JavaScript

basic hello word in JavaScript in code

JavaScript programs can be inserted almost anywhere into an HTML document using the `<script>` tag

```
<!DOCTYPE HTML>
<html>
<body>
  <p>Before the script...</p>
  <script>
    alert( 'Hello, world!' );
  </script>
  <p>...After the script.</p>
</body>
</html>
```

Comments before and after scripts.
In really ancient books and guides, you may find comments inside `<script>` tags, like this:
What does `type="module"` mean?

```
<script type="module" src="/main.js"></script>
```

This tells the browser:
Use ES6 Modules: This enables import/export syntax in your JavaScript files.
Scope: Code in modules is scoped (like functions), so variables won't leak into the global scope.
Runs in strict mode by default.
Statements
Statements are syntax constructs and commands that perform actions.
We've already seen a statement, alert('Hello, world!'), which shows the message "Hello, world!".
We can have as many statements in our code as we want. Statements can be separated with a semicolon.
For example, here we split "Hello World" into two alerts:

```
alert('Hello'); alert('World');
```

The modern mode, "use strict"

`use strict`
The directive looks like a string: `use strict` or 'use strict'. When it is located at the top of a script, the whole script works the “modern” way.
In main file `use strict`

Data Types

JavaScript has 8 basic data types, divided into primitive and non-primitive types.

1. Primitive Data Types (7 total)
   These types represent single values and are immutable.
   a. `number`

- Used for all kinds of numeric values: integers and floating-point numbers.
- Example: `42`, `3.14`
- Note: Integers are limited by ±(2^{53} - 1).
  b. `bigint`
- Used for integers of arbitrary length.
- Example: `1234567890123456789012345678901234567890n`
- Useful for very large numbers beyond `number` limits.
  c. `string`
- Represents a sequence of characters.
- Can be enclosed in single quotes (`'...'`), double quotes (`"..."`), or backticks (`...`).
- Example: `"Hello"`, `'World'`, `` `Hi ${name}` ``
  d. `boolean`
- Logical type representing only `true` or `false`.
- Used in conditional logic.
- Example: `let isLoggedIn = true;`
  e. `null`
- Represents an intentional absence of a value.
- It’s a standalone type with a single value: `null`.
- Commonly used to reset or clear a variable.
  f. `undefined`
- A variable that has been declared but not assigned any value.
- JavaScript automatically assigns `undefined` to uninitialized variables.
  g. `symbol`
- Used to create unique identifiers for object properties.
- Every symbol is guaranteed to be unique.
- Example: `let id = Symbol('id');`

2. Non-Primitive Data Type (1 total)
   a. object

Used to store collections of data and more complex entities.
Can hold key-value pairs, functions, arrays, other objects, etc.

```
let user = {
  name: "Alice",
  age: 30
};
```

`typeof` Operator
The typeof operator returns the data type of a variable as a string.
Syntax: typeof x or typeof(x)

```
typeof 123           // "number"
typeof "hello"       // "string"
typeof true          // "boolean"
typeof undefined     // "undefined"
typeof null          // "object" ← (known language quirk)
typeof {a: 1}        // "object"
typeof Symbol()      // "symbol"
```

Basic operators, maths

We know many operators from school. They are things like addition `+`, multiplication `*`, subtraction `-`, and so on.

Terms: “unary”, “binary”, “operand”

Before we move on, let’s grasp some common terminology.
An operand – is what operators are applied to. For instance, in the multiplication of 5 * 2 there are two operands: the left operand is 5 and the right operand is 2. Sometimes, people call these “arguments” instead of “operands”.
An operator is unary if it has a single operand. For example, the unary negation - reverses the sign of a number:

```
let x = 1;
x = -x;
alert( x ); // -1, unary negation was applied
```

An operator is *binary* if it has two operands. The same minus exists in binary form as well:

```
let x = 1, y = 3;
alert( y - x ); // 2, binary minus subtracts values
```

Maths

The following math operations are supported:
`Addition +`
`Subtraction -`,
`Multiplication *`,
`Division /`,
`Remainder %`,
`Exponentiation .`
The first four are straightforward, while % and ** need a few words about them.

Remainder %

The remainder operator %, despite its appearance, is not related to percents.
The result of a % b is the remainder of the integer division of a by b.
For instance:

```
alert( 5 % 2 ); // 1, the remainder of 5 divided by 2
alert( 8 % 3 ); // 2, the remainder of 8 divided by 3
alert( 8 % 4 ); // 0, the remainder of 8 divided by 4
```

Exponentiation **

The exponentiation operator `a ** b` raises `a` to the power of `b`.
In school maths, we write that as a^b.

```
alert( 2 ** 2 ); // 2² = 4
alert( 2 ** 3 ); // 2³ = 8
alert( 2 ** 4 ); // 2⁴ = 16
```

Just like in maths, the exponentiation operator is defined for non-integer numbers as well.
For example, a square root is an exponentiation by ½:

```
alert( 4 ** (1/2) ); // 2 (power of 1/2 is the same as a square root)
alert( 8 ** (1/3) ); // 2 (power of 1/3 is the same as a cubic root)
```

String concatenation with binary +

Let's meet the features of JavaScript operators that are beyond school arithmetics.
Usually, the plus operator + sums numbers.
But, if the binary + is applied to strings, it merges (concatenates) them:

```
let s = "my" + "string";
alert(s); // mystring
```

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
alert(2 + 2 + '1' ); // "41" and not "221"
alert('1' + 2 + 2); // "122" and not "14"
alert( 6 - '2' ); // 4, converts '2' to a number
alert( '6' / '2' ); // 3, converts both operands to numbers
```

Numeric conversion, unary +

The plus `+` exists in two forms: the binary form that we used above and the unary form.
The unary plus or, in other words, the plus operator + applied to a single value, doesn't do anything to numbers. But if the operand is not a number, the unary plus converts it into a number.

```
// No effect on numbers
let x = 1;
alert( +x ); // 1
let y = -2;
alert( +y ); // -2
// Converts non-numbers
alert( +true ); // 1
alert( +"" );   // 0
```

If we want to treat them as numbers, we need to convert and then sum them:

```
let apples = "2";
let oranges = "3";
// both values converted to numbers before the binary plus
alert( +apples + +oranges ); // 5
// the longer variant
// alert( Number(apples) + Number(oranges) ); // 5
```

From a mathematician's standpoint, the abundance of pluses may seem strange. But from a programmer's standpoint, there's nothing special: unary pluses are applied first, they convert strings to numbers, and then the binary plus sums them up.
Why are unary pluses applied to values before the binary ones? As we're going to see, that's because of their higher precedence.

Assignment

Let's note that an assignment = is also an operator. It is listed in the precedence table with the very low priority of 2.

Chaining assignments

```
let a, b, c;
a = b = c = 2 + 2;
alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

Modify-in-place

```
let n = 2;
n = n + 5;
n = n * 2;
let n = 2;
n += 5; // now n = 7 (same as n = n + 5)
n *= 2; // now n = 14 (same as n = n * 2)
alert( n ); // 14
```

Short “modify-and-assign” operators exist for all arithmetical and bitwise operators: /=, -=, etc.
Such operators have the same precedence as a normal assignment, so they run after most other calculations:

```
let n = 2;
n *= 3 + 5; // right part evaluated first, same as n *= 8
alert( n ); // 16
```

Increment/decrement

Increasing or decreasing a number by one is among the most common numerical operations.
Increment ++ increases a variable by 1:

```
// Increment `++` increases a variable by 1:
let counter = 2;
counter++;        // works the same as counter = counter + 1, but is
shorter
alert( counter ); // 3
//Decrement`--` decreases a variable by 1:
let counter = 2;
counter--;        // works the same as counter = counter - 1, but is
shorter
alert( counter ); // 1
```

Important:

Increment/decrement can only be applied to variables. Trying to use it on a value like `5++` will give an error.
The operators ++ and -- can be placed either before or after a variable.

When the operator goes after the variable, it is in `postfix form`: counter++.
The `prefix form` is when the operator goes before the variable: ++counter.
Is there any difference?
Yes, but we can only see it if we use the returned value of ++/--.
all operators return a value. Increment/decrement is no exception. The prefix form returns the new value while the postfix form returns the old value (prior to increment/decrement).
To see the difference, here's an example:

```
let counter = 1;
let a = ++counter; // (*)
alert(a); // 2
//let's use the postfix form
let counter = 1;
let a = counter++; // (*) changed ++counter to counter++
alert(a); // 1
```

Bitwise operators

Bitwise operators treat arguments as 32-bit integer numbers and work on the level of their binary representation.
The list of operators:
AND (&)
OR (|)
XOR (^)
NOT (~)
LEFT SHIFT (<<)
RIGHT SHIFT (>>)
ZERO-FILL RIGHT SHIFT(>>>)

Comma

The comma operator , is one of the rarest and most unusual operators. Sometimes, it's used to write shorter code, so we need to know it in order to understand what's going on.
The comma operator allows us to evaluate several expressions, dividing them with a comma ,. Each of them is evaluated but only the result of the last one is returned.

```
let a = (1 + 2, 3 + 4);
```

```
alert( a ); // 7 (the result of 3 + 4)
```

Comparisons

We know many comparison operators from maths.
In JavaScript they are written like this:
Greater/less than: `a > b`, `a < b`.
Greater/less than or equals: `a >= b`, `a <= b.`
Equals: `a == b`, please note the double equality sign `==` means the equality test, while a single one `a = b` means an assignment.
Not equals: In maths the notation is ≠, but in JavaScript it's written as `a != b.`

Boolean is the result

All comparison operators return a boolean value:
`true` – means "yes", "correct" or "the truth".
`false` – means "no", "wrong" or "not the truth".

```
alert( 2 > 1 );  // true (correct)
alert( 2 == 1 ); // false (wrong)
alert( 2 != 1 ); // true (correct)
```

String comparison

To see whether a string is greater than another, JavaScript uses the so-called "dictionary" or "lexicographical" order.

```
alert( 'Z' > 'A' ); // true
alert( 'Glow' > 'Glee' ); // true
alert( 'Bee' > 'Be' ); // true
```

Strict equality

A regular equality check == has a problem. It cannot differentiate 0 from false:

```
alert( 0 == false ); // true
alert( '' == false ); // true
alert( 0 === false ); // false, because the types are different
alert( null === undefined ); // false
alert( null == undefined ); // true
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

Conditional branching

Sometimes, we need to perform different actions based on different conditions.
To do that, we can use the if statement and the conditional operator `?`, that's also called a question mark operator.

Boolean conversion

A number `0`, an empty string `""`, `null`, `undefined`, and `NaN` all become false.
Because of that they are called "`falsy`" values.
Other values become true, so they are called "`truthy`".

```
if (0) { // 0 is falsy
   ...
}
if (1) { // 1 is truthy
   ...
}
```

Conditional operator '?'

```
let accessAllowed;
let age = prompt('How old are you?', '');
if (age > 18) {
  accessAllowed = true;
} else {
  accessAllowed = false;
}
alert(accessAllowed);
let result = condition ? value1 : value2;
let accessAllowed = (age > 18) ? true : false;
```

Logical operators

There are four logical operators in JavaScript: || (OR), && (AND), ! (NOT), ?? (Nullish Coalescing).

|| (OR)

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

&& (AND)

```
result = a && b;
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
let hour = 12;
let minute = 30;

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

! (NOT)

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

```
result = !value;
alert( !true ); // false
alert( !0 ); // true
```

Nullish coalescing operator '??'

The `nullish` coalescing operator is written as two question marks `??`.
As it treats null and undefined similarly, we'll use a special term here, in this article. For brevity, we'll say that a value is "defined" when it's neither `null` nor `undefined`.
We can rewrite `result = a ?? b` using the operators that we already know, like this:

```
result = (a !== null && a !== undefined) ? a : b;
let user;
alert(user ?? "Anonymous"); // Anonymous (user is undefined)
let user = "John";
alert(user ?? "Anonymous"); // John (user is not null/undefined)
let firstName = null;
let lastName = null;
let nickName = "Supercoder";
// shows the first defined value:
alert(firstName ?? lastName ?? nickName ?? "Anonymous"); // Supercoder
```

Loops: while and for

We often need to repeat actions.

The "while" loop

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

Important

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

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

The “for” loop

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

```
for (begin; condition; step) {
  // ... loop body ...
};
for (let i = 0; i < 3; i++) { // shows 0, then 1, then 2
  alert(i);
}
```

Inline variable declaration

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

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

for of loop

There is another loop we can use to iterate over the elements of an array: the for of loop. It cannot be used to change the value associated with the index as we can do with the regular loop, but for processing values it is a very nice and readable loop

```
let arr = [some array];
for (let variableName of arr) {
 // code to be executed
 // value of variableName gets updated every iteration
 // all values of the array will be variableName once
}
let names = ["Chantal", "John", "Maxime", "Bobbi", "Jair"];
for (let name of names){
 console.log(name);
}
```

for in loop

Manipulating objects with loops can also be done with another variation of the for loop, the for in loop. The for in loop is somewhat similar to the for of loop. Again here, we need to specify a temporary name, also referred to as a key, to store each property name in

```
let car = {
 model: "Golf",
 make: "Volkswagen",
 year: 1999,
 color: "black",
};
for (let prop in car){
 console.log(car[prop]);
}
 // Golf Volkswagen 1999 black
 for (let prop in car){
 console.log(prop);
}
// model make year color
```

```
###### 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.

```
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++ );
}
// 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 );

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
}
```

The "switch" statement

A switch statement can replace multiple if checks.
It gives a more descriptive way to compare a value with multiple variants.

```
switch(x) {
  case 'value1':  // if (x === 'value1')
```

```
    ...
    [break]
  case 'value2':  // if (x === 'value2')
    ...
    [break]
  default:
    ...
    [break]
}
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Too small' );
    break;
  case 4:
    alert( 'Exactly!' );
    break;
  case 5:
    alert( 'Too big' );
    break;
  default:
    alert( "I don't know such values" );
}
```

Functions

Quite often we need to perform a similar action in many places of the script.
For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.
Functions are the main `building blocks` of the program. They allow the code to be called many times without repetition.

Function Declaration

To create a function we can use a function declaration.

```
function showMessage() {
  alert( 'Hello everyone!' );
}
```

The function keyword goes first, then goes the name of the function, then a list of parameters between the parentheses (comma-separated, empty in the example above, we'll see examples later) and finally the code of the function, also named "the function body", between curly braces.

```
function name(parameter1, parameter2, ... parameterN) {
 // body
```

```
}
function showMessage() {
  alert( 'Hello everyone!' );
}
showMessage();
showMessage();
```

Local variables

A variable declared inside a function is only visible inside that function.

