

These are just the main things to know before you can jump into react or node, just the let's say starter kit, otherwise JavaScript is very vast and ever changing.

- Interpreted v/s compiled languages
- Single threaded nature of JS
- variables(let,var,const)
- Data Types(string,numbers,booleans)
- if/else
- Loops
- Arrays
- Objects
- Functions
- Callbacks
- forEach
- String Functions
- parseFunctions
- Array Functions
- Intro to class
- Date object
- Math object
- Object methods
- Async
- Map/Filter/SpreAD/DESTRUCTURING

*JavaScript* was initially created to “**make web pages alive**”.

The programs in this language are called **scripts**. They can be written right in a web page’s HTML and run automatically as the page loads.

Scripts are provided and executed as plain text. They don’t need special preparation or compilation to run.

In this aspect, JavaScript is very different from another language called [Java](#).

Today, JavaScript can execute not only in the browser, but also on the server, or actually on any device that has a special program called [the JavaScript engine](#).

The browser has an embedded engine sometimes called a “JavaScript virtual machine”.

Different engines have different “codenames”. For example:

- [V8](#) – in Chrome, Opera and Edge.
- [SpiderMonkey](#) – in Firefox.
- ...There are other codenames like “Chakra” for IE, “JavaScriptCore”, “Nitro” and “SquirrelFish” for Safari, etc.

## What can in-browser JavaScript do?

Modern JavaScript is a “safe” programming language. It does not provide low-level access to memory or the CPU, because it was initially created for browsers which do not require it.

JavaScript’s capabilities greatly depend on the environment it’s running in. For instance, [Node.js](#) supports functions that allow JavaScript to read/write arbitrary files, perform network requests, etc.

In-browser JavaScript can do everything related to webpage manipulation, interaction with the user, and the webserver.

For instance, in-browser JavaScript is able to:

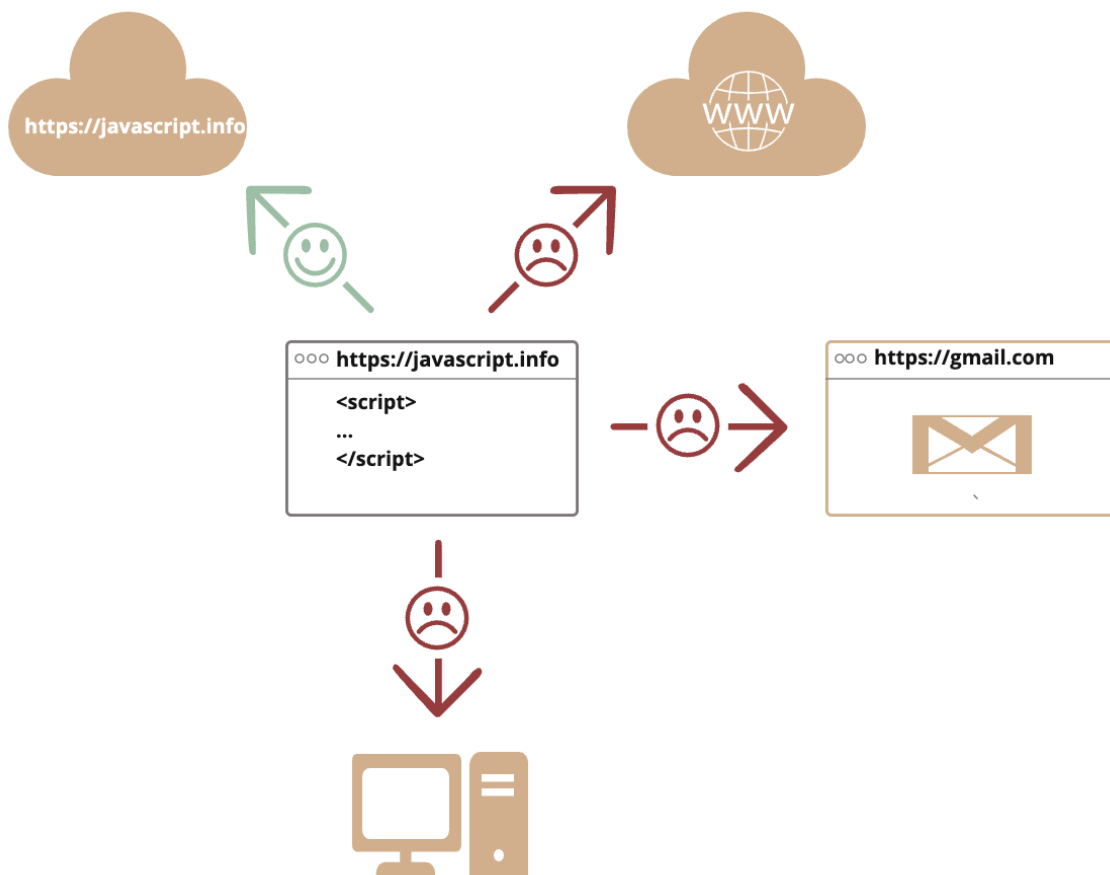
- **Add new HTML to the page, change the existing content, modify styles.**
- **React to user actions, run on mouse clicks, pointer movements, key presses.**
- **Send requests over the network to remote servers, download and upload files (so-called [AJAX](#) and [COMET](#) technologies).**
- **Get and set cookies, ask questions to the visitor, show messages.**
- **Remember the data on the client-side (“local storage”).**

