Basic Operator

☑ Basic operator, math

덧셈 +, 곱셈 *, 뺄셈 -과 같은 연산은 학교에서 배워서 이미 알고 있을 것이다.

이 장에서는 이런 기본 operator를 시작으로 학교에선 다루지 않았던 JavaScript에서만 제공하는 operator에 대해 학습하도록 한다.

❖ 용어: 'unary(단항)', 'binary(이항)', 'operand(피연산자)'

operator에 대해 학습하기 전에, 앞으로 자주 등장하게 될 용어 몇 가지를 정리해 보자.

☞ operand는 operator가 연산을 수행하는 대상이다. 5 * 2에는 왼쪽 operand 5와 오른쪽 operand 2, 총 두 개의 operand가 있다. 'operand'는 'argument(인수)'라는 용어로 불리기도 한다.

☞ operand를 하나만 받는 operator는 단항(unary) operator 라고 부른다. operand의 부호를 뒤집는 unary negation operator -는 단항 operator의 대표적인 예이다.

```
let x = 1;

x = -x;
alert( x );      // -1, unary negation operator는 부호를 뒤집는다.
```

☞ 두 개의 operand를 받는 operator는 이항(binary) operator 라고 부른다. minus operator는 아래와 같이 binary operator로 쓸 수도 있다.

```
let x = 1, y = 3;
alert( y - x );  // 2, binary minus operator는 뺄셈을 해준다.
```

위와 같이 부호를 반전해주는 unary negation operator와 뺄셈에 쓰이는 binary minus operator(뺄셈 operator)는 기호는 같지만 수행하는 연산이 다르다. 두 연산을 구분하는 기준은 operand의 개수이다.

❖ Maths

JavaScript에서 지원하는 수학 operator는 다음과 같다.

- ☞ Addition(덧셈) operator +,
- ☞ Substraction(뺄셈) operator -,
- ☞ Multiplication(곱셈) operator *,
- ☞ Division(나눗셈) operator /,
- ☞ Remainder(나머지) operator %,
- ☞ Exponentiation(거듭제곱) operator **

앞쪽 4 operator는 설명이 필요 없겠지만, % 와 **는 약간의 설명이 필요할 것 같다.

☞ Remainder operator %

remainder operator는 % 기호로 나타내지만, 비율을 나타내는 퍼센트와 관련이 없다.
remainder operator를 사용한 표현식 a % b는 a를 b로 나눈 후 그 나머지(remainder)를 정수로 반환해 준다.

[예시]

```
alert( 5 % 2 );   // 5를 2로 나눈 후의 나머지인 1을 출력
alert( 8 % 3 );   // 8을 3으로 나눈 후의 나머지인 2를 출력
```

☞ Exponentiation(거듭제곱) operator **

exponentiation operator를 사용한 a ** b를 평가하면 a를 b번 곱한 값이 반환된다.

[예시]

```
alert( 2 ** 2 );  // 4  (2 * 2)
alert( 2 ** 3 );  // 8  (2 * 2 * 2)
alert( 2 ** 4 );  // 16 (2 * 2 * 2 * 2)
```

exponentiation operator는 정수가 아닌 숫자에 대해서도 동작한다. 1/2을 사용하면 square root(제곱근)을 구할 수 있다.

```
alert( 4 ** (1/2) );      // 2 (1/2 거듭제곱은 제곱근)
alert( 8 ** (1/3) );      // 2 (1/3 거듭제곱은 세제곱근)
```

❖ binary operator '+'와 string concatenation

JavaScript가 제공하는 특별한 operator 기능에 대해 살펴보자.

plus operator +는 대개 number를 더한 결과를 반환한다.
그런나 binary operator +의 operand로 string이 전달되면 plus operator는 plus가 아닌 string을 병합(concatenattion-연결)한다.

```
let s = "my" + "string";
alert(s); // mystring
```

따라서 binary operator +를 사용할 때는 operand 중 하나가 string이면 다른 하나도 string으로 변환된다는 점에 주의해야 한다.

[예시]

```
alert( '1' + 2 );        // "12"
alert( 2 + '1' );        // "21"
```

첫 번째 operand가 string인지, 두 번째 operand가 string인지는 중요하지 않다. operand 중 어느 하나가 string이면 다른 하나도 string로 변환된다.

