# Java의 정석

Chapter 03~04

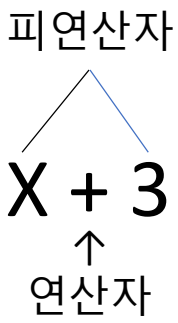
# 연산자

# 연산자

## □ 연산자와 피 연산자

-**연산자** : 연산을 수행하는 기호 (+, -, * , /) =>

-**피 연산자** : 연산자의 작업 대상



# 연산자의 종류

- 단항 연산자 : + - ++ -- !
- 이항 연산자
  - 산술 : + - * / %
  - 비교 : < > >= <= == =!
  - 논리 : && || & |
- 삼항 연산자 : ? :
- 대입 연산자 : =

# 우선순위와 결합규칙

□ **괄호> 산술 > 비교 > 논리 > 대입**

ex) X > 3 && X < 5

-> 논리 연산자 '&&' 보다 비교 연산자 '<, > ' 가 먼저 수행 된다

□ 단항 > 이항 > 삼항

□ 연산의 진행 방향은 왼쪽에서 오른쪽

□ 단항 연산자와 대입 연산자의 경우는 오른쪽에서 왼쪽

ex) result = x + y + 3

-> 우변의 산술 연산자는 왼쪽에서 오른쪽으로 연산한 후 좌변의 result에 값을 저장

# 대입 연산자

x의 값이 5일때,
4 * x + 3
→23
(결과값은 있지만 쓰이진 않는 값)

Y = 4 * x + 3
Y = 23

```
y = 4 * x +3;
System.out.println(y);
```

# 단항 연산자

## □ 증감 연산자 ++ --

- 전위형 : 값이 참조되기 **전에** 증가시킨다
**ex) j = ++i;**

- 후위형 : 값이 참조된 **후에** 증가시킨다.
**ex) j = i++;**

# 나머지 연산자 %

- 나누기한 나머지를 반환하는 연산자
- 주로 홀수, 짝수 등 배수검사에 사용

ex) int x = 10 % 8

```
System.out.println(-10%8);
System.out.println(-10%-8);
System.out.println(10%8);
```

-2
-2
2

# 비교 연산자

## □ 비교 연산자 : < > <= >= == !=

- 피연산자를 같은 타입으로 변환한 후에 비교한다.
- 결과 값은 true 또는 false
- Boolean을 제외한 나머지 자료형에서 사용가능 (참조형X)
- ==, !=은 참조형을 포함한 모든 자료형에서 사용 할 수 있다.

| 수 식 | 연 산 결 과 |
|---|---|
| x > y | x가 y보다 클 때 true, 그 외에는 false |
| x < y | x가 y보다 작을 때 true, 그 외에는 false |
| x >= y | x가 y보다 크거나 같을 때 true, 그 외에는 false |
| x <= y | x가 y보다 작거나 같을 때 true, 그 외에는 false |
| x == y | x와 y가 같을 때 true, 그 외에는 false |
| x != y | x와 y가 다를 때 true, 그 외에는 false |

[표3-11] 비교연산자의 연산결과

# 논리 연산자

- 피연산자는 반드시 boolean형이며 연산결과도 boolean이다
- 우선순위 : && > ||

▶ OR연산자(||) : 피연산자중 한쪽이 true이면 true
▶ AND연산자 (&&) : 피연산자 양 쪽이 true이면 true

| x | y | x \|\| y | x && y |
|---|---|---|---|
| true | true | true | true |
| true | false | true | false |
| false | true | true | false |
| false | false | false | false |

# 논리연산자

Int i = 8

i > 5 && i < 9

i > 9 || i < 0

# 비트연산자

## □ 비트연산자 & | ^ ~ >> <<

- 피연사자를 비트단위로 논리 연산한다.
- 실수는 허용하지 않으며, 정수(문자 포함)만 허용한다.

▶ OR연산자(|) : 피연산자 중 어느 한 쪽이 1이면 1이다.

▶ AND연산자(&) : 피연산자 양 쪽 모두 1이면 1이다.

▶ XOR연산자(^) : 피연산자가 서로 다를 때 1이다.

