수로 바꾸면 피연산자 / $2^{(점프한\ 수)}$와 같다.
>>> 연산자	• 데이터를 오른쪽으로 이동한다. • 점프하고 남은 자릿수는 부호에 상관 없이 0으로 채운다. • 항상 양수를 유지하므로 unsigned shift operator이다.

△ 표 3-10 시프트 연산자의 종류와 설명

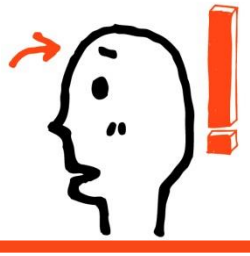
알아두면 편리한 시프트 연산자와 3항 연산자



시프트 연산자

```
class BitShiftOp
{
    public static void main(String[] args)
    {
        System.out.println(2 << 1);    // 4 출력
        System.out.println(2 << 2);    // 8 출력
        System.out.println(2 << 3);    // 16 출력

        System.out.println(8 >> 1);    // 4 출력
        System.out.println(8 >> 2);    // 2 출력
        System.out.println(8 >> 3);    // 1 출력
        System.out.println(-8 >> 1);   // -4 출력
        System.out.println(-8 >> 2);   // -2 출력
        System.out.println(-8 >> 3);   // -1 출력

        System.out.println(-8 >>> 1);  // 2147483644 출력
    }
}
```

String을 사용한 기본 문자열 연산 익히기



문자열

큰따옴표로 감싼 문자들을 문자열이라고 함.

문자열은 char 타입에 저장할 수 없음

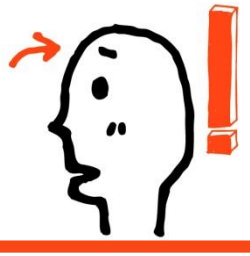
```
char var1 = "A";
char var2 = "홍길동";
```

String 타입

문자열을 String 타입 변수에 저장

```
String var1 = "A";
String var2 = "홍길동";
```

String을 사용한 기본 문자열 연산 익히기



- \+ 연산

  피연산자가 모두 숫자일 경우 덧셈 연산

  피연산자 중 하나가 문자열일 경우 나머지 피연산자도 문자열로 자동 변환되고 문자열 결합 연산

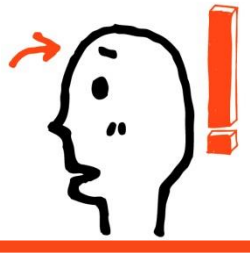
```
int value = 3 + 7;     → int value = 10;
String str = "3" + 7;  → String str = "3" + "7"; → String str = "37";
String str = 3 + "7";  → String str = "3" + "7"; → String str = "37";
```

- \+ 연산은 앞에서부터 순차적으로 수행

  먼저 수행된 연산이 결합 연산인 경우 이후 모든 연산이 결합 연산이 됨

```
int value = 1 + 2 + 3;     → int value = 3 + 3;      → int value = 6;
String str = 1 + 2 + "3";  → String str = 3 + "3";   → String str = "33";
String str = 1 + "2" + 3;  → String str = "12" + 3;  → String str = "123";
String str = "1" + 2 + 3;  → String str = "12" + 3;  → String str = "123";
```

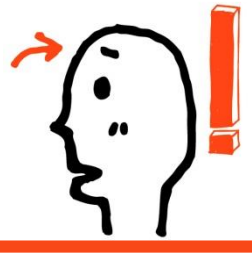
String을 사용한 기본 문자열 연산 익히기



문자열을 기본 타입으로 강제 변환

변환 타입	사용 예
String → byte	`String str = "10";` `byte value = Byte.parseByte(str);`
String → short	`String str = "200";` `short value = Short.parseShort(str);`
String → int	`String str = "300000";` `int value = Integer.parseInt(str);`
String → long	`String str = "40000000000";` `long value = Long.parseLong(str);`
String → float	`String str = "12.345";` `float value = Float.parseFloat(str);`
String → double	`String str = "12.345";` `double value = Double.parseDouble(str);`
String → boolean	`String str = "true";` `boolean value = Boolean.parseBoolean(str);`

String을 사용한 기본 문자열 연산 익히기



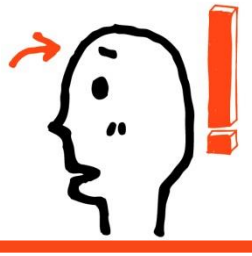
- 문자열이 숫자 외 요소를 포함할 경우 숫자 타입 변환 시도할 경우 숫자 형식 예외 발생

```
String str = "1a";
int value = Integer.parseInt(str);    //NumberFormatException 발생
```

