● **비교 연산자**: 주로 조건문과 반복문의 조건식에 사용되며, 연산결과는 오직 true와 false 둘 중 하나다. 비교 연산자는 **이항 연산자**이므로 비교하는 피연산자의 타입이 서로 다를 경우, 자료형의 범위가 큰 쪽으로 자동 형변환하여 피연산자의 타입을 일치시킨 후에 비교한다.

1. **대소비교 연산자 > < <= >=: 기본형** 중에서는 **boolean은 제외한 나머지 자료형**에 다 사용할 수 있지만, **참조형에는 사용할 수 없다**.

2. **등가비교 연산자 == !=: 기본형의 경우 변수에 저장되어 있는 값이 같은지를 알 수 있고, 참조형의 경우 객체의 주소값을 저장하기 때문에 두 개의 피연산자(참조변수)가 같은 객체를 가르키고 있는지**를 알 수 있다. 기본형과 참조형은 서로 형변환이 불가하기 때문에 등가비교 연산자로 비교가 불가능하다.
- **실수형끼리의 비교**: 10.0==10.0f는 true이지만 0.1==0.1f는 false이다. **정수형과 닫리 실수형은 근사값으로 저장되므로 오차가 받생할 수 있기 때문이다**. 10.0f는 오차없이 저장할 수 있는 값이라서 double로 형변환해도 그대로 10.0이 되지만, 0.1f는 저장할 때 2진수로 변환하는 과정에서 오차가 발생한다. double타입의 상수인 0.1도 저장되는 과정에서 오차가 발생하지만, float타입의 리터럴인 0.1f보다 적은 오차로 저장된다. float타입의 값을 double타입으로 형변환하면, 부호와 지수는 달라지지 않고 그저 가수의 빈자리를 0으로 채울 뿐이므로 0.1f를 double타입으로 형변환해도 그 값은 전혀 달라지지 않는다. 즉, float타입의 값을 정밀도가 더 높은 dobule타입으로 형변환했다고 해서 오차가 적어지는 것이 아니라는 이야기다. **float타입의 값과 double타입의 값은 오류 없이 비교하려면 double타입의 값은 float타입으로 형변환한 다음에 비교해야 한다**. 또는 어느 정도의 오차는 무시하고 두 타입의 값을 앞에서 몇 자리만 잘라서 비교할 수도 있다.
- **문자열의 비교**: "=="대신 euqals()라는 메서드를 사용해서 문자열의 내용이 같은지 비교해야 한다. **equals()는 객체가 달라도 내용이 같으면 true은 반환한다**. 만일 **대소문자를 구분하지 않고 비교하고 싶으면 equalsIgnoreCase()**를 사용하면 된다.

● **논리 연산자: 피연산자로 boolean형 또는 boolean형 값은 결과로 하는 조건식만은 허용**한다.

1. AND&& OR||, !

- 하나의 식에 &&와 ||가 같이 포함된 경우, &&가 우선순위가 높지만 &&가 먼저 연산되어야 하는 경우라도 **괄호를 사용해서 우선순위를 명확하게 해 주는 것이 좋다**.
☞ 사용자로부터 입력된 문자가 숫자('0'~'9')인지 확인하는 식:

```
'0' <= ch && ch <= '9'
```

유니코드에서 문자'0'(48)부터 '9(57)'까지 연속적으로 배치되어 있기 때문에 가능한 식이다. 문자 'a'부터 'z'까지(97-122), 문자 **'A'(65)**부터 'Z'(90)까지도 연속적으로 배치되어 있어 같은 방식으로 식을 쓸 수 있다.

```
('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z')
```

```java
public class CheckChar {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        System.out.print("문자 하나를 입력하세요 : ");
        char ch = scan.nextLine().charAt(0);      // String으로 입력 받아 문자로 전환

        if ('0' <= ch && ch <= '9') {
            System.out.println("입력하신 문자는 숫자입니다.");
        } else if (('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z')) {
            System.out.println("입력하신 문자는 알파벳입니다.");
        } else {
            System.out.println("입력하신 문자는 한글입니다.");
        }

    }
}
```