좀 더 복잡한 예시를 살펴보자.

```
alert(2 + 2 + '1' );       // '221'이 아니라 '41'이 출력된다.
```

연산은 왼쪽에서 오른쪽으로 순차적으로 진행되기 때문에 이런 결과가 나왔다. 두 개의 숫자 뒤에 string이 오는 경우, 숫자가 먼저 더해지고, 그 후 더해진 숫자와 string과의 병합이 일어난다.

이처럼 binary plue operator +는 string 연결과 변환이라는 특별한 기능을 제공한다. 다른 arithmetic operator(산술 연산자)가 오직 숫자형의 operand만 다루고, operand가 number가 아닌 경우에 그 type을 number로 바꾸는 것과는 대조적이다.

아래는 뺄셈 -과 나눗셈 / operator가 어떻게 string operand를 다루는지를 보여준다.

```
alert( 6 - '2' );        // 4, '2'를 number로 바꾼 후 연산이 진행된다.
alert( '6' / '2' );      // 3, 두 operand가 number로 바뀐 후 연산이 진행된다.
```

❖ unary operator +와 Numeric conversion

plus operator +는 binary operator뿐만 아니라 unary operator로도 사용할 수 있다.

number에 unary plus operator를 붙이면 이 operator는 아무런 동작도 하지 않는다. 그러나 operand가 number가 아닌 경우엔 number type으로의 변환이 일어난다.

[예시]

```
// 숫자에는 아무런 영향을 미치지 않는니다.
let x = 1;
alert( +x );     // 1

let y = -2;
alert( +y );     // -2

// 숫자형이 아닌 operand는 숫자형으로 변화한다.
alert( +true ); // 1
alert( +"" );   // 0
```

unary plus operator는 짧은 문법으로도 Number(...)와 동일한 일을 할 수 있게 해준다.

개발을 하다 보면 string을 number로 변환해야 하는 경우가 자주 생긴다. HTML 폼(form) 필드에서 값을 가져왔는데, 그 값이 string일 때 같이 상황이다. 실제로 form에서 가지고 온 값은 대개 string 형태이다.

binary plus operator를 사용하면 아래와 같이 값이 string로 변해서 연결될 것이다.

```
let apples = "2";
let oranges = "3";

alert( apples + oranges ); // 23, binary plus operator는 string을 연결한다.
```

원하는 대로 값을 더해주려면, unary plus operator를 사용해 operand를 number type으로 변화시키면 된다.

```
let apples = "2";
let oranges = "3";

// binary plus operator가 적용되기 전에, 두 operand는 number type으로 변화한다.
alert( +apples + +oranges ); // 5

// `Number(...)`를 사용해서 같은 동작을 하는 코드를 작성할 수 있지만, 코드가 더 길다.
// alert( Number(apples) + Number(oranges) ); // 5
```

위 식을 수학자가 본다면 불필요한 덧셈 기호에 대해 언급하며 식이 이상하다고 지적할 것이다. Programmer 라면 아닐 것이다. 위 식은 우리가 의도한 대로 unary plus operator가 먼저 string을 number로 변환시키고, binary plus operator가 그 결과들을 더해주고 있다.

그런데 왜 binary plus operator가 적용되기 전에 unary plus operator가 먼저 적용될까? 그 이유는 이제 학습하게 될 operator 우선순위 때문이다.

❖ Operator precedence

하나의 표현식에 둘 이상의 operator가 있는 경우, 실행 순서는 operator의 우선순위(precedence) 에 의해 결정된다.

1 + 2 * 2라는 식이 있을 때 곱셈이 먼저, 그 후에 덧셈이 일어난다는 것이다. 이런 개념이 operator 우선순위이다. 여기서 곱셈은 덧셈보다 더 높은 우선순위를 가진다.

JavaScript에서 정의한 operator 우선순위가 마음에 들지 않는다면, 괄호를 사용하면 된다. 괄호는 모든 operator보다 우선순위가 높기 때문에 JavaScript에서 정의한 operator 우선순위를 무력화시킨다. 표현식 (1 + 2) * 2에서 괄호로 둘러싼 덧셈 operator가 먼저 수행되는 것과 같다.

JavaScript는 다양한 operator를 제공하는데, 이 모든 operator엔 우선순위가 매겨져 있다. 우선순위 숫자가 클수록 먼저 실행된다. 순위가 같으면 왼쪽부터 시작해서 오른쪽으로 연산이 수행된다.

