

제 4 장

연산자와 연산식

연산자

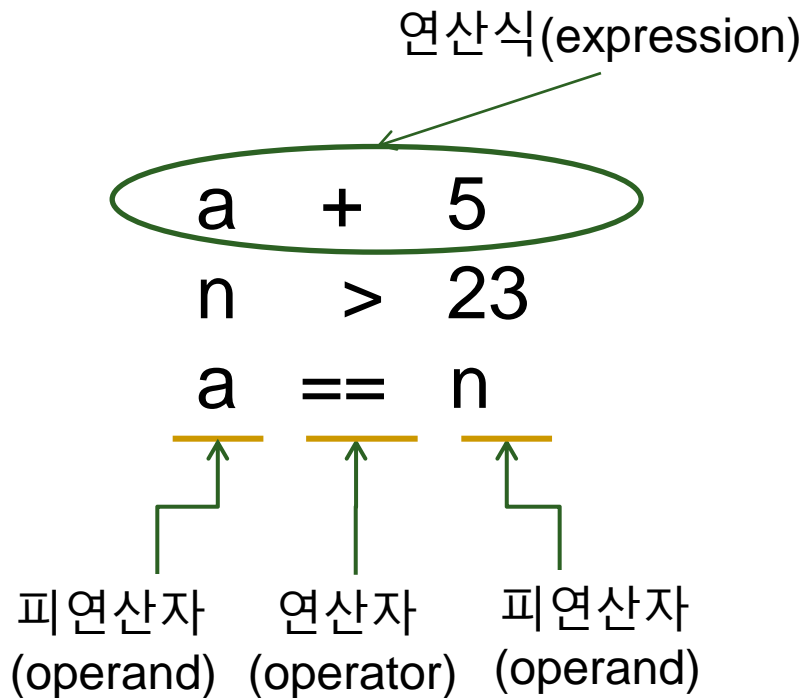
- 연산
 - 변수들의 가공 및 조작을 통해 목적하는 결과를 도출하기 위한 모든 동작
- 연산자(operator)
 - 연산을 의미하는 특정 기호
 - 연산자와 더불어 연산을 위해 필요한 변수를 피연산자(operand)라고 한다
 - 단항연산자 (unary operator) : 하나의 피연산자를 필요
 - 이항연산자(binary operator) : 2개의 피연산자를 필요
 - 삼항연산자(ternary operator) : 3개의 피연산자를 필요

피연산자 수에 따른 연산자의 분류

연산자명			연산자	비고
단항연산자	부호연산자		+, -	boolean, 문자형 제외
	증감연산자		++expr, --expr,	선 증감
			expr++, expr--	후 증감
	논리 NOT(부정) 연산자		!	boolean 타입에 사용
	비트 NOT(반전) 연산자		~	정수타입 피연산자
이항연산자	산술연산자		+, -, *, /, %	자동 타입 변환
	문자열 연결 연산자		+	문자열 2개를 연결
	비교 연산자		==, !=, <, >, <=, >=, instanceof	비교결과가 참이면 true, 거짓이면 false
	논리연산자		&&,	논리AND, 논리OR
	비트연산자	비트논리연산자	&, , ^	비트 AND, 비트 OR, 비트 XOR
		비트이동연산자	<<, >>, >>>	비트 이동
삼항연산자			?:	조건값에 따른 연산값 할당
대입연산자	단순대입연산자		=	우변의 값을 좌변 변수에 저장
	복합대입연산자		+= -= *= /= %= &= ^= = <<= >>= >>>=	우변의 값을 좌변의 변수와 표시된 연산 후에 좌변의 변수에 저장

연산식과 명령문

- 연산식 : 연산자와 피연산자 및 메소드 호출 등을 조합하여 목적하는 결과를 반환하는 표현식



연산자의 종류	연산자
증감	++ --
산술	+ - * / %
시프트	>> << >>>
비교	> < >= <= == !=
비트	& ^ ~
논리	&& ! ^
조건	? :
대입	= *= /= += -= &= ^= = <<= >>= >>>=

■ 복합식(compound expression)

- 2개 이상의 연산자와 피연산자 등을 사용하여 이루어지는 연산식
- 예: $x + y * z$, $(a + b) * 10$
- 연산자 우선순위가 필요
- 결합 우선순위 (좌측우선, 우측우선)