- 효율적인 연산(short circuit evaluation): OR연산'||'의 경우, 두 피연산자 중 어느 한 쪽만 참(true)이어도 전체 연산결과가 '참'이므로 좌측 피연산자가 'true(참)'이면, 우측 피연산자의 값은 평가하지 않는다. AND연산'&&'의 경우, 어느 한쪽만 '거짓(false)'이어도 전체 연산결과가 '거짓'이므로 좌측 피연산자가 '거짓(false)'이면, 우측 피연산자는 평가하지 않는다. 그래서 같은 조건식이라도 피연산자의 위치에 따라서 연산속도가 달라질 수 있는 것이다. OR연산'||'의 경우에는 연산의 결과가 '참'일 확률이 높은 피연산자를 연산자의 왼쪽에 놓아야 더 빠른 연산결과를 얻을 수 있다.
- 논리 부정 연산자 !: 어떤 값에 논리 부정 연산자!를 반복적으로 적용하면, 참과 거짓이 차례대로 반복된다. 이 연산자의 이러한 성질을 이용하면, 한번 누르면 켜지고 다시 누르면 꺼지는 TV의 전원버튼과 같은 토글 버튼(toggle button)을 논리적으로 구현할 수 있다. 논리 부정 연산자!가 **주로 사용되는 곳은 조건문과 반복문의 조건식**이며, 잘 이용하면 조건식이 보다 이용하기 쉬워진다 (OR대신 사용).
ㄴ !!x가 평가되는 과정: 단항연산자는 결합방향이 오른쪽에서 왼쪽이므로 피연산자와 가까운 것부터 먼저 연산된다.

2. **비트 연산자 & | ^ ~ <<>>: 피연산자를 비트단위로 논리 연산**한다. **피연산자로 실수는 허용하지 않고, 정수(문자 포함)만 허용**된다. 피연산자를 이진수로 표현했을 때, 각 자리를 아래의 규칙에 따라 연산 수행한다. int타입(4byte)간의 연산이라 32자리로 표현되며, **비트연산에서도 피연산자의 타입을 일치시키는 '산술 변환'**이 일어날 수 있다.

비트 연산자	규칙	사용처	사용처 예
| (OR비트연산자)	피연산자 중 한쪽의 값이 1이면, 1을 결과로 얻고, 그 외에는 0을 얻음.	주로 특정 비트의 값을 변경할 때 사용.	0xAB | 0xF = 0xAF (피연산자의 마지막 4bit을 F로 변경하기)
& (AND비트연산자)	피연산자 양 쪽이 모두 1이어야만 1을 결과로 얻고, 그 외에는 0을 얻음.	주로 특정 비트의 값을 뽑아낼 때 사용.	0XAB : 0xF = 0xB (피연산자의 마지막 4bit의 값 찾기)
^ (XOR연산자: 배타적 eXclusive OR)	피연산자의 값이 서로 다를 때만(배터직인 경우에만) 1을 결과로 얻고, 같을 때는 0을 얻음.	같은 값을 두고 XOR연산을 수행하면 원래의 값으로 돌아오는 특징이 있어 간단한 암호화에 사용.	0XAB ^ 0XF = 0XA4 OXAB ^ OXF ^ OXF = oXAB
~ (비트전환연산자, 혹은 '1의 보수'연산자)	피연산자를 2진수로 표현했을 때 0은 1로 1은 0으로 바꿈. 부호 있는 타입의 피연산자는 부호가 반대로 변경됨. 즉, 피연산자의 '1의 보수'를 얻을	양의 정수 p가 있을 때, p에 대한 음의 정수를 얻으려면 ~p+1을 계산하면 됨. 반대로 음의 정수 n이 있을 때, n에 대한 양의 정수를 얻으려면 ~(n-1)을 계산하면 됨. ~~p는 변수 p에 비트 전환 연산을 두 번 적용하여 원래의 값이 되지만, 연산결과 타입이 byte가 아닌 int임.	