## What CAN'T in-browser JavaScript do?

JavaScript's abilities in the browser are limited to protect the user's safety. The aim is to prevent an evil webpage from accessing private information or harming the user's data.

Examples of such restrictions include:

- JavaScript on a webpage may not read/write arbitrary files on the hard disk, copy them or execute programs. It has no direct access to OS functions. Modern browsers allow it to work with files, but the access is limited and only provided if the user does certain actions, like "dropping" a file into a browser window or selecting it via an `<input>` tag. There are ways to interact with the camera/microphone and other devices, but they require a user's explicit permission. So a JavaScript-enabled page may not sneakily enable a web-camera, observe the surroundings and send the information to the [NSA](#).
- Different tabs/windows generally do not know about each other. Sometimes they do, for example when one window uses JavaScript to open the other one. But even in this case, JavaScript from one page may not access the other page if they come from different sites (from a different domain, protocol or port). This is called the "Same Origin Policy". To work around that, *both pages* must agree for data exchange and must contain special JavaScript code that handles it. We'll cover that in the tutorial. This limitation is, again, for the user's safety. A page from `http://anysite.com` which a user has opened must not be able to access another browser tab with the URL `http://gmail.com`, for example, and steal information from there.
- JavaScript can easily communicate over the net to the server where the current page came from. But its ability to receive data from other sites/domains is crippled. Though possible, it requires explicit agreement (expressed in HTTP headers) from the remote side. Once again, that's a safety limitation.



Such limitations do not exist if JavaScript is used outside of the browser, for example on a server. Modern browsers also allow plugins/extensions which may ask for extended permissions.

## Summary

- JavaScript was initially created as a browser-only language, but it is now used in many other environments as well.
- Today, JavaScript has a unique position as the most widely-adopted browser language, fully integrated with HTML/CSS.
- There are many languages that get “transpiled” to JavaScript and provide certain features. It is recommended to take a look at them, at least briefly, after mastering JavaScript.

## Specification

The [ECMA-262 specification](#) contains the most in-depth, detailed and formalized information about JavaScript. It defines the language.

# Variables

We can declare variables to store data by using the `var`, `let`, or `const` keywords.

`let` – is a modern variable declaration.

`var` – is an old-school variable declaration. Normally we don't use it at all, but we'll cover subtle differences from `let` in the chapter [The old "var"](#), just in case you need them.

`const` – is like `let`, but the value of the variable can't be changed.

Variables should be named in a way that allows us to easily understand what's inside them.

## The old "var"

The `var` declaration is similar to `let`. Most of the time we can replace `let` by `var` or vice-versa and expect things to work

But internally `var` is a very different beast, that originates from very old times. It's generally not used in modern scripts, but still lurks in the old ones.

