Chapter 3. 연산자

3.1 연산자와 연산식

연산 : 프로그램에서 데이터를 처리하여 결과를 산출하는 것

연산자 : 연산에서 사용되는 표시나 기호

피연산자 : 연산되는 데이터

연산식 : 연산자와 피연산자를 이용하여 연산의 과정을 기술한 것

- **다양한 연산자**

연산자 종류	연산자	기능 설명
산술	+, - , * , / , %	사칙 연산 및 나머지 계산
부호	+, -	음수와 양수의 부호
대입	=, +\, -=, *=, /= , %=, &= , ^= , |= , <<= >>=, >>>=	우변의 값을 좌변의 변수에 대입
증감	++, --	1만큼 증가/감소
비교	== , != , > , < , >= , <= , instanceof	값의 비교
논리	! , & , | , && , ||	논리적 NOT, AND, OR 연산
조건	(조건식) ? A : B	조건식에 따라 A 또는 B 중 하나 선택
비트	~ , & , | , ^	비트 NOT, AND, OR, XOR 연산
쉬프트	>> , << , >>>	비트를 좌측/우측 으로 밀어서 이동

- **단항, 이항, 삼항 연산자** : 피연산자의 개수에 따라
  - **단항 연산자** : ++x
  - **이항 연산자** : x + y
  - **삼항 연산자** : (sum > 90) ? "A" : "B"

3.2 연산의 방향과 우선순위

- **연산식 예시**

```
X > 0 && Y < 0
```

> **연산 순서 :** > , < , && **연산 방향** : 왼쪽에서 오른쪽으로

```
100 * 2 / 3 % 5
```

> **연산 순서** : * , / , % (세개 모두 같은 우선순위를 가지고 있다.)
>
> > **연산 방향** : 왼쪽에서 오른쪽으로

```
a = b = c = 5
```

> **연산 순서** : c = 5 , b = c , a = b
>
> > **연산 방향** : 오른쪽에서 왼쪽

- **우선순위 정리표**

연산자	연산 방향	우선순위
증감(++, --), 부호(+, -) , 비트(~) , 논리(!)	<<(오른쪽에서 왼쪽)	1
산술(*, / , %)	>>(왼쪽에서 오른쪽)	2
쉬프트(<< , >> , >>>)	>>	3
비교(< , > , <= , >= , instanceof)	>>	4
비교(== , !=)	>>	5
논리(&)	>>	6
논리(^)	>>	7
논리(&&)	>>	8
논리(||)	>>	9
조건(?:)	>>	10
대입(= , += , -= , *= , /= , %=)	>>	11

- **우선순위 정리**

1. 단항, 이항, 삼항 연산자 순으로 우선순위를 가진다.
2. 산술, 비교, 논리, 대입 연산자 순으로 우선순위를 가진다.
3. 단항과 대입 연산자를 제외한 모든 연산의 방향은 왼쪽에서 오른쪽이다.
4. 복잡한 연산식에는 괄호( )를 사용해서 우선순위를 정해준다.

3.3 단항 연산자

: 피연산자가 단 하나뿐인 연산자.

3.3.3) 부호 연산자(+ , -)

: 양수 및 음수를 표시한다. boolean 타입과 char타입을 제외한 나머지 기본 타입에 사용.

- **부호 연산자 사용시 주의할 점**

```
short s = 100;
short result = -s;  // 컴파일 에러!!
```

> 부호 연산자의 산출 타입은 **int 타입**이므로 short 타입 값을 부호 연산하면 int 타입 값으로 바뀌기 때문에 컴파일 에러가 발생한다.

수정한 코드

```
short s = 100;
int result3 = -s;
```

- **부호 연산자 예제 코드**

```
public class SignOperatorExample {
    public static void main(String[] args) {
        int x = -100;
        int result1 = +x;
        int result2 = -x;
        System.out.println("result1 = " + result1);
        System.out.println("reuslt2 = " + result2);

        short s = 100;
        // short result3 = -s;   컴파일 에러!!
        int result3 = -s;
        System.out.println("result3 = " + result3);
    }
}
```

실행 결과

```
result1 = -100
reuslt2 = 100
result3 = -100
```

3.3.2) 증감 연산자(++, --)

: 변수의 값을 1 증가시키거나 감소시키는 연산자. boolean타입을 제외한 모든 기본 타입의 피연산자에 사용 가능.