```
function showMessage() {
  let message = "Hello, I'm JavaScript!"; // local variable
  alert( message );
}
showMessage(); // Hello, I'm JavaScript!
alert( message ); // <-- Error! The variable is local to the function
```

Outer variables

```
let userName = 'John';
function showMessage() {
  let message = 'Hello, ' + userName;
  alert(message);
}
showMessage(); // Hello, John

// The function has full access to the outer variable. It can modify it as
well.
let userName = 'John';

function showMessage() {
  userName = "Bob"; // (1) changed the outer variable
  let message = 'Hello, ' + userName;
  alert(message);
}
alert( userName ); // John before the function call
showMessage();
alert( userName ); // Bob, the value was modified by the function
```

Note

Global variables
Variables declared outside of any function, such as the outer userName in the code above, are called global.
Global variables are visible from any function (unless shadowed by locals).
It’s a good practice to minimize the use of global variables. Modern code has few

or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

Default values

If a function is called, but an argument is not provided, then the corresponding value becomes `undefined`.
For instance, the aforementioned function showMessage(from, text) can be called with a single argument:

```
showMessage("Ann");
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}
showMessage("Ann"); // Ann: no text given
```

Returning a value

A function can return a value back into the calling code as the result.
The simplest example would be a function that sums two values:

```
function sum(a, b) {
  return a + b;
}
let result = sum(1, 2);
alert( result ); // 3
function checkAge(age) {
  if (age >= 18) {
    return true;
  } else {
    return confirm('Do you have permission from your parents?');
  }
}
let age = prompt('How old are you?', 18);
if ( checkAge(age) ) {
  alert( 'Access granted' );
} else {
  alert( 'Access denied' );
}
```

The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).

A function with an empty return or without it returns undefined

If a function does not return a value, it is the same as if it returns undefined:

```
function doNothing() { /* empty */ }
alert( doNothing() === undefined ); // true
```

An empty return is also the same as return undefined:

```
function doNothing() {
  return;
}
alert( doNothing() === undefined ); // true
```

Naming a function

Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.

One function – one action

Function starting with...
"get..." – return a value,
"calc..." – calculate something,
"create..." – create something,
"check..." – check something and return a boolean, etc.

Summary

```
function name(parameters, delimited, by, comma) {
  /* code */
}
```

Values passed to a function as parameters are copied to its local variables.
A function may access outer variables. But it works only from inside out. The code outside of the function doesn't see its local variables.
A function can return a value. If it doesn't, then its result is undefined.
To make the code clean and easy to understand, it's recommended to use mainly local variables and parameters in the function, not outer variables.

It is always easier to understand a function which gets parameters, works with them and returns a result than a function which gets no parameters, but modifies outer variables as a side effect.

Function naming:

A name should clearly describe what the function does. When we see a function call in the code, a good name instantly gives us an understanding what it does and returns.

A function is an action, so function names are usually verbal.
There exist many well-known function prefixes like create..., show..., get..., check... and so on. Use them to hint what a function does.
Functions are the main building blocks of scripts. Now we've covered the basics, so we actually can start creating and using them. But that's only the beginning of the path. We are going to return to them many times, going more deeply into their advanced features.

Function expressions

In JavaScript, a function is not a "magical language structure", but a special kind of value.

```
function sayHi() {
  alert( "Hello" );
}
```

There is another syntax for creating a function that is called a **Function Expression.**

Function is a value

```
function sayHi() {
  alert( "Hello" );
}

alert( sayHi ); // shows the function code
```

Please note that the last line does not run the function, because there are no parentheses after sayHi. There are programming languages where any mention of a function name causes its execution, but JavaScript is not like that.
Surely, a function is a special value, in the sense that we can call it like `sayHi()`.

```
function sayHi() {   // (1) create
  alert( "Hello" );
}
let func = sayHi;    // (2) copy
func(); // Hello     // (3) run the copy (it works)!
sayHi(); // Hello    //     this still works too (why wouldn't it)
```

Why is there a semicolon at the end?

You might wonder, why do Function Expressions have a semicolon ; at the end, but Function Declarations do not:

```
function sayHi() {
  // ...
```

```
}
let sayHi = function() {
  // ...
};
```

The answer is simple: a Function Expression is created here as function(...) {...} inside the assignment statement: let sayHi = ...;. The semicolon ; is recommended at the end of the statement, it's not a part of the function syntax.
The semicolon would be there for a simpler assignment, such as let sayHi = 5;, and it's also there for a function assignment.

Function Expression vs Function Declaration

A Function Expression is created when the execution reaches it and is usable only from that moment.
Once the execution flow passes to the right side of the assignment let sum = function... – here we go, the function is created and can be used (assigned, called, etc.) from now on.
*A Function Declaration can be called earlier than it is defined.

Information

For example, a global Function Declaration is visible in the whole script, no matter where it is.
That's due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an "initialization stage".
And after all Function Declarations are processed, the code is executed. So it has access to these functions.

```
sayHi("John"); // Hello, John

function sayHi(name) {
  alert( `Hello, ${name}` );
}
```

The Function Declaration sayHi is created when JavaScript is preparing to start the script and is visible everywhere in it.

Note

In strict mode, when a Function Declaration is within a code block, it's visible everywhere inside that block. But not outside of it.

Arrow functions, the basics

There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.
It's called "arrow functions", because it looks like this:\

```
let func = (arg1, arg2, ..., argN) => expression;

let sum = (a, b) => a + b;
/* This arrow function is a shorter form of:
let sum = function(a, b) {
  return a + b;
};
*/
alert( sum(1, 2) ); // 3
let sum = (a, b) => {  // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, then we need an explicit
"return"
};

alert( sum(1, 2) ); // 3
```

Objects: the basics

Objects

```
let user = new Object(); // "object constructor" syntax
let user = {};  // "object literal" syntax
let user = {     // an object
  name: "John",  // by key "name" store value "John"
  age: 30        // by key "age" store value 30
};
```

```
let user = {};
// set
user["likes birds"] = true;
// get
alert(user["likes birds"]); // true
// delete
delete user["likes birds"];
let key = "likes birds";

// same as user["likes birds"] = true;
user[key] = true;
```

Here, the variable `key` may be calculated at run-time or depend on the user input. And then we use it to access the property. That gives us a great deal of flexibility.

```
let user = { name: "John", age: 30 };
alert( "age" in user ); // true, user.age exists
alert( "blabla" in user ); // false, user.blabla doesn't exist

let user = { age: 30 };

let key = "age";
alert( key in user ); // true, property "age" exists
```

Why does the in operator exist? Isn't it enough to compare against undefined?
Well, most of the time the comparison with undefined works fine. But there's a special case when it fails, but "`in`" works correctly.

It's when an object property exists, but stores `undefined`:

```
let obj = {
  test: undefined
};
alert( obj.test ); // it's undefined, so - no such property?
alert( "test" in obj ); // true, the property does exist!
```

In the code above, the property obj.test technically exists. So the in operator works right.
Situations like this happen very rarely, because undefined should not be explicitly assigned. We mostly use null for "unknown" or "empty" values. So the in operator is an exotic guest in the code.

The "for..in" loop

To walk over all keys of an object, there exists a special form of the loop: `for..in`. This is a completely different thing from the `for(;;)` construct that we studied before.

```
for (key in object) {
  // executes the body for each key among object properties
}

let user = {
  name: "John",
  age: 30,
  isAdmin: true
};
for (let key in user) {
  // keys
```

```
  alert( key );  // name, age, isAdmin
  // values for the keys
  alert( user[key] ); // John, 30, true
}
```

Note that all “for” constructs allow us to declare the looping variable inside the loop, like let key here.
Also, we could use another variable name here instead of key. For instance, "for (let prop in obj)" is also widely used.

Ordered like an object

Are objects ordered? In other words, if we loop over an object, do we get all properties in the same order they were added? Can we rely on this?
The short answer is: “ordered in a special fashion”: integer properties are sorted, others appear in creation order. The details follow.

```
let codes = {
  "49": "Germany",
  "41": "Switzerland",
  "44": "Great Britain",
  // ..,
  "1": "USA"
};

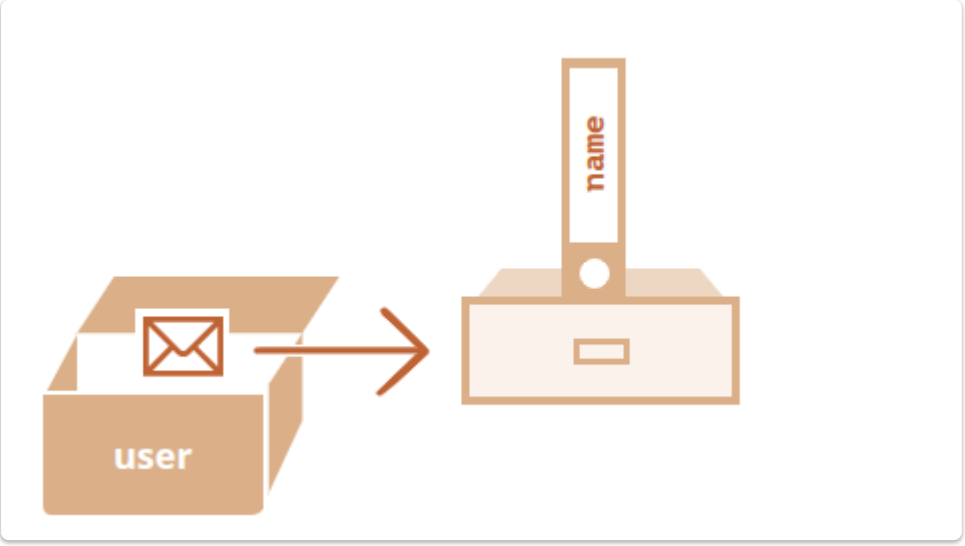
for (let code in codes) {
  alert(code); // 1, 41, 44, 49
}
```

Object references and copying

One of the fundamental differences of objects versus primitives is that objects are stored and copied “`by reference`”, whereas primitive values: strings, numbers, boolean, etc – are always copied “as a whole value”.

> ***A variable assigned to an object stores not the object itself, but its “address in memory” – in other words “a reference” to it.***

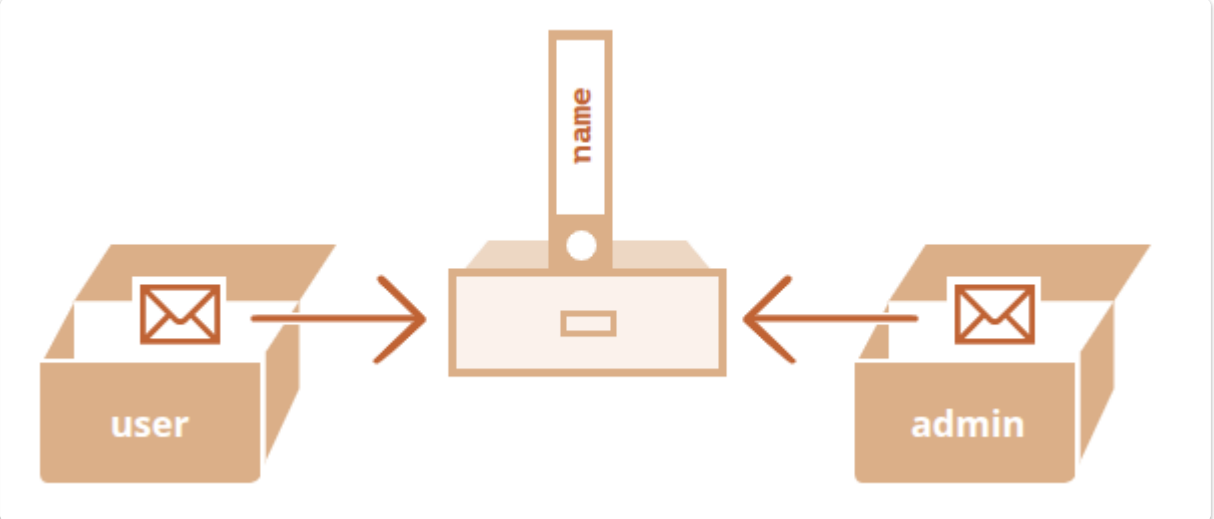
```
let user = {
  name: "John"
};
```



The object is stored somewhere in memory (at the right of the picture), while the user variable (at the left) has a “reference” to it.
We may think of an object variable, such as `user`, like a sheet of paper with the address of the object on it.
When we perform actions with the object, e.g. take a property `user.name`, the JavaScript engine looks at what’s at that address and performs the operation on the actual object.

> ***When an object variable is copied, the reference is copied, but the object itself is not duplicated.***

```
let user = { name: "John" };
let admin = user; // copy the reference
```



As you can see, there’s still one object, but now with two variables that reference it.

```
let user = { name: 'John' };
let admin = user;
admin.name = 'Pete'; // changed by the "admin" reference
alert(user.name); // 'Pete', changes are seen from the "user" reference
```

Comparison by reference

Two objects are equal only if they are the same object.

Const objects can be modified

An important side effect of storing objects as references is that an object declared as `const` *can* be modified.

```
const user = {
  name: "John"
};
user.name = "Pete"; // (*)
alert(user.name); // Pete
```

It might seem that the line `(*)` would cause an error, but it does not. The value of `user` is constant, it must always reference the same object, but properties of that object are free to change.
In other words, the `const user` gives an error only if we try to set `user=...` as a whole.

Cloning and merging, Object.assign

copying an object variable creates one more reference to the same object.

But what if we need to duplicate an object?

We can create a new object and replicate the structure of the existing one, by iterating over its properties and copying them on the primitive level.

```
let user = {
  name: "John",
  age: 30
};
//1. Shallow clone using for...in loop
let clone1 = {};
for (let key in user) {
  clone1[key] = user[key];
}

// 2. Shallow clone using spread syntax
let clone2 = { ...user };
// Modify clones
clone1.name = "Alice";
clone2.age = 99;
// Check original
console.log(user.name); // "John" — not affected
```

```
console.log(user.age);  // 30 — not affected
```

Above is shallow copy

Shallow Copy (Surface-Level Only)

Copies only the top-level properties of an object. If a property is a primitive (like a number or string), it gets copied by value. If a property is an object or array, only the reference (pointer) is copied — not the actual content.Changes to top-level properties in the copy don't affect the original. But changes to nested objects/arrays will affect the original.

```
let user = {
 name: "John",
 address: {
   city: "Delhi"
 }
};
let shallow = { ...user }; // shallow copy
shallow.name = "Alice"; // OK, no effect on original
shallow.address.city = "Mumbai"; // ❗ Affects original
console.log(user.address.city); // "Mumbai"
```

Deep Copy (Full Clone)

Copies everything, including nested objects and arrays. The copy is fully independent — changing it does not affect the original.

```
let user = {
  name: "John",
  address: {
    city: "Delhi"
  }
};
let deep = structuredClone(user); // deep copy
deep.address.city = "Mumbai";
console.log(user.address.city); // "Delhi" — original is safe
```

What are JSON.stringify() and JSON.parse()?
`JSON.stringify(obj)` → Converts a JavaScript object into a JSON string
`JSON.parse(json)` → Converts a JSON string back into a JavaScript object.
Together, they convert an object into text and back again, creating a brand-new copy in the process.

