30th November 2019

Logical operators

There are three logical operators in JavaScript: `||` (OR), `&&` (AND), `!` (NOT).

Although they are called “logical”, they can be applied to values of any type, not only boolean. Their result can also be of any type.

Let’s see the details.

|| (OR)

The “OR” operator is represented with two vertical line symbols:

```
result = a || b;
```

In classical programming, the logical OR is meant to manipulate boolean values only. If any of its arguments are `true`, it returns `true`, otherwise it returns `false`.

In JavaScript, the operator is a little bit trickier and more powerful. But first, let’s see what happens with boolean values.

There are four possible logical combinations:

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

As we can see, the result is always `true` except for the case when both operands are `false`.

If an operand is not a boolean, it’s converted to a boolean for the evaluation.

For instance, the number `1` is treated as `true`, the number `0` as `false`:

```
if (1 || 0) { // works just like if( true || false )
  alert( 'truthy!' );
}
```

Most of the time, OR `||` is used in an `if` statement to test if *any* of the given conditions is `true`.

For example:

```
let hour = 9;

if (hour < 10 || hour > 18) {
  alert( 'The office is closed.' );
}
```

We can pass more conditions:

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'The office is closed.' ); // it is the weekend
}
```

OR “||” finds the first truthy value

The logic described above is somewhat classical. Now, let’s bring in the “extra” features of JavaScript.

The extended algorithm works as follows.

Given multiple OR’ed values:

```
result = value1 || value2 || value3;
```

The OR `||` operator does the following:

- Evaluates operands from left to right.
- For each operand, converts it to boolean. If the result is `true`, stops and returns the original value of that operand.
- If all operands have been evaluated (i.e. all were `false`), returns the last operand.

A value is returned in its original form, without the conversion.

In other words, a chain of OR `"||"` returns the first truthy value or the last one if no truthy value is found.

For instance:

```
alert( 1 || 0 ); // 1 (1 is truthy)
alert( true || 'no matter what' ); // (true is truthy)

alert( null || 1 ); // 1 (1 is the first truthy value)
alert( null || 0 || 1 ); // 1 (the first truthy value)
alert( undefined || null || 0 ); // 0 (all falsy, returns the last value)
```

This leads to some interesting usage compared to a “pure, classical, boolean-only OR”.

1. **Getting the first truthy value from a list of variables or expressions.**

Imagine we have a list of variables which can either contain data or be `null/undefined`. How can we find the first one with data?

We can use OR `||`:

```
let currentUser = null;
let defaultUser = "John";

let name = currentUser || defaultUser || "unnamed";

alert( name ); // selects "John" – the first truthy value
```

If both `currentUser` and `defaultUser` were falsy, `"unnamed"` would be the result.

2. **Short-circuit evaluation.**

Operands can be not only values, but arbitrary expressions. OR evaluates and tests them from left to right. The evaluation stops when a truthy value is reached, and the value is returned. This process is called “a short-circuit evaluation” because it goes as short as possible from left to right.

This is clearly seen when the expression given as the second argument has a side effect like a variable assignment.

In the example below, `x` does not get assigned:

```
let x;

true || (x = 1);

alert(x); // undefined, because (x = 1) not evaluated
```

If, instead, the first argument is `false`, `||` evaluates the second one, thus running the assignment:

```
let x;

false || (x = 1);

alert(x); // 1
```

An assignment is a simple case. There may be side effects, that won’t show up if the evaluation doesn’t reach them.

As we can see, such a use case is a "shorter way of doing `if`". The first operand is converted to boolean. If it’s false, the second one is evaluated.

Most of time, it’s better to use a “regular” `if` to keep the code easy to understand, but sometimes this can be handy.

&& (AND)

The AND operator is represented with two ampersands `&&`:

```
result = a && b;
```

In classical programming, AND returns `true` if both operands are truthy and `false` otherwise:

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
```

An example with `if`:

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( 'The time is 12:30' );
}
```

Just as with OR, any value is allowed as an operand of AND:

```
if (1 && 0) { // evaluated as true && false
  alert( "won't work, because the result is falsy" );
}
```

AND “&&” finds the first falsy value

Given multiple AND’ed values:

```
result = value1 && value2 && value3;
```

The AND `&&` operator does the following:

- Evaluates operands from left to right.
- For each operand, converts it to a boolean. If the result is `false`, stops and returns the original value of that operand.
- If all operands have been evaluated (i.e. all were truthy), returns the last operand.

In other words, AND returns the first falsy value or the last value if none were found.

The rules above are similar to OR. The difference is that AND returns the first *falsy* value while OR returns the first *truthy* one.

Examples:

```
// if the first operand is truthy,
// AND returns the second operand:
```

```
3  alert( 1 && 0 ); // 0
4  alert( 1 && 5 ); // 5
5
6  // if the first operand is falsy,
7  // AND returns it. The second operand is ignored
8  alert( null && 5 ); // null
9  alert( 0 && "no matter what" ); // 0
```

We can also pass several values in a row. See how the first falsy one is returned:

```
1  alert( 1 && 2 && null && 3 ); // null
```

When all values are truthy, the last value is returned:

```
1  alert( 1 && 2 && 3 ); // 3, the last one
```

Precedence of AND `&&` is higher than OR `||`

The precedence of AND `&&` operator is higher than OR `||`.

So the code `a && b || c && d` is essentially the same as if the `&&` expressions were in parentheses: `(a && b) || (c && d)`.

Just like OR, the AND `&&` operator can sometimes replace `if`.

For instance:

```
1  let x = 1;
2
3  (x > 0) && alert( 'Greater than zero!' );
```

The action in the right part of `&&` would execute only if the evaluation reaches it. That is, only if `(x > 0)` is true.

So we basically have an analogue for:

```
1  let x = 1;
2
3  if (x > 0) {
4    alert( 'Greater than zero!' );
5  }
```

The variant with `&&` appears shorter. But `if` is more obvious and tends to be a little bit more readable.

So we recommend using every construct for its purpose: use `if` if we want if and use `&&` if we want AND.

! (NOT)

The boolean NOT operator is represented with an exclamation sign `!`.

The syntax is pretty simple:

```
result = !value;
```

The operator accepts a single argument and does the following:

1. Converts the operand to boolean type: `true/false`.
2. Returns the inverse value.

For instance:

```
alert( !true ); // false
alert( !0 ); // true
```

A double NOT `!!` is sometimes used for converting a value to boolean type:

```
alert( !!"non-empty string" ); // true
alert( !!null ); // false
```

That is, the first NOT converts the value to boolean and returns the inverse, and the second NOT inverses it again. In the end, we have a plain value-to-boolean conversion.

There's a little more verbose way to do the same thing – a built-in `Boolean` function:

```
alert( Boolean("non-empty string") ); // true
alert( Boolean(null) ); // false
```

The precedence of NOT `!` is the highest of all logical operators, so it always executes first, before `&&` or `||`.

Tasks

What's the result of OR?

importance: 5

What is the code below going to output?

```
alert( null || 2 || undefined );
```

solution

What's the result of OR'ed alerts?

importance: 3

What will the code below output?

```
1  alert( alert(1) || 2 || alert(3) );
```

solution

What is the result of AND?

importance: 5

What is this code going to show?

```
1  alert( 1 && null && 2 );
```

solution

What is the result of AND'ed alerts?

importance: 3

What will this code show?

```
1  alert( alert(1) && alert(2) );
```

solution

The result of OR AND OR

importance: 5

What will the result be?

```
1  alert( null || 2 && 3 || 4 );
```

solution

Check the range between

importance: 3

Write an “if” condition to check that `age` is between `14` and `90` inclusively.

“Inclusively” means that `age` can reach the edges `14` or `90`.

solution

Check the range outside

importance: 3

Write an `if` condition to check that `age` is NOT between 14 and 90 inclusively.

Create two variants: the first one using NOT `!`, the second one – without it.

solution

A question about "if"

importance: 5

Which of these `alert`s are going to execute?

What will the results of the expressions be inside `if(...)`?