- **증감 연산자 예제 코드**

```
public class IncreaseDecreaseOperatorExample {
    public static void main(String[] args) {
        int x = 10;
        int y = 10;
        int z;

        x++;    // x = 11
        ++x;    // x = 12
        System.out.println("x = " + x + '\n');


        y--;    // y = 9
        --y;    // y = 8
        System.out.println("y = " + y+ '\n');

        z = x++;    // x = 13, z = 12
        System.out.println("z = " + z);
        System.out.println("x = " + x+ '\n');

        z = ++x;    // x = 14, z = 14
        System.out.println("z = " + z);
        System.out.println("x = " + x+ '\n');

        z = ++x + y++;  // x = 15, y = 9 , z = 23
        System.out.println("z = " + z);
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}
```

실행 결과

```
x = 12

y = 8

z = 12
x = 13

z = 14
x = 14

z = 23
x = 15
y = 9
```

> ++ 피연산자 는 다른 연산을 수행하기 전에 피연산자의 값을 1 증가시키고
>
> 피연산자++ 은 다른 연산을 수행한 후에 피연산자의 값을 1 증가시킨다.

3.3.3) 논리 부정 연산자(!)

: true를 false로, false를 true로 변경한다. boolean 타입에만 사용

- **논리 부정 연산자 예제 코드**

```java
public class DenyLogicOperatorExample {
    public static void main(String[] args) {
        boolean play = true;
        System.out.println(play);

        play = !play;
        System.out.println(play);

        play = !play;
        System.out.println(play);
    }
}
```

실행 결과

```
true
false
true
```

> 조건문과 제어문에서 사용되어 조건식의 값을 부정하도록 해서 **실행 흐름을 제어**할 때 주로 사용된다.

3.3.4) 비트 반전 연산자(~)

: 정수 타입(byte, short, int, long) 의 피연산자에만 사용되며, 피연산자를 2진수로 변환한 뒤 0을 1로, 1은 0으로 반전한다. 그러므로 부호가 반대인 새로운 값이 산출된다.

- **비트 반전 연산자 예시**

```
byte v1 = 10;
byte v2 = ~v1;  // 컴파일 에러
```

> 비트 반전을 할 때 피연산자는 연산을 수행하기 전에 **int 타입으로 변환**되므로 컴파일 에러가 발생한다.

수정한 코드

```
byte v1 = 10;
int v2 = ~v1;
```

- **Integer.toBinaryString( ) 메소드**

  : 정수값을 총 32비트의 이진 문자열로 리턴하는 메소드이다. 앞의 비트가 0이면 0은 생략되고 나머지 문자열만 리턴한다.

  - **Integer.toBinaryString( ) 메소드 사용한 추가 메소드**

```
public class BinaryStringExample {
    public static String toBinaryString(int value) {
        String str = Integer.toBinaryString(value);
        // 문자 수가 32보다 작으면 앞에 0을 붙이도록 한다.
        while(str.length() < 32){
            str = "0" + str;
        }
        return str;
    }
}
```

> Integer.toBinaryString( ) 메소드는 앞의 비트가 모두 0이면 0은 생략되고 나머지 문자열만 리턴하기 때문에 총 32개의 문자열을 모두 얻기 위해서는 위와 같은 메소드가 필요하다.

- **비트 반전 연산자 예제 코드**

```
public class BitReverseOperatorExample {
    public static void main(String[] args) {
        int v1 = 10;
        int v2 = ~v1;
        int v3 = ~v1 + 1;
        System.out.println(toBinaryString(v1) + "(십진수 : " + v1 + " )" );
        System.out.println(toBinaryString(v2) + "(십진수 : " + v2 + " )" );
        System.out.println(toBinaryString(v3) + "(십진수 : " + v3 + " )" );
```

```
        int v4 = -10;
        int v5 = ~v4;
        int v6 = ~v4 + 1;
        System.out.println(toBinaryString(v4) + "(십진수 : " + v4 + " )" );
        System.out.println(toBinaryString(v5) + "(십진수 : " + v5 + " )" );
        System.out.println(toBinaryString(v6) + "(십진수 : " + v6 + " )" );
    }

    public static String toBinaryString(int value){
        String str = Integer.toBinaryString(value);
        while(str.length()<32){
            str = "0" + str;
        }
        return str;
    }
}
```