아래는 우선순위 테이블(precedence table)의 일부를 발췌한 표이다. 순서를 기억할 필요는 없지만, 동일한 기호의 unary operator는 binary operator보다 우선순위가 더 높다는 것에 주목하자.

Precedence	Name	Sign
…	…	…
15	unary plus	+
15	unary negation	-
14	exponentiation	**
13	multiplication	*
13	division	/
12	addition	+
12	subtraction	-
…	…	…
2	assignment	=
…	…	…

'unary plus operator'는 우선순위 15로, '(binary) plus operator'의 우선순위 12보다 높다. 표현식 "+apples + +oranges"에서 unary plus operator가 plus보다 먼저 수행되는 이유가 바로 이 때문이다.

❖ Assignment operator

무언가를 assignment할 때 쓰이는 = 도 operator 이다. 이 operator는 assignment(할당) operator라고 불리는데, 우선순위는 3으로 아주 낮다.

x = 2 * 2 + 1과 같은 표현식에서 계산이 먼저 이뤄지고, 그 결과가 x에 assignment되는 이유가 바로 이 때문이다.

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

☞ Assignment operator = return a value

= 는 operator이기 때문에 흥미로운 함축성을 내포하고 있다.
JavaScript에서 대부분의 operator들은 값을 반환한다. +와 -뿐만 아니라 = 역시 값을 반환한다.
x = value을 호출하면 value가 x에 쓰여지고, 이에 더하여 value가 반환된다.

assignment operator의 이런 특징을 이용한 복잡한 표현식을 살펴보자.

```
let a = 1;
let b = 2;

let c = 3 - (a = b + 1);

alert( a );       // 3
alert( c );       // 0
```

위 예제에서 표현식 (a = b + 1)은 a에 값을 assignment하고, 그 값인 3을 반환한다. 그리고 반환 값은 이어지는 표현식에 사용된다.

괴상한 코드라고 느껴지겠지만, 여러 JavaScript Library에서 이런 식으로 assignment operator를 사용하고 있기 때문에 동작 원리를 이해할 수 있어야 한다.

다만, 직접 코드를 작성할 땐 이런 방식을 사용하지 않도록 한다. 이런 트릭을 사용하면 코드가 명확하지 않을 뿐만 아니라 가독성도 떨어지기 때문이다.

☞ assignment operator Chaining

assignment operator는 아래와 같이 여러 개를 연결할 수도 있다(chaining).

```
let a, b, c;

a = b = c = 2 + 2;

alert( a );      // 4
alert( b );      // 4
alert( c );      // 4
```

이렇게 assignment operator를 여러 개 연결한 경우, 평가는 우측부터 진행된된. 먼저 가장 우측의 2 + 2가 평가되고, 그 결과가 좌측의 c, b, a에 순차적으로 assignment 된다. 모든 변수가 단일 값을 공유하게 된다..

그런데 되도록이면 operator를 chaining 하는것 보다 가독성을 위해 아래와 같이 줄을 나눠 코드를 작성하길 권장한다.

```
c = 2 + 2;
b = c;
a = c;
```

이렇게 작성하면 읽기도 쉽고, 눈을 빠르게 움직이며 코드를 읽을 수 있다.

❖ Modify-in-place(복합 assignment operator)

프로그램을 짜다 보면, 변수에 operator를 적용하고 그 결과를 같은 변수에 저장해야 하는 경우가 종종 생긴다.

```
let n = 2;
n = n + 5;
n = n * 2;
```

이때, += 와 *= operator를 사용하면 짧은 문법으로 동일한 연산을 수행할 수 있다.

```
let n = 2;
n += 5;          // n은 7이 된다(n = n + 5와 동일).
n *= 2;          // n은 14가 된다(n = n * 2와 동일).

alert( n );      // 14
```

이런 'modify-and-assign' operator는 arithmetical operator와 bitwise operator에도 적용할 수 있다. /=, -= 등의 operator를 만들 수 있다.

modify-and-assign operator의 우선순위는 assignment operator와 동일하다. 따라서 대부분 다른 operator가 실행된 후에 modify-and-assign operator가 실행된다.

```
let n = 2;

n *= 3 + 5;

alert( n );      // 16  (*=의 우측이 먼저 평가되므로, 위 식은 n *= 8과 동일하다.)
```

❖ increment/descrement operator

숫자를 하나 늘리거나 줄이는 것은 자주 사용되는 연산이다. JavaScript에서는 이런 연산을 해주는 operator를 제공한다.

