● **연산자(operator)**: 연산을 수행하는 기호(+, -, *, / 등). 피연산자로 연산을 수행하고 나면 항상 결과값을 반환한다.
● **피연산자(operand)**: 연산자의 작업 대상(변수, 상수, 리터럴, 수식)
ㄴ **식(expression)**: 연산자와 피연산자를 조합하여 계산하고자 하는 바를 표현한 것. 식의 끝에 ';'을 붙여서 **문장(statement)**으로 만들어 프로그램에 포함시킨다.
ㄴ **식을 평가(evaluation)**: 식을 계산하여 결과를 얻는 것.

종류	연산자	설명
산술 연산자	+ - * / % << >>	사칙 연산과 나머지 연산
비교 연산자	> < >= <= == !=	크고 작음과 같고 다름을 비교
논리 연산자	&& || ! & | ^ ~	그리고(AND)와 또는(OR)으로 조건을 연결
대입 연산자	=	우변의 값을 좌변에 저장
기타	(type) ?: instanceof	형변환 연산자, 삼항 연산자, instanceof연산자

● **연산자의 우선순위:**
1. **단항(1)** > 이항(2) > 삼항(3)
2. **산술 > 비교 > 논리 > 대입.**
ㄴ 산술: 곱셈, 나눗셈 > 덧셈, 뺄셈
ㄴ 산술: 덧셈 > 쉬프트 연산자 << >>
ㄴ **논리: AND은 의미하는 &와 && > OR은 의미하는 |와 ||**
ㄴ 논리: 비교 연산자 == > 비트 연산자 &
3. 연산자 결합규칙(같은 우선순위의 연산자들의 수행 순서 규칙): 단항 연산자와 대입 연산자를 제외한 모든 연산의 진행방향은 **왼쪽에서 오른쪽이다**.

존재하지 않는 이미지입니다.

● **산술 변환**(usual arithmetic conversion): 연산 전에 피연산자 타입의 일치를 위해 자동 형변환하는 것.

1) **두 연산자의 타입은 보다 큰 타입으로 같게 일치**시킨다 (이항 연산자는 두 피연산자의 타입이 일치해야 연산이 가능하다). 더 큰 타입으로 일치시키는 이유는 형변환 시 값의 손실을 막기 위해서이다.
2) **피연산자의 타입이 int보다 작은 타입이면 int로 변환한다:** byte, char, short는 표현 범위가 좁아서 연산 중에 오버플로우가 발생한 가능서이 높기 때문에 int형으로 변환한다.
- 모든 연산에서 '산술 변환'이 일어나지만, 쉬프트 연산자 <<, >>, 증감 연산자(++ --)은 예외이다.

● **단항 연산자:**
1. **증감 연산자: 피연산자에 저장된 값은 1 증가 또는 감소**시킨다. **피연산자로는 정수와 실수** 모두 가능하지만, 상수는 값을 변경할 수 없으므로 가능하지 않다.
ㄴ **증가 연산자(++): 피연산자의 값은 1 증가시킨다**
ㄴ **감소 연산자(--): 피연산자의 값은 1 감소시킨다**
ㄴ **전위형(prefix): 피연산자의 왼쪽**에 위치한다.
ㄴ **후위형(postfix): 피연산자의 오른쪽**에 위치한다.
- 증감 연산자가 다른 수식이나 메서드 호출에 포함된 경우, **전위형일 때(값이 참조되기 전에 증가시킴)와 후위형일 때(값이 참조된 후에 증가시킴)의 결과가 다르다. 전위형은 변수(피연산자)의 값을 먼저 증가시킨 후에 변수의 값을 읽어오는 반면, 후위형은 변수의 값을 먼저 읽어온 후에 값을 증가시킨다.**
- 그러나 '++i'나 'i++'처럼 증감연산자가 수식이나 메서드 호출에 포함되지 않고 독립적인 하나의 문장으로 쓰인 경우에는 전위형과 후위형의 차이가 없다.
- 식에 두 번 이상 포함된 변수에 증감연사자를 사용하는 것은 피해야 한다.
2. **부호 연산자: '-'는 피연산자의 부호를 반대로 변경한 결과를 반환한다. 피연산자가 음수면 양수, 양수면 음수가 연산의 결과가 된다.** (부호연산자'+'는 하는 일이 없으며, 쓰이는 일도 없다.)

● **산술 연산자:**
1. **사칙 연산자:**
- 피연산자가 정수형인 경우, 나누는 수로 0을 사용할 경우 컴파일은 정상적으로 되지만 실행 시 오듀(ArithmeticException)가 발생한다. **부동 소수점값인 0.0f, 0.0d로 나누는 것은 가능하지만 그 결과는 Infinit(무한대)**이다.
- 두 피연산자가 모두 int타입인 경우, 연산 결과 역시 int타입이다. 또한 이런 경우 소수점 이하는 모두 버려진다. 또한 연산의 결과가 overflow로 int가 저장할 수 있는 범위를 넘어설 경우, 아무리 그 결과 값을 long로 자동 형변환해도 overflow된 값은 변하지 않는다. 이 오버플로우 때문에 같은 의미의 식이라도 연산의 순서에 따라서 다든 결과를 얻을 수 있다.
- 사칙연산의 피연산자로 문자도 가능하다. 실제로 해당 문자의 유니코드(부호 없는 정수)로 바뀌어 저장되므로 문자간의 사칙 연산은 정수간의 연산과 동일하다. 문자 '숫자'를 숫자로 변환하려면 문자 '0'을 빼주면 되는데, 이는 '0'~'9'까지의 문자의 유니코드가 연속적으로 배치되어 있어서, 해당 문자에서 '0'을 빼주면 숫자로 변환하기 때문이다. '0'(48)~'9'(57) 뿐만 아니라, 'A'(65)~'Z'(90), 'a'(97)~'z'(122)역시 연속적으로 코드가 지정되어 있다.
ㄴ **문자 리터럴과 숫자 리터럴의 연산은 컴파일 에러가 발생하지 않는다. 상수 또는 리터럴 간의 연산은 실행 과정동안 변하는 값이 아니기 때문에, 컴파일러가 미리 연산을 수행하고 실행 시에는 연산의 결과를 변수에 저장하기 때문이다. 반면 수식에 변수가 듣어가 있는 경우에는 컴파일러가 미리 계산을 할 수 없기 때문에 컴파일 에러가 발생한다. 이 경우 반드시 형변환 연산자를 사용해서 형변환을 해 주어야 한다.**
- float 리터럴간의 연산에 +0.5를 해서 반올림 하여 int로 형변환하면 소수점 나머지를 버리게 되므로 이런 방식으로 반올림을 해 줄 수 있다.
- Math.round(): 매게변수로 받은 값은 소수점 첫째자리에서 반올림은 하고 그 결과를 정수로 돋려주는 메서드이다.

2. **나머지 연산자 %:**
- 왼쪽의 피연산자를 오든쪽 피연산자로 나누고 난 나머지 값을 결과로 반환하는 연산자. 짝수, 홀수, 또는 배수 검사 등에 주로 사용된다.
- 연산자는 나누는 수로 음수도 허용하지만, 부호는 무시되므로 결과는 음수의 절대값으로 나눈 나머지와 결과가 같다. 결과를 음수로 표시하고 싶을 경우 피연산자의 부호를 모두 무시하고, 나머지 연산을 한 결과에 왼쪽 피연산자(나눠지는 수)의 부호를 붙이면 된다.