

Lesson - 15

Logical Operators
Loops.



Lesson Plan

- Logical Operators
- Loops
- While
- For
- Switch Statement

Logical Operators

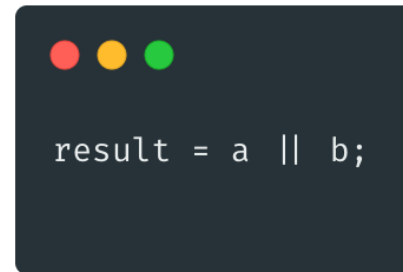
There are three logical operators in JavaScript: `||` (OR), `&&` (AND), `!` (NOT).

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.

|| (OR)

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

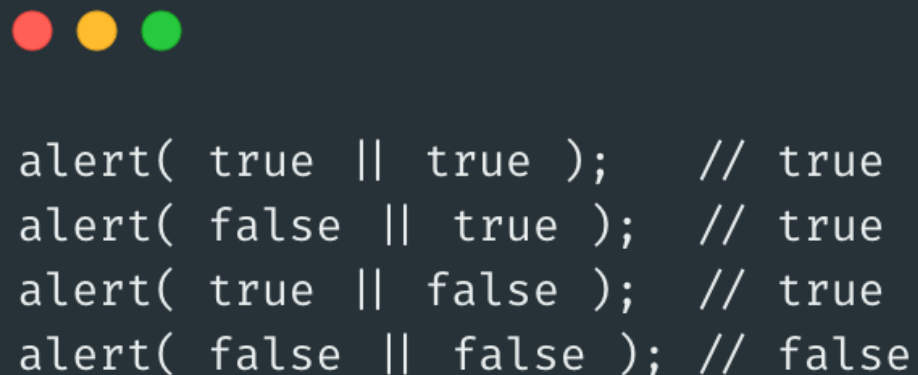
A small code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the code `result = a || b;` in a light-colored monospace font.

```
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:

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains four lines of JavaScript code demonstrating the OR operator with true and false values, each followed by a comment showing the result.


```
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`.

|| (OR)


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`.



```
let hour = 9;  
  
if (hour < 10 || hour > 18) {  
  alert( 'The office is closed.' );  
}
```

|| (OR)

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.

|| (OR)

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”.

|| (OR)

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.

|| (OR)

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
```

|| (OR)

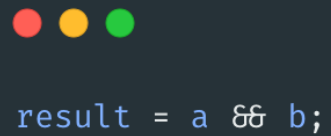
An assignment is a simple case. There may be side effects, that won't show up if the evaluation doesn't reach them.

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.

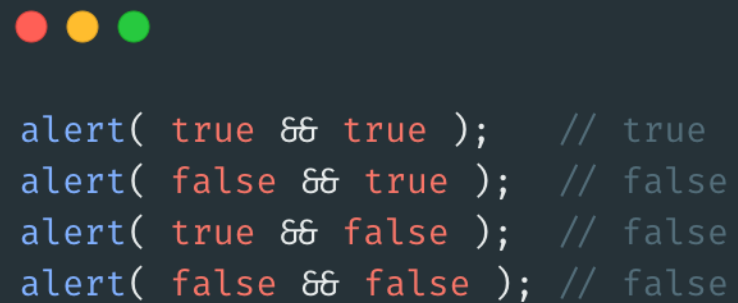
&& (AND)

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



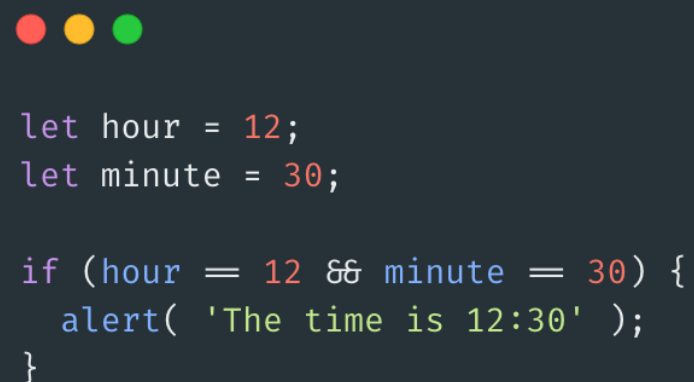
```
result = a && b;
```

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' );
}
```

&& (AND)

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 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.