☞ 증가(increment) operator ++는 변수를 1 증가시킨다.

```
let counter = 2;
counter++;       // counter = counter + 1과 동일하게 동작한다. 하지만 식은 더 짧다.
alert( counter ); // 3
```

☞ 감소(decrement) operator --는 변수를 1 감소시킵니다.

```
let counter = 2;
counter--;      // counter = counter - 1과 동일하게 동작한다. 하지만 식은 더 짧다.
alert( counter ); // 1
```

중요한 점은 increment/decrement operator는 변수에만 쓸 수 있다. 5++와 같이 값에 사용하려고 하면 에러가 발생한다.

++와-- operator는 변수 앞이나 뒤에 올 수 있다.

☞ counter++와 같이 operand 뒤에 올 때는, '후위형(postfix form)'이라고 부른다.
☞ ++counter와 같이 operand 앞에 올 때는, '전위형(prefix form)'이라고 부른다.

postfix과 prefix는 operand인 counter를 1만큼 증가시켜 준다는 점에서 동일한 일을 한다.

두 형의 차이는 ++/--의 반환 값을 사용할 때 드러난다.

다시 상기해 보도록 한다. 이미 배운 바와 같이 모든 operator는 값을 반환한다. increment/decrement operator도 마찬가지입니다. prefix은 increment/decrement 후의 새로운 값을 반환하는 반면, postfix은 increment/decrement 전의 기존 값을 반환한다.

아래 예시를 통해 차이점을 직접 살펴보도록 한다.

```
let counter = 1;
let a = ++counter; // (1)

alert(a);        // 2
```

(1)로 표시한 줄의 prefix ++counter는 counter를 증가시키고 새로운 값 2를 반환한다. 따라서 alert는 2를 표시한다.

이제 postfix을 살펴본다.

```
let counter = 1;
let a = counter++; // (2) ++counter를 counter++로 바꿈

alert(a); // 1
```

(2)로 표시한 줄의 postfix counter++는 counter를 증가시키긴 하지만, 증가 전의 기존 값을 반환한다. 따라서 alert는 1을 표시한다.

increment/decrement operator에 대한 내용을 정리하면 아래와 같다.

반환 값을 사용하지 않는 경우라면, prefix과 postfix엔 차이가 없다.

```
let counter = 0;
counter++;
++counter;
alert( counter ); // 2, 위 두 라인은 동일한 연산을 수행한다.
```

값을 증가시키고 난 후, 증가한 값을 바로 사용하려면 prefix 증가 operator를 사용하면 된다.

```
let counter = 0;
alert( ++counter ); // 1
```

값을 증가시키지만, 증가 전의 기존값을 사용하려면 postfix 증가 operator를 사용하면 된다.

```
let counter = 0;
alert( counter++ ); // 0
```

⊙ 다른 operator 사이의 increment/decrement operator

++/-- operator를 표현식 중간에 사용하는 것도 가능하다. 이때, increment/decrement operator의 우선순위는 다른 대부분의 arithmetical operator보다 높기 때문에, 평가가 먼저 이뤄진다.

[예시]

```
let counter = 1;
alert( 2 * ++counter ); // 4
```

위 예시를 아래와 비교해 보자.

```
let counter = 1;
alert( 2 * counter++ ); // counter++는 '기존'값을 반환하기 때문에 2가 출력된다.
```

이렇게 코드를 작성하는 게 기술적으로 문제가 있는 것은 아니지만, 한 줄에서 여러 가지 일을 동시에 하고 있기 때문에 코드의 가독성이 떨어진다.

코드를 읽을 때 눈을 '수직으로' 빠르게 움직이다 보면 counter++와 같은 것을 놓치기 쉽다. 변수가 증가했다는 것을 놓칠 수 있다.

'코드 한 줄엔, 특정 동작 하나'에 관련된 내용만 작성하는 게 좋다.

```
let counter = 1;
alert( 2 * counter );
counter++;
```

❖ Bitwise operator

bitwise operator는 인수를 32비트 정수로 변환하여 binary 연산을 수행한다.
이런 bit 조작 관련 operator는 JavaScript 뿐만 아니라 대부분의 프로그래밍 언어에서 지원한다.