```
let user = {
  name: "John",
  age: 30,
  address: {
    city: "Delhi"
  }
};
let deepClone = JSON.parse(JSON.stringify(user));
deepClone.address.city = "Mumbai";

console.log(user.address.city); // Delhi ✅ (original not changed)
```

`JSON.stringify()` removes all references and turns the object into plain text.
`JSON.parse()` builds a brand-new object with its own nested structures.
Limitations You Must Know:

Case Result
Functions Not included in clone
undefined values Ignored
Symbols Ignored
Circular references (e.g., obj.a = obj) Throws error
Plain objects, arrays, numbers, strings Works perfectly

```
let user = {
  name: "John",
  greet: function () { console.log("Hi"); }
};
let clone = JSON.parse(JSON.stringify(user));
console.log(clone.greet); // undefined
```

- *`structuredClone()` is built-in in modern browsers and Node.js (v17+)*

Object.assign(dest, ...sources)

The first argument dest is a target object.
Further arguments is a list of source objects.
It copies the properties of all source objects into the target dest, and then returns it as the result.

```
let user = { name: "John" };
let permissions1 = { canView: true };
let permissions2 = { canEdit: true };
// copies all properties from permissions1 and permissions2 into user
Object.assign(user, permissions1, permissions2);
// now user = { name: "John", canView: true, canEdit: true }
```

```
alert(user.name); // John
alert(user.canView); // true
alert(user.canEdit); // true
```

If the copied property name already exists, it gets overwritten:

```
let user = { name: "John" };
Object.assign(user, { name: "Pete" });
alert(user.name); // now user = { name: "Pete" }
```

We also can use Object.assign to perform a simple object cloning:

```
let user = {
  name: "John",
  age: 30
};

let clone = Object.assign({}, user);
alert(clone.name); // John
alert(clone.age); // 30
```

Garbage collection

Memory management in JavaScript is performed automatically and invisibly to us. We create primitives, objects, functions... All that takes memory.
What happens when something is not needed any more? How does the JavaScript engine discover it and clean it up?

Reachability

The main concept of memory management in JavaScript is `reachability`.
Simply put, "`reachable`" values are those that are accessible or usable somehow.
They are guaranteed to be stored in memory.
Reachable means: "Can I still access this value in memory?"
If **yes → it's kept in memory**
If **no → JavaScript will delete it from memory (garbage collection)**

Imagine JavaScript Memory as a Warehouse:

- Inside the warehouse are **boxes (objects, arrays, variables, etc.)**
- Boxes are connected with **ropes (references)**
- If a box has **no rope leading to it**, it's considered "useless"
- The janitor (garbage collector) removes unconnected boxes.

Roots (Starting Points)

Some things are always reachable, no matter what:

- Current function's variables
- Parameters of the function
- All parent functions still running
- Global variables (like let user = ...)
  We call these roots. They are where JavaScript starts when checking what should stay in memory.

```
let user = {
  name: "John"
};
```

Because user is a global variable, and global variables are roots
2. Local Scope = Temporary Root

```
function greet() {
  let user = { name: "John" };
  console.log("Hello");
}
greet();
```

Here, user exists only inside the function.
Once the function ends, user is out of scope.
The object { name: "John" } becomes unreachable.

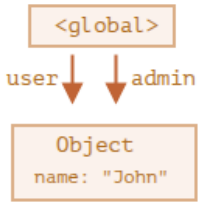
3. But... if you keep a reference (closure), it stays

```
function outer() {
  let user = { name: "John" };
  return function inner() {
    console.log(user.name); // this function still uses 'user'
  };
}
let sayName = outer(); // outer runs, returns inner
sayName(); // "John"
```

Even though outer() has finished,
The inner() function remembers the user object
So user is still reachable
Garbage collector keeps it alive
This is called a closure — and it keeps variables in memory as long as they're needed.

Two references

```
// user has a reference to the object
let user = {
  name: "John"
};
let admin = user;
```



`user = null;`

...Then the object is still reachable via admin global variable, so it must stay in memory. If we overwrite admin too, then it can be removed.

Interlinked objects

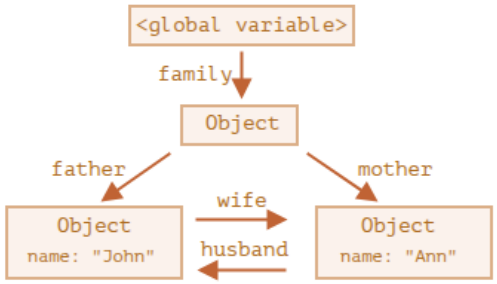
```
function marry(man, woman) {
  woman.husband = man;
  man.wife = woman;

  return {
    father: man,
    mother: woman
  }
}

let family = marry(
  { name: "John" },
  { name: "Ann" }
);
```

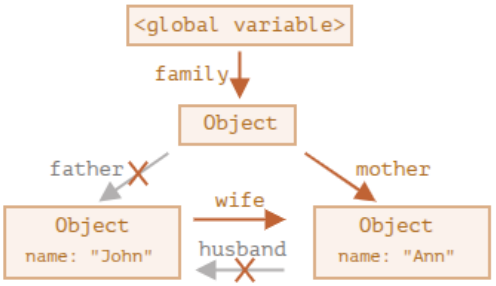
What Does the Memory Look Like?

```
family ⟶ {
  father ⟶ John ⟶ wife ⟶ Ann ⟶ husband ⟶ John (again!)
  mother ⟶ Ann
}
```



This is called a circular reference
comes Garbage Collection

```
delete family.father;
delete family.mother.husband;
```



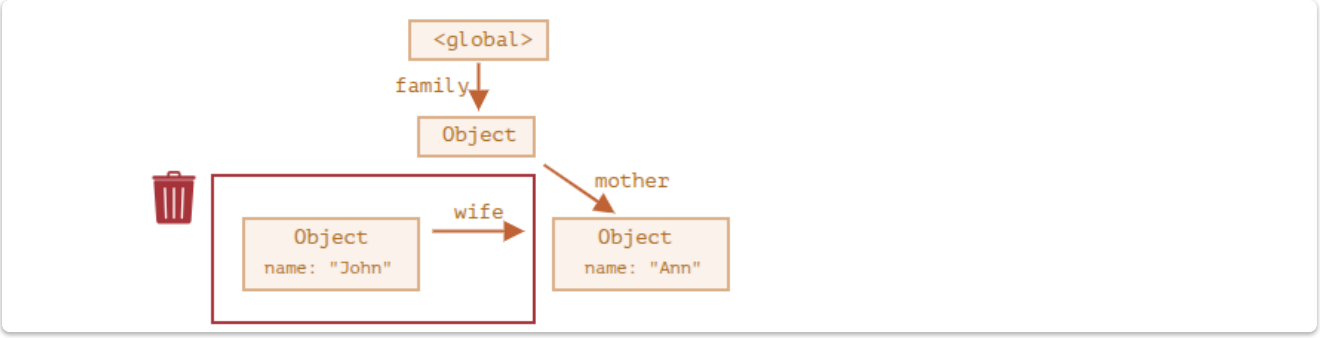
Outgoing references do not matter. Only incoming ones can make an object reachable. So, John is now unreachable and will be removed from the memory with all its data that also became unaccessible.
what left ?

```
family ⟶ {
  mother ⟶ Ann
}

Ann has no more `.husband` property
John is not stored in `family` anymore
```

Important rule

Garbage collector only cares about incoming references — who's pointing to the object.

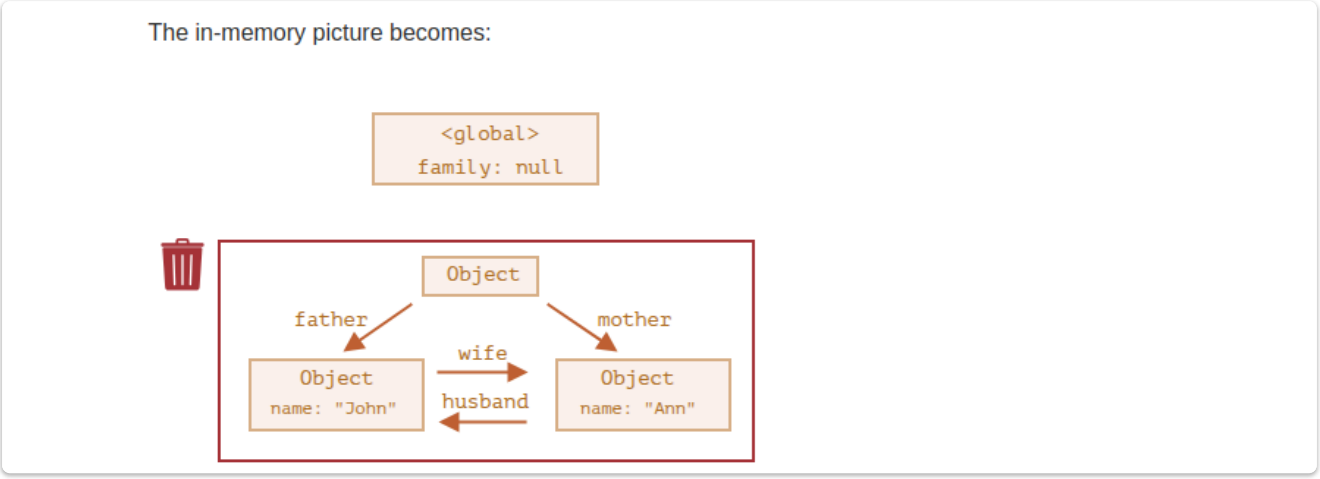


What is a Circular Link?
A circular link (also called a circular reference) happens when:
Two (or more) objects reference each other, forming a loop

Unreachable island

It is possible that the whole island of interlinked objects becomes unreachable and is removed from the memory.

```
family = null;
```



It's obvious that John and Ann are still linked, both have incoming references. But that's not enough.
The former "family" object has been unlinked from the root, there's no reference to it any more, so the whole island becomes unreachable and will be removed.

Internal algorithms

The basic garbage collection algorithm is called "mark-and-sweep".

The following "garbage collection" steps are regularly performed:
The garbage collector takes roots and "marks" (remembers) them.
Then it visits and "marks" all references from them.
Then it visits marked objects and marks their references. All visited objects are remembered, so as not to visit the same object twice in the future.
...And so on until every reachable (from the roots) references are visited.
All objects except marked ones are removed.

Object methods, `this`

Objects are usually created to represent entities of the real world, like users, orders and so on

```
let user = {
  name: "John",
  age: 30
}
```

And, in the real world, a user can act: select something from the shopping cart, login, logout etc.
Actions are represented in JavaScript by functions in properties.

Method examples

```
let user = {
  name: "John",
  age: 30
};
user.sayHi = function() {
  alert("Hello!");
};
user.sayHi(); // Hello!
```

Here we've just used a Function Expression to create a function and assign it to the property `user.sayHi` of the object.
Then we can call it as `user.sayHi()`. The user can now speak!
A function that is a property of an object is called its *method.*

`this` in methods

It's common that an object method needs to access the information stored in the object to do its job.
For instance, the code inside `user.sayHi()` may need the name of the `user`.

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    // "this" is the "current object"
    alert(this.name);
  }
};
user.sayHi(); // John
```

To access the object, a method can use the this keyword.
Here during the execution of `user.sayHi()`, the value of `this` will be `user`.

Technically, it's also possible to access the object without this, by referencing it via the outer variable:

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    alert(user.name); // "user" instead of "this"
  }

};
```

...But such code is unreliable. If we decide to copy `user` to another variable, e.g. `admin = user` and overwrite `user` with something else, then it will access the wrong object.

```
let user = {
  name: "John",
  age: 30,
  sayHi() {
    alert( user.name ); // leads to an error
  }
};
let admin = user;
user = null; // overwrite to make things obvious

admin.sayHi(); // TypeError: Cannot read property 'name' of null
```

If we used `this.name` instead of `user.name` inside the `alert`, then the code would work.

`this` is not bound

In JavaScript, keyword this behaves unlike most other programming languages. It can be used in any function, even if it's not a method of an object.

```
function sayHi() {
  alert( this.name );
}
```

The value of `this` is evaluated during the run-time, depending on the context.
For instance, here the same function is assigned to two different objects and has different "this" in the calls:

```
let user = { name: "John" };
let admin = { name: "Admin" };
```

```
function sayHi() {
  alert( this.name );
}
// use the same function in two objects
user.f = sayHi;
admin.f = sayHi;
// these calls have different this
// "this" inside the function is the object "before the dot"
user.f(); // John  (this == user)
admin.f(); // Admin  (this == admin)
admin['f'](); // Admin (dot or square brackets access the method — doesn't
matter)
```

Info

Calling without an object: `this == undefined`

```
function sayHi() {
  alert(this);
}

sayHi(); // undefined
```

In this case this is undefined in strict mode. If we try to access this.name, there will be an error.
In non-strict mode the value of this in such case will be the global object (window in a browser). This is a historical behavior that "use strict" fixes.
Usually such call is a programming error. If there's this inside a function, it expects to be called in an object context.

Attention

If you come from another programming language, then you are probably used to the idea of a "bound `this`", where methods defined in an object always have `this` referencing that object.
In JavaScript `this` is "free", its value is evaluated at call-time and does not depend on where the method was declared, but rather on what object is "before the dot".
The concept of run-time evaluated `this` has both pluses and minuses. On the one hand, a function can be reused for different objects. On the other hand, the greater flexibility creates more possibilities for mistakes.
Here our position is not to judge whether this language design decision is good or

bad. We'll understand how to work with it, how to get benefits and avoid problems.

Arrow functions have no `this`

Arrow functions are special: they don't have their "own" `this`. If we reference `this` from such a function, it's taken from the outer "normal" function.

```
let user = {
  firstName: "Ilya",
  sayHi() {
    let arrow = () => alert(this.firstName);
    arrow();
  }
};

user.sayHi(); // Ilya
```

Constructor, operator `new`

The regular {...} syntax allows us to create one object. But often we need to create many similar objects, like multiple users or menu items and so on.
That can be done using constructor functions and the `new` operator.
Constructor functions technically are regular functions. There are two conventions though:

- They are named with capital letter first.
- They should be executed only with `new` operator.