실행 결과

```
00000000000000000000000000001010(십진수 : 10 )
11111111111111111111111111110101(십진수 : -11 )
11111111111111111111111111110110(십진수 : -10 )
11111111111111111111111111110110(십진수 : -10 )
00000000000000000000000000001001(십진수 : 9 )
00000000000000000000000000001010(십진수 : 10 )
```

3.4 이항 연산자

: 피연산자가 두 개인 연산자

3.4.1) 산술 연산자(+ , - , * , / , %)

- **% 연산**

```
int result = num % 3;
```

> **result** = 0, 1, 2 중이 한 값
>
> **num % 3** = num을 3으로 나눈 나머지

- **산술 연산자 규칙**

1. 피연산자들이 모두 **정수 타입**이고, int 타입(4 byte)보다 크기가 작은 타입일 경우 모두 **int 타입으로 변환 후**, 연산을 수행한다. 따라서 연산의 **산출 타입은 int**이다.

   ex) byte + byte --> int + int = int

2. long 타입이 있을 경우 모두 **long 타입으로 변환 후**, 연산 수행. 따라서 연산의 **산출 타입은 long** 이다.

   ex) ing + long --> long + long = long

3. 피연산자 중 실수 타입(float, double)이 있을 경우, **크기가 큰 실수 타입으로 변환 후**, 연산 수행. 따라서 연산의 **산출 타입은 실수 타입**이다.

   ex) int + double --> double + double = double

- **산술 에러 코드 예시**

  1. byte + byte

```
byte byte1 = 1;
byte byte2 = 1;
byte byte3 = byte1 + byte2;     // 컴파일 에러!!
```

> 산술 연산시 int 타입으로 산출이 되기 때문에 에러가 발생한다.

2. int1 / int2

```
int int1 = 10;
int int2 = 4;
int result2 = int1 / int2;
double result3 = int1 / int2;
```

result2, 3 출력 결과

```
2
2.0
```

> int1 / int 2 의 결과가 2 이므로 double로 자동 타입 변환을 해도 2.5가 아닌 2.0이 나오게 된다.

```
**수정한 코드**

```java
double result3 = (int1 * 1.0) / int2;
System.out.println(result3);

result3 = (double) int1 / int2;
System.out.println(result3);

result3 = int1 / (double) int2;
System.out.println(result3);
```

실행 결과

```
```
```

```
2.5
2.5
2.5
```

- **산술 연산자 예제 코드1**

```
public class ArithmeticOperatorExample2 {
    public static void main(String[] args) {
        int v1 = 5;
        int v2 = 2;

        int result1 = v1 + v2;
        System.out.println("result1 = " + result1);

        int result2 = v1 - v2;
        System.out.println("result2 = " + result2);

        int result3 = v1 * v2;
        System.out.println("result3 = " + result3);

        int result4 = v1 / v2;
        System.out.println("result4 = " + result4);

        int result5 = v1 % v2;
        System.out.println("result5 = " + result5);

        double result6 = (double) v1/v2;
        System.out.println("result6 = " + result6);
    }
}
```

실행 결과

```
result1 = 7
result2 = 3
result3 = 10
result4 = 2
result5 = 1
result6 = 2.5
```

- **산술 연산자 예제 코드2**

```java
public class CharOperationExample {
    public static void main(String[] args) {
        char c1 = 'A' + 1;
        char c2 = 'A';
        // char c3 = c2 + 1;    컴파일 에러
        System.out.println("c1 : " + c1);
        System.out.println("c2 : " + c2);
        // System.out.println("c3 : " + c3);
    }
}
```

실행 결과

```
c1 : B
c2 : A
```

> **char c3 = c2 + 1 이 컴파일 에러나는 이유**는 산술 연산시 int 타입으로 자동 변환 되기 때문에 컴파일 에러가 발생한다. **하지만** char c1 = 'A' + 1 가 에러나지 않는 이유는 리터럴 끼리 산술 연살 했을 때는 타입 변환이 일어나지 않기 때문이다.
>
> 컴파일 에러가 나지 않게 하려면 **char c3 = (char) (c2 + 1)** 로 바꿔야한다.