아래는 bit 연산 시 쓰이는 operator 목록이다.

- ☞ 비트 AND (&)
- ☞ 비트 OR (¦)
- ☞ 비트 XOR (^)
- ☞ 비트 NOT (~)
- ☞ 왼쪽 시프트(LEFT SHIFT) (<<)
- ☞ 오른쪽 시프트(RIGHT SHIFT) (>>)
- ☞ 부호 없는 오른쪽 시프트(ZERO-FILL RIGHT SHIFT) (>>>)

bitwise operator는 저수준(2진 표현)에서 숫자를 다뤄야 할 때 쓰이므로 흔하게 쓰이진 않는다. Web 개발 시엔 이런 일이 자주 일어나지 않기 때문에, bitwise operator를 만날 일은 거의 없다. 그렇긴 해도 암호를 다뤄야 할 땐 bitwise operator가 유용하ek.

❖ Comma operator

comma operator(,)는 좀처럼 보기 힘들고, 특이한 operator 중 하나이다. 코드를 짧게 쓰려는 의도로 가끔 사용된다. 이런 코드를 만났을 때, 어떤 연산 결과가 도출되는지 알아야 함으로 comma operator에 대해 알아보도록 한다.

comma operator(,)는 여러 표현식을 코드 한 줄에서 평가할 수 있게 해준다. 이때 표현식 각각이 모두 평가되지만, 마지막 표현식의 평가 결과만 반환되는 점에 유의해야 한다.

[예시]

```
let a = (1 + 2, 3 + 4);

alert( a );      // 7 (3 + 4의 결과)
```

위 예시에서 첫 번째 표현식 1 + 2은 평가가 되지만 그 결과는 버려진다. 3 + 4만 평가되어 a에 assignment 된다.

⊙ 쉼표의 우선순위는 매우 낮다.

comma operator(,)의 operator 우선순위는 매우 낮습니다. assignment operator = 보다 더 낮다. 따라서 위 예시에선 괄호가 중요한 역할을 한다.
괄호가 없으면 a = 1 + 2, 3 + 4에서 +가 먼저 수행되어 a = 3, 7이 된다. assignment operator = 는 comma operator(,)보다 우선순위가 높기 때문에 a = 3이 먼저 실행되고, 나머지(7)는 무시되죠. (a = 1 + 2), 3 + 4를 연산한 것처럼 될 것이다.

이렇게 마지막 표현식을 제외한 모든 것을 버리는 operator는 어디서 사용되는 걸까?
여러 동작을 하나의 줄에서 처리하려는 복잡한 구조에서 이를 사용한다.

```
// 한 줄에서 세 개의 연산이 수행됨
for (a = 1, b = 3, c = a * b; a < 10; a++) {
 ...
}
```

comma operator(,)를 사용한 트릭은 여러 JavaScript Framework에서 볼 수 있다. 이 operator의 사용 빈도가 높지 않지만, 언급하고 넘어가는 이유이ek. comma operator(,)는 코드 가독성에 도움이 되지 않는다. 따라서 곰곰이 생각해 본 후, 진짜 필요한 경우에만 사용하는 것을 권장한다.

[과제]

☞ Prefix와 Postfix

★ **중요도:** 5

아래 코드가 실행된 후, 변수 a, b, c, d엔 각각 어떤 값들이 저장될까?

```
let a = 1, b = 1;

let c = ++a; // ?
let d = b++; // ?
```

[해답]

✓ a = 2

✓ b = 2

✓ c = 2

✓ d = 1

```
let a = 1, b = 1;

alert( ++a );     // 2, prefix는 증가 후의 값을 반환한다.
alert( b++ );     // 1, postfix는 증가 전의 값을 반환한다.

alert( a );       // 2, 값이 1만큼 증가한다.
alert( b );       // 2, 값이 1만큼 증가한다.
```

☞ assignment 후 결과 예측하기

★ **중요도:** 3

아래 코드가 실행되고 난 후, a와 x엔 각각 어떤 값이 저장될까?

```
let a = 2;

let x = 1 + (a *= 2);
```

[해답]

a = 4 (기존 값(2)에 2를 곱한 4)

x = 5 (1 + 4의 결과)

☞ Type conversion

★ **중요도:** 5

아래 표현식들의 결과를 예측해 보자.