```
function User(name) {
  this.name = name;
  this.isAdmin = false;
}

let user = new User("Jack");

alert(user.name); // Jack
alert(user.isAdmin); // false
```

When a function is executed with new, it does the following steps:

- A new empty object is created and assigned to this.
- The function body executes. Usually it modifies this, adds new properties to it.
- The value of this is returned.
  So `let user = new User("Jack")` gives the same result as:

```
let user = {
  name: "Jack",
  isAdmin: false
};
```

Now if we want to create other users, we can call `new User("Ann")`, `new User("Alice")` and so on. Much shorter than using literals every time, and also easy to read.

> *That's the main purpose of constructors – to implement reusable object creation code.*

Let's note once again – technically, any function (except arrow functions, as they don't have `this`) can be used as a constructor. It can be run with `new`, and it will execute the algorithm above. The "capital letter first" is a common agreement, to make it clear that a function is to be run with `new`.

Return from constructors

Usually, constructors do not have a `return` statement. Their task is to write all necessary stuff into `this`, and it automatically becomes the result.
But if there is a `return` statement, then the rule is simple:

- If `return` is called with an object, then the object is returned instead of `this`.
- If `return` is called with a primitive, it's ignored.
  In other words, `return` with an object returns that object, in all other cases `this` is returned.

```
function BigUser() {
  this.name = "John";
  return { name: "Godzilla" };  // <-- returns this object
}
alert( new BigUser().name );  // Godzilla, got that object
```

And here's an example with an empty `return` (or we could place a primitive after it, doesn't matter):

```
function SmallUser() {
  this.name = "John";
  return; // <-- returns this
}
alert( new SmallUser().name );  // John
```

Usually constructors don't have a return statement. Here we mention the special behavior with returning objects mainly for the sake of completeness

Omitting parentheses

By the way, we can omit parentheses after `new`:

```
let user = new User; // <-- no parentheses
// same as
let user = new User();
```

Methods in constructor

Using constructor functions to create objects gives a great deal of flexibility. The constructor function may have parameters that define how to construct the object, and what to put in it.
Of course, we can add to `this` not only properties, but methods as well.

```
function User(name) {
  this.name = name;
  this.sayHi = function() {
    alert( "My name is: " + this.name );
  };
}
let john = new User("John");

john.sayHi(); // My name is: John
/*
john = {
   name: "John",
   sayHi: function() { ... }
}
*/
```

Optional chaining '?.'

The optional chaining `?.` is a safe way to access nested object properties, even if an intermediate property doesn't exist.

The "non-existing property" problem

let's say we have `user` objects that hold the information about our users.
Most of our users have addresses in `user.address` property, with the street `user.address.street`, but some did not provide them.

```
let user = {}; // a user without "address" property
alert(user.address.street); // Error!
```

That's the expected result. JavaScript works like this. As user.address is undefined, an attempt to get user.address.street fails with an error.

In many practical cases we'd prefer to get undefined instead of an error here (meaning "no street").
The obvious solution would be to check the value using `if` or the conditional operator `?`, before accessing its property.

```
let user = {};
alert(user.address ? user.address.street : undefined);
```

Optional chaining

The optional chaining `?.` stops the evaluation if the value before `?.` is `undefined` or `null` and returns `undefined`.

> ⚠ **for brevity, we'll be saying that something "exists" if it's not `null` and not `undefined`.**

In other words, `value?.prop`:

- works as `value.prop`, if `value` exists,
- otherwise (when `value` is `undefined/null`) it returns `undefined`.

```
let user = {}; // user has no address
alert( user?.address?.street ); // undefined (no error)
let user = null;
alert( user?.address ); // undefined
alert( user?.address.street ); // undefined
```

> ⚠ **Warning**
>
> Don't overuse the optional chaining
> We should use `?.` only where it's ok that something doesn't exist.
> For example, if according to our code logic `user` object must exist, but `address` is optional, then we should write `user.address?.street`, but not `user?.address?.street`.
> Then, if `user` happens to be undefined, we'll see a programming error about it and fix it. Otherwise, if we overuse `?.`, coding errors can be silenced where not appropriate, and become more difficult to debug.

The variable before `?.` must be declared
If there's no variable `user` at all, then `user?.anything` triggers an error:

```
// ReferenceError: user is not defined
user?.address;
```

The variable must be declared (e.g. `let/const/var user` or as a function parameter). The optional chaining works only for declared variables.

Short-circuiting

As it was said before, the ?. immediately stops (`short-circuits`) the evaluation if the left part doesn't exist.
So, if there are any further function calls or operations to the right of ?., they won't be made.

```
let user = null;
let x = 0;
user?.sayHi(x++); // no "user", so the execution doesn't reach sayHi call
and x++
alert(x); // 0, value not incremented
```

Data types

A primitive

- Is a value of a primitive type.
- There are 7 primitive
  types: `string` , `number` , `bigint` , `boolean` , `symbol` , `null` and `undefined` .
  An object
- Is capable of storing multiple values as properties.
- Can be created with `{}` , for instance: `{name: "John", age: 30}` . There are other kinds of objects in JavaScript: functions, for example, are objects.

Numbers

Regular numbers in JavaScript are stored in 64-bit format IEEE-754, also known as `double` precision floating point `numbers` .
BigInt numbers represent integers of arbitrary length. They are sometimes needed because a regular integer number can't safely exceed (2^53-1) or be less than -(2^53-1), as we mentioned earlier in the chapter Data types. As bigints are used in a few special areas, we devote them to a special chapter BigInt.
`toString(base)`
The method `num.toString(base)` returns a string representation of `num` in the numeral system with the given `base` .

```
let num = 255;
alert( num.toString(16) );  // ff
alert( num.toString(2) );   // 11111111
```

The `base` can vary from `2` to `36`. By default, it's `10`.
Common use cases for this are:

- base=16 is used for hex colors, character encodings etc, digits can be `0..9` or `A..F`.
- base=2 is mostly for debugging bitwise operations, digits can be `0` or `1`.
- base=36 is the maximum, digits can be `0..9` or `A..Z`. The whole Latin alphabet is used to represent a number. A funny, but useful case for `36` is when we need to turn a long numeric identifier into something shorter, for example, to make a short url. Can simply represent it in the numeral system with base `36`:

```
alert( 123456..toString(36) ); // 2n9c
```

Note

Two dots to call a method
Please note that two dots in `123456..toString(36)` is not a typo. If we want to call a method directly on a number, like `toString` in the example above, then we need to place two dots `..` after it.
If we placed a single dot: `123456.toString(36)`, then there would be an error, because JavaScript syntax implies the decimal part after the first dot. And if we place one more dot, then JavaScript knows that the decimal part is empty and now uses the method.
Also could write `(123456).toString(36)`.

Rounding

One of the most used operations when working with numbers is rounding.
`Math.floor`
Rounds down: `3.1` becomes `3`, and `-1.1` becomes `-2`.
`Math.ceil`
Rounds up: `3.1` becomes `4`, and `-1.1` becomes `-1`.
`Math.round`
Rounds to the nearest integer: `3.1` becomes `3`, `3.6` becomes `4`. In the middle cases `3.5` rounds up to `4`, and `-3.5` rounds up to `-3`.
`Math.trunc` (not supported by Internet Explorer)
Removes anything after the decimal point without

rounding: `3.1` becomes `3`, `-1.1` becomes `-1`.

	Math.floor	Math.ceil	Math.round	Math.trunc
3.1	3	4	3	3
3.5	3	4	4	3
3.6	3	4	4	3
-1.1	-2	-1	-1	-1
-1.5	-2	-1	-1	-1
-1.6	-2	-1	-2	-1

For instance, we have 1.2345 and want to round it to 2 digits, getting only 1.23.
There are two ways to do so:
Multiply-and-divide.
For example, to round the number to the 2nd digit after the decimal, we can multiply the number by 100, call the rounding function and then divide it back.

```
let num = 1.23456;
alert( Math.round(num * 100) / 100 ); // 1.23456 -> 123.456 -> 123 -> 1.23
```

The method `toFixed(n)` rounds the number to n digits after the point and returns a string representation of the result.

```
let num = 12.34;
alert( num.toFixed(1) ); // "12.3"
```

Important

Please note that the result of toFixed is a string. If the decimal part is shorter than required, zeroes are appended to the end:

```
let num = 12.34;
alert( num.toFixed(5) ); // "12.34000", added zeroes to make exactly 5 digits
```

Imprecise calculations

Internally, a number is represented in 64-bit format IEEE-754,
so there are exactly `64 bits`
to store a number: 52 of them are used to `store the digits`,
11 of them store the position of the decimal point,
and 1 bit is for the sign.
`alert( 1e500 ); // Infinity`
What may be a little less obvious, but happens quite often, is the loss of precision.

`alert( 0.1 + 0.2 == 0.3 ); // false`
That's right, if we check whether the sum of 0.1 and 0.2 is 0.3, we get *false.*
Strange! What is it then if not 0.3?
`alert( 0.1 + 0.2 ); // 0.30000000000000004`

Ouch! Imagine you're making an e-shopping site and the visitor puts $0.10 and $0.20 goods into their cart. The order total will be `$0.30000000000000004.` That would surprise anyone.
A number is stored in memory in its binary form, a sequence of bits – ones and zeroes. But fractions like 0.1, 0.2 that look simple in the decimal numeric system are actually unending fractions in their binary form.

```
alert(0.1.toString(2));  //
0.0001100110011001100110011001100110011001100110011001101
alert(0.2.toString(2)); //
0.0011001100110011001100110011001100110011001100110011001101
alert((0.1 + 0.2).toString(2)); //
0.0100110011001100110011001100110011001100110011001101
```

What is 0.1? It is one divided by ten 1/10, one-tenth. In the decimal numeral system, such numbers are easily representable. Compare it to one-third: 1/3. It becomes an endless fraction 0.33333(3).
So, division by powers 10 is guaranteed to work well in the decimal system, but division by 3 is not. For the same reason, in the binary numeral system, the division by powers of 2 is guaranteed to work, but 1/10 becomes an endless binary fraction.
There's just no way to store exactly 0.1 or exactly 0.2 using the binary system, just like there is no way to store one-third as a decimal fraction.
The numeric format IEEE-754 solves this by rounding to the nearest possible number. These rounding rules normally don't allow us to see that `tiny precision loss`, but it exists.

```
alert( 0.1.toFixed(20) ); // 0.10000000000000000555
```

And when we sum two numbers, their `precision losses` add up.
That's why 0.1 + 0.2 is not exactly 0.3.

Not only JavaScript

The same issue exists in many other programming languages.
PHP, Java, C, Perl, and Ruby give exactly the same result, because they are based on the same numeric format.

Can we work around the problem? Sure, the most reliable method is to round the result with the help of a method `toFixed(n)`

```
let sum = 0.1 + 0.2;
alert( sum.toFixed(2) ); // "0.30"
```

> ⚠ **Warning**
>
> Please note that toFixed always returns a string. It ensures that it has 2 digits after the decimal point.

we can use the unary plus to coerce it into a number:

```
let sum = 0.1 + 0.2;
alert( +sum.toFixed(2) ); // 0.3
```

Tests: `isFinite` and `isNaN`

Remember these two special numeric values?

- `Infinity` (and `-Infinity`) is a special numeric value that is greater (less) than anything.
- `NaN` represents an error.
  `isNaN(value)` converts its argument to a number and then tests it for being `NaN`:

```
alert( isNaN(NaN) ); // true
alert( isNaN("str") ); // true
```

But do we need this function? Can't we just use the comparison `=== NaN`? Unfortunately not. The value `NaN` is unique in that it does not equal anything, including itself:

```
alert( NaN === NaN ); // false
```

`isFinite(value)` converts its argument to a number and returns `true` if it's a regular number, not `NaN/Infinity/-Infinity`:

```
alert( isFinite("15") ); // true
alert( isFinite("str") ); // false, because a special value: NaN
alert( isFinite(Infinity) ); // false, because a special value: Infinity
```

Number.isNaN and Number.isFinite — they're just stricter, safer versions of isNaN and isFinite.

`isNaN()` vs `Number.isNaN()`

`isNaN(value)`

- Converts the value to a number.
- Returns true if the result is `NaN`

```
isNaN("str") // true -> because "str" becomes NaN when converted
isNaN(NaN)
```

`Number.isNaN(value)`

- Does NOT convert value.
- Only returns `true` if:
  - Value is of type number
  - AND it is NaN

```
Number.isNaN("str")   // false -> it's a string, not a number
Number.isNaN(NaN)     // true
Number.isNaN("str" / 2) // true -> because "str" / 2 results in NaN (and
it's type number)
```

`isFinite()` vs `Number.isFinite()`

`isFinite(value)`
Converts value to a number
Returns true if the result is a finite number (not NaN/Infinity/-Infinity)

```
isFinite("123")  // true  -> becomes 123
isFinite("abc")  // false -> becomes NaN
```

`Number.isFinite(value)`

```
Number.isFinite("123")  // false -> it's a string
Number.isFinite(123)    // true
Number.isFinite(2 / 0)  // false -> Infinity
```

Comparison with `Object.is`

There is a special built-in method `Object.is` that compares values like `===`, but is more reliable for two edge cases:

1. It works with `NaN`: `Object.is(NaN, NaN) === true`, that's a good thing.
2. Values `0` and `-0` are different: `Object.is(0, -0) === false`, technically that's correct because internally the number has a sign bit that may be different even if

all other bits are zeroes.
In all other cases, `Object.is(a, b)` is the same as `a === b`.

We mention `Object.is` here, because it's often used in JavaScript specification. When an internal algorithm needs to compare two values for being exactly the same, it uses `Object.is` (internally called SameValue

parseInt and parseFloat

Numeric conversion using a plus `+` or `Number()` is strict. If a value is not exactly a number, it fails:
`alert( +"100px" ); // NaN`
But in real life, we often have values in units, like `"100px"` or `"12pt"` in CSS. Also in many countries, the currency symbol goes after the amount, so we have `"19€"` and would like to extract a numeric value out of that.
That's what **`parseInt`** and **`parseFloat`** are for.
They "read" a number from a string until they can't. In case of an error, the gathered number is returned. The function `parseInt` returns an integer, whilst `parseFloat` will return a floating-point number:

```
alert( parseInt('100px') ); // 100
alert( parseFloat('12.5em') ); // 12.5
alert( parseInt('12.3') ); // 12, only the integer part is returned
alert( parseFloat('12.3.4') ); // 12.3, the second point stops the reading
```

There are situations when `parseInt/parseFloat` will return `NaN`. It happens when no digits could be read:

```
alert( parseInt('a123') ); // NaN, the first symbol stops the process
```

The second argument of `parseInt(str, radix)`
The `parseInt()` function has an optional second parameter. It specifies the base of the numeral system, so `parseInt` can also parse strings of hex numbers, binary numbers and so on:

```
alert( parseInt('0xff', 16) ); // 255
alert( parseInt('ff', 16) ); // 255, without 0x also works

alert( parseInt('2n9c', 36) ); // 123456
```

Strings

In JavaScript, the textual data is stored as strings. There is no separate type for a single character.
The internal format for strings is always `UTF-16`, it is not tied to the page encoding.

Quotes

Strings can be enclosed within either single quotes, double quotes or backticks:

```
let single = 'single-quoted';
let double = "double-quoted";
let backticks = `backticks`;
function sum(a, b) {
  return a + b;
}
alert(`1 + 2 = ${sum(1, 2)}.`); // 1 + 2 = 3.
```

Another advantage of using backticks is that they allow a string to span multiple lines

```
let guestList = `Guests:
 * John
 * Pete
 * Mary
`;

alert(guestList); // a list of guests, multiple lines
```

Looks natural, right? But single or double quotes do not work this way.
If we use them and try to use multiple lines, there'll be an error:

```
let guestList = "Guests: // Error: Unexpected token ILLEGAL
  * John"
```

String length

```
alert( `My\n`.length ); // 3
```

> ⚠ **Warning**
>
> `length` is a property
> People with a background in some other languages sometimes mistype by calling `str.length()` instead of just `str.length`. That doesn't work.
> Please note that `str.length` is a numeric property, not a function. There is no need to add parenthesis after it. Not `.length()`, but `.length`.

Accessing characters

To get a character at position pos, use square brackets [pos] or call the method str.at(pos). The first character starts from the zero position:

```
let str = `Hello`;
// the first character
alert( str[0] ); // H
alert( str.at(0) ); // H
// the last character
alert( str[str.length - 1] ); // o
alert( str.at(-1) );

let str = `Hello`;

alert( str[-2] ); // undefined
alert( str.at(-2) ); // l
for (let char of "Hello") {
  alert(char); // H,e,l,l,o (char becomes "H", then "e", then "l" etc)
}
```

Strings are immutable

Strings can't be changed in JavaScript. It is impossible to change a character.