```
1  if (-1 || 0) alert( 'first' );
2  if (-1 && 0) alert( 'second' );
3  if (null || -1 && 1) alert( 'third' );
```

solution

Check the login

importance: 3

Write the code which asks for a login with `prompt`.

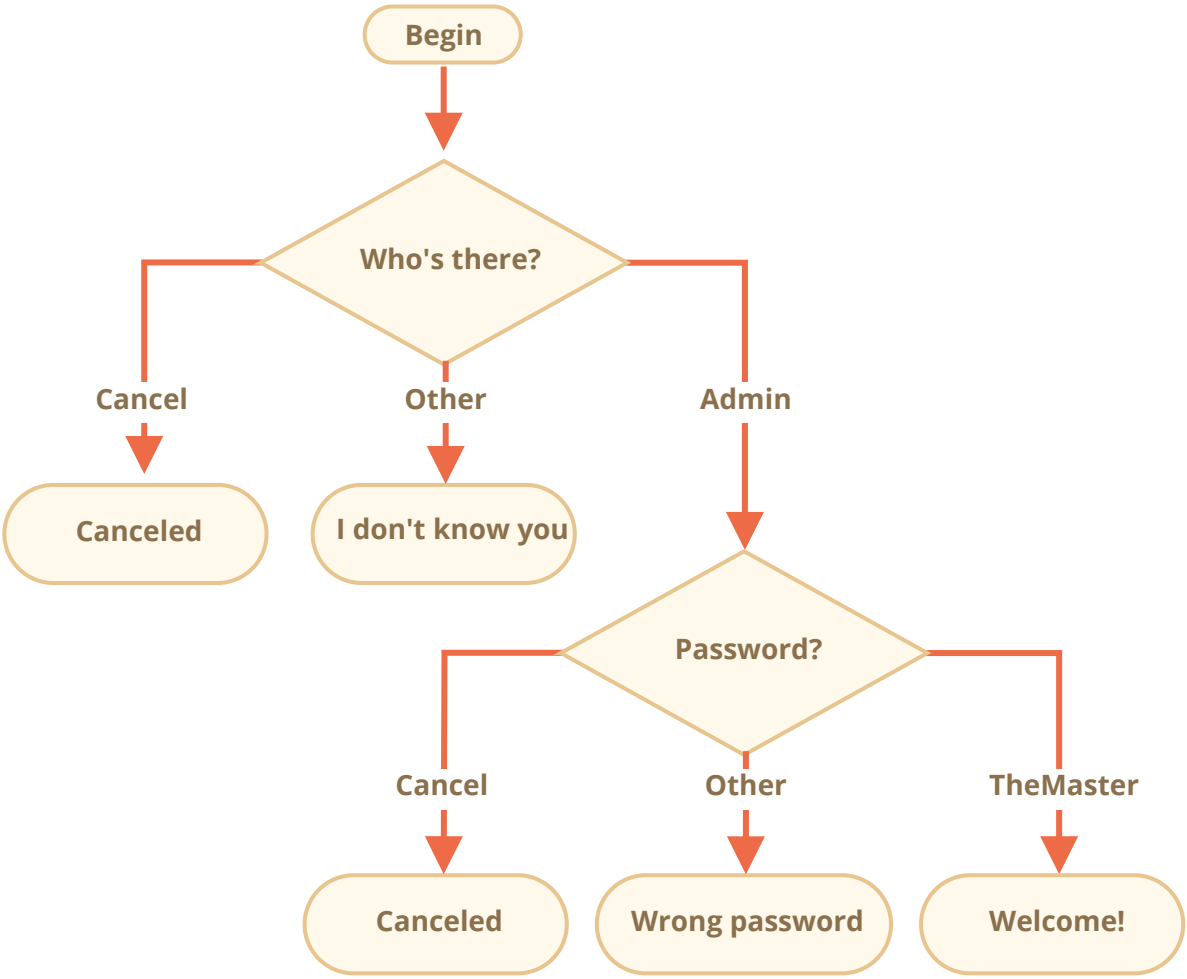
If the visitor enters `"Admin"`, then `prompt` for a password, if the input is an empty line or `Esc` – show “Canceled”, if it’s another string – then show “I don’t know you”.

The password is checked as follows:

- If it equals “TheMaster”, then show “Welcome!”,
- Another string – show “Wrong password”,

- For an empty string or cancelled input, show “Canceled”

The schema:



Please use nested `if` blocks. Mind the overall readability of the code.

Hint: passing an empty input to a prompt returns an empty string `''`. Pressing `ESC` during a prompt returns `null`.

Run the demo

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)