- String.valueOf() 메소드 사용하여 기본 타입을 문자열로 변환

```
String str = String.valueOf(3);
```

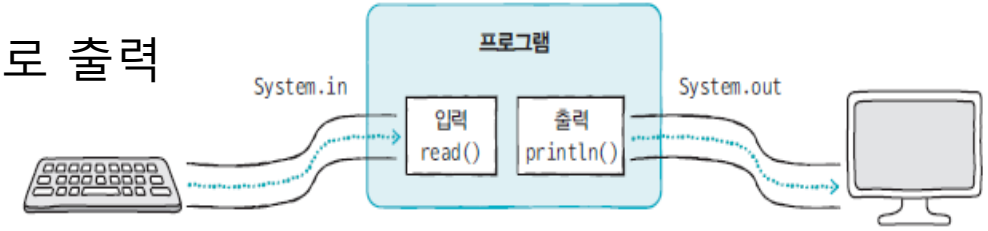
시스템 표준 입 출력 장치의 이해



- System.out

  시스템의 표준 출력 장치로 출력

```
System.out.println("출력 내용");
```

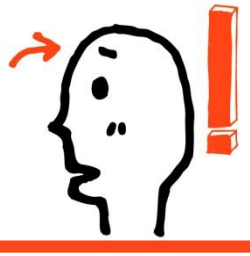


- System.in

  시스템의 표준 입력 장치에서 읽음

```
System.in.read();
```

시스템 표준 입 출력 장치의 이해



println() 메소드

괄호 안에 리터럴 넣으면 그대로 출력 / 변수 넣으면 저장된 값 출력

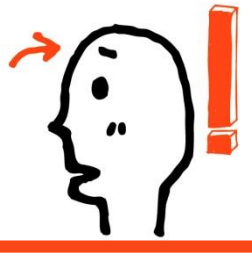
```
System. + out. + println(리터럴 또는 변수);
```

System. : 시스템이 가지고 있는
out. : 출력 장치로
println(리터럴 또는 변수); : 괄호 안의 내용을 출력하고 행을 바꿔라

다양한 출력 메소드

메소드	의미
println(내용);	괄호 안의 내용을 출력하고 행을 바꿔라
print(내용);	괄호 안의 내용을 출력만 해라
printf("형식문자열", 값1, 값2, …);	괄호 안의 첫 번째 문자열 형식대로 내용을 출력해라

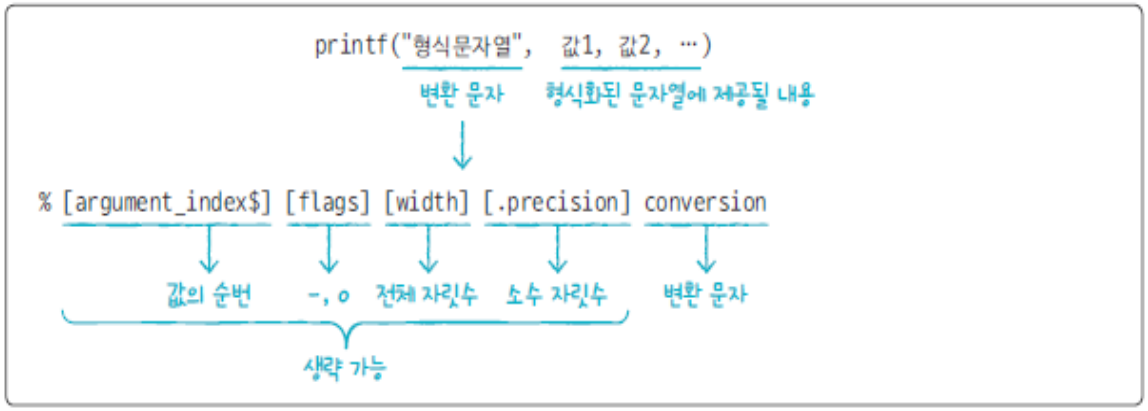
시스템 표준 입 출력 장치의 이해



- printf() 메소드

  개발자가 원하는 형식화된 문자열 (formal string) 출력

  (전체 출력 자리수 및 소수 자릿수 제한)

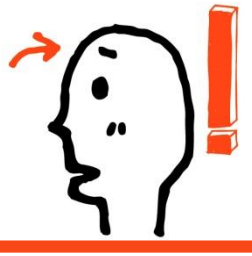


- 형식 문자열에서 %와 conversion 외에는 모두 생략 가능

  conversion에는 제공되는 값의 타입에 따라 d(정수), f(실수), s(문자열) 입력

```
System.out.printf("이름: %s",  "감자바"); → 이름: 감자바
System.out.printf("나이: %d",  25 );     → 나이: 25
```