■ 명령문(statement)

- 연산식(혹은 복합식)으로 구성되는 온전한 실행단위로 ‘;’으로 끝나며, 어떤 변수에 동작 결과를 저장하는 상황을 초래
- 선언 명령문 (declaration) : 보통 변수 선언시 사용
- 프로그램 흐름제어(flow control) 명령문 : if, while, for, switch, ..
- 블록(block) : 중괄호({, })로 둘러싸인 명령문들의 집합
 - 하나의 명령문처럼 동작

명령문 예

```
// 할당 명령문
aValue = 8933.234;
// 증감 명령문
aValue++;
// 메소드 호출 명령문
System.out.println("Hello World!");
// 객체 생성 명령문
Bicycle myBike = new Bicycle();
```


선언 명령문 예

```
double aValue = 8933.234;
```

블록 예

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { // begin block 1
            System.out.println("Condition is true.");
        } // end block 1
        else { // begin block 2
            System.out.println("Condition is false.");
        } // end block 2
    }
}
```

연산자 우선 순위

<div>높음</div>  <div>낮음</div>	++(postfix) -- (postfix)
	+(양수 부호) -(양수, 음수 부호) ++(prefix) --(prefix) ~ !
	형 변환(type casting)
	* / %
	+(덧셈) -(뺄셈)
	<< >> >>>
	< > <= >= instanceof
	== !=
	&(비트 AND)
	^(비트 XOR)
	(비트 OR)
	&&(논리 AND)
	(논리 OR)
	? : (조건)
	= += -= *= /= %= &= ^= = <<= >>= >>>=

- 같은 우선순위의 연산자가 2개 이상 존재할 경우 결합 방향을 결정하는 우선순위
 - 좌측 우선 처리
 - 예외) 우측 우선
 - 대입 연산자, --, ++, +, -(양수 음수 부호), !, 형 변환은 오른쪽에서 왼쪽으로 처리
- 괄호는 최우선순위
 - 괄호가 다시 괄호를 포함한 경우는 가장 안쪽의 괄호부터 먼저 처리

연산자 우선순위와 결합방향 우선순위

연산자명		연산자	결합우선순위	연산자우선순위
단항연산자	부호연산자	+, -	← (우측에서 좌측으로)	가장높음 (highest)
	증감연산자	++, --		
	논리 NOT(부정) 연산자	!		
	비트 NOT(반전) 연산자	~		
이항연산자	산술연산자 (문자열연결 연산자 포함)	*, /, %	→ (좌측에서 우측으로)	하향 낮은 우선순위
		+, -,	→	
	비트쉬프트연산자	<<, >>, >>>	→	
	비교 연산자	<, >, <=, >=, instanceof	→	
		==, !=	→	상향 높은 우선순위
	비트 AND 연산자	&	→	
	비트 XOR 연산자	^	→	
	비트 OR 연산자		→	
	논리 AND 연산자	&&,	→	
	논리 OR 연산자		→	
삼항연산자	조건 연산자	?:	→	가장낮음 (lowest)
대입연산자	대입연산자	= += -= *= /= %= &= ^= = <<= >>= >>>=	←	

단항 연산자(Unary Operator)

- 하나의 피연산자만을 필요로 하는 연산자
- 부호 연산자(+, -) : 양수 혹은 음수
- **증감연산자** (++ , --)
 - 변수를 1만큼 증가 혹은 감소
 - prefix(선증감) 혹은 postfix(후증감)
 - 예 :
 - `a++` : 변수 `a`를 연산식에 (증가시키지 않은 상태에서 먼저)적용한 후에 증가
 - `++a` : 변수 `a`를 먼저 증가시키고 연산식에 적용
 - 연산식이 없이 단순히 `a++` 혹은 `++a` 만 있는 경우는 동일한 결과 값
 - `a--`와 `-a` 인 경우에도 동일한 개념을 적용한다. 다만 1만큼 감소

