&& and || operators in Javascript

The logical and (&&) and or (||) are logical operators in JavaScript. Normally, you're using these operators on booleans:

```
true && true   // => true
true && false  // => false
true || true   // => true
true || false  // => true
```

However, can you use && and || with operands other than booleans? Turns out, you can!

This post explains in detail how && and || operators work in JavaScript.

Before jumping into how the operators work, let's start with the basic concepts of truthy and falsy.

1. **Falsy value**

Because JavaScript is a loosely typed language, logical operations can be performed on any type. The expressions like `1 && 2`, `null || undefined`, `'hello' && true` are weird, but still valid in JavaScript.

To perform logical operations on any type, JavaScript decides whether a particular value can be considered falsy (an equivalent of `false`) or truthy (an equivalent of `true`).

Falsy is a value for which `Boolean(value)` returns `false`. Falsy values in JavaScript are only `false`, `0`, `''`, `null`, `undefined` and `NaN`.

```
Boolean(false);     // => false
Boolean(0);         // => false
Boolean('');        // => false
Boolean(null);      // => false
```

```
Boolean(undefined); // => false
Boolean(NaN);       // => false
```

2. **Truthy value**

Truthy is a value for which `Boolean(value)` returns `true`. Saying it differently, truthy are the non-falsy values.

Examples of truthy values are `true`, `4`, `'Hello'`, `{ name: 'John' }` and everything else that's not falsy.

```
Boolean(true);            // => true
Boolean(4);               // => true
Boolean('Hello');         // => true
Boolean({ name: 'John' }); // => true
```

3. **How && operator works**

Now let's continue with proper learning of how `&&` operator works. Note that the operator works in terms of truthy and fasly, rather than `true` and `false`.

Here's the syntax of the `&&` operator in a chain:

```
operand1 && operand2 && ... && operandN
```

The expression is evaluated as follows:

Starting from left and moving to the right, return the first operand that is falsy. If no falsy operand was found, return the latest operand.
Let's see how the algorithm works in a few examples.

When the operands are booleans, it's simple:

```
true && false && true; // => false
```

The evaluation starts from left and moves to the right. The first `true` operand is passed. However, the second operand `false` is a falsy value, and evaluation stops. `false`

becomes the result of the entire expression. The third operand `true` is not evaluated.

When operands are numbers:

```
3 && 1 && 0 && 10; // => 0
```

The evaluation is performed from left to right. `3` and `1` are passed because they are truthy. But the evaluation stops at the third operand `0` since it's falsy. `0` becomes the result of the entire expression. The fourth operand `10` is not evaluated.

A slighly more complex example with different types:

```
true && 1 && { name: 'John' }
```

Again, from left to right, the operands are checked for falsy. No operand is falsy, so the last operand is returned. The evaluation result is `{ name: 'John' }`.

3.1 Skipping operands

The `&&` evaluation algorithm is optimal because the evaluation of operands stops as soon as a falsy value is encountered.

Let's consider an example:

```
const person = null;
person && person.address && person.address.street; // => null
```

The evaluation of the long logical expression `person && person.address && person.address.street` stops right at the first operand `person` (which is `null` - i.e. falsy). `person.address` and `person.address.street` operands are skipped.

4. **How || operator works**

Here's a generalized syntax of || operator in chain:

```
operand1 || operand2 || ... || operandN
```

The evaluation of || happens this way:

Starting from left and moving to the right, return the first operand that is truthy. If no truthy operand was found, return the latest operand.
|| works the same way as &&, with the only difference that || stops evaluation when encounters a truthy operand.

Let's study some || examples.

A simple expression having 2 booleans:

```
true || false; // => true
```

The evaluation starts from left and moves to the right. Luckily, the first operand `true` is a truthy value, so the whole expression evaluates to `true`. The second operand `false` is not checked.

Having some numbers as operands:

```
0 || -1 || 10; // => -1
```

The first operand `0` is falsy, so the evaluation continues. The second argument `-1` is already truthy, so the evaluation stops, and the result is `-1`.

4.1 Default value when accessing properties

You can use a side-effect of the || evaluation to access an object property providing a default value when the property is missing.

For example, let's access the properties `name` and `job` of the `person` object. When the property is missing, simply

default to a string `'Unknown'`. Here's how you could use `||` operator to achieve it:

```
const person = {
  name: 'John'
};
person.name || 'Unknown'; // => 'John'
person.job  || 'Unknown'; // => 'Unknown'
```

Let's look at the expression `person.name || 'Unknown'`. Because the first operand `person.name` is `'John'` (a truthy value), the evaluation early exists with the truthy value as a result. The expression evaluates to `'John'`.

`person.job || 'Unknown'` operates differently. `person.job` is `undefined`, so the expression is the same as `undefined || 'Unknown'`. The `||` evaluation algorithm says that the expression should return the first operand that is truthy, which in this case is `'Unknown'`.

5. **Summary**

Because JavaScript is a loosely typed language, the operands of `&&` and `||` can be of any type.

The concepts of falsy and truthy are handy to deal with types conversion within logical operators. Falsy values are `false`, `0`, `''`, `null`, `undefined` and `NaN`, while the rest of values are truthy.

`&&` operator evaluates the operands from left to right and returns the first falsy value encountered. If no operand is falsy, the latest operand is returned.

The same way `||` operator evaluates the operands from left to right but returns the first truthy value encountered. If no truthy value was found, the latest operand is returned.

While `&&` and `||` evaluation algorithms seem weird at first, in my opinion, they’re quite efficient. The algorithms perform early exit, which is a good performance optimization.

In terms of usage, I recommend to stick to booleans as operands for both `&&` and `||`, and avoid other types if possible. Logical expressions that operate only on booleans are easier to understand.