시스템 표준 입 출력 장치의 이해



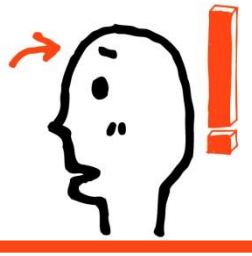
- 형식 문자열에 포함될 값 2개 이상인 경우 값의 순번(argument_index$) 표시해야

```
System.out.printf("이름: %1$s, 나이: %2$d",  "김자바",  25); → 이름: 김자바, 나이: 25
```

- 다양한 형식 문자열

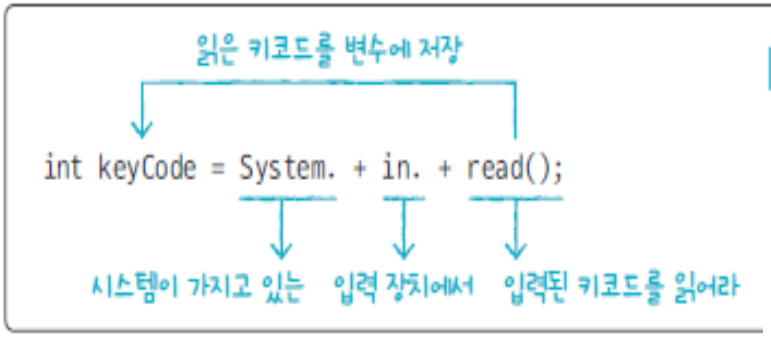
형식화된 문자열		설명	출력 형태
정수	%d	정수	123
	%6d	6자리 정수. 왼쪽 빈 자리 공백	___123
	%-6d	6자리 정수. 오른쪽 빈 자리 공백	123___
	%06d	6자리 정수. 왼쪽 빈 자리 0 채움	000123
실수	%10.2f	소수점 이상 7자리, 소수점 이하 2자리. 왼쪽 빈 자리 공백	____123.45
	%-10.2f	소수점 이상 7자리, 소수점 이하 2자리. 오른쪽 빈 자리 공백	123.45____
	%010.2f	소수점 이상 7자리, 소수점 이하 2자리. 왼쪽 빈 자리 0 채움	0000123.45
문자열	%s	문자열	abc
	%6s	6자리 문자열. 왼쪽 빈 자리 공백	___abc
	%-6s	6자리 문자열. 오른쪽 빈 자리 공백	abc___
특수 문자	\t	탭(tab)	
	\n	줄 바꿈	
	%%	%	%

시스템 표준 입 출력 장치의 이해



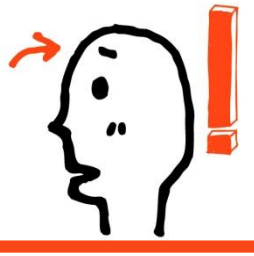
- 키코드
  키보드에서 키를 입력할 때 프로그램에서 숫자로 된 키코드를 읽음
  System.in의 read() 사용