## "var" has no block scope

Variables, declared with `var`, are either function-scoped or global-scoped. They are visible through blocks.

For instance:

```
if (true) {  
  var test = true; // use "var" instead of "let"  
}
```

```
alert(test); // true, the variable lives after if
```

=====

=====

As `var` ignores code blocks, we've got a global variable `test`.

If we used `let test` instead of `var test`, then the variable would only be visible inside `if`:

```
if (true) {  
  let test = true; // use "let"  
}
```

```
alert(test); // ReferenceError: test is not defined
```

```
=====
```

```
=====
```

The same thing for loops: `var` cannot be block- or loop-local:

```
for (var i = 0; i < 10; i++) {
```

```
  var one = 1;
```

```
  // ...
```

```
}
```

```
alert(i); // 10, "i" is visible after loop, it's a global  
variable
```

```
alert(one); // 1, "one" is visible after loop, it's a global  
variable
```

If a code block is inside a function, then `var` becomes a function-level variable:

```
function sayHi() {
```

```
  if (true) {
```

```
    var phrase = "Hello";
```

```
  }
```

```
  alert(phrase); // works
```

```
}
```

```
sayHi();
```

```
alert(phrase); // ReferenceError: phrase is not defined
```

As we can see, `var` pierces through `if`, `for` or other code blocks. That's because a long time ago in JavaScript, blocks had no Lexical Environments, and `var` is a remnant of that.

```
=====
```



## “var” tolerates redeclarations

If we declare the same variable with `let` twice in the same scope, that’s an error:

```
let user;
```

```
let user; // SyntaxError: 'user' has already been declared
```

With `var`, we can redeclare a variable any number of times. If we use `var` with an already-declared variable, it’s just ignored:

```
var user = "Pete";
```

```
var user = "John"; // this "var" does nothing (already declared)
```

```
// ...it doesn't trigger an error
```

```
alert(user); // John
```

## “var” variables can be declared below their use

`var` declarations are processed when the function starts (or script starts for globals).

In other words, `var` variables are defined from the beginning of the function, no matter where the definition is (assuming that the definition is not in the nested function).

So this code:

```
function sayHi() {
```

```
    phrase = "Hello";
```

```
    alert(phrase);
```

```
    var phrase;
```

```
}
```

```
sayHi();
```

**...Is technically the same as this (moved var phrase above):**

```
function sayHi() {
```

```
    var phrase;
```

```
    phrase = "Hello";
```

```
alert (phrase) ;
```

```
}
```

```
sayHi () ;
```

...Or even as this (remember, code blocks are ignored):

```
function sayHi () {
```

```
    phrase = "Hello"; // (*)
```

```
    if (false) {
```

```
        var phrase;
```

```
    }
```

```
    alert (phrase) ;
```

```
}
```

```
sayHi () ;
```

People also call such behavior “hoisting” (raising), because all `var` are “hoisted” (raised) to the top of the function.

So in the example above, `if (false)` branch never executes, but that doesn’t matter. The `var` inside it is processed in the beginning of the function, so at the moment of (\*) the variable exists.

Declarations are hoisted, but assignments are not.

That’s best demonstrated with an example:

```
function sayHi() {
```

```
    alert(phrase);
```

```
    var phrase = "Hello";
```

```
}
```

```
sayHi();
```

The line `var phrase = "Hello"` has two actions in it:

1. Variable declaration `var`
2. Variable assignment `=`.

The declaration is processed at the start of function execution (“hoisted”), but the assignment always works at the place where it appears. So the code works essentially like this:

```
function sayHi() {
```

```
    var phrase; // declaration works at the start...
```

```
    alert(phrase); // undefined
```

```
    phrase = "Hello"; // ...assignment - when the execution  
    reaches it.
```

```
}
```

```
sayHi();
```

Because all `var` declarations are processed at the function start, we can reference them at any place. But variables are undefined until the assignments.

In both examples above, `alert` runs without an error, because the variable `phrase` exists. But its value is not yet assigned, so it shows `undefined`.