오버플로우 탐지 (쓰레기 값 발생 점검)

: 산술 연산을 할 때 주의할 점은 연산 후의 산출값이 충분히 표현 가능한 범위인지 살펴야함.

- **오버플로우 예제 코드**

```java
public class OverflowExample {
    public static void main(String[] args) {
        int x = 1000000;
        int y = 1000000;
        int z = x * y;
        System.out.println(z);
    }
}
```

실행 결과

```
-727379968      // 쓰레기 값 발생
```

- **오버플로우 해결 코드**

```
public class OverflowSolutionExample {
    public static void main(String[] args) {
        long x = 1000000;
        long y = 1000000;
        long z = x * y;
        System.out.println(z);
    }
}
```

실행 결과

```
1000000000000
```

> 위에서와 같이 코드에서 피연산자의 값을 직접 리터럴로 주는 경우는 드물기 때문에 **메소드를 이용**하여 산술 연산을 하기 전에 **피연산자들의 값을 조사**해 오버플로우를 탑지하는 것이 좋다.

- **산술 연산 전에 오버플로우를 탐지**

```
public class CheckOverflowExample {
    public static void main(String[] args) {
        try{
            int result = safeAdd(2000000000, 2000000000);
            System.out.println(result);
        }catch (ArithmeticException e){
            System.out.println("오버플로우가 발생하여 정확하게 계산할 수 없음");
        }
    }

    public static int safeAdd(int left, int right){
        if((right>0)){
            // 왼쪽 값 > int 타입 최대 값 - 오른쪽 값
            if(left > (Integer.MAX_VALUE - right)){
                throw new ArithmeticException("오버플로우 발생");
            }
        } else{
            // 왼쪽 값 < int 타입 최대 값 - 오른쪽 값
            if(left<(Integer.MIN_VALUE - right)){
                throw new ArithmeticException("오버플로우 발생");
            }
        }
        return left + right;
    }
}
```

실행 결과

```
오버플로우가 발생하여 정확하게 계산할 수 없음
```

> safeAdd( ) 메소드가 두 개의 매개 값을 더해도 안전한 경우에만 더한 결과를 리턴한다.

정확한 계산은 정수 사용

: 정확하게 계산해야 할 때는 부동소수점(실수) 타입을 사용하지 않는 것이 좋다.

- **예제 코드**

```
public class AccuracyExample {
    public static void main(String[] args) {
        int apple = 1;
        double pieceUnit = 0.1;
        int number = 7;

        double result = apple - number * pieceUnit;

        System.out.println("사과 한개에서 ");
        System.out.println("0.7 조각을 빼면, ");
        System.out.println(result + "조각이 남는다.");
    }
}
```

실행 결과

```
사과 한개에서
0.7 조각을 빼면,
0.29999999999999993조각이 남는다.
```

> 부동소수점 타입(float, double) 은 0.1을 정확히 표현할 수 없어 근사치로 처리하기 때문에 정확히 0.3이 되지 않는다.

수정한 코드

```
public class AccuracyExample2 {
    public static void main(String[] args) {
        int apple = 1;

        int totalPieces = apple * 10;
        int number = 7;
        int temp = totalPieces - number;

        double result = temp / 10.0;

        System.out.println("사과 한개에서 ");
        System.out.println("0.7 조각을 빼면, ");
        System.out.println(result + "조각이 남는다.");
    }
}
```

실행 결과

```
사과 한개에서
0.7 조각을 빼면,
0.3조각이 남는다.
```

NaN과 Infinity 연산

- **/ 또는 % 연산자 와 0**

```
5 / 0   // ArithmeticException 예외 발생
5 % 0   // ArithmeticException 예외 발생
```

> 예외가 발생하면 프로그램 실행이 즉시 멈추고 종료된다. 종료되지 않도록 하려면 예외 처리를 해야한다.