  얻은 키코드는 대입 연산자 사용하여 int 변수에 저장

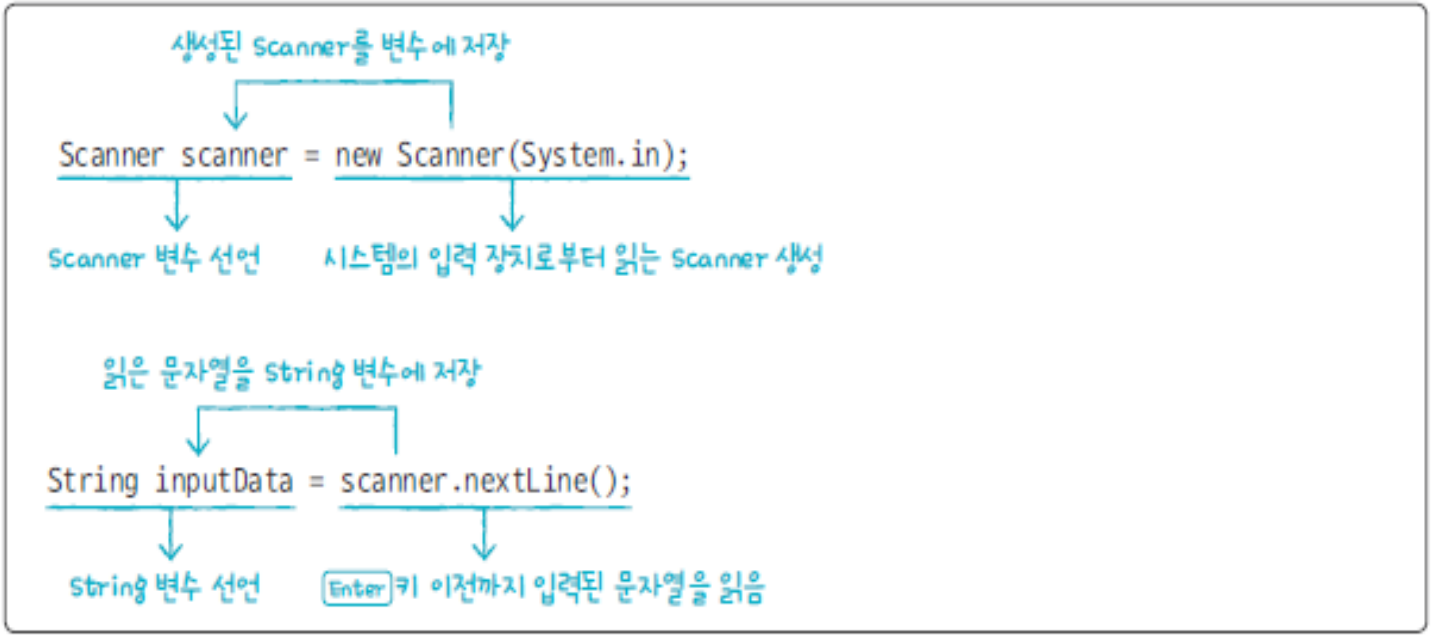


숫자	알파벳				기능키	방향키
0 = 48	A = 65	N = 78	a = 97	n = 110	BACK SPACE = 8	← = 37
1 = 49	B = 66	O = 79	b = 98	o = 111	TAB = 9	↑ = 38
2 = 50	C = 67	P = 80	c = 99	p = 112	ENTER = [CR=13, LF=10]	→ = 39
3 = 51	D = 68	Q = 81	d = 100	q = 113	SHIFT = 16	↓ = 40
4 = 52	E = 69	R = 82	e = 101	r = 114	CONTROL = 17	
5 = 53	F = 70	S = 83	f = 102	s = 115	ALT = 18	
6 = 54	G = 71	T = 84	g = 103	t = 116	ESC = 27	
7 = 55	H = 72	U = 85	h = 104	u = 117	SPACE = 32	
8 = 56	I = 73	V = 86	i = 105	v = 118	PAGEUP = 33	
9 = 57	J = 74	W = 87	j = 106	w = 119	PAGEDN = 34	
	K = 75	X = 88	k = 107	x = 120		
	L = 76	Y = 89	l = 108	y = 121		
	M = 77	Z = 90	m = 109	z = 122		

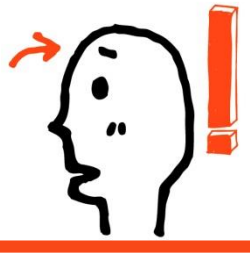
시스템 표준 입 출력 장치의 이해



- System.in.read()의 단점
  - 2개 이상 키가 조합된 한글 읽을 수 없음
  - 키보드로 입력된 내용을 통문자열로 읽을 수 없음
- Scanner 로 해결
  - 자바가 제공하는 Scanner 클래스를 이용하면 입력된 통문자열을 읽을 수 있음