## Summary

There are two main differences of `var` compared to `let/const`:

1. `var` variables have no block scope, their visibility is scoped to current function, or global, if declared outside function.
2. `var` declarations are processed at function start (script start for globals).

```
=====
```

# Data types

A value in JavaScript is always of a certain type. For example, a string or a number.

There are eight basic data types in JavaScript. Here, we'll cover them in general and in the next chapters we'll talk about each of them in detail.

We can put any type in a variable. For example, a variable can at one moment be a string and then store a number:

```
// no error
```

```
let message = "hello";
```

```
message = 123456;
```

There are 8 basic data types in JavaScript.

- Seven primitive data types:
  - **number** for numbers of any kind: integer or floating-point, integers are limited by  $\pm(2^{53}-1)$ .
  - **bigint** for integer numbers of arbitrary length.
  - **string** for strings. A string may have zero or more characters, there's no separate single-character type.
  - **boolean** for `true/false`.
  - **null** for unknown values – a standalone type that has a single value `null`.
  - **undefined** for unassigned values – a standalone type that has a single value `undefined`.
  - **symbol** for unique identifiers.
- And one non-primitive data type:
  - **object** for more complex data structures.

The `typeof` operator allows us to see which type is stored in a variable.

- Usually used as `typeof x`, but `typeof(x)` is also possible.
- Returns a string with the name of the type, like `"string"`.
- For `null` returns `"object"` – this is an error in the language, it's not actually an object.

=====

=====

# Interaction: alert, prompt, confirm

We covered 3 browser-specific functions to interact with visitors:

## **alert**

shows a message.

## **prompt**

shows a message asking the user to input text. It returns the text or, if Cancel button or Esc is clicked, `null`.

## **confirm**

shows a message and waits for the user to press "OK" or "Cancel". It returns `true` for OK and `false` for Cancel/Esc.

All these methods are modal: they pause script execution and don't allow the visitor to interact with the rest of the page until the window has been dismissed.

There are two limitations shared by all the methods above:

1. The exact location of the modal window is determined by the browser. Usually, it's in the center.
2. The exact look of the window also depends on the browser. We can't modify it.

That is the price for simplicity. There are other ways to show nicer windows and richer interaction with the visitor, but if "bells and whistles" do not matter much, these methods work just fine.

=====



# Type Conversions

Most of the time, operators and functions automatically convert the values given to them to the right type.

For example, `alert` automatically converts any value to a string to show it. Mathematical operations convert values to numbers.

There are also cases when we need to explicitly convert a value to the expected type.

The three most widely used type conversions are to string, to number, and to boolean.

**String Conversion** – Occurs when we output something. Can be performed with `String(value)`. The conversion to string is usually obvious for primitive values.

**Numeric Conversion** – Occurs in math operations. Can be performed with `Number(value)`.

The conversion follows the rules:

Value	Becomes...
<code>undefined</code>	<code>NaN</code>
<code>null</code>	<code>0</code>
<code>true</code> / <code>false</code>	<code>1</code> / <code>0</code>
<code>string</code>	The string is read “as is”, whitespaces (includes spaces, tabs <code>\t</code> , newlines <code>\n</code> etc.) from both sides are ignored. An empty string becomes <code>0</code> . An error gives <code>NaN</code> .

**Boolean Conversion** – Occurs in logical operations. Can be performed with `Boolean(value)`.

Follows the rules:

Value	Becomes...
0, null, undefined, NaN, ""	false
any other value	true

Most of these rules are easy to understand and memorize. The notable exceptions where people usually make mistakes are:

`undefined` is NaN as a number, not 0.  
"0" and space-only strings like " " are true as a boolean.

Objects aren't covered here. We'll return to them later in the chapter [Object to primitive conversion](#) that is devoted exclusively to objects after we learn more basic things about JavaScript.

=====  
=====

# Basic operators, maths

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

## 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  $b$ .

## Exponentiation \*\*

The exponentiation operator  $a ** b$  raises  $a$  to the power of  $b$ .

In school maths, we write that as  $a^b$ .