```
let str = 'Hi';
str[0] = 'h'; // error
alert( str[0] ); // doesn't work
```

The usual workaround is to create a whole new string and assign it to `str` instead of the old one.

```
let str = 'Hi';
str = 'h' + str[1]; // replace the string
alert( str ); // hi
```

Changing the case

```
alert( 'Interface'.toUpperCase() ); // INTERFACE
alert( 'Interface'.toLowerCase() ); // interface
alert( 'Interface'[0].toLowerCase() ); // 'i'
```

Searching for a substring

`str.indexOf`
The first method is `str.indexOf(substr, pos)`.
It looks for the `substr` in `str`, starting from the given position `pos`, and returns the position where the match was found or -1 if nothing can be found.

```
let str = 'Widget with id';
alert( str.indexOf('Widget') ); // 0, because 'Widget' is found at the
```

```
beginning
alert( str.indexOf('widget') ); // -1, not found, the search is case-
sensitive
alert( str.indexOf("id") ); // 1, "id" is found at the position 1 (..idget
with id)
```

Arrays

Objects allow you to store keyed collections of values. That's fine.
But quite often we find that we need an *ordered collection*, where we have a 1st, a 2nd, a 3rd element and so on.
For example, we need that to store a list of something: users, goods, HTML elements etc.

> *It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can't insert a new property "between" the existing ones. Objects are just not meant for such use.*

Declaration

```
let arr = new Array();
let arr = [];
```

Almost all the time, the second syntax is used. We can supply initial elements in the brackets:
`let fruits = ["Apple", "Orange", "Plum"];`
Array elements are numbered, starting with zero.
We can get an element by its number in square brackets:

```
let fruits = ["Apple", "Orange", "Plum"];
alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
alert( fruits[2] ); // Plum
// We can replace an element:
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
 // …Or add a new one to the array:
fruits[3] = 'Lemon'; // now ["Apple", "Orange", "Pear", "Lemon"]
```

The total count of the elements in the array is its length:

```
let fruits = ["Apple", "Orange", "Plum"];
alert( fruits.length ); // 3
```

An array can store elements of any type.

```
// mix of values
let arr = [ 'Apple', { name: 'John' }, true, function() { alert('hello');
} ];
// get the object at index 1 and then show its name
alert( arr[1].name ); // John
// get the function at index 3 and run it
arr[3](); // hello
```

> ⚠ **Warning**
>
> Trailing comma
> An array, just like an object, may end with a comma:

```
let fruits = [
  "Apple",
  "Orange",
  "Plum",
];
```

Get last elements with “at”

> ⚠ **Warning**
>
> A recent addition
> This is a recent addition to the language. Old browsers may need polyfills.

Let’s say we want the last element of the array.
Some programming languages allow the use of negative indexes for the same purpose, like fruits[-1].
Although, in JavaScript it won’t work. The result will be undefined, because the index in square brackets is treated literally.
We can explicitly calculate the last element index and then access it:
fruits[fruits.length - 1].

```
let fruits = ["Apple", "Orange", "Plum"];
alert( fruits[fruits.length-1] ); // Plum
```

Luckily, there’s a shorter syntax: `fruits.at(-1)`:

```
let fruits = ["Apple", "Orange", "Plum"];
// same as fruits[fruits.length-1]
alert( fruits.at(-1) ); // Plum
```

Internals

An array is a special kind of object. The square brackets used to access a property arr[0] actually come from the object syntax. That's essentially the same as obj[key], where arr is the object, while numbers are used as keys.
They extend objects providing special methods to work with ordered collections of data and also the length property. But at the core it's still an object.
Remember, there are only eight basic data types in JavaScript . Array is an object and thus behaves like an object.

```
let fruits = ["Banana"]
let arr = fruits; // copy by reference (two variables reference the same
array)
alert( arr === fruits ); // true
arr.push("Pear"); // modify the array by reference
alert( fruits ); // Banana, Pear - 2 items now
```

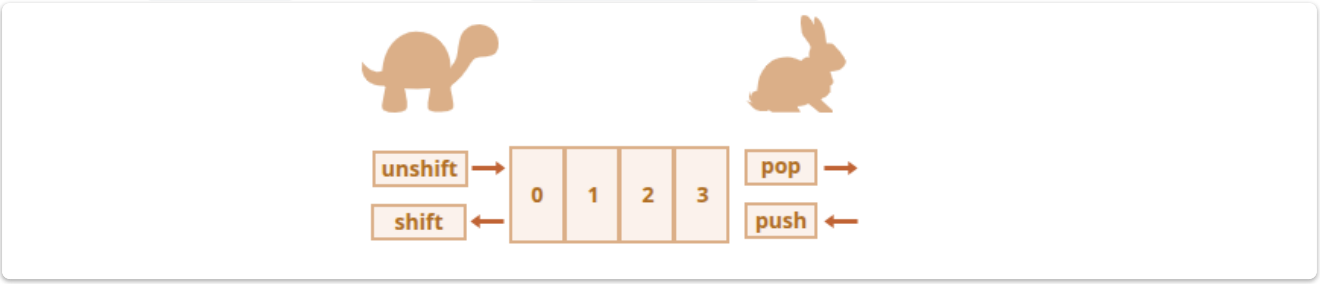
...But what makes arrays really special is their internal representation. The engine tries to store its elements in the contiguous memory area, one after another,and there are other optimizations as well, to make arrays work really fast.

```
let fruits = []; // make an array
fruits[99999] = 5; // assign a property with the index far greater than
its length
fruits.age = 25; // create a property with an arbitrary name
```

That's possible, because arrays are objects at their base. We can add any properties to them.
But the engine will see that we're working with the array as with a regular object.
Array-specific optimizations are not suited for such cases and will be turned off, their benefits disappear.
The ways to misuse an array:
Add a non-numeric property like arr.test = 5.
Make holes, like: add arr[0] and then arr[1000] (and nothing between them).
Fill the array in the reverse order, like arr[1000], arr[999] and so on.
Please think of arrays as special structures to work with the ordered data. They provide special methods for that. Arrays are carefully tuned inside JavaScript engines to work with contiguous ordered data, please use them this way. And if you need arbitrary keys, chances are high that you actually require a regular object {}.

Performance

Methods `push/pop` run fast, while `shift/unshift` are slow.



Why is it faster to work with the end of an array than with its beginning? Let's see what happens during the execution:

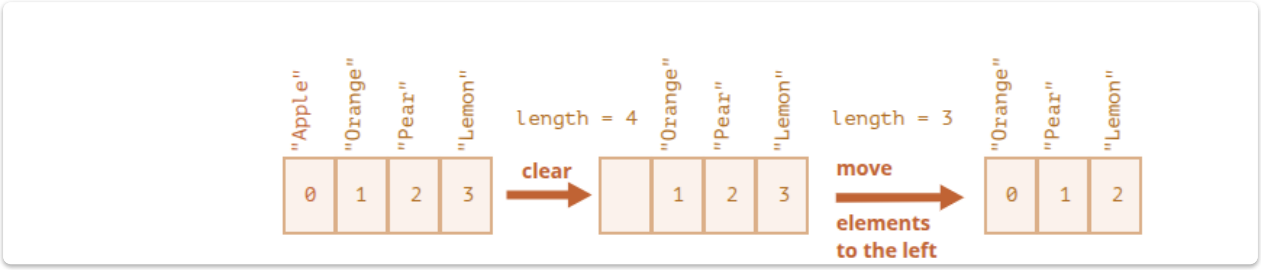
```
fruits.shift(); // take 1 element from the start
```

It's not enough to take and remove the element with the index 0. Other elements need to be renumbered as well.

The shift operation must do 3 things:

- Remove the element with the index 0.
- Move all elements to the left, renumber them from the index 1 to 0, from 2 to 1 and so on.

  Update the length property.



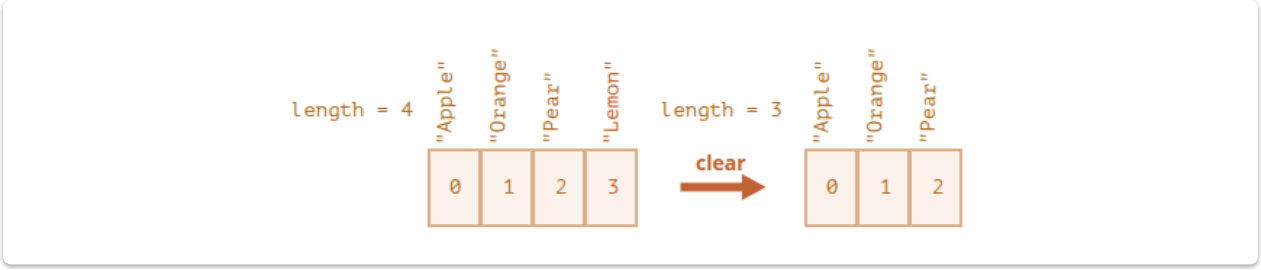
The more elements in the array, the more time to move them, more in-memory operations.

The similar thing happens with `unshift`: to add an element to the beginning of the array, we need first to move existing elements to the right, increasing their indexes.

And what's with `push/pop`? They do not need to move anything. To extract an element from the end, the `pop` method cleans the index and shortens `length`.

The actions for the `pop` operation:

```
fruits.pop(); // take 1 element from the end
```



The `pop` method does not need to move anything, because other elements keep their indexes. That's why it's blazingly fast.

Loops

One of the oldest ways to cycle array items is the for loop over indexes:

```
let arr = ["Apple", "Orange", "Pear"];
for (let i = 0; i < arr.length; i++) {
  alert( arr[i] );
}
// But for arrays there is another form of loop, `for..of`:
let fruits = ["Apple", "Orange", "Plum"];

// iterates over array elements
for (let fruit of fruits) {
  alert( fruit );
}
let arr = ["Apple", "Orange", "Pear"];
for (let key in arr) {
  alert( arr[key] ); // Apple, Orange, Pear
}
```

The `for..of` doesn't give access to the number of the current element, just its value, but in most cases that's enough. And it's shorter.
But that's actually a bad idea. There are potential problems with it:

1. The loop `for..in` iterates over *all properties*, not only the numeric ones.
   There are so-called "array-like" objects in the browser and in other environments, that *look like arrays*. That is, they have `length` and indexes properties, but they may also have other non-numeric properties and methods, which we usually don't need. The `for..in` loop will list them though. So if we need to work with array-like objects, then these "extra" properties can become a problem.
2. The `for..in` loop is optimized for generic objects, not arrays, and thus is 10-100 times slower. Of course, it's still very fast. The speedup may only matter in bottlenecks. But still we should be aware of the difference.
   Generally, we shouldn't use `for..in` for arrays.

A word about "length"

The `length` property automatically updates when we modify the array. To be precise, it is actually not the count of values in the array, but the greatest numeric index plus one.

```
let fruits = [];
fruits[123] = "Apple";
alert( fruits.length ); // 124
```

Another interesting thing about the length property is that it's writable.
If we increase it manually, nothing interesting happens. But if we decrease it, the array is truncated. *The process is irreversible*

```
let arr = [1, 2, 3, 4, 5];
arr.length = 2; // truncate to 2 elements
alert( arr ); // [1, 2]
arr.length = 5; // return length back
alert( arr[3] ); // undefined: the values do not return
```

`new Array()`

If new Array is called with a single argument which is a number, then it creates an array without items, but with the given length.

```
let arr = new Array("Apple", "Pear", "etc");
let arr = new Array(2); // will it create an array of [2] ?
alert( arr[0] ); // undefined! no elements.
alert( arr.length ); // length 2
```

toString

Arrays have their own implementation of `toString` method that returns a comma-separated list of elements.

```
let arr = [1, 2, 3];
alert( arr ); // 1,2,3
alert( String(arr) === '1,2,3' ); // true
alert( [] + 1 ); // "1"
alert( [1] + 1 ); // "11"
alert( [1,2] + 1 ); // "1,21"
alert( "" + 1 ); // "1"
alert( "1" + 1 ); // "11"
alert( "1,2" + 1 ); // "1,21"
```

Don’t compare arrays with ==

Arrays in JavaScript, unlike some other programming languages, shouldn’t be compared with operator `==`.
This operator has no special treatment for arrays, it works with them as with any objects.

```
alert( [] == [] ); // false
alert( [0] == [0] ); // false
```

These arrays are technically different objects. So they aren’t equal. The `==` operator doesn’t do item-by-item comparison.

```
alert( 0 == [] ); // true
alert('0' == [] ); // false
```

Array methods

Add/remove items

We already know methods that add and remove items from the beginning or the end:

- `arr.push(...items)` – adds items to the end,
- `arr.pop()` – extracts an item from the end,
- `arr.shift()` – extracts an item from the beginning,
- `arr.unshift(...items)` – adds items to the beginning.

splice

How to delete an element from the array?
The arrays are objects, so we can try to use delete

```
let arr = ["I", "go", "home"];
delete arr[1]; // remove "go"
alert( arr[1] ); // undefined
// now arr = ["I",  , "home"];
alert( arr.length ); // 3
```

The `arr.splice` method is a Swiss army knife for arrays. It can do everything: insert, remove and replace elements.

```
arr.splice(start[, deleteCount, elem1, ... , elemN])
```

It modifies `arr` starting from the index `start`: removes `deleteCount` elements and then inserts `elem1, ..., elemN` at their place. Returns the array of removed elements.