```
class PrePostInc {  
    public static void main(String[] args) {  
        int i = 3;    //지역변수 선언 및 초기화  
  
        i++;          // 후증감이지만 연산식 적용 없음  
  
        System.out.println(i); // prints 4  
        ++i;  
        System.out.println(i); // prints 5  
        System.out.println(++i);    // prints 6  
        System.out.println(i++);    // prints 6  
        System.out.println(i); // prints 7  
    }  
}
```

■ 논리-NOT(!), 비트-NOT(~) 연산자

□ 논리-NOT 연산자

- 논리값 true를 false로, false는 true로 변환
- boolean 타입의 피연산자에만 적용

□ 비트-NOT 연산자

- 정수 타입의(long, int, short, byte) 피연산자에만 적용 가능
- 피연산자 값의 2진수 비트 패턴에서 0은 1로, 1은 0으로 반전
- 연산 후의 타입은 int 타입이 되므로, 결과는 int 타입의 변수에 저장

```
byte b1 = 35;           // 00100011
```

```
byte b2 = ~b1;          // 컴파일 오류 발생 (~b1 = 11111111 11111111 11111111 11011100)
```

```
int b2 = ~b1;           // 정상 컴파일
```

논리 NOT 연산자

a	!a	예제
true	false	!(3 < 5)는 false
false	true	!(3 > 5)는 true

비트 NOT 연산자

byte b1; 이라고 가정

b1	~b1
35(00100011)	11111111 11111111 11111111 11011100

이항 연산자(binary Operator)

- 2개의 피연산자를 필요로 하는 연산자
- 다양한 연산자가 존재 (p.60을 참조)
- 산술 및 문자열 연결 연산자

산술 연산자	의미	예	결과값
+	덧셈 및 문자연결	25.5 + 3.6 System.out.println("This year is " + year);	29.1
-	뺄셈	3 - 5	-2
*	곱셈	2.5 * 4	10.0
/	나눗셈(0 혹은 0.0으로 나누는 경우를 주의)	5/2 (5.0/0.0)	2(NaN or Infinity)
%	모듈로(나머지)	5%2	1

□ / 와 % 연산자의 특이성

■ 정수 연산에서 사용할 경우

- / 은 정수 몫을. %는 정수 나머지

■ 실수 연산에서 사용할 경우

- / 은 실수 연산을 수행하며. %는 몫을 정수로 구했을 때의 실수 나머지
- $5.5/3.3 = 1.6666666$ <----- 나머지 없는 실수 몫 값
- $5.5\%3.3 = 2.2$ <----- 정수 몫 1를 구한 나머지 실수 몫 값($5.5-3.3 = 2.2$)
- $1.6666666*3.3 + 2.2 \neq 5.5$: 주의 !!

■ % 연산은 정수연산에서의 사용을 원칙으로 한다

- 산술 연산자는 피연산자들의 타입이 서로 다를 경우, 그들의 타입을 통일시킨 후 연산을 실행

❑ 타입을 통일시키는 규칙은 다음과 같다

■ 피연산자들이 모두 정수 타입이면서

- ❑ long 타입이 존재 → 모두 long 타입으로 변환한 후 연산하며 결과도 long 타입에 저장
- ❑ 모두 int 타입보다 범위가 작은 경우 → 모두 int 타입으로 변환한 후 연산하며 결과도 int 타입에 저장

■ 피연산자들에 정수 타입과 실수 타입이 혼재하면서

- ❑ 모두 실수 타입인 경우 → 범위가 큰 실수 타입으로 변환한 후 연산하며, 결과도 범위가 큰 실수 타입에 저장

❑ 문자열 연결

```
class ConcatTest {  
    public static void main(String[] args){  
        String firstString = "This is";  
        String secondString = " a concatenated string";  
        String thirdString = firstString + secondString;  
        System.out.println(thirdString);  
    }  
}
```

산술 연산 예제

정수의 몫과 나머지를 이용하여 500초는 몇 시간, 몇 분, 몇 초인가를 구하는 프로그램을 작성하시오.