- **예외 처리 코드**

```
try{
    // int z = x / y;
    int z = x % y;
    System.out.println("z: " + z);
}catch(ArithmeticException e){
    System.out.println("0으로 나누면 안됨");
}
```

> 자세한 내용은 10장에서 학습

- **Infinity(무한대), NaN(Not a Number)**

```
5 / 0.0     // Infinity
5 % 0.0     // NaN
```

> 0.0 또는 0.0f 로 나누면 예외가 발생하지 않고, / 연산의 결과는 **무한대** 값을 가지며, % 연산의 결과는 **NaN** 을 가진다.
>
> **주의할 점** : 결과가 Infinity 이거나 NaN이 나오면 다음 연산을 수행해서는 안 된다. 왜냐하면 어떤 수와 연산하더라도 Infinity 와 NaN 이 산출되기 때문이다.

- **예제 코드**

```
public class InfinityAndNaNCheckExample {
    public static void main(String[] args) {
        int x = 5;
        double y = 0.0;

        double z1 = x / y;
        double z2 = x % y;
```

```
        System.out.println(Double.isInfinite(z1));
        System.out.println(Double.isNaN(z1));
        System.out.println();

        System.out.println(Double.isInfinite(z2));
        System.out.println(Double.isNaN(z2));
        System.out.println();

        System.out.println(z1 + 2);  // 문제의 코드
    }
}
```

실행 결과

```
true
false

false
true

Infinity
```

> z1 = Infinity 이기 때문에 z1 + 2 를 해도 Infinity 이다. 그러므로 if문을 사용해서 실행 흐름을 변경해야 한다.

````
**값 산출 점검 if문**

```java
if( Double.isInfinite(z) || Double.isNaN(z)){   // false 일 경우에만 계산
    System.out.println("값 산출 불가");
}else{
    System.out.println(z + 2);
}
```
````

입력값의 NaN 검사

: 부동소수점(실수)을 입력받을 때는 반드시 NaN 검사를 해야 한다.

- **NaN 문자열의 문제점**

```
public class InputDataCheckNaNExample {
    public static void main(String[] args) {
        String userInput = "NaN";
        double val = Double.valueOf(userInput); // 입력값을 double 타입으로

        double currentBalance = 10000.0;

        currentBalance += val;  // currentBalance = NaN
        System.out.println(currentBalance);
    }
}
```

실행 결과

```
NaN
```

> "NaN" 문자열을 메소드를 이용하여 double 타입으로 변환하면 NaN이 된다.
>
> 따라서 NaN과 어떠한 수가 연산되면 결과는 결국 **NaN이 산출**되어 데이터가 엉망이 된다.

수정한 코드

```
public class InputDataCheckNaNExample2 {
    public static void main(String[] args) {
        String userInput = "NaN";
        double val = Double.valueOf(userInput);

        double currentBalance = 10000.0;

        if(Double.isNaN(val)){
            System.out.println("NaN이 입력되어 처리할 수 없음");
            val = 0.0;
        }

        currentBalance += val;
        System.out.println(currentBalance);
    }
}
```

실행 결과

```
NaN이 입력되어 처리할 수 없음
10000.0
```

> NaN인지 검사하기 위해 **Double.isNaN( ) 메소드**를 이용한다. 그 후 매개값이 NaN이면 true를 리턴하게되 NaN 대신 0.0이 들어가게 되어 currentBalance는 **원래 값을 유지**하게 된다.

3.4.2) 문자열 연결 연산자(+)

: 문자열을 서로 결합하는 연산자이다.

- **문자열 연결 연산자 예제 코드**

```
public class StringConcatExample {
    public static void main(String[] args) {
        String str1 = "JDK" + 6.0;
        String str2 = str1 + "특징";
        System.out.println(str2);

        String str3 = "JDK" + 3 + 3.0;
        String str4 = 3 + 3.0 + "JDK";
        System.out.println(str3);
        System.out.println(str4);
    }
}
```

실행 결과

```
JDK6.0특징
JDK33.0             // 3 + 3.0 = 33.0 , 문자열로 인식
6.0JDK              // 3 + 3.0 = 6.0  , 숫자로 인식
```

> 문자열과 숫자가 혼합된 + 연산식은 **어떤 것이 먼저 연산되느냐에** 따라 다른 결과가 나오므로 주의해야 한다.