```
let arr = ["I", "study", "JavaScript"];
arr.splice(1, 1); // from index 1 remove 1 element
alert( arr ); // ["I", "JavaScript"]
```

we remove 3 elements and replace them with the other two:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
// remove 3 first elements and replace them with another
arr.splice(0, 3, "Let's", "dance");
alert( arr ) // now ["Let's", "dance", "right", "now"]
```

```
let arr = ["I", "study", "JavaScript"];

// from index 2
// delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");

alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

> ⚠ **Warning**
>
> Negative indexes allowed
> Here and in other array methods, negative indexes are allowed. They specify the position from the end of the array, like here:

```
let arr = [1, 2, 5];
// from index -1 (one step from the end)
// delete 0 elements,
// then insert 3 and 4
arr.splice(-1, 0, 3, 4);
alert( arr ); // 1,2,3,4,5
```

slice

The method `arr.slice` is much simpler than the similar-looking `arr.splice`.
`arr.slice([start], [end])`
It returns a new array copying to it all items from index start to end (not including end). Both start and end can be negative, in that case position from array end is assumed.

```
let arr = ["t", "e", "s", "t"];
alert( arr.slice(1, 3) ); // e,s (copy from 1 to 3)
alert( arr.slice(-2) ); // s,t (copy from -2 till the end)
```

concat

The method `arr.concat` creates a new array that includes values from other arrays and additional items.
`arr.concat(arg1, arg2...)`
Normally, it only copies elements from arrays. Other objects, even if they look like arrays, are added as a whole:

```
let arr = [1, 2];
let arrayLike = {
  0: "something",
```

```
  length: 1
};
alert( arr.concat(arrayLike) ); // 1,2,[object Object]
```

...But if an array-like object has a special Symbol.isConcatSpreadable property, then it's treated as an array by concat: its elements are added instead:

```
let arr = [1, 2];
let arrayLike = {
  0: "something",
  1: "else",
  [Symbol.isConcatSpreadable]: true,
  length: 2
};
alert( arr.concat(arrayLike) ); // 1,2,something,else
```

Iterate: `forEach`

```
arr.forEach(function(item, index, array) {
  // ... do something with an item
});
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
  alert(`${item} is at index ${index} in ${array}`);
});
```

Searching in array

`indexOf/lastIndexOf` and `includes`

The methods `arr.indexOf` and `arr.includes` have the similar syntax and do essentially the same as their string counterparts

- `arr.indexOf(item, from)` – looks for `item` starting from index `from`, and returns the index where it was found, otherwise `-1`.
- `arr.includes(item, from)` – looks for `item` starting from index `from`, returns `true` if found.

```
let arr = [1, 0, false];
alert( arr.indexOf(0) ); // 1
alert( arr.indexOf(false) ); // 2
alert( arr.indexOf(null) ); // -1
alert( arr.includes(1) ); // true
```

Please note that `indexOf` uses the strict equality `===` for comparison. So, if we look for `false`, it finds exactly `false` and not the zero.
If we want to check if `item` exists in the array and don't need the index,

then `arr.includes` is preferred.
The method `arr.lastIndexOf` is the same as `indexOf`, but looks for from right to left.

```
let fruits = ['Apple', 'Orange', 'Apple']
alert( fruits.indexOf('Apple') ); // 0 (first Apple)
alert( fruits.lastIndexOf('Apple') ); // 2 (last Appl
```

The `includes` method handles `NaN` correctly
A minor, but noteworthy feature of `includes` is that it correctly handles `NaN`, unlike `indexOf`:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (wrong, should be 0)
alert( arr.includes(NaN) );// true (correct)
```

find and `findIndex/findLastIndex`

Imagine we have an array of objects. How do we find an object with a specific condition?
Here the `arr.find(fn)` method comes in handy.

```
let result = arr.find(function(item, index, array) {
  // if true is returned, item is returned and iteration is stopped
  // for falsy scenario returns undefined
});
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
let user = users.find(item => item.id == 1);
alert(user.name); // John
```

In real life, arrays of objects are a common thing, so the `find` method is very useful.
Note that in the example we provide to `find` the function `item => item.id == 1` with one argument. That's typical, other arguments of this function are rarely used.

filter

The `find` method looks for a single (first) element that makes the function return `true`.

If there may be many, we can use `arr.filter(fn)`
The syntax is similar to `find`, but `filter` returns an array of all matching elements:

```
let results = arr.filter(function(item, index, array) {
  // if true item is pushed to results and the iteration continues
  // returns empty array if nothing found
});
let users = [
  {id: 1, name: "John"},
  {id: 2, name: "Pete"},
  {id: 3, name: "Mary"}
];
// returns array of the first two users
let someUsers = users.filter(item => item.id < 3);
alert(someUsers.length); // 2
```

Transform an array

Let's move on to methods that transform and reorder an array.

map

The `arr.map` method is one of the most useful and often used.
It calls the function for each element of the array and returns the array of results.

```
let result = arr.map(function(item, index, array) {
  // returns the new value instead of item
});
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

sort(fn)

The call to `arr.sort()` sorts the array in place, changing its element order.
It also returns the sorted array, but the returned value is usually ignored, as arr itself is modified.

```
let arr = [ 1, 2, 15 ];
// the method reorders the content of arr
arr.sort();
alert( arr );  // 1, 15, 2
```

Did you notice anything strange in the outcome?
The order became 1, 15, 2. Incorrect. But why?
The items are sorted as strings by default.
`arr.sort( (a, b) => a - b );`

reverse

The method arr.reverse reverses the order of elements in arr.

```
let arr = [1, 2, 3, 4, 5];
arr.reverse();

alert( arr ); // 5,4,3,2,1
```

split and join

We are writing a messaging app, and the person enters the comma-delimited list of receivers: `John, Pete, Mary`. But for us an array of names would be much more comfortable than a single string. How to get it?
``JS let names = 'Bilbo, Gandalf, Nazgul'; let arr = names.split(', '); for (let name of arr) { alert(A message to ${name}.`); // A message to Bilbo (and other names)
}

```
Split into letters
The call to `split(s)` with an empty `s` would split the string into an
array of letters:
```JS
let str = "test";

alert( str.split('') ); // t,e,s,t
```

The call `arr.join(glue)` does the reverse to split. It creates a string of arr items joined by glue between them.

```
let str = "test";
alert( str.split('') ); // t,e,s,t
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
let str = arr.join(';'); // glue the array into a string using ;
alert( str ); // Bilbo;Gandalf;Nazgul
```

`reduce/reduceRight`

When we need to iterate over an array – we can use `forEach`, `for` or `for..of`.
The methods `arr.reduce` and `arr.reduceRight` also belong to that breed, but are a little bit more intricate. They are used to calculate a single value based on the array.

```
let value = arr.reduce(function(accumulator, item, index, array) {
  // ...
}, [initial]);
```

The function is applied to all array elements one after another and “carries on” its result to the next call.
```

Arguments:

- `accumulator` – is the result of the previous function call, equals `initial` the first time (if `initial` is provided).
- `item` – is the current array item.
- `index` – is its position.
- `array` – is the array.

```
const arr = [1, 2, 3];

const sum = arr.reduce((acc, item) => acc + item, 0);
console.log(sum); // 6
//Without initial value:
const sum2 = arr.reduce((acc, item) => acc + item);
console.log(sum2); // 6 (still works!)
[].reduce((acc, item) => acc + item); // ❌ TypeError: Reduce of empty
array with no initial value
```

Building an object

```
const arr = ["a", "b", "c"];
const result = arr.reduce((acc, item, index) => {
  acc[item] = index;
  return acc;
}, {}); // ← initial is an empty object

console.log(result);
// { a: 0, b: 1, c: 2 }
```

If you don't pass initial = `{}`:
The first item "a" would be used as acc, which causes bugs.

Array.isArray

Arrays do not form a separate language type. They are based on objects.
So `typeof` does not help to distinguish a plain object from an array:

```
alert(Array.isArray({})); // false
alert(Array.isArray([])); // true
```

Map and Set

`Objects` are used for storing keyed collections.
`Arrays` are used for storing ordered collections.
But that's not enough for real life. That's why `Map` and `Set` also exist.

Map

`Map is a collection of keyed data items, just like an Object. But the main difference is that Map allows keys of any type.` new Map() `– creates the map.` map.set(key, value) `– stores the value by the key.` map.get(key) `– returns the value by the key, undefined if key doesn’t exist in map.` map.has(key) `– returns true if the key exists, false otherwise.` map.delete(key) `– removes the element (the key/value pair) by the key.` map.clear() `– removes everything from the map.` map.size` – returns the current element count.

```
let map = new Map();
map.set('1', 'str1');   // a string key
map.set(1, 'num1');     // a numeric key
map.set(true, 'bool1'); // a boolean key
// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:
alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'
alert( map.size ); // 3
```

> ⚠ **Warning**
>
> map[key] isn’t the right way to use a Map
> Although map[key] also works, e.g. we can set map[key] = 2, this is treating map as a plain JavaScript object, so it implies all corresponding limitations (only string/symbol keys and so on).
> So we should use map methods: set, get and so on.

Map can also use objects as keys.

```
let john = { name: "John" };
// for every user, let's store their visits count
let visitsCountMap = new Map();
// john is the key for the map
visitsCountMap.set(john, 123);
alert( visitsCountMap.get(john) ); // 123
```

Using objects as keys is one of the most notable and important `Map` features. The same does not count for `Object`. String as a key in `Object` is fine, but we can’t use another `Object` as a key in `Object`.

```
let john = { name: "John" };
let ben = { name: "Ben" };
let visitsCountObj = {}; // try to use an object
```

```
visitsCountObj[ben] = 234; // try to use ben object as the key
visitsCountObj[john] = 123; // try to use john object as the key, ben
object will get replaced

// That's what got written!
alert( visitsCountObj["[object Object]"] ); // 123
```

Note

How Map compares keys

To test keys for equivalence, Map uses the algorithm SameValueZero. It is roughly the same as strict equality ===, but the difference is that NaN is considered equal to NaN. So NaN can be used as the key as well.

This algorithm can't be changed or customized.

```
Every map.set call returns the map itself, so we can "chain" the calls:
map.set('1', 'str1')
  .set(1, 'num1')
  .set(true, 'bool1');
```

Iteration over Map

For looping over a map, there are 3 methods:

`map.keys()` – returns an iterable for keys,

`map.values()` – returns an iterable for values,

`map.entries()` – returns an iterable for entries [key, value], it's used by default in for..of.

```
let recipeMap = new Map([
  ['cucumber', 500],
  ['tomatoes', 350],
  ['onion',    50]
]);
// iterate over keys (vegetables)
for (let vegetable of recipeMap.keys()) {
  alert(vegetable); // cucumber, tomatoes, onion
}
// iterate over values (amounts)
for (let amount of recipeMap.values()) {
  alert(amount); // 500, 350, 50
}
// iterate over [key, value] entries
for (let entry of recipeMap) { // the same as of recipeMap.entries()
```

```
  alert(entry); // cucumber,500 (and so on)
}
```

The iteration goes in the same order as the values were inserted. `Map` preserves this order, unlike a regular `Object`.

Besides that, Map has a built-in `forEach` method, similar to Array:

```
recipeMap.forEach( (value, key, map) => {
  alert(`${key}: ${value}`); // cucumber: 500 etc
});
```

Object.entries: Map from Object

```
// array of [key, value] pairs
let map = new Map([
  ['1',  'str1'],
  [1,    'num1'],
  [true, 'bool1']
]);
alert( map.get('1') ); // str1
```

If we have a plain object, and we'd like to create a Map from it, then we can use built-in method `Object.entries(obj)` that returns an array of key/value pairs for an object exactly in that format.

```
let obj = {
  name: "John",
  age: 30
};
let map = new Map(Object.entries(obj));
alert( map.get('name') ); // John
```

Here, Object.entries returns the array of key/value pairs: [["name","John"], ["age", 30]]. That's what Map needs.

Object.fromEntries: Object from Map

We've just seen how to create Map from a plain object with `Object.entries(obj).` There's `Object.fromEntries` method that does the reverse: given an array of [key, value] pairs, it creates an object from them:

```
let prices = Object.fromEntries([
  ['banana', 1],
  ['orange', 2],
  ['meat', 4]
]);
// now prices = { banana: 1, orange: 2, meat: 4 }
alert(prices.orange); // 2
```

We can use `Object.fromEntries` to get a plain object from `Map`.
E.g. we store the data in a `Map`, but we need to pass it to a 3rd-party code that expects a plain object.

```
let map = new Map();
map.set('banana', 1);
map.set('orange', 2);
map.set('meat', 4);
let obj = Object.fromEntries(map.entries()); // make a plain object (*)
// done!
// obj = { banana: 1, orange: 2, meat: 4 }
alert(obj.orange); // 2
```

A call to map.entries() returns an iterable of key/value pairs, exactly in the right format for `Object.fromEntries.`

Set

A `Set` is a special type collection – `set of values` (without keys), where each value may occur only once.
Its main methods are:

- `new Set([iterable])` – creates the set, and if an iterable object is provided (usually an array), copies values from it into the set.
- `set.add(value)` – adds a value, returns the set itself.
- `set.delete(value)` – removes the value, returns true if value existed at the moment of the call, otherwise false.
- `set.has(value)` – returns true if the value exists in the set, otherwise false.
- `set.clear()` – removes everything from the set.
  `set.size` – is the elements count.

The main feature is that repeated calls of `set.add(value)` with the same value don't do anything. That's the reason why each value appears in a `Set` only once.

```
let set = new Set();
let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };
// visits, some users come multiple times
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);
// set keeps only unique values
alert( set.size ); // 3
for (let user of set) {
  alert(user.name); // John (then Pete and Mary)
}
```

The alternative to `Set` could be an array of users, and the code to check for duplicates on every insertion using `arr.find` But the performance would be much worse, because this method walks through the whole array checking every element. `Set` is much better optimized internally for uniqueness checks.

Iteration over Set

We can loop over a set either with `for..of` or using `forEach`:

```
let set = new Set(["oranges", "apples", "bananas"]);
for (let value of set) alert(value);
// the same with forEach:
set.forEach((value, valueAgain, set) => {
  alert(value);
});
```

Note the funny thing. The callback function passed in `forEach` has 3 arguments: a `value`, then *the same value* `valueAgain`, and then the target object. Indeed, the same value appears in the arguments twice.

That's for compatibility with `Map` where the callback passed `forEach` has three arguments. Looks a bit strange, for sure. But this may help to replace `Map` with `Set` in certain cases with ease, and vice versa.
The same methods `Map` has for iterators are also supported:

- `set.keys()` – returns an `iterable` object for values,
- `set.values()` – same as `set.keys()`, for compatibility with `Map`,
- `set.entries()` – returns an `iterable` object for entries `[value, value]`, exists for compatibility with `Map`.

Object.keys, values, entries

In the previous chapter we saw methods `map.keys()`, `map.values()`, `map.entries()`. These methods are generic, there is a common agreement to use them for data structures. If we ever create a data structure of our own, we should implement them too.
They are supported for:

- `Map`
- `Set`
- `Array`

Object.keys, values, entries

For plain objects, the following methods are available:
Object.keys(obj) – returns an array of keys.
`Object.values(obj)` – returns an array of values.
`Object.entries(obj)` – returns an array of [key, value] pairs.

The first difference is that we have to call `Object.keys(obj)`, and not `obj.keys()`
Why so? The main reason is flexibility. Remember, objects are a base of all complex structures in JavaScript. So we may have an object of our own like `data` that implements its own `data.values()` method. And we still can call `Object.values(data)` on it.