```
public class ArithmeticOperator {  
    public static void main (String[] args) {  
        final int TIME = 500;  
        int second;  
        int minute;  
        int hour  
        second = TIME % 60;  
        minute = (TIME / 60) % 60;  
        hour = (TIME / 60) / 60;  
        System.out.print(TIME + "초는 ");  
        System.out.print(hour + "시간, ");  
        System.out.print(minute + "분, ");  
        System.out.println(second + "초입니다.");  
    }  
}
```

500초는 0시간, 8분, 20초입니다.

■ 비교 연산자

- 피연산자들 간의 다양한 관계를 true 혹은 false 값으로 도출

비교 연산자	내용	예제	결과
$a < b$	a가 b보다 작으면 true 아니면 false	$3 < 5$	true
$a > b$	a가 b보다 크면 true 아니면 false	$3 > 5$	false
$a \leq b$	a가 b보다 작거나 같으면 true 아니면 false	$1 \leq 0$	false
$a \geq b$	a가 b보다 크거나 같으면 true 아니면 false	$10 \geq 10$	true
$a == b$	a가 b와 같으면 true 아니면 false	$1 == 3$	false
$a != b$	a가 b와 같지 않으면 true 아니면 false	$1 != 3$	true

비교 연산 예제

```
class ComparisonTest {  
    public static void main(String[] args){  
        int value1 = 1;  
        int value2 = 2;  
        if(value1 == value2)  
            System.out.println("value1 == value2");  
        if(value1 != value2)  
            System.out.println("value1 != value2");  
        if(value1 > value2)  
            System.out.println("value1 > value2");  
        if(value1 < value2)  
            System.out.println("value1 < value2");  
        if(value1 <= value2)  
            System.out.println("value1 <= value2");  
    }  
}
```

value1 != value2

value1 < value2

value1 <= value2

■ 논리 (이항)연산자

- 논리-AND(&&)와 논리-OR(||) (논리-NOT은 단항연산자)
- 피연산자들 간의 논리관계를 true 혹은 false 값으로 도출
- AND와 OR 그리고 단항연산자인 NOT(!)이 있다
- AND와 OR 논리연산시에 축약(short-circuiting) 연산 동작 수행

```
class LogicalTest {  
    public static void main(String[] args){  
        int value1 = 1;  
        int value2 = 2;  
        if((value1 == 1) && (value2 == 2))  
            System.out.println("value1 is 1 AND value2 is 2");  
        if((value1 == 1) || (value2 == 1))  
            System.out.println("value1 is 1 OR value2 is 1");  
    }  
}
```

논리 연산 예제

a	b	a b	예제
true	true	true	(3<5) (1==1)은 true
true	false	true	(3<5) (1==2)은 true
false	true	true	(3>5) (1==1)은 true
false	false	false	(3>5) (1==2)은 false

a	b	a && b	예제
true	true	true	(3<5) (1==1)은 true
true	false	false	(3<5) (1==2)은 false
false	true	false	(3>5) (1==1)은 false
false	false	false	(3>5) (1==2)은 false

■ 객체 타입비교 연산자 : instanceof

- 2 객체 간 타입 비교
- 동일한 타입이면 true 아니면 false 값을 도출한다
- 어떤 객체가 어느 클래스의 인스턴스인지, 어느 서브클래스의 인스턴스인지 혹은 특정 인터페이스를 구현한 클래스의 인스턴스인지를 테스트하는 용도로 자주 사용
- null 객체는 어떤 것의 인스턴스도 아니다

```
class InstanceofTest {
    public static void main(String[] args) {
        Parent obj1 = new Parent();
        Parent obj2 = new Child();
        System.out.println("obj1 instanceof Parent: " + (obj1 instanceof Parent));
        System.out.println("obj1 instanceof Child: " + (obj1 instanceof Child));
        System.out.println("obj1 instanceof MyInterface: " + (obj1 instanceof MyInterface));
        System.out.println("obj2 instanceof Parent: " + (obj2 instanceof Parent));
        System.out.println("obj2 instanceof Child: " + (obj2 instanceof Child));
        System.out.println("obj2 instanceof MyInterface: " + (obj2 instanceof MyInterface));
    }
}

