Some more operators

Enrich your code with more operators that will help you change values and formulate better conditions.

We will soon write conditions and loops. In these two control structures, a few operators come handy:

- `+=`, `-=`, `*=`, `/=`, `%=` etc. are abbreviations for adding a value to a variable. `a += 1` is the same as writing `a = a + 1`.
- `++x` increases the value of `x` by `1`, then returns the increased value
- `x++` returns the original value of `x`, then increases its value by `1`
- `--x` decreases the value of `x` by `1`, then returns the decreased value
- `x--` returns the original value of `x`, then decreases its value by `1`

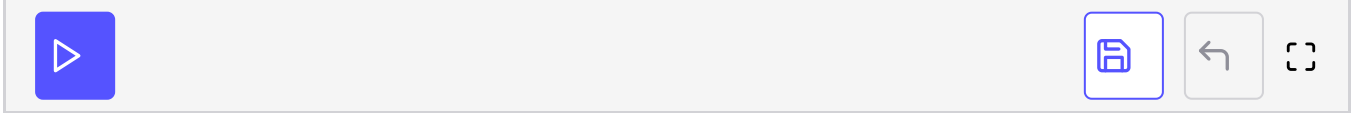
++x and x++

I know, the difference between `++x` and `x++` may not make sense to you right now. I argue that in most cases, it should not even make a difference as long as you want to write readable code.

Both `x++` and `++x` have a *main effect* and a *side effect*. As a main effect, they return a value. Either `x`, or `x + 1`, depending on whether you write the ++ after the x or before. Basically the difference comes because of precedence,

- `x++` executes the statement and then increments the value.
- `++x` increments the value and then executes the statement.

```
let a = 1;
a *= 2; // a = a * 2;
console.log( '2++: ', a++ );
console.log( '--3: ', --a );
```



`console.log(a++)` writes the value of `a` to the console before incrementing it

`console.log(a++)` writes the value of `a` to the console before incrementing it. The value of `a` was `2` after the second line of the program. After the value of `a` is written to the console, the value of `a` is incremented and it now becomes 3. Then `console.log(--a)` first decrements the value making it 2 and then writes it to the console.

Quiz Time!

Q

Determine the value of `a` at the end of this program.

```
var a = 3;
++a;
a *= 3;
a %= 5;
```

COMPLETED 0%

1 of 1

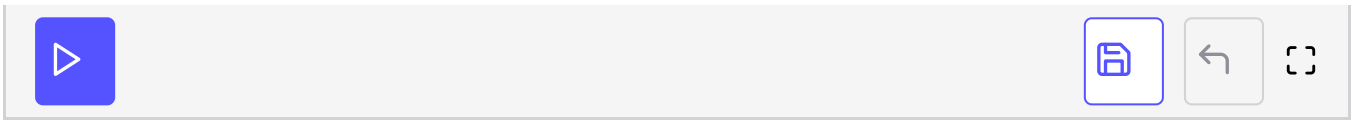
The && operator

There are two more operators that come handy when formulating boolean conditions.

Suppose you are checking whether a variable is an array **and** it has a positive length. The **and** conjunction is denoted by the `&&` operator:

```
function isNonEmptyArray( x ) {
    return Array.isArray( x ) && x.length > 0;
}

console.log(isNonEmptyArray( 5 )) //false not an array and does not have positive value
console.log(isNonEmptyArray( [] )) //false
console.log(isNonEmptyArray( [1] ))//true
```



- `isNonEmptyArray(5)` returns `false` because `5` is not an array and does not have a positive value
- `isNonEmptyArray([])` returns `false` because it is an array but does not have any positive value
- `isNonEmptyArray([1])` returns `true` because it satisfies both the conditions as `[1]` is an array and it also contains a positive value

The and shortcut

Also, please note that the value `5` is not an array, so `Array.isArray( 5 )` becomes false. A `false` value in conjunction with anything is `false`:

```
> false && false
false
> false && true
false
```

Therefore, the second operand of the `&&` operator is not even executed. This is good, because `5.length` would have *thrown an error*. We can rely on the `&&` operator not executing the right hand side expression in case the left side is evaluated as `false`. This simplification is called a *shortcut*.

If the left hand side is true, the right hand side is executed. In this case, the value of the `and` expression becomes the value of the right hand side expression:

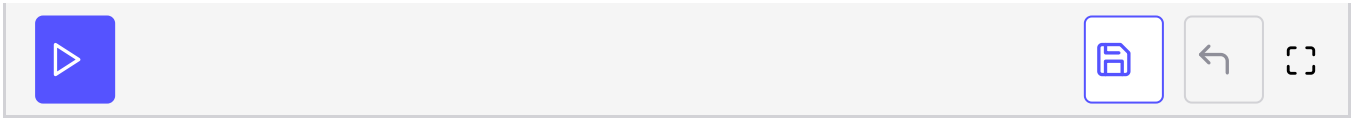
```
 > true && false
false
> true && true
true
```

The || operator

We can also define an **or** relationship between operands. For instance, suppose we have a function that returns true for numbers and strings:

```
function numberOrString( x ) {
    return typeof x === 'number' || typeof x === 'string'
}

console.log(numberOrString( NaN )) //true
console.log(numberOrString( '' )) //true
console.log(numberOrString( null )) //false
```



- `numberOrString( NaN )` returns `true` because it has the `number` type
- `numberOfString( ‘’ )` returns `true` because it has the `string` type
- `numberOrString( null )` returns `false` because it does not match the `number` nor the `string` type

The or *shortcut*

The or operator works as follows: in case of `a || b`, if `a` is `true`, then `b` is not even evaluated. This is the or *shortcut*. Once we know that a variable is a number, we don’t have to check if the same variable is a string too. We already know that the function should return `true`.

How to check for NaN value?

The value `NaN` was mentioned. When formulating a condition that checks if a value is `NaN`, we have the problem that `NaN == NaN` is false. The value `NaN` does not equal to itself. How can we then check if a value is `NaN`? The answer is, use `Number.isNaN()`.

```
console.log(Number.isNaN( NaN ))
```