```
let user = {
  name: "John",
  age: 30
};
```

- `Object.keys(user) = ["name", "age"]`
- `Object.values(user) = ["John", 30]`
- `Object.entries(user) = [ ["name","John"], ["age",30] ]`
  Here's an example of using Object.values to loop over property values:

```
let user = {
  name: "John",
  age: 30
};
// loop over values
for (let value of Object.values(user)) {
  alert(value); // John, then 30
}
```

```
let prices = {
  banana: 1,
  orange: 2,
  meat: 4,
}
```

Goal: Double all prices

```
{
 banana: 2,
 orange: 4,
 meat: 8
}
```

Step-by-step Breakdown

`Object.entries(prices)`
This converts the object to an array of [key, value] pairs:

```
[
  ["banana", 1],
  ["orange", 2],
  ["meat", 4]
]
```

`Step 2: .map(entry => [entry[0], entry[1] * 2])`
You are looping through each [key, value] pair and transforming it.

```
.map(entry => [entry[0], entry[1] * 2])
.map(([key, value]) => [key, value * 2])

// result
[
  ["banana", 2],
  ["orange", 4],
  ["meat", 8]
]
```

Step 3: `Object.fromEntries(...)`
Now you want to convert that array of `key-value` pairs back into an object.

```
Object.fromEntries([
  ["banana", 2],
```

```
  ["orange", 4],
  ["meat", 8]
])

{
  banana: 2,
  orange: 4,
  meat: 8
}
```

Final code recap:

```
let doublePrices = Object.fromEntries(
  Object.entries(prices).map(([key, value]) => [key, value * 2])
);
```

Destructuring assignment

The two most used data structures in JavaScript are `Object` and `Array`. `Objects` allow us to create a single entity that stores data items by `key`. `Arrays` allow us to gather data items into an ordered list.

Array destructuring

```
let arr = ["John", "Smith"]
// destructuring assignment
// sets firstName = arr[0]
// and surname = arr[1]
let [firstName, surname] = arr;
alert(firstName); // John
alert(surname);  // Smith
```

It looks great when combined with `split` or other array-returning methods:

```
let [firstName, surname] = "John Smith".split(' ');
alert(firstName); // John
alert(surname);  // Smith
```

> **Danger**
>
> “Destructuring” does not mean “destructive”.

It’s called “`destructuring` assignment,” because it “`destructurizes`” by copying items into variables. However, the array itself is not modified.

Assign to anything at the left-side

```
let user = {};
[user.name, user.surname] = "John Smith".split(' ');

alert(user.name); // John
alert(user.surname); // Smith
```

`The rest '…'`

Date and time

Let's meet a new built-in object: `Date`. It stores the date, time and provides methods for date/time management.
For instance, we can use it to store creation/modification times, to measure time, or just to print out the current date.

Creation

To create a new `Date` object call `new Date()` with one of the following arguments:
`new Date()`

```
let now = new Date();
alert( now ); // shows current date/time
```

`new Date(year, month, date, hours, minutes, seconds, ms)`
Create the date with the given components in the local time zone. Only the first two arguments are obligatory.

- The `year` should have 4 digits. For compatibility, 2 digits are also accepted and considered `19xx`, e.g. `98` is the same as `1998` here, but always using 4 digits is strongly encouraged.
- The `month` count starts with `0` (Jan), up to `11` (Dec).
- The `date` parameter is actually the day of month, if absent then `1` is assumed.
- If `hours/minutes/seconds/ms` is absent, they are assumed to be equal `0`.

```
new Date(2011, 0, 1, 0, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // the same, hours etc are 0 by default
```

The maximal precision is 1 ms (1/1000 sec):

```
let date = new Date(2011, 0, 1, 2, 3, 4, 567);
alert( date ); // 1.01.2011, 02:03:04.567
```

Access date components

There are methods to access the year, month and so on from the Date object:

`getFullYear()`
Get the year (4 digits)
`getMonth()`
Get the month, from 0 to 11.
`getDate()`
Get the day of month, from 1 to 31, the name of the method does look a little bit strange.
`getHours(), getMinutes(), getSeconds(), getMilliseconds()`
Get the corresponding time components.

JSON methods, toJSON

Let's say we have a complex object, and we'd like to convert it into a string, to send it over a network, or just to output it for logging purposes.
Naturally, such a string should include all important properties.

```
let user = {
  name: "John",
  age: 30,
  toString() {
    return `{name: "${this.name}", age: ${this.age}}`;
  }
};
alert(user); // {name: "John", age: 30}
```

JSON.stringify

The `JSON` (JavaScript Object Notation) is a general format to represent values and objects. It is described as in RFC 4627 standard. Initially it was made for JavaScript, but many other languages have libraries to handle it as well. So it's easy to use `JSON` for data exchange when the client uses JavaScript and the server is written on Ruby/PHP/Java/Whatever.
JavaScript provides methods:

- `JSON.stringify` to convert objects into JSON.
- `JSON.parse` to convert JSON back into an object.

For instance, here we `JSON.stringify` a student:

```
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
```

```
  spouse: null
};
let json = JSON.stringify(student);
alert(typeof json); // we've got a string!
alert(json);
```

The method `JSON.stringify(student)` takes the object and converts it into a string.
The resulting `json` string is called a *JSON-encoded* or *serialized* or *stringified* or *marshalled* object. We are ready to send it over the wire or put into a plain data store.
Please note that a JSON-encoded object has several important differences from the object literal:

- Strings use double quotes. No single quotes or backticks in JSON.
  So `'John'` becomes `"John"`.
- Object property names are double-quoted also. That's obligatory.
  So `age:30` becomes `"age":30`.
  `JSON.stringify` can be applied to primitives as well.
  `JSON` supports following data types:
- Objects `{ ... }`
- Arrays `[ ... ]`
- Primitives:
  - strings,
  - numbers,
  - boolean values `true/false`,
  - `null`.

```
// a number in JSON is just a number
alert( JSON.stringify(1) ) // 1
// a string in JSON is still a string, but double-quoted
alert( JSON.stringify('test') ) // "test"
alert( JSON.stringify(true) ); // true
alert( JSON.stringify([1, 2, 3]) ); // [1,2,3]
```

JSON is data-only language-independent specification, so some JavaScript-specific object properties are skipped by `JSON.stringify`.
Namely:

- Function properties (methods).
- Symbolic keys and values.
- Properties that store `undefined`.

```
let user = {
  sayHi() { // ignored
```

```
    alert("Hello");
  },
  [Symbol("id")]: 123, // ignored
  something: undefined // ignored
};
```

Let's go over each property in `user`:

1. `sayHi()`
   - It's a function.
   - `JSON.stringify()` ignores functions.
2. `[Symbol("id")]`
   - This is a symbol-keyed property.
   - Symbol properties are ignored by `JSON.stringify()`.
3. *`something: undefined`*
   - `undefined` values are ignored.

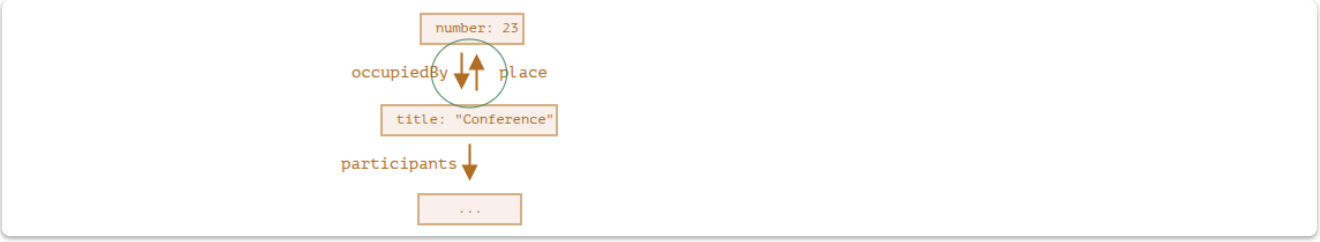
The great thing is that nested objects are supported and converted automatically.

```
let meetup = {
  title: "Conference",
  room: {
    number: 23,
    participants: ["john", "ann"]
  }
};
alert( JSON.stringify(meetup) );
/* The whole structure is stringified:
{
  "title":"Conference",
  "room":{"number":23,"participants":["john","ann"]},
}
*/
```

The important limitation: there must be no circular references.

```
let room = {
  number: 23
};
let meetup = {
  title: "Conference",
  participants: ["john", "ann"]
};
meetup.place = room;       // meetup references room
room.occupiedBy = meetup; // room references meetup
JSON.stringify(meetup); // Error: Converting circular structure to JSON
```

Here, the conversion fails, because of circular reference: `room.occupiedBy` references `meetup`, and `meetup.place` references `room`:



Excluding and transforming: replacer

```
let json = JSON.stringify(value[, replacer, space])
```

`value`
A value to encode.
`replacer`
Array of properties to encode or a mapping function `function(key, value)`.
`space`
Amount of space to use for formatting
Most of the time, `JSON.stringify` is used with the first argument only. But if we need to fine-tune the replacement process, like to filter out circular references, we can use the second argument of `JSON.stringify`.
If we pass an array of properties to it, only these properties will be encoded.

JSON.parse

To decode a JSON-string, we need another method named `JSON.parse.`

```
let value = JSON.parse(str[, reviver]);
// stringified array
let numbers = "[0, 1, 2, 3]";
numbers = JSON.parse(numbers);
alert( numbers[1] ); // 1

let userData = '{ "name": "John", "age": 35, "isAdmin": false, "friends":
[0,1,2,3] }';

let user = JSON.parse(userData);

alert( user.friends[1] ); // 1
```

Advanced working with functions

Recursion and stack

When a function solves a task, in the process it can call many other functions. A partial case of this is when a function calls itself. That's called `recursion`.

Two ways of thinking

For something simple to start with – let's write a function `pow(x, n)` that raises `x` to a natural power of `n`. In other words, multiplies `x` by itself `n` times.

```
pow(2, 2) = 4
pow(2, 3) = 8
pow(2, 4) = 16
```

There are two ways to implement it.

1. 1.Iterative thinking: the `for` loop:

```
function pow(x, n) {
let result = 1;
// multiply result by x n times in the loop
for (let i = 0; i < n; i++) {
  result *= x;
}
return result;
}
alert( pow(2, 3) ); // 8
```

Recursive thinking: simplify the task and call self:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}

alert( pow(2, 3) ); //
```

Rest parameters and spread syntax

Rest parameters ...

A function can be called with any number of arguments, no matter how it is defined.

```
function sum(a, b) {
  return a + b;
}

alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of “excessive” arguments. But of course in the result only the first two will be counted, so the result in the code above is `3`.
The rest of the parameters can be included in the function definition by using three dots `...` followed by the name of the array that will contain them. The dots literally mean “gather the remaining parameters into an array”.

```
function sumAll(...args) { // args is the name for the array
  let sum = 0;

  for (let arg of args) sum += arg;

  return sum;
}

alert( sumAll(1) ); // 1
alert( sumAll(1, 2) ); // 3
alert( sumAll(1, 2, 3) ); // 6
```

Danger

The rest parameters must be at the end
The rest parameters gather all remaining arguments, so the following does not make sense and causes an error:

The `...rest` must always be last.

Spread

```
let str = "Hello";
alert( [...str] ); // H,e,l,l,o
```

The spread syntax internally uses iterators to gather elements, the same way as `for..of` does.
So, for a string, `for..of` returns characters
and `...str` becomes `"H","e","l","l","o"`. The list of characters is passed to array initializer `[...str]`.
For this particular task we could also use `Array.from`, because it converts an iterable (like a string) into an array:

```
let str = "Hello";
// Array.from converts an iterable into an array
alert( Array.from(str) ); // H,e,l,l,o
```

Copy an array/object

```
let arr = [1, 2, 3];
let arrCopy = [...arr]; // spread the array into a list of parameters
                        // then put the result into a new array

// do the arrays have the same contents?
alert(JSON.stringify(arr) === JSON.stringify(arrCopy)); // true
// are the arrays equal?
alert(arr === arrCopy); // false (not same reference)
// modifying our initial array does not modify the copy:
arr.push(4);
alert(arr); // 1, 2, 3, 4
alert(arrCopy); // 1, 2, 3
```

Note that it is possible to do the same thing to make a copy of an object:

```
let obj = { a: 1, b: 2, c: 3 };
let objCopy = { ...obj }; // spread the object into a list of parameters
                          // then return the result in a new object

// do the objects have the same contents?
alert(JSON.stringify(obj) === JSON.stringify(objCopy)); // true
// are the objects equal?
alert(obj === objCopy); // false (not same reference)
// modifying our initial object does not modify the copy:
obj.d = 4;
alert(JSON.stringify(obj)); // {"a":1,"b":2,"c":3,"d":4}
alert(JSON.stringify(objCopy)); // {"a":1,"b":2,"c":3}
```

Variable scope, closure

Code blocks

If a variable is declared inside a code block `{...}`, it's only visible inside that block.