&& (AND)

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.



```
// if the first operand is truthy,  
// AND returns the second operand:  
alert( 1 && 0 ); // 0  
alert( 1 && 5 ); // 5  
  
// if the first operand is falsy,  
// AND returns it. The second operand is ignored  
alert( null && 5 ); // null  
alert( 0 && "no matter what" ); // 0
```

&&(AND)

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



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

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

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


i 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)`.

&& (AND)

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



```
let x = 1;

(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:

let x = 1;

if (x > 0) {
    alert( 'Greater than zero!' );
}
```

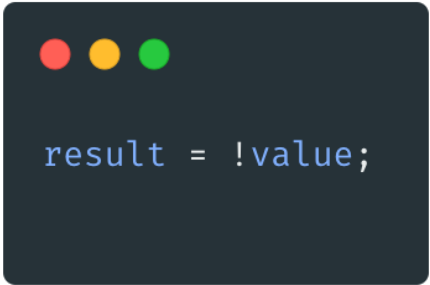
The variant with && appears shorter. But if is more obvious and tends to be a little bit more readable.

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 `!`.

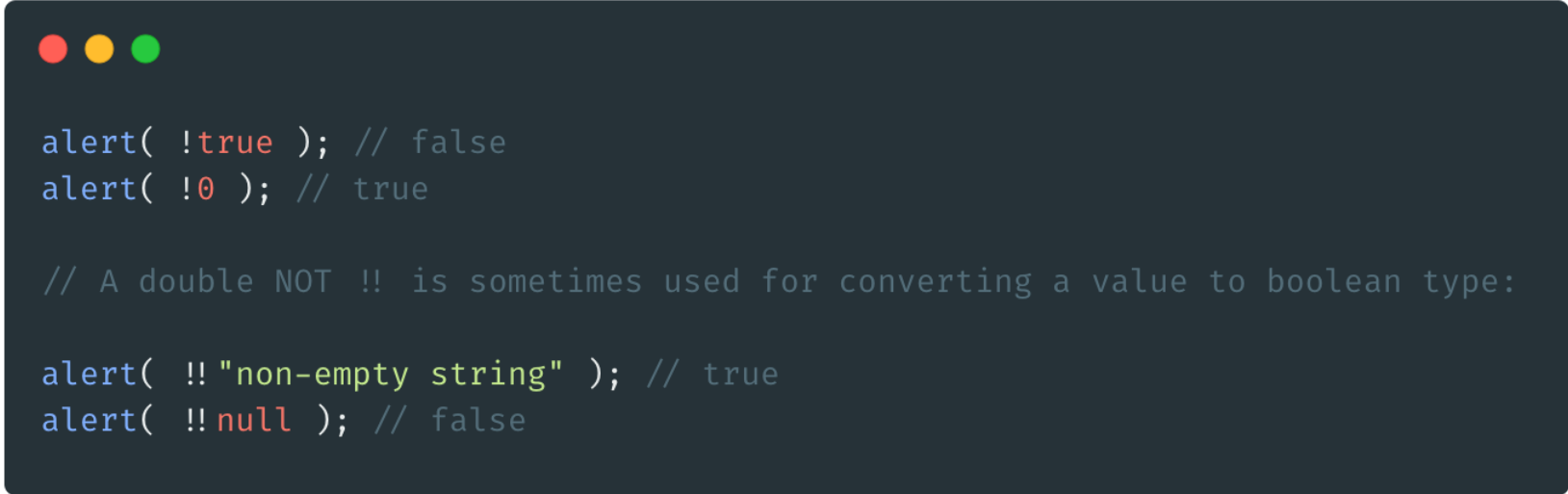
The syntax is pretty simple:



```
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.