3.4.3) 비교 연산자(< , <= , > , >= , == , !=)

: 대소 또는 동등을 비교해서 boolean 타입인 true/false 산출한다. 주로 실행 흐름을 제어할 때 사용된다.

- **char 타입 대소 비교**

```
('A' < 'B') --> (65 < 66)
```

> 유니코드로 대소를 비교한다.

- **비교 연산자 예제 코드**

```
public class CompareOperatorExample {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 10;
        boolean result1 = (num1 == num2);
        boolean result2 = (num1 != num2);
        boolean result3 = (num1 <= num2);
        System.out.println("result1 = " + result1);
```

```
        System.out.println("result2 = " + result2);
        System.out.println("result3 = " + result3);

        char char1 = 'A';
        char char2 = 'B';
        boolean result4 = (char1 < char2);
        System.out.println("result4 = " + result4);
    }
}
```

실행 결과

```
result1 = true
result2 = false
result3 = true
result4 = true
```

- **피연산자 타입 변환**

  : 비교 연산을 수행하기 전에 타입 변환을 통해 피연산자의 타입을 일치시킨다.

```
'A' == 65    --> true
3 == 3.0     --> true
0.1 == 0.1f  --> false
```

> **'A' == 65** : int 타입이 char 타입보다 크기 때문에 int 타입으로 변환하여 비교한다.
>
> **3 == 3.0** : double 타입이 int 타입보다 크기 때문에 double 타입으로 변환하여 비교한다.
>
> **0.1 == 0.1f** : 부동소수점 타입은 0.1을 정확히 표현할 수 없기 때문에 0.1f 는 0.1 보다 큰 값이 되어 false 가 산출된다.

- **비교 연산자 예제 코드**

```
public class CompareOperatorExample2 {
    public static void main(String[] args) {
        int v2 = 1;
        double v3 = 1.0;
        System.out.println(v2 == v3);   // true

        double v4 = 0.1;
        float v5 = 0.1f;

        System.out.println(v4 == v5);   // false
        System.out.println((float)v4 == v5);    // true
        System.out.println((int)(v4*10) == (int)(v5*10));   // true
    }
}
```

실행 결과

```
true
false
true
true
```

> 부동소수점 타입끼리 비교할 때는 예제 코드와 같이 **강제 타입 변환한 후** 비교 연산을 하거나 **정수로 변환**해서 비교해야 한다.

- **String 타입의 문자열 비교**

  : 대소 연산자를 사용할 수 없고 동등, 비교 연산자는 사용할 수 있으나 문자열이 같은지, 다른지 비교하는 용도로는 사용되지 않는다.

  - **String ' == ' 연산자 비교**

    ```
    String strVar1 = "신민철";
    String strVar2 = "신민철";
    String strvar3 = new String("신민철");
    ```

    > strVar1 과 strVar2 와 비교하면 true가 나오지만 strVar1과 strVar3와 비교하면 false가 나온다. 그 이유는 strVar1 과 strVar2 는 **같은 객체의 번지값을 참조**하지만 strVar1과 strVar3 는 **다른 객체의 번지값을 참조**하기 때문이다. 그러므로 문자열이 같은지 비교하기 위해서는 **equals 메소드**를 사용해야 한다.

- **문자열 비교 예제 코드**

```
public class StringEqualsExample {
    public static void main(String[] args) {
        String strVar1 = "신민철";
        String strVar2 = "신민철";
        String strvar3 = new String("신민철");

        System.out.println(strVar1 == strVar2);
        System.out.println(strVar1 == strvar3);
        System.out.println();
        System.out.println(strVar1.equals(strVar2));
        System.out.println(strVar1.equals(strvar3));
    }
}
```

실행 결과

```
true
false

true
true
```

3.4.4) 논리 연산자(&&, ||, &, |, ^, !)

: 논리곱(&&), 논리합(||), 배타적 논리합(^), 논리 부정(!), 피연산자는 boolean 타입만 가능.