# 비트 연산자

| 식 | 2진수 | 10진수 |
|---|---|---|
| 3 \| 5 = 7 | 0 0 0 0 0 0 1 1 | 3 |
| | \| ) 0 0 0 0 0 1 0 1 | 5 |
| | 0 0 0 0 0 1 1 1 | 7 |
| 3 & 5 = 1 | 0 0 0 0 0 0 1 1 | 3 |
| | & ) 0 0 0 0 0 1 0 1 | 5 |
| | 0 0 0 0 0 0 0 1 | 1 |
| 3 ^ 5 = 6 | 0 0 0 0 0 0 1 1 | 3 |
| | ^ ) 0 0 0 0 0 1 0 1 | 5 |
| | 0 0 0 0 0 1 1 0 | 6 |

**[표3-14]** 비트연산자의 연산결과

# 비트 연산자

▶ 쉬프트 연산자( >> <<) : 2진수의 각 자리를 오른쪽(왼쪽)으로 이동

- 10진수 8의 2진수를 왼쪽으로 2자리 이동 ( 8 << 2)

| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

# 조건 연산자

## □ 조건 연산자 ? :

**(조건식) ? 식1 : 식2**

- 조건식의 평가결과가 true이면 식1이, false이면 식2가 연산결과가 된다.

```
int x = 5, y = 3, result;
Result = (x > y) ? x : y  // 조건식이 참이므로 연산결과는 식1
```

# 조건문

# If문

## ☐ if, if – else, if – else if

```
if(조건식) {
    // 조건식의 결과가 true일 때 수행될 문장들
}


if(조건식) {
    // 조건식의 결과가 true일 때 수행될 문장들
} else {
    // 조건식의 결과가 false일 때 수행될 문장들
}

if(조건식1) {
    // 조건식1의 결과가 true일 때 수행될 문장들
} else if(조건식2) {
    // 조건식2의 결과가 true일 때 수행될 문장들
    // (조건식1의 결과는 false)
} else if(조건식3) {
    // 조건식3의 결과가 true일 때 수행될 문장들
    // (조건식1과 조건식2의 결과는 false)
} else {
    // 모든 조건식의 결과가 false일 때 수행될 문장들
}
```

# 중첩 if문

- If문의 블록 내에는 또 다른 if문을 넣을 수 있다.
- 횟수에는 거의 제한이 없다.

```
if (조건식1) {
      // 조건식1의 연산결과가 true일 때 수행될 문장들을 적는다.
      if (조건식2) {
            // 조건식1과 조건식2가 모두 true일 때 수행될 문장들
      } else {
            // 조건식1이 true이고, 조건식2가 false일 때 수행되는 문장들
      }
} else {
      // 조건식1이 false일 때 수행되는 문장들
}
```

# 중첩 if문

```
if (score >= 90)     {        // score가 90점 보다 같거나 크면 A학점(grade)
    grade = "A";

    if ( score >= 98) {       // 90점 이상 중에서도 98점 이상은 A+
            grade += "+";     // grade = grade + "+";
    } else if ( score < 94) {
            grade += "-";
    }
} else if  (score >= 80){     // score가 80점 보다 같거나 크면 B학점(grade)
    grade = "B";

    if ( score >= 88) {
        grade += "+";
    } else if ( score < 84) {
        grade += "-";
    }
} else {                      // 나머지는 C학점(grade)
    grade = "C";
}
```

# Switch문

- 조건식을 계산한다
- > 계산결과와 일치하는 case문으로 이동
- > 문장 수행
- > Break문을 만나면 문장을 빠져나간다.
- > 일치하는 case문 값이 없을 경우 default로 이동한다.

```
switch (조건식) {
     case 값1 :
          // 조건식의 결과가 값1과 같을 경우 수행될 문장들
          //...
          break;
     case 값2 :
          // 조건식의 결과가 값2와 같을 경우 수행될 문장들
          //...
          break;
     //...
     default :
          // 조건식의 결과와 일치하는 case문이 없을 때 수행될 문장들
          //...
}
```

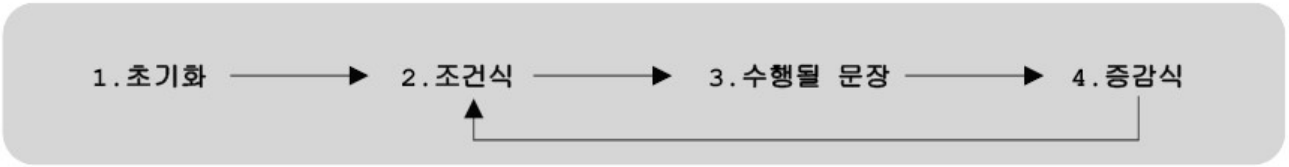
※switch문의 제약조건
- 조건식 결과는 정수 또는 문자열이어야 한다.
- Case문의 값은 정수 상수만 가능하며, 중복되지 않아야 한다.