class Parent {} // empty class
class Child extends Parent implements MyInterface {} //empty class
interface MyInterface {} // empty interface
```

obj1 instanceof Parent: true
obj1 instanceof Child: false
obj1 instanceof MyInterface: false
obj2 instanceof Parent: true
obj2 instanceof Child: true
obj2 instanceof MyInterface: true

■ 비트 논리 연산자

- 피 연산자의 각 비트들을 대상으로 논리 연산을 실행

비트 논리 연산자	내용
비트-AND $a \& b$	a와 b의 각 비트들의 AND 연산. 두 비트 모두 1일 때만 1이 되며 나머지는 0
비트-OR $a b$	a와 b의 각 비트들의 OR 연산. 두 비트 모두 0일 때만 0이 되며 나머지는 1
비트-XOR $a \wedge b$	a와 b의 각 비트들의 XOR 연산. 두 비트가 서로 다르면 1, 같으면 0 (exclusive OR)
비트-NOT $\sim a$	단항연산자로, a의 각 비트들에 NOT 연산. 1을 0으로, 0을 1로 변환 (1's complement: 1의 보수)

비트 연산자의 사례

$$\begin{array}{r} 01101010 \\ \& 11001101 \\ \hline 01001000 \end{array}$$

모두 1이므로 결과는 1 둘 중 하나라도 0이 되면
결과는 0

$$\begin{array}{r} 01101010 \\ | 11001101 \\ \hline 11101111 \end{array}$$

모두 0이므로 결과는 0 둘 중 하나라도 1이 되면
결과는 1

$$\begin{array}{r} 01101010 \\ \wedge 11001101 \\ \hline 10100111 \end{array}$$

두 비트가 서로 다르므로
결과는 1 두 비트가 모두 같으므로
결과는 0

$$\begin{array}{r} \sim 01101010 \\ \hline 10010101 \end{array}$$

0은 1로 변환 1은 0으로 변환

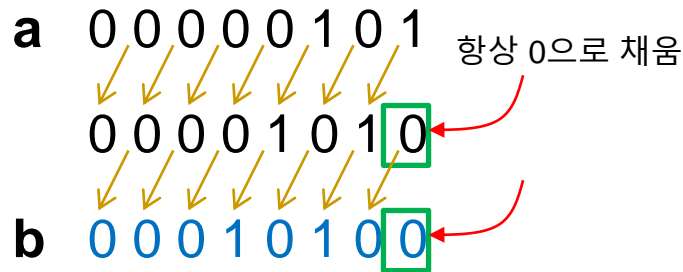
■ 비트 시프트(shift) 연산자

- 피 연산자의 각 비트들을 대상으로 논리 연산을 실행
- byte, short, char 타입의 시프트 연산 시 주의 사항
 - int 타입으로 변환되어 연산이 일어나므로 원하지 않는 결과 발생 가능

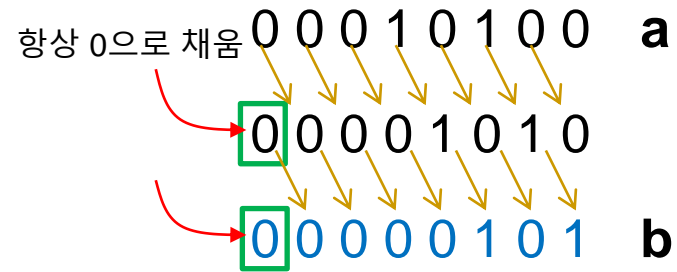
비트시프트 연산자	내용
$a \gg b$	<ul style="list-style-type: none">- a의 각 비트를 오른쪽으로 b 번 시프트한다.- 최상위 비트의 빈자리는 시프트 전의 최상위 비트로 다시 채운다.- 산술적 오른쪽 시프트 (2로 나누는 효과)
$a \ggg b$	<ul style="list-style-type: none">- a의 각 비트를 오른쪽으로 b 번 시프트한다.- 최상위 비트의 빈자리는 0으로 채운다.- 논리적 오른쪽 시프트.
$a \ll b$	<ul style="list-style-type: none">- a의 각 비트를 왼쪽으로 b 번 시프트한다.- 최하위 비트의 빈자리는 0으로 채운다.- 산술적 왼쪽 시프트 (2를 곱하는 효과)- 음수(최상위 비트가 1)는 시프트 결과 최상위 비트가 0인 양수가 되는 오버플로 발생 가능 주의

비트시프트 연산자의 사례

```
byte a = 5; // 5
byte b = (byte)(a << 2); // 20
```

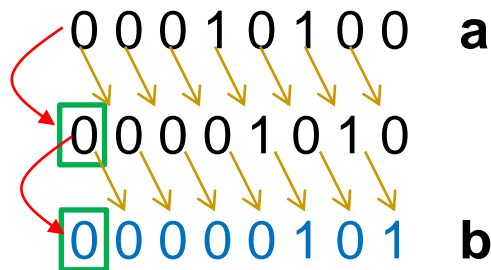