	수 있음. 피연산자의 타입이 int보다 작으면 int로 자동 형변환(산술 변환) 후에 연산하여 연산결과는 32자리의 2진수.
<< >> (쉬프트 연산자)	오든쪽(>>) 또는 왼쪽(>>)으로 피연산자의 각 자리를 이동(shift)한다. 1) 10진수를 2진수로 변환 2) 2진수를 왼쪽 또는 오든쯕으로 이동 3) 자리이동으로 인해 저장범위를 벗어나는 값은 버리고, 빈 자리는 양수일 때는 0, 음수일 때는 1로 채움 - 죄측 피연산자는 산술변환이 적용되어 int보다 작은 타입은 int타입으로 자동변환되고, 연산결과 역시 int타입임. 그러나 다든 이항연산자들과 달리 피연산자의 타입을 일치시킬 필요가 없어 우측 피연산자에는 산술변환이 적용되지 않음. Ex) '8 << 2' 1) 00001000 2) 001000 3) 00100000 (10진수로 32) - x << n은 x * 2ⁿ의 결과와 같다. - x >> n은 x / 2ⁿ의 결과와 같다. - 쉬프트 연산자를 사용하는 것이 나눗셈 또는 곱셈 연산자를 사용하는 것보다 빠르다. 그러나 코드의 가독성(readability)도 중요하기 때문에, 곱셈 또는 나눗셈 연산자를 주로 사용하되 보다 빠든 실행속도가 요구되어지는 곳만 쉬프트 연산자를 사용하는 것이 좋다. - n의 값이 자료형의 bit수보다 크면, 자료형의 bit수로 나눈 나머지 만큼만 이동한다. - n은 정수만 가능하며 음수인 경우, 부호없는 정수로 자동 변환된다.

● 그 외 연산자

1. 조건 연산자 ? : 조건식, 식1, 식2 모두 세 개의 피연산자를 필요로 하는 삼항 연산자.

(조건식) ? 식1 : 식2

첫 번째 피연산자인 조건식의 평가결과에 따라 다든 결과를 반환. 조건식이 true이면 식1이, false이면 식2가 연산 결과가 된다. 가독성을 높이기 위해 조건식을 괄호()로 둘러싸는 경우가 많다.

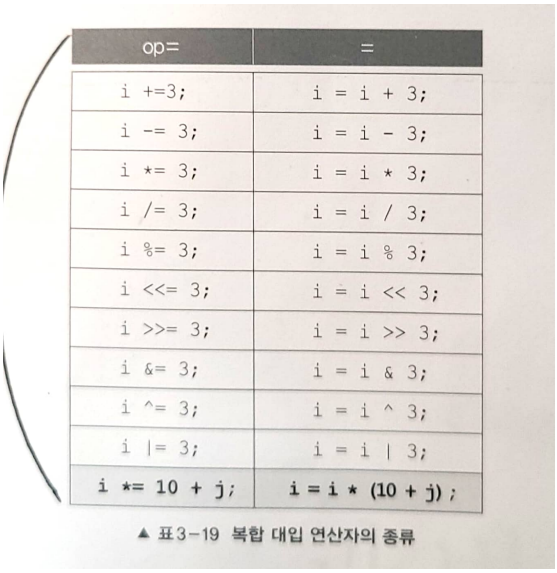
조건 연산자를 중첩해서 사용하면 셋 이상 중의 하나를 결과로 얻을 수 있다. 그리고 조건 연산자의 식1과 식2, 두 피연산자의 타입이 다든 경우, 산술 변환이 일어난다.

조건식1 ? 식1 : (조건식2 ? 식2 : 식3);

조건식1이 true일 경우 식1, false일 경우 - 조건식2가 true일 경우 식2, false일 경우 식3

2. 대입 연산자 =, op=: 변수와 같은 저장공간에 값 또는 수식의 연산결과를 저장하는데 사용된다. 오른쪽 피연산자의 값(식이라면 평가값)을 왼쪽 피연산자에 저장한다. 대입 연산자의 **왼쪽 피연산자를 'lvalue(left value)'라고 하고, 오른쪽 피연산자를 'rvalue(right value)**라고 한다. **대입연산자의 rvalue는 변수뿐만 아니라 식이나 상수 등이 모두 가능한 반면, lvalue는 반드시 변수처럼 값을 저장할 수 있는 것이어야 한다 - 리터럴이나 상수(변수 앞에 키워드 final을 붙임)같이 값을 저장할 수 없거나 변경할 수 없는 것들은 lvalue가 될 수 없다.**

- 복합 대입 연산자: 다른 연산자(op)와 결합하여 'op='와 같은 방식으로 사용될 수 있다.



op=	=
i +=3;	i = i + 3;
i -= 3;	i = i - 3;
i *= 3;	i = i * 3;
i /= 3;	i = i / 3;
i %= 3;	i = i % 3;
i <<= 3;	i = i << 3;
i >>= 3;	i = i >> 3;
i &= 3;	i = i & 3;
i ^= 3;	i = i ^ 3;
i |= 3;	i = i | 3;
i *= 10 + j;	**i = i * (10 + j);**

▲ 표3-19 복합 대입 연산자의 종류

Resource: 자바의 정석