```
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
```

! (NOT)

That is, the first NOT converts the value to boolean and returns the inverse, and the second NOT inverts 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 `||`.

Loops. While and for(;;)

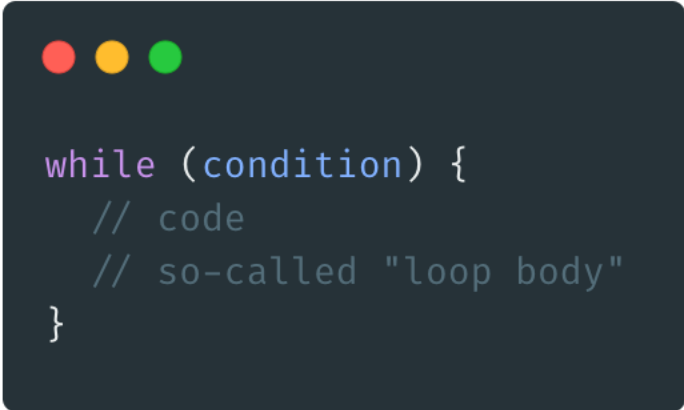
We often need to repeat actions.

For example, outputting goods from a list one after another or just running the same code for each number from 1 to 10.

Loops are a way to repeat the same code multiple times.

While loops

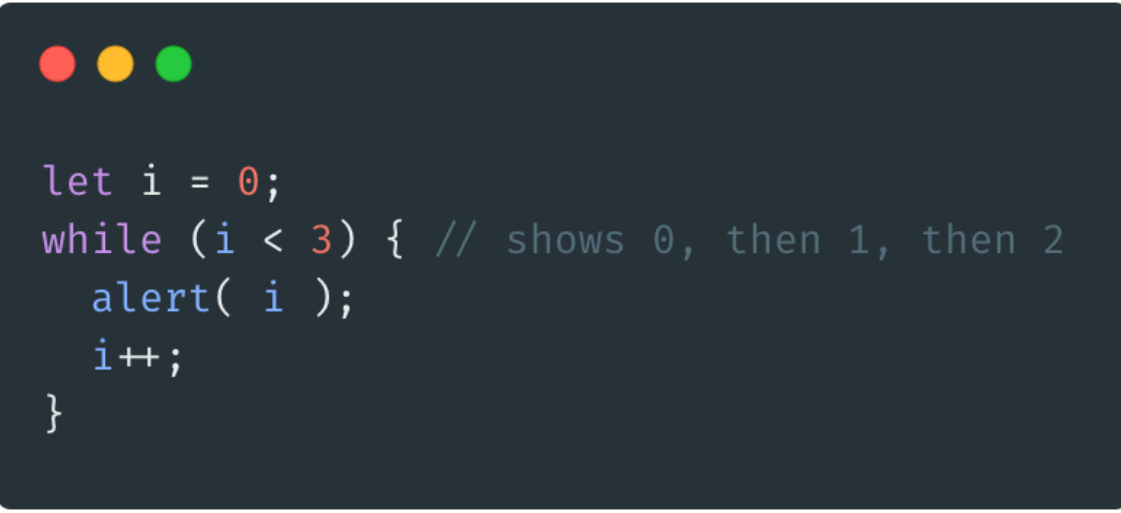
The `while` loop has the following syntax:



```
while (condition) {  
  // code  
  // so-called "loop body"  
}
```

While the `condition` is truthy, the `code` from the loop body is executed.

For instance, the loop below outputs `i` while `i < 3`:



```
let i = 0;  
while (i < 3) { // shows 0, then 1, then 2  
  alert( i );  
  i++;  
}
```

While loops

A single execution of the loop body is called *an iteration*. The loop in the example above makes three iterations.

If `i++` was missing from the example above, the loop would repeat (in theory) forever. In practice, the browser provides ways to stop such loops, and in server-side JavaScript, we can kill the process.

Any expression or variable can be a loop condition, not just comparisons: the condition is evaluated and converted to a boolean by `while`.

For instance, a shorter way to write `while (i !== 0)` is `while (i)`:

```
let i = 3;
while (i) { // when i becomes 0, the condition becomes falsy, and the loop stops
  alert( i );
  i--;
}
```

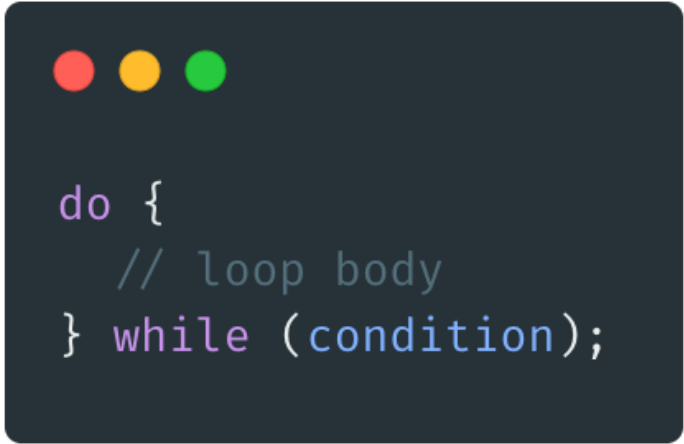
i Curly braces are not required for a single-line body

If the loop body has a single statement, we can omit the curly braces `{...}`:

```
1 let i = 3;
2 while (i) alert(i--);
```

Do While loops

The condition check can be moved *below* the loop body using the `do...while` syntax:



```
do {  
    // loop body  
} while (condition);
```

The loop will first execute the body, then check the condition, and, while it's truthy, execute it again and again.



```
let i = 0;  
do {  
    alert( i );  
    i++;  
} while (i < 3);
```

This form of syntax should only be used when you want the body of the loop to execute **at least once** regardless of the condition being truthy. Usually, the other form is preferred: `while(...) {...}`.

The for(;;) loop


The `for` loop is more complex, but it's also the most commonly used loop.



```
for (begin; condition; step) {  
    // ... loop body ...  
}
```


The for loop

Let's learn the meaning of these parts by example. The loop below runs `alert(i)` for `i` from `0` up to (but not including) `3`:



```
for (let i = 0; i < 3; i++) { // shows 0, then 1, then 2
  alert(i);
}
```

Let's examine the `for` statement part-by-part:

part

begin	<code>i = 0</code>	Executes once upon entering the loop.
condition	<code>i < 3</code>	Checked before every loop iteration. If false, the loop stops.
body	<code>alert(i)</code>	Runs again and again while the condition is truthy.
step	<code>i++</code>	Executes after the body on each iteration.

The for loop

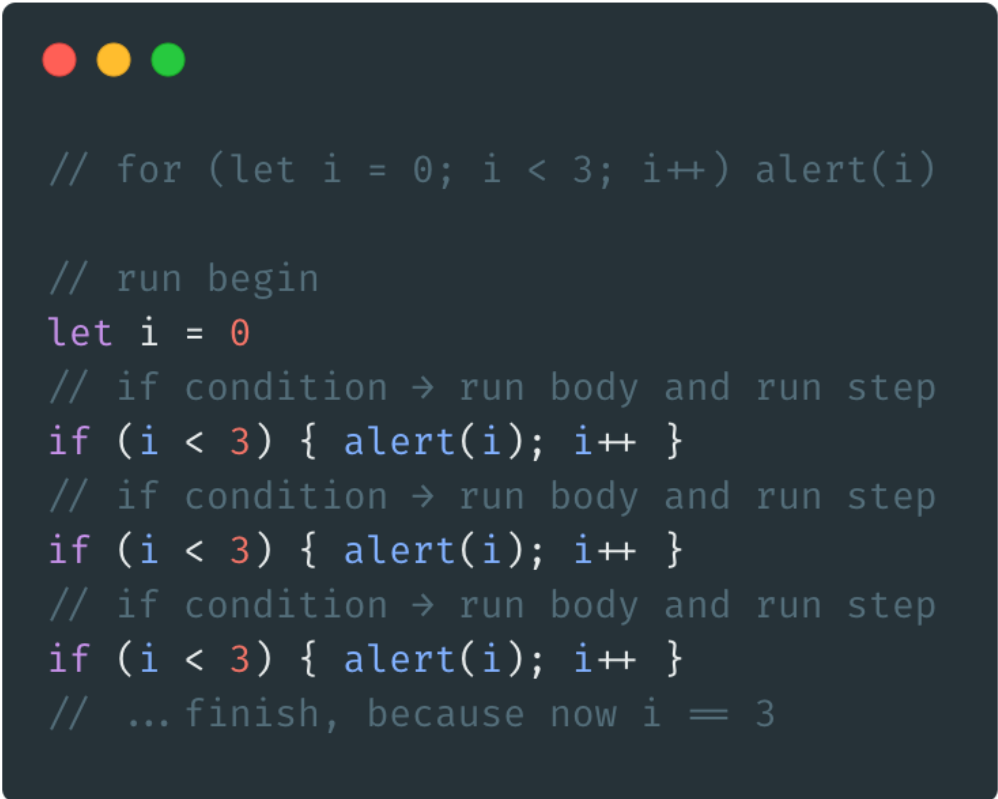
The general loop algorithm works like this:

```
1 Run begin
2 → (if condition → run body and run step)
3 → (if condition → run body and run step)
4 → (if condition → run body and run step)
5 → ...
```

That is, `begin` executes once, and then it iterates: after each `condition` test, `body` and `step` are executed.

If you are new to loops, it could help to go back to the example and reproduce how it runs step-by-step on a piece of paper.

Here's exactly what happens in our case:



```
// for (let i = 0; i < 3; i++) alert(i)

// run begin
let i = 0
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// if condition → run body and run step
if (i < 3) { alert(i); i++ }
// ...finish, because now i = 3
```

The for loop

Inline variable declaration

Here, the “counter” variable `i` is declared right in the loop. This is called an “inline” variable declaration. Such variables are visible only inside the loop.

```
1 for (let i = 0; i < 3; i++) {  
2   alert(i); // 0, 1, 2  
3 }  
4 alert(i); // error, no such variable
```

Instead of defining a variable, we could use an existing one:

```
1 let i = 0;  
2  
3 for (i = 0; i < 3; i++) { // use an existing variable  
4   alert(i); // 0, 1, 2  
5 }  
6  
7 alert(i); // 3, visible, because declared outside of the loop
```


The for loop

Skipping parts

Any part of `for` can be skipped.

For example, we can omit `begin` if we don't need to do anything at the loop start.

Like here



```
let i = 0; // we have i already declared and assigned

for (; i < 3; i++) { // no need for "begin"
  alert( i ); // 0, 1, 2
}

// We can also remove the step part:

let i = 0;

for (; i < 3;) {
  alert( i++ );
}
```

The for loop

We can actually remove everything, creating an infinite loop:




```
for (;;) {  
    // repeats without limits  
}
```

Breaking the loop

Normally, a loop exits when its condition becomes falsy.

But we can force the exit at any time using the special `break` directive.

For example, the loop below asks the user for a series of numbers, “breaking” when no number is entered:



```
let sum = 0;

while (true) {

    let value = +prompt("Enter a number", '');

    if (!value) break; // (*)

    sum += value;