```
byte a = 20; // 20
byte b = (byte)(a >>> 2); // 5
```



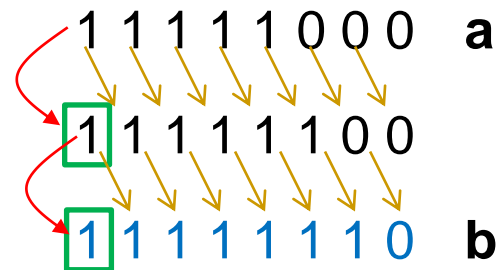
```
byte a = 20; // 20
byte b = (byte)(a >> 2); // 5
```

최상위비트로 채움



```
byte a = (byte)0xf8; // -8
byte b = (byte)(a >> 2); // -2
```

최상위비트로 채움



비트 논리 및 시프트 예제

다음 소스의 실행 결과는 무엇인가?

```
public class BitShiftOperator {  
    public static void main (String[] args) {  
        short a = (short)0x55ff;  
        short b = 0x00ff;  
        // 비트 연산  
        System.out.printf("%x\n", a & b);  
        System.out.printf("%x\n", a | b);  
        System.out.printf("%x\n", a ^ b);  
        System.out.printf("%x\n", ~a);  
        byte c = 20; // 0x14  
        byte d = -8; // 0xf8  
        // 시프트 연산  
        System.out.println(c << 2); // c를 2비트 왼쪽 시프트  
        System.out.println(c >> 2); // c를 2비트 오른쪽 시프트. 0 삽입  
        System.out.println(d >> 2); // d를 2비트 오른쪽 시프트. 1 삽입  
        System.out.printf("%x\n", d >>> 2); // d를 2비트 오른쪽 시프트. 0 삽입  
    }  
}
```

printf("%x\n", ...)는
16진수 형식으로
출력

최상위 비트에 0 삽입
나누기 효과는 나타나
지 않음.

ff
55ff
5500
ffffaa00
80
5
-2
3fffffe

■ 단순 대입 및 복합 대입 연산자

- '=' 를 사용

대입 연산자	내용
$a = b$	b의 값을 a에 대입
$a += b$	$a = a + b$ 과 동일
$a -= b$	$a = a - b$ 과 동일
$a *= b$	$a = a * b$ 과 동일
$a /= b$	$a = a / b$ 과 동일
$a \% = b$	$a = a \% b$ 과 동일
$a \& = b$	$a = a \& b$ 과 동일
$a \wedge = b$	$a = a \wedge b$ 과 동일
$a = b$	$a = a b$ 과 동일
$a << = b$	$a = a << b$ 과 동일
$a >> = b$	$a = a >> b$ 과 동일
$a >>> = b$	$a = a >>> b$ 과 동일

대입연산자의 사례

다음 소스의 실행 결과는 무엇인가?

```
public class UnaryOperator {  
    public static void main(String[] args){  
        int opr = 0;  
        System.out.println(opr++);  
        System.out.println(opr);  
        System.out.println(++opr);  
        System.out.println(opr);  
        System.out.println(opr--);  
        System.out.println(opr);  
        System.out.println(--opr);  
        System.out.println(opr);  
    }  
}
```

0
1
2
2
2
2
1
0
0

삼항연산자(ternary operator)

- 조건연산자 (? :)
- 형식 -- opr1?opr2:opr3
 - 세 개의 피연산자로 구성
 - opr1이 true이면 값은 opr2, false이면 opr3.
 - 조건 연산자를 활용하면 변수에 값을 대입하는 연산 시 조건에 따라 다른 값을 대입할 수가 있다.
 - if-else에 비해 문장이 간결해짐

```
int x = 5;  
int y = 3;  
int s = (x>y)?1:-1; // x가 y보다 크기 때문에 1이 s에 대입된다.
```

다음 소스의 실행 결과는 무엇인가?

```
public class TernaryOperator {  
    public static void main (String[] args) {  
        int a = 3, b = 5;  
        System.out.println("두 수의 차는 " + ((a>b)?(a-b):(b-a)));  
    }  
}
```

두 수의 차는 2