- **&& 와 & 의 차이(|| 와 | 의 차이 동일)**

  && : 앞의 피연산자가 false라면 뒤의 피연산자 평가 X

  & : 두 피연산자 모두 평가

- **논리 연산자 예제 코드**

```
public class LogicalOperatorExample {
    public static void main(String[] args) {
        int charCode = 'A';

        if((charCode>=65) & (charCode<=90)){
            System.out.println("대문자");
        }

        if((charCode>=97) && (charCode<=122)){
            System.out.println("소문자");
        }

        if(!(charCode<48) && !(charCode>57)){
            System.out.println("0~9 숫자");
        }

        int value = 6;

        if((value % 2 == 0) | (value % 3 == 0 )){
            System.out.println("2 또는 3의 배수");
        }

        if((value % 2 == 0) || (value % 3 == 0)){
            System.out.println("2 또는 3의 배수");
        }


    }
}
```

실행 결과

```
대문자
2 또는 3의 배수
2 또는 3의 배수
```

3.4.5) 비트 연산자(&, |, ^, ~, <<, >>, >>>)

: 비트 단위로 연산한다. 정수 타입만 연산을 할 수 있다.

비트 논리 연산자(& , | , ^)

: 피연산자가 boolean 타입일 경우에는 일반 논리 연산자이고, 피연산자가 정수 타입일 경우에는 비트 논리 연산자로 사용된다.

- **비트 논리 연산자 예시**

```
byte num1 = 45;
byte num2 = 25;
byte result = num1 & num2;  // 컴파일 에러!!
int result = num1 & num2;   // 정상 작동
```

> 컴파일 에러가 발생하는 이유는 byte, short, char 타입을 비트 논리 연산하면 그 결과는 int 타입이 되기 때문이다.

- **비트 논리 연산자 예제**

```
public class BitLogicExample {
    public static void main(String[] args) {
        System.out.println("45 & 25 = " + (45 & 25));
        System.out.println("45 | 25 = " + (45 | 25));
        System.out.println("45 ^ 25 = " + (45 ^ 25));
        System.out.println("~45 = " + (~45));
    }
}
```

실행 결과

```
45 & 25 = 9
45 | 25 = 61
45 ^ 25 = 52
~45 = -46
```

비트 이동 연산자 (<< , >> , >>>)

: 정수 데이터의 비트를 좌측 또는 우측으로 밀어서 이동시키는 연산을 수행.

- **<< 예시**

```
int result = 1 << 3;    // result = 8
```

- **>> 예시**

```
int result = -8 >> 3;   // result = -1
```

- **>>> 예시**

```
int result = -8 >>> 3;  // result = 536870911
```

> 맨 오른쪽 3비트는 밀려서 버려지고, 맨 왼쪽에 새로 생기는 3비트는 무조건 0으로 채워진다.

- **비트 이동 연산자 예제**

```
public class BitShiftExample {
    public static void main(String[] args) {
        System.out.println("1 << 3 = " + (1<<3));
        System.out.println("-8 >> 3 = " + (-8>>3));
        System.out.println("-8 >>> 3 = " + (-8 >>> 3));
    }
}
```

실행 결과

```
1 << 3 = 8
-8 >> 3 = -1
-8 >>> 3 = 536870911
```

3.4.6) 대입 연산자(= , += , -= , *= , /= , %= , &= , ^= , |= , <<= , >>= , >>>=)

: 오른쪽 피연산자의 값을 좌측 피연산자인 변수에 저장. 대입 연산자는 모든 연산자들 중에서 가장 낮은 연산 순위를 가지고 있기 때문에.

- **대입 연산자 예제**

```
public class AssignmentOperatorExample {
    public static void main(String[] args) {
        int result = 0;
        result += 10;
        System.out.println("result = " + result);

        result -= 5;
        System.out.println("result = " + result);

        result *= 3;
        System.out.println("result = " + result);

        result /= 5;
        System.out.println("result = " + result);

        result %= 3;
        System.out.println("result = " + result);
```

```
    }
}
```

실행 결과

```
result = 10
result = 5
result = 15
result = 3
result = 0
```

3.5) 삼항 연산자

: 세 개의 피연산자를 필요로 한다.

```
"조건식" ? "값 또는 연산식" : "값 또는 연산식"
피연산자1      피연산자2          피연산3
              True일때         False일때
```

