**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.

**The for…of and for…in loops**

A small announcement for advanced readers.

This article covers only basic loops: while, do..while and for(..;..;..).

If you came to this article searching for other types of loops, here are the pointers:

* See [for…in](https://javascript.info/object#forin) to loop over object properties.
* See [for…of](https://javascript.info/array#loops) and [iterables](https://javascript.info/iterable) for looping over arrays and iterable objects.

Otherwise, please read on.

**[The “while” loop](https://javascript.info/while-for" \l "the-while-loop)**

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++;

}

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--;

}

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

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

let i = 3;

while (i) alert(i--);

**[The “do…while” loop](https://javascript.info/while-for" \l "the-do-while-loop)**

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.

For example:

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](https://javascript.info/while-for" \l "the-for-loop)**

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

It looks like this:

for (begin; condition; step) {

// ... loop body ...

}

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

The general loop algorithm works like this:

Run begin

→ (if condition → run body and run step)

→ (if condition → run body and run step)

→ (if condition → run body and run step)

→ ...

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

**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.

for (let i = 0; i < 3; i++) {

alert(i); // 0, 1, 2

}

alert(i); // error, no such variable

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

let i = 0;

for (i = 0; i < 3; i++) { // use an existing variable

alert(i); // 0, 1, 2

}

alert(i); // 3, visible, because declared outside of the loop

**[Skipping parts](https://javascript.info/while-for" \l "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++ );

}

This makes the loop identical to while (i < 3).

We can actually remove everything, creating an infinite loop:

for (;;) {

// repeats without limits

}

Please note that the two for semicolons ; must be present. Otherwise, there would be a syntax error.

**[Breaking the loop](https://javascript.info/while-for" \l "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 to the next iteration](https://javascript.info/while-for" \l "continue)**

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 continue directive helps decrease nesting**

A loop that shows odd values could look like this:

for (let i = 0; i < 10; i++) {

if (i % 2) {

alert( i );

}

}

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.

**No break/continue to the right side 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:

if (i > 5) {

alert(i);

} else {

continue;

}

…and rewrite it using a question mark:

(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/continue](https://javascript.info/while-for" \l "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):

for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`Value at coords (${i},${j})`, '');

// what if we want to exit from here to Done (below)?

}

}

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!

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

labelName: for (...) {

...

}

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

outer: for (let i = 0; i < 3; i++) {

for (let j = 0; j < 3; j++) {

let input = prompt(`Value at coords (${i},${j})`, '');

// if an empty string or canceled, then break out of both loops

if (!input) break outer; // (\*)

// do something with the value...

}

}

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!').

We can also move the label onto a separate line:

outer:

for (let i = 0; i < 3; i++) { ... }

The continue directive can also be used with a label. In this case, code execution jumps to the next iteration of the labeled loop.

**Labels do not allow to “jump” anywhere**

Labels do not allow us to jump into an arbitrary place in the code.

For example, it is impossible to do this:

break label; // jump to the label below (doesn't work)

label: for (...)

A break directive must be inside a code block. Technically, any labelled code block will do, e.g.:

label: {

// ...

break label; // works

// ...

}

…Although, 99.9% of the time break is used inside loops, as we’ve seen in the examples above.

A continue is only possible from inside a loop.

**[Summary](https://javascript.info/while-for" \l "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.

[**Tasks**](https://javascript.info/while-for#tasks)

**[Last loop value](https://javascript.info/while-for" \l "last-loop-value)**

**importance: 3**

What is the last value alerted by this code? Why?

let i = 3;

while (i) {

alert( i-- );

}

**solution**

The answer: 1.

let i = 3;

while (i) {

alert( i-- );

}

Every loop iteration decreases i by 1. The check while(i) stops the loop when i = 0.

Hence, the steps of the loop form the following sequence (“loop unrolled”):

let i = 3;

alert(i--); // shows 3, decreases i to 2

alert(i--) // shows 2, decreases i to 1

alert(i--) // shows 1, decreases i to 0

// done, while(i) check stops the loop

**[Which values does the while loop show?](https://javascript.info/while-for" \l "which-values-does-the-while-loop-show)**

**importance: 4**

For every loop iteration, write down which value it outputs and then compare it with the solution.

Both loops alert the same values, or not?

1. The prefix form ++i:
2. let i = 0;

while (++i < 5) alert( i );

1. The postfix form i++
2. let i = 0;

while (i++ < 5) alert( i );

**solution**

The task demonstrates how postfix/prefix forms can lead to different results when used in comparisons.

1. **From 1 to 4**
2. let i = 0;

while (++i < 5) alert( i );

The first value is i = 1, because ++i first increments i and then returns the new value. So the first comparison is 1 < 5 and the alert shows 1.

Then follow 2, 3, 4… – the values show up one after another. The comparison always uses the incremented value, because ++ is before the variable.