}

alert( 'Sum: ' + sum );
```

The `break` directive is activated at the line `(*)` if the user enters an empty line or cancels the input. It stops the loop immediately, passing control to the first line after the loop. Namely, `alert`.

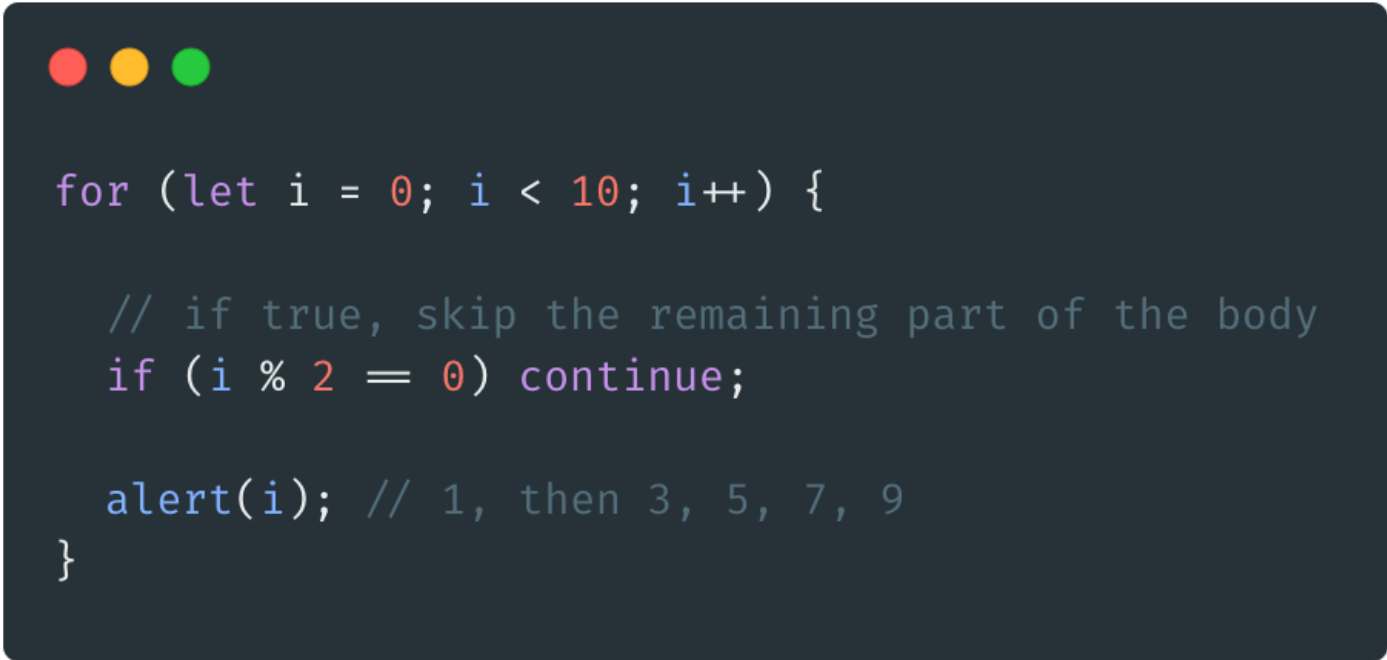
The combination “infinite loop + `break` as needed” is great for situations when a loop’s condition must be checked not in the beginning or end of the loop, but in the middle or even in several places of its body.

Continue on next iteration

The `continue` directive is a “lighter version” of `break`. It doesn’t stop the whole loop. Instead, it stops the current iteration and forces the loop to start a new one (if the condition allows).

We can use it if we’re done with the current iteration and would like to move on to the next one.

The loop below uses `continue` to output only odd values:



```
for (let i = 0; i < 10; i++) {  
  
    // if true, skip the remaining part of the body  
    if (i % 2 == 0) continue;  
  
    alert(i); // 1, then 3, 5, 7, 9  
}
```

For even values of `i`, the `continue` directive stops executing the body and passes control to the next iteration of `for` (with the next number). So the `alert` is only called for odd values.

The for loop

The `continue` directive helps decrease nesting

A loop that shows odd values could look like this:

```
1 for (let i = 0; i < 10; i++) {  
2  
3   if (i % 2) {  
4     alert( i );  
5   }  
6  
7 }
```

From a technical point of view, this is identical to the example above. Surely, we can just wrap the code in an `if` block instead of using `continue`.

But as a side-effect, this created one more level of nesting (the `alert` call inside the curly braces). If the code inside of `if` is longer than a few lines, that may decrease the overall readability.

The for loop

⚠ No `break/continue` to the right of `'?'`

Please note that syntax constructs that are not expressions cannot be used with the ternary operator `?`. In particular, directives such as `break/continue` aren't allowed there.

For example, if we take this code:

```
1  if (i > 5) {  
2    alert(i);  
3  } else {  
4    continue;  
5  }
```

...and rewrite it using a question mark:

```
1  (i > 5) ? alert(i) : continue; // continue isn't allowed here
```

...it stops working: there's a syntax error.

This is just another reason not to use the question mark operator `?` instead of `if`.

Labels for break

Labels for break/continue

Sometimes we need to break out from multiple nested loops at once.

For example, in the code below we loop over `i` and `j`, prompting for the coordinates `(i, j)` from `(0,0)` to `(2,2)`:

```
1 for (let i = 0; i < 3; i++) {  
2  
3   for (let j = 0; j < 3; j++) {  
4  
5     let input = prompt(`Value at coords (${i},${j})`, '');  
6  
7     // what if we want to exit from here to Done (below)?  
8   }  
9 }  
10  
11 alert('Done!');
```

We need a way to stop the process if the user cancels the input.

The ordinary `break` after `input` would only break the inner loop. That's not sufficient—labels, come to the rescue!

Labels for break

A *label* is an identifier with a colon before a loop:

```
1  labelName: for (...) {  
2    ...  
3  }
```

The `break <labelName>` statement in the loop below breaks out to the label:

```
1  outer: for (let i = 0; i < 3; i++) {  
2  
3    for (let j = 0; j < 3; j++) {  
4  
5        let input = prompt(`Value at coords (${i},${j})`, '');  
6  
7        // if an empty string or canceled, then break out of both loops  
8        if (!input) break outer; // (*)  
9  
10       // do something with the value...  
11    }  
12 }  
13 alert('Done!');
```

In the code above, `break outer` looks upwards for the label named `outer` and breaks out of that loop.

So the control goes straight from `(*)` to `alert('Done!')`.

Summary

We covered 3 types of loops:

- `while` – The condition is checked before each iteration.
- `do..while` – The condition is checked after each iteration.
- `for (;;)` – The condition is checked before each iteration, additional settings available.

To make an “infinite” loop, usually the `while(true)` construct is used. Such a loop, just like any other, can be stopped with the `break` directive.

If we don't want to do anything in the current iteration and would like to forward to the next one, we can use the `continue` directive.

`break/continue` support labels before the loop. A label is the only way for `break/continue` to escape a nested loop to go to an outer one.

The Switch

A `switch` statement can replace multiple `if` checks.

It gives a more descriptive way to compare a value with multiple variants.

The syntax

The `switch` has one or more `case` blocks and an optional default.

It looks like this:

```
1 switch(x) {  
2   case 'value1': // if (x === 'value1')  
3     ...  
4     [break]  
5  
6   case 'value2': // if (x === 'value2')  
7     ...  
8     [break]  
9  
10  default:  
11    ...  
12    [break]  
13 }
```

The switch

- The value of `x` is checked for a strict equality to the value from the first `case` (that is, `value1`) then to the second (`value2`) and so on.
- If the equality is found, `switch` starts to execute the code starting from the corresponding `case`, until the nearest `break` (or until the end of `switch`).
- If no case is matched then the `default` code is executed (if it exists).

An example

An example of `switch` (the executed code is highlighted):

```
1  let a = 2 + 2;  
2  
3  switch (a) {  
4    case 3:  
5      alert( 'Too small' );  
6      break;  
7    case 4:  
8      alert( 'Exactly!' );  
9      break;  
10   case 5:  
11     alert( 'Too large' );  
12     break;  
13   default:  
14     alert( "I don't know such values" );  
15 }
```

Here the `switch` starts to compare `a` from the first `case` variant that is `3`. The match fails.

Then `4`. That's a match, so the execution starts from `case 4` until the nearest `break`.

The switch

If there is no `break` then the execution continues with the next `case` without any checks.

An example without `break`:

```
1 let a = 2 + 2;
2
3 switch (a) {
4   case 3:
5     alert( 'Too small' );
6   case 4:
7     alert( 'Exactly!' );
8   case 5:
9     alert( 'Too big' );
10  default:
11    alert( "I don't know such values" );
12 }
```

In the example above we'll see sequential execution of three `alert` s:

```
1 alert( 'Exactly!' );
2 alert( 'Too big' );
3 alert( "I don't know such values" );
```

The switch

Grouping of "case"

Several variants of `case` which share the same code can be grouped.

For example, if we want the same code to run for `case 3` and `case 5`:

```
1  let a = 3;
2
3  switch (a) {
4    case 4:
5      alert('Right!');
6      break;
7
8    case 3: // (*) grouped two cases
9    case 5:
10     alert('Wrong!');
11     alert("Why don't you take a math class?");
12     break;
13
14   default:
15     alert('The result is strange. Really.');
```

Now both `3` and `5` show the same message.

The ability to "group" cases is a side-effect of how `switch/case` works without `break`. Here the execution of `case 3` starts from the line `(*)` and goes through `case 5`, because there's no `break`.

The switch

Type matters

Let's emphasize that the equality check is always strict. The values must be of the same type to match.

For example, let's consider the code:

```
1 let arg = prompt("Enter a value?");
2 switch (arg) {
3   case '0':
4   case '1':
5     alert( 'One or zero' );
6     break;
7
8   case '2':
9     alert( 'Two' );
10    break;
11
12   case 3:
13     alert( 'Never executes!' );
14     break;
15   default:
16     alert( 'An unknown value' );
17 }
```

1. For `0`, `1`, the first `alert` runs.
2. For `2` the second `alert` runs.
3. But for `3`, the result of the `prompt` is a string `"3"`, which is not strictly equal `===` to the number `3`. So we've got a dead code in `case 3`! The `default` variant will execute.

Learning Resources

1. Logical Operators:

1. <https://javascript.info/logical-operators>
2. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_Operators

2. Loops:

1. <https://javascript.info/while-for>
2. https://www.w3schools.com/js/js_loop_for.asp
3. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration

3. Switch statement:

1. <https://javascript.info/switch>
2. https://www.w3schools.com/js/js_switch.asp

Home Work

1. Create a new project in GitHub lesson-15-hw
2. Create a index.html file
3. Create a script.js file and attach it to index.html
4. Output even(divisible by 2) numbers in for loop
 1. Use the for loop to output(console.log) even numbers from 2 to 10.
5. Rewrite following code from **for** to **while**:
 1. Rewrite the code changing the `for` loop to `while` without altering its behavior (the output should stay same).

```
for (let i = 0; i < 3; i++) {  
  alert( `number ${i}!` );  
}
```
6. Write a for loop that calculates [factorial](#) of a number. For example declare a variable **number** which equals to 5, and a **result** variable which equals to 1(starting point), the result of for loop should be $1 * 2 * 3 * 4 * 5 = 120$. Print to console.log result of

Home Work pt2

1. Rewrite the code below using a single switch statement:

```
let a = +prompt('a?', '');

if (a == 0) {
  alert( 0 );
}

if (a == 1) {
  alert( 1 );
}

if (a == 2 || a == 3) {
  alert( '2,3' );
}
```

Home Work pt3

1. Write the code using `if...else` which would correspond to the following switch:

```
switch (browser) {  
  case 'Edge':  
    alert( "You've got the Edge!" );  
    break;  
  
  case 'Chrome':  
  case 'Firefox':  
  case 'Safari':  
  case 'Opera':  
    alert( 'Okay we support these browsers too' );  
    break;  
  
  default:  
    alert( 'We hope that this page looks ok!' );  
}
```