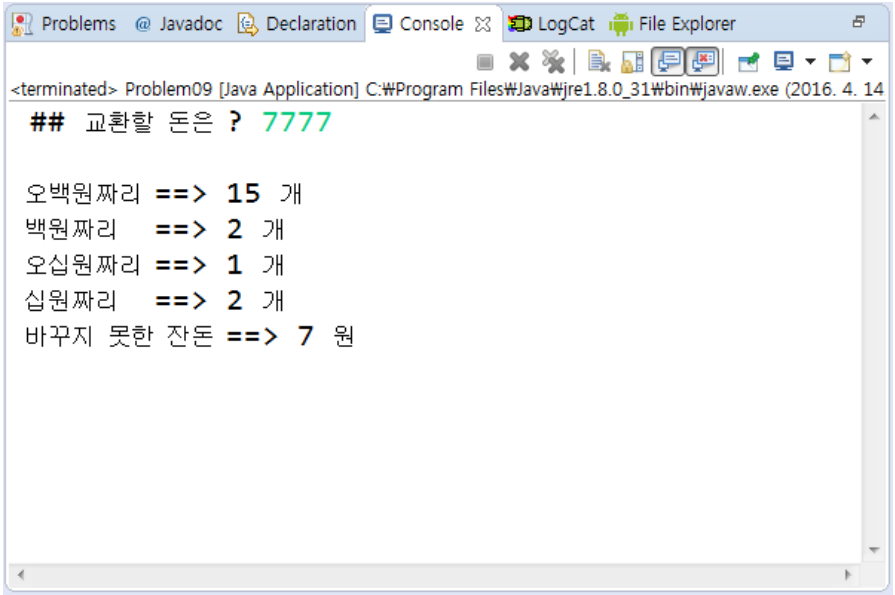


동전교환 프로그램

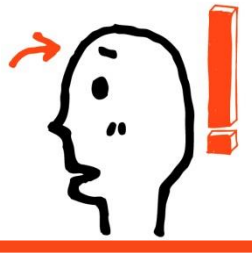


입력된 액수만큼 500원, 100원, 50원, 10원짜리 동전으로 교환해 주는 프로그램을 작성해보자

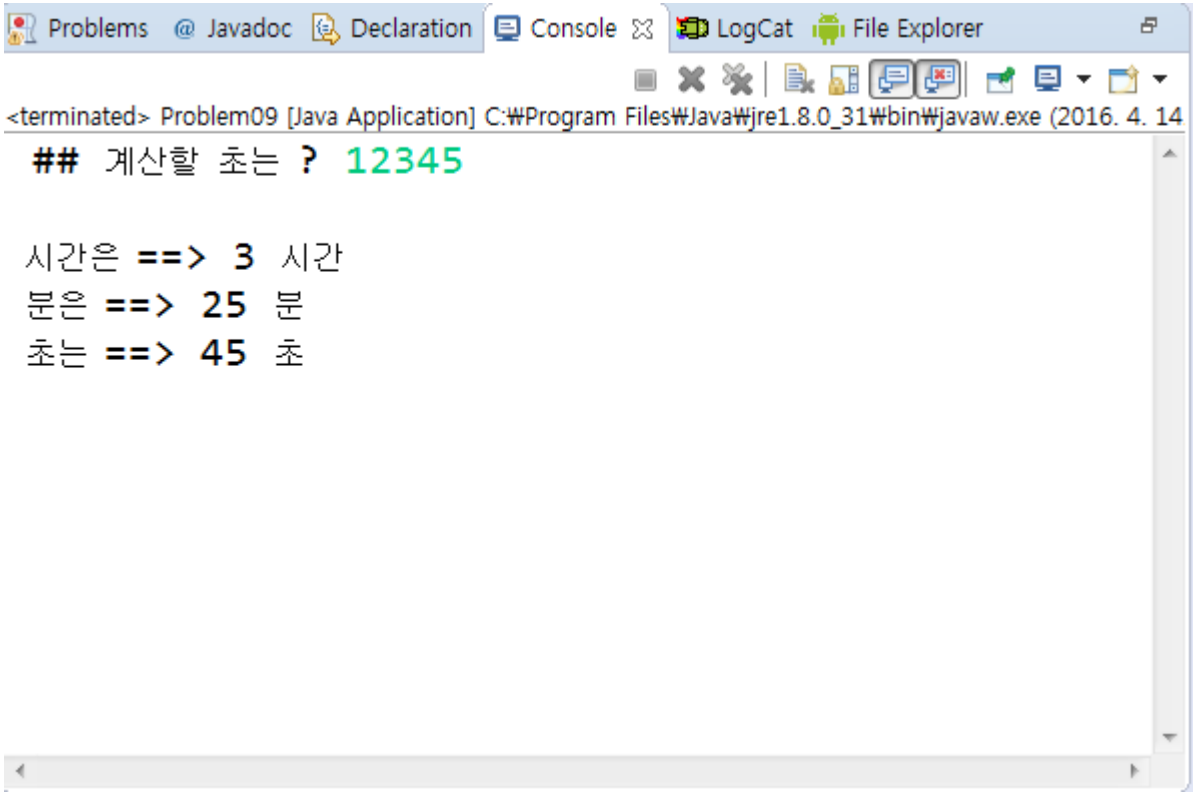
1. 동전의 총 개수는 최소화한다.
2. 고액의 동전을 우선적으로 교환해준다.



시,분,초 프로그램



- 다음과 같이 초를 입력받으면 시, 분, 초로 분할해서 출력하는 프로그램을 작성하시오



THANK YOU

실무에서 알아야 할 기술은 따로 있다! 자바를 다루는 기술