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
7 / 0
"  -9  " + 5
"  -9  " - 5
null + 1
undefined + 1
" \t \n" - 2
```

예측한 결과를 적어본 후, 해답과 비교해 본다.

[해답]

```
"" + 1 + 0 = "10"                // (1)
"" - 1 + 0 = -1                  // (2)
true + false = 1
6 / "3" = 2
"2" * "3" = 6
4 + 5 + "px" = "9px"
"$" + 4 + 5 = "$45"
"4" - 2 = 2
"4px" - 2 = NaN
7 / 0 = Infinity
"  -9  " + 5 = "  -9  5"         // (3)
"  -9  " - 5 = -14               // (4)
null + 1 = 1                     // (5)
undefined + 1 = NaN              // (6)
" \t \n" - 2 = -2                // (7)
```

(1) operand 중 하나가 string인 "" + 1에서 1은 string으로 변환된다. 따라서 공백과 string 1을 더한, "" + 1 = "1"과 같은 효과를 발휘한다. 그 다음 연산 "1" + 0에도 같은 규칙이 적용된다.

(2) subtraction operator -는 기타 수학 operator처럼 숫자형만을 인수로 받는다. empty string ""는 숫자 0으로 변환되기 때문에 결과는 -1이 된다.

(3) operand 중 하나가 string이므로 숫자 5가 string로 변환된다.

(4) subtraction operator는 인수를 숫자형으로 변화시키므로 " -9 "는 숫자 -9로 변한다. 앞, 뒤 공백은 제거된다.
(5) numerix(숫자형)으로 변환 시 null은 0이 된다.
(6) undefined는 numeric(숫자형)으로 변환시 NaN이 된다.
(7) string이 number로 변할 땐 string 앞뒤의 공백이 삭제된다. subtraction operator 앞의 operand는 공백을 만드는 문자 \t와 \n, 그 사이의 "일반적인" 공백으로 구성된다. 따라서 " \t \n"는 number로 변환 시 길이가 0인 string로 취급되어 숫자 0이 된다.

☞ 덧셈 고치기

★ **중요도:** 5

아래 코드는 사용자에게 숫자 2개를 입력받은 다음 그 합을 보여준다.
그런데 의도한 대로 예시가 동작하지 않는다. prompt 창에 세팅한 기본값을 수정하지 않은 경우 덧셈의 결과는 12가 된다.
왜 그럴까? 예시가 제대로 동작하도록 코드를 수정해 보자. 결과는 3이 되어야 한다.

```
let a = prompt("덧셈할 첫 번째 숫자를 입력해주세요.", 1);
let b = prompt("덧셈할 두 번째 숫자를 입력해주세요.", 2);

alert(a + b); // 12
```

[해답]

의도한 대로 덧셈이 되지 않는 이유는 prompt 함수가 사용자 입력을 string으로 반환하기 때문이다.
따라서 prompt 창에서 입력한 변수들은 각각 string인 "1"과 "2"가 된다.

```
let a = "1"; // prompt("덧셈할 첫 번째 숫자를 입력해주세요.", 1);
let b = "2"; // prompt("덧셈할 두 번째 숫자를 입력해주세요.", 2);

alert(a + b); // 12
```

예시가 제대로 동작하게 하려면 덧셈 연산 +가 수행되기 전에 string을 number로 변환해야 한다. 이때 Number()를 사용하거나 변수 앞에 +를 붙여줄 수 있다.

아래 코드에선 prompt 함수 바로 앞에서 string을 number로 변환했다.

```
let a = +prompt("덧셈할 첫 번째 숫자를 입력해주세요.", 1);
let b = +prompt("덧셈할 두 번째 숫자를 입력해주세요.", 2);

alert(a + b); // 3
```

아래 코드에선 alert 함수 안에서 string을 number로 변환해 보았다.

```
let a = prompt("덧셈할 첫 번째 숫자를 입력해주세요.", 1);
let b = prompt("덧셈할 두 번째 숫자를 입력해주세요.", 2);

alert(+a + +b); // 3
```

코드 한 줄 안에서 unary, binary + operator를 한꺼번에 사용하는 것은 이상하다.