- **삼함 연산자 예제**

```
public class ConditionalOperationExample {
    public static void main(String[] args) {
        int score = 85;
        char grade = (score > 90) ? 'A' : ( (score > 80) ? 'B' : 'C');
        System.out.println(score + "점은 " + grade + "등급입니다.");
    }
}
```

실행 결과

```
85점은 B등급입니다.
```

확인문제

1. 연산자와 연산식에 대한 설명 중 틀린 것은 무엇입니까?
    1. 연산자는 피연산자의 수에 따라 단항, 이항, 삼항 연산자로 구분된다.
    2. 비교 연산자와 논리 연산자의 산출 타입은 boolean(true/false) 이다.
    3. **연산식은 하나 이상의 값을 산출할 수도 있다. (X , 연산식은 하나의 값만 산출한다.)**
    4. 하나의 값이 올 수 있는 자리라면 연산식도 올 수 있다.

2. 다음 코드를 실행했을 때 출력 결과는 무엇입니까?

```java
public class Exercise02 {
    public static void main(String[] args) {
        int x = 10;
        int y = 20;
        int z= (++x) + (y--);
        System.out.println(z);
    }
}
```

출력 결과

```
31
```

3. 다음 코드를 실행했을 때 출력 결과는 무엇입니까?

```java
public class Exercise03 {
    public static void main(String[] args) {
        int score = 85;
        String result = (!(score>90) ? "가":"나");
        System.out.println(result);
    }
}
```

실행 결과

```
가
```

4. 534 자루의 연필을 30명의 학생들에게 똑같은 개수로 나누어 줄 때 학생당 몇 개를 가질 수 있고, 최종적으로 몇 개가 남는지를 구하는 코드이다. 괄호에 들어갈 알맞은 코드를 작성하세요.

```java
public class Exercise04 {
    public static void main(String[] args) {
        int pencils = 534;
        int student = 30;

        // 학생 한 명이 가지는 연필 수
        int pencilsPerStudent = pencils / student;  // 빈칸
        System.out.println(pencilsPerStudent);

        // 남은 연필 수
        int pencilsLeft = pencils % student;        // 빈칸
        System.out.println(pencilsLeft);
    }
}
```

실행 결과

```
17
24
```

5. 다음은 십의 자리 이하를 버리는 코드입니다. 변수 value의 값이 356이라면 300이 나올 수 있도록 빈칸에 알맞은 코드를 작성하세요(산술 연산자만 사용하세요).

```
public class Exercise05 {
    public static void main(String[] args) {
        int value = 356;
        System.out.println(value - value%100);  // 빈칸
    }
}
```

실행 결과

```
300
```

6. 다음 코드는 사다리꼴의 넓이를 구하는 코드입니다. 정확히 소수자릿수가 나올 수 있도록 빈칸에 알맞은 코드를 작성하세요.

```
public class Exercise06 {
    public static void main(String[] args) {
        int lengthTop = 5;
        int lengthBottom = 10;
        int height = 7;
        double area = (lengthTop + lengthBottom) * height * 1.0 / 2; // 빈칸
        System.out.println(area);
    }
}
```

실행 결과

```
52.5
```

7. 다음 코드는 비교 연산자와 논리 연산자의 복합 연산식입니다. 연산식의 출력 결과를 나타내시오.

```java
public class Exercise07 {
    public static void main(String[] args) {
        int x = 10;
        int y = 5;

        System.out.println((x>7) && (y<=5));
        System.out.println((x%3 == 2) || (y%2 != 1));
    }
}
```

실행 결과

```
true
false
```

8. 다음은 % 연산을 수행한 결과값에 10을 더하는 코드입니다. NaN 값을 검사해서 올바른 결과가 출력될 수 있도록 빈칸에 들어갈 NaN을 검사하는 코드를 작성하세요

```java
public class Exercise08 {
    public static void main(String[] args) {
        double x = 5.0;
        double y = 0.0;

        double z = x % y;

        if((Double.isNaN(z))){  // 빈칸
            System.out.println("0.0으로 나눌 수 없습니다.");
        }else{
            double result = z + 10;
            System.out.println("결과 : " + result);
        }
    }
}
```

실행 결과

```
0.0으로 나눌 수 없습니다.
```