# 반복문(for, while)

# for문

```
for (초기화;조건식;증감식) {
     // 조건식이 true일 때 수행될 문장들을 적는다.
}
```

**[참고]** 반복하려는 문장이 단 하나일 때는 중괄호{}를 생략할 수 있다.




Ex)1부터 10까지의 양수 더하기

```
int sum = 0;

for(int i=1; i<=10; i++) {
    sum += i; // sum = sum + i;
}
```

# while문

- 조건식이 '참인 동안' 블록 내의 문장을 반복 수행

```
while (조건식) {
      // 조건식의 연산결과가 true일 때 수행될 문장들을 적는다.
}
```

```
for (int i = 0; i<=10; i++)
{
    System.out.println(i);
}
```

```
int i = 1;

while(i<=10)
{
    System.out.println(i);
    i++;
}
```

# 심화 – lambda

## □ 람다식이란?

- 함수를 간단한 식으로 표현하는 방법
- 익명함수 (메서드의 이름과 반환값 X)
- 함수형 프로그래밍에 적합한 문법적 표현

```
int max(int a, int b) {
    return a > b ? a : b;
}
```

```
int max(int a, int b) -> {
    return a > b ? a : b;
}
```

# 심화 – lambda

▶ For문 사용

```
for (int i = 0; i < 10; i++) { System.out.println(i); }
```

▶ 람다식 사용

```
IntStream.range(0, 10).forEach((int value) -> System.out.println(value));
```

▶ 메소드 참조 사용

```
IntStream.range(0, 10).forEach(System.out::println);
```

# 심화 – lambda

```
avajavajava//TestInterface.java
@FunctionalInterface
public interface TestInterface{
    public int plusAandB(int a, int b);
}

// TestInterfaceImpl.java
public class TestInterfaceImpl implements TestInterface{

    @Override
    public int plusAandB(int a, int b){
        return a + b;
    }
}

// Main.java
public class Main{
    public static void main(String[] args){
        TestInterface t1 = new TestInterfaceImpl();
        System.out.println(t1.plusAandB(3, 4));
    }
```

```
// Main.java
public class Main{
    public static void main(String[] args){
//      TestInterface t1 = new TestInterfaceImpl();
//      System.out.println(t1.plusAandB(3, 4));

//      TestInterface t2 = new TestInterface(){
//
//          @Override
//          public int plusAandB(int a, int b){
//              return a + b;
//          }
//      }
        TestInterface t3 = (a, b) ->{return a + b; };
        System.out.println(t3.plusAandB(3, 4));
    }
```

람다식을 이용해 간결하게 표현

# 심화 - stream

## □스트림이란?

- 데이터를 다루는데 자주 사용되는 메서드들을 정의해 놓은 것.
- 람다를 이용해 코드를 간결하게 바꿀 수 있다.
- 배열과 컬렉션을 함수화 할 수 있다.
- 병렬 처리가 가능하다.

# 심화 - stream

□예제

```
List<String> list = new ArrayList<String>();
list.add("볼펜");
list.add("지우개");
list.add("샤프");
list.add("형광펜");
list.add("커터칼")
```

```
Iterator<String> iter = list.iterator();
while(iter.hasNext()) {
System.out.println(iter.next());
```

이터레이터 사용

```
Stream<String> stream = list.stream();
stream.forEach(s -> System.out.println(s));
```

스트림 사용 (람다식 포함)

# 심화 - stream

## □ Stream VS for loop

▶ 처리속도 : stream < for loop ( 3~15배)

- 컴파일러가 for loop에 최적화 돼 있다. (스트림은 비교적 최근)
- 상황에 따라 적절히 사용

▶ 디버깅 : stream < for loop

- 함수형은 객체형에 비해 비교적 관리가 어렵다.