Finally, i = 4 is incremented to 5, the comparison while(5 < 5) fails, and the loop stops. So 5 is not shown.

1. **From 1 to 5**
2. let i = 0;

while (i++ < 5) alert( i );

The first value is again i = 1. The postfix form of i++ increments i and then returns the *old* value, so the comparison i++ < 5 will use i = 0 (contrary to ++i < 5).

But the alert call is separate. It’s another statement which executes after the increment and the comparison. So it gets the current i = 1.

Then follow 2, 3, 4…

Let’s stop on i = 4. The prefix form ++i would increment it and use 5 in the comparison. But here we have the postfix form i++. So it increments i to 5, but returns the old value. Hence the comparison is actually while(4 < 5) – true, and the control goes on to alert.

The value i = 5 is the last one, because on the next step while(5 < 5) is false.

**[Which values get shown by the "for" loop?](https://javascript.info/while-for" \l "which-values-get-shown-by-the-for-loop)**

**importance: 4**

For each loop write down which values it is going to show. Then compare with the answer.

Both loops alert same values or not?

1. The postfix form:

for (let i = 0; i < 5; i++) alert( i );

1. The prefix form:

for (let i = 0; i < 5; ++i) alert( i );

**solution**

**The answer: from 0 to 4 in both cases.**

for (let i = 0; i < 5; ++i) alert( i );

for (let i = 0; i < 5; i++) alert( i );

That can be easily deducted from the algorithm of for:

1. Execute once i = 0 before everything (begin).
2. Check the condition i < 5
3. If true – execute the loop body alert(i), and then i++

The increment i++ is separated from the condition check (2). That’s just another statement.

The value returned by the increment is not used here, so there’s no difference between i++ and ++i.

**[Output even numbers in the loop](https://javascript.info/while-for" \l "output-even-numbers-in-the-loop)**

**importance: 5**

Use the for loop to output even numbers from 2 to 10.

[Run the demo](https://javascript.info/while-for)

**solution**

for (let i = 2; i <= 10; i++) {

if (i % 2 == 0) {

alert( i );

}

}

We use the “modulo” operator % to get the remainder and check for the evenness here.

**[Replace "for" with "while"](https://javascript.info/while-for" \l "replace-for-with-while)**

**importance: 5**

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}!` );

}

**solution**

let i = 0;

while (i < 3) {

alert( `number ${i}!` );

i++;

}

**[Repeat until the input is correct](https://javascript.info/while-for" \l "repeat-until-the-input-is-correct)**

**importance: 5**

Write a loop which prompts for a number greater than 100. If the visitor enters another number – ask them to input again.

The loop must ask for a number until either the visitor enters a number greater than 100 or cancels the input/enters an empty line.

Here we can assume that the visitor only inputs numbers. There’s no need to implement a special handling for a non-numeric input in this task.

[Run the demo](https://javascript.info/while-for)

**solution**

let num;

do {

num = prompt("Enter a number greater than 100?", 0);

} while (num <= 100 && num);

The loop do..while repeats while both checks are truthy:

1. The check for num <= 100 – that is, the entered value is still not greater than 100.
2. The check && num is false when num is null or an empty string. Then the while loop stops too.

P.S. If num is null then num <= 100 is true, so without the 2nd check the loop wouldn’t stop if the user clicks CANCEL. Both checks are required.

**[Output prime numbers](https://javascript.info/while-for" \l "output-prime-numbers)**

**importance: 3**

An integer number greater than 1 is called a [prime](https://en.wikipedia.org/wiki/Prime_number) if it cannot be divided without a remainder by anything except 1 and itself.

In other words, n > 1 is a prime if it can’t be evenly divided by anything except 1 and n.

For example, 5 is a prime, because it cannot be divided without a remainder by 2, 3 and 4.

**Write the code which outputs prime numbers in the interval from 2 to n.**

For n = 10 the result will be 2,3,5,7.

P.S. The code should work for any n, not be hard-tuned for any fixed value.

**solution**

There are many algorithms for this task.

Let’s use a nested loop:

For each i in the interval {

check if i has a divisor from 1..i

if yes => the value is not a prime

if no => the value is a prime, show it

}

The code using a label:

let n = 10;

nextPrime:

for (let i = 2; i <= n; i++) { // for each i...

for (let j = 2; j < i; j++) { // look for a divisor..

if (i % j == 0) continue nextPrime; // not a prime, go next i

}

alert( i ); // a prime

}

There’s a lot of space to optimize it. For instance, we could look for the divisors from 2 to square root of i. But anyway, if we want to be really efficient for large intervals, we need to change the approach and rely on advanced maths and complex algorithms like [Quadratic sieve](https://en.wikipedia.org/wiki/Quadratic_sieve), [General number field sieve](https://en.wikipedia.org/wiki/General_number_field_sieve) etc.

**[Comments](https://javascript.info/while-for" \l "comments)**

read this before commenting…