```
{
  // do some job with local variables that should not be seen outside
  let message = "Hello"; // only visible in this block
  alert(message); // Hello
}
alert(message); // Error: message is not defined
```

We can use this to isolate a piece of code that does its own task, with variables that only belong to it:

```
{
  // show message
  let message = "Hello";
```

```
  alert(message);
}
{
  // show another message
  let message = "Goodbye";
  alert(message);
}
```

There'd be an error without blocks

```
// show message
let message = "Hello";
alert(message);

// show another message
let message = "Goodbye"; // Error: variable already declared
alert(message);
```

For if, for, while and so on, variables declared in {...} are also only visible inside:

```
if (true) {
  let phrase = "Hello!";

  alert(phrase); // Hello!
}

alert(phrase); // Error, no such variable!

for (let i = 0; i < 3; i++) {
  // the variable i is only visible inside this for
  alert(i); // 0, then 1, then 2
}

alert(i); // Error, no such variable
```

Nested functions

```
function sayHiBye(firstName, lastName) {
  // helper nested function to use below
  function getFullName() {
    return firstName + " " + lastName;
  }
  alert( "Hello, " + getFullName() );
  alert( "Bye, " + getFullName() );

}
```

Lexical Environment

Step 1. Variables

In JavaScript, every running function, code block `{...}`, and the script as a whole have an internal (hidden) associated object known as the *Lexical Environment*.
The Lexical Environment object consists of two parts:

1. *Environment Record* – an object that stores all local variables as its properties (and some other information like the value of `this`).
2. A reference to the *outer lexical environment*, the one associated with the outer code.

> **Note**
> A "variable" is just a property of the special internal object, Environment Record. "To get or change a variable" means "to get or change a property of that object".

In this simple code without functions, there is only one Lexical Environment:

This is the so-called *global* Lexical Environment, associated with the whole script.
On the picture above, the rectangle means Environment Record (variable store) and the arrow means the outer reference. The global Lexical Environment has no outer reference, that's why the arrow points to `null`.

Rectangles on the right-hand side demonstrate how the global Lexical Environment changes during the execution:

1. When the script starts, the Lexical Environment is pre-populated with all declared variables.
   - Initially, they are in the "Uninitialized" state. That's a special internal state, it means that the engine knows about the variable, but it cannot be referenced until it has been declared with `let`. It's almost the same as if the variable didn't exist.
2. Then `let phrase` definition appears. There's no assignment yet, so its value is `undefined`. We can use the variable from this point forward.

3. `phrase` is assigned a value.
4. `phrase` changes the value.
   Everything looks simple for now, right?

- A variable is a property of a special internal object, associated with the currently executing block/function/script.
- Working with variables is actually working with the properties of that object.

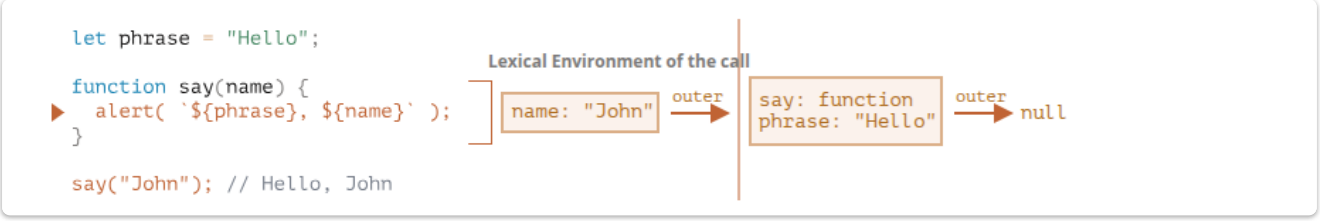
Step 2. Function Declarations

A function is also a value, like a variable.
The difference is that a Function Declaration is instantly fully initialized.
When a Lexical Environment is created, a Function Declaration immediately becomes a ready-to-use function (unlike `let`, that is unusable till the declaration).
That's why we can use a function, declared as Function Declaration, even before the declaration itself.

Naturally, this behavior only applies to Function Declarations, not Function Expressions where we assign a function to a variable, such as `let say = function(name)...`.

Step 3. Inner and outer Lexical Environment

When a function runs, at the beginning of the call, a new Lexical Environment is created automatically to store local variables and parameters of the call.
For instance, for `say("John")`, it looks like this (the execution is at the line, labelled with an arrow):



During the function call we have two Lexical Environments: the inner one (for the function call) and the outer one (global):

- The inner Lexical Environment corresponds to the current execution of `say`. It has a single property: `name`, the function argument. We called `say("John")`, so the value of the `name` is `"John"`.
- The outer Lexical Environment is the global Lexical Environment. It has the `phrase` variable and the function itself.

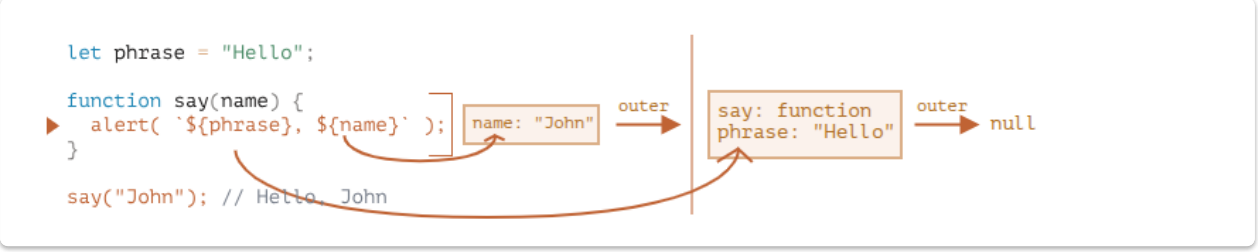
The inner Lexical Environment has a reference to the `outer` one.

When the code wants to access a variable – the inner Lexical Environment is searched first, then the outer one, then the more outer one and so on until the global one.

If a variable is not found anywhere, that's an error in strict mode (without `use strict`, an assignment to a non-existing variable creates a new global variable, for compatibility with old code).

In this example the search proceeds as follows:

- For the `name` variable, the `alert` inside `say` finds it immediately in the inner Lexical Environment.
- When it wants to access `phrase`, then there is no `phrase` locally, so it follows the reference to the outer Lexical Environment and finds it there.



Step 4. Returning a function

Let's return to the `makeCounter` example.

```
function makeCounter() {
  let count = 0;

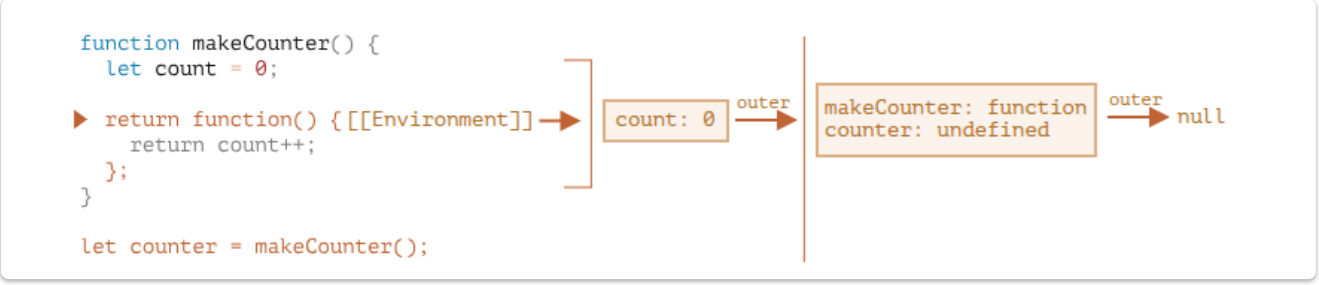
  return function() {
    return count++;
  };
}
let counter = makeCounter();
```

At the beginning of each `makeCounter()` call, a new Lexical Environment object is created, to store variables for this `makeCounter` run.

So we have two nested Lexical Environments, just like in the example above:

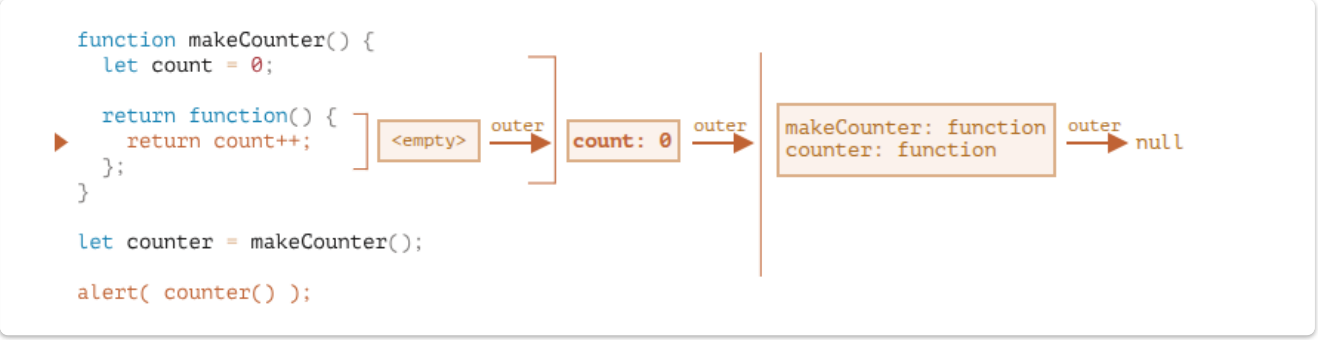
What's different is that, during the execution of `makeCounter()`, a tiny nested function is created of only one line: `return count++`. We don't run it yet, only create.

All functions remember the Lexical Environment in which they were made. Technically, there's no magic here: all functions have the hidden property named `[[Environment]]`, that keeps the reference to the Lexical Environment where the function was created:



So, `counter.[[Environment]]` has the reference to `{count: 0}` Lexical Environment. That's how the function remembers where it was created, no matter where it's called. The `[[Environment]]` reference is set once and forever at function creation time.

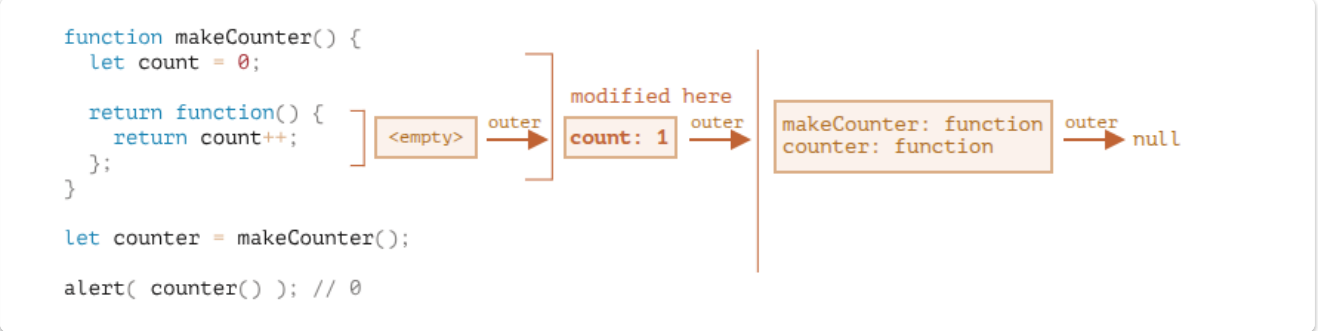
Later, when `counter()` is called, a new Lexical Environment is created for the call, and its outer Lexical Environment reference is taken from `counter.[[Environment]]`:



Now when the code inside `counter()` looks for `count` variable, it first searches its own Lexical Environment (empty, as there are no local variables there), then the Lexical Environment of the outer `makeCounter()` call, where it finds and changes it.

A variable is updated in the Lexical Environment where it lives.

Here's the state after the execution:



If we call `counter()` multiple times, the `count` variable will be increased to `2`, `3` and so on, at the same place.

Closure

There is a general programming term "`closure`", that developers generally should know.

A closure is a function that remembers its outer variables and can access them. In

some languages, that's not possible, or a function should be written in a special way to make it happen. But as explained above, in JavaScript, all functions are naturally closures

Garbage collection

Usually, a Lexical Environment is removed from memory with all the variables after the function call finishes. That's because there are no references to it. As any JavaScript object, it's only kept in memory while it's reachable.
However, if there's a nested function that is still reachable after the end of a function, then it has `[[Environment]]` property that references the lexical environment.
In that case the Lexical Environment is still reachable even after the completion of the function, so it stays alive.

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // g.[[Environment]] stores a reference to the Lexical
Environment
// of the corresponding f() call
```

Please note that if `f()` is called many times, and resulting functions are saved, then all corresponding Lexical Environment objects will also be retained in memory.

```
function f() {
  let value = Math.random();

  return function() { alert(value); };
}

// 3 functions in array, every one of them links to Lexical Environment
// from the corresponding f() run
let arr = [f(), f(), f()];
```

A Lexical Environment object dies when it becomes unreachable (just like any other object). In other words, it exists only while there's at least one nested function referencing it.

after the nested function is removed, its enclosing Lexical Environment (and hence the `value`) is cleaned from memory:

```
function f() {
  let value = 123;

  return function() {
    alert(value);
  }
}

let g = f(); // while g function exists, the value stays in memory

g = null; // ...and now the memory is cleaned up
```

Global object

The global object provides variables and functions that are available anywhere. By default, those that are built into the language or the environment.
In a browser it is named `window`, for Node.js it is `global`, for other environments it may have another name.
Recently, `globalThis` was added to the language, as a standardized name for a global object, that should be supported across all environments. It's supported in all major browsers.
We'll use `window` here, assuming that our environment is a browser. If your script may run in other environments, it's better to use `globalThis` instead.

```
alert("Hello");
// is the same as
window.alert("Hello");
```

In a browser, global functions and variables declared with `var` (not `let/const`!) become the property of the global object:

```
var gVar = 5;
alert(window.gVar); // 5 (became a property of the global object)
```

Function declarations have the same effect (statements with function keyword in the main code flow, not function expressions).
Please don't rely on that! This behavior exists for compatibility reasons. Modern scripts use JavaScript modules where such a thing doesn't happen.
If we used let instead, such thing wouldn't happen:

```
let gLet = 5;
alert(window.gLet); // undefined (doesn't become a property of the global
object)
```

If a value is so important that you'd like to make it available globally, write it directly as a property:

```
// make current user information global, to let all scripts access it
window.currentUser = {
  name: "John"
};
// somewhere else in code
alert(currentUser.name);  // John
// or, if we have a local variable with the name "currentUser"
// get it from window explicitly (safe!)
alert(window.currentUser.name); // John
```

That said, using global variables is generally discouraged. There should be as few global variables as possible. The code design where a function gets "input" variables and produces certain "outcome" is clearer, less prone to errors and easier to test than if it uses outer or global variables.

Function object, NFE

As we already know, a function in JavaScript is a value.
Every value in JavaScript has a type. What type is a function?
In JavaScript, functions are `objects`.
A good way to imagine functions is as callable "action objects". We can not only call them, but also treat them as objects: add/remove properties, pass by reference etc.

The "name" property

Function objects contain some useable properties.
For instance, a function's name is accessible as the "name" property:

```
function sayHi() {
  alert("Hi");
}
alert(sayHi.name); // sayHi
```

What's kind of funny, the name-assigning logic is smart. It also assigns the correct name to a function even if it's created without one, and then immediately assigned:

```
let sayHi = function() {
  alert("Hi");
};
alert(sayHi.name); // sayHi (there's a name!)
```

It also works if the assignment is done via a default value:

```
function f(sayHi = function() {}) {
  alert(sayHi.name); // sayHi (works!)
}
f();
```

In the specification, this feature is called a "`contextual name`". If the function does not provide one, then in an assignment it is figured out from the context.

```
let user = {
  sayHi() {
    // ...
  },
  sayBye: function() {
    // ...
  }
}
alert(user.sayHi.name); // sayHi
alert(user.sayBye.name); // sayBye
```

There's no magic though. There are cases when there's no way to figure out the right name. In that case, the name property is empty, like here:

```
// function created inside array
let arr = [function() {}];
alert( arr[0].name ); // <empty string>
// the engine has no way to set up the right name, so there is none
```

The "length" property

There is another built-in property "length" that returns the number of function parameters, for instance:

```
function f1(a) {}
function f2(a, b) {}
function many(a, b, ...more) {}

alert(f1.length); // 1
alert(f2.length); // 2
alert(many.length); // 2
```