## 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
```

Note that if any of the operands is a string, then the other one is converted to a string too.

For example:

```
alert( '1' + 2 ); // "12"
```

```
alert( 2 + '1' ); // "21"
```

See, it doesn't matter whether the first operand is a string or the second one.

Here's a more complex example:

```
alert(2 + 2 + '1' ); // "41" and not "221"
```

Here, operators work one after another. The first + sums two numbers, so it returns 4, then the next + adds the string 1 to it, so it's like `4 + '1' = '41'`.

```
alert('1' + 2 + 2); // "122" and not "14"
```

Here, the first operand is a string, the compiler treats the other two operands as strings too. The 2 gets concatenated to '1', so it's like `'1' + 2 = "12"` and `"12" + 2 = "122"`.

The binary + is the only operator that supports strings in such a way. Other arithmetic operators work only with numbers and always convert their operands to numbers.

Here's the demo for subtraction and division:

```
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.

For example:

```
// No effect on numbers
```

```
let x = 1;
```

```
alert( +x ); // 1
```

```
let y = -2;
```

```
alert( +y ); // -2
```

```
// Converts non-numbers
```

```
alert( +true ); // 1
```

```
alert( +"" ); // 0
```

It actually does the same thing as Number(...), but is shorter.

# Operator precedence

Precedence	Name	Sign
...	...	...
14	unary plus	+
14	unary negation	-
13	exponentiation	**
12	multiplication	*
12	division	/
11	addition	+
11	subtraction	-
...	...	...
2	assignment	=
...	...	...

## 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.

That's why, when we assign a variable, like `x = 2 * 2 + 1`, the calculations are done first and then the = is evaluated, storing the result in `x`.

```
let x = 2 * 2 + 1;
```

```
alert( x ); // 5
```

## Assignment = returns a value

The fact of = being an operator, not a “magical” language construct has an interesting implication.

All operators in JavaScript return a value. That's obvious for + and -, but also true for =.

The call `x = value` writes the `value` into `x` *and then returns it*.

Here's a demo that uses an assignment as part of a more complex expression:

```
let a = 1;  
let b = 2;
```

```
let c = 3 - (a = b + 1);
```

```
alert( a ); // 3
```

```
alert( c ); // 0
```

In the example above, the result of expression `(a = b + 1)` is the value which was assigned to `a` (that is 3). It is then used for further evaluations.

Funny code, isn't it? We should understand how it works, because sometimes we see it in JavaScript libraries.

Although, please don't write the code like that. Such tricks definitely don't make code clearer or readable.

## Chaining assignments

Another interesting feature is the ability to chain assignments:

```
let a, b, c;
```

```
a = b = c = 2 + 2;
```

```
alert( a ); // 4
```

```
alert( b ); // 4
```

```
alert( c ); // 4
```

Chained assignments evaluate from right to left. First, the rightmost expression `2 + 2` is evaluated and then assigned to the variables on the left: `c`, `b` and `a`. At the end, all the variables share a single value.

Once again, for the purposes of readability it's better to split such code into few lines:

```
c = 2 + 2;
```

```
b = c;
```

```
a = c;
```

That's easier to read, especially when eye-scanning the code fast.

## Modify-in-place

We often need to apply an operator to a variable and store the new result in that same variable.

For example:

```
let n = 2;
```

```
n = n + 5;
```

```
n = n * 2;
```

This notation can be shortened using the operators `+=` and `*=`:

```
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.

So, there are special operators for it:

**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
```

## Bitwise operators

Bitwise operators treat arguments as 32-bit integer numbers and work on the level of their binary representation.

These operators are not JavaScript-specific. They are supported in most programming languages.

The list of operators:

- AND ( & )
- OR ( | )
- XOR ( ^ )
- NOT ( ~ )
- LEFT SHIFT ( << )
- RIGHT SHIFT ( >> )
- ZERO-FILL RIGHT SHIFT ( >>> )

These operators are used very rarely, when we need to fiddle with numbers on the very lowest (bitwise) level. We won't need these operators any time soon, as web development has little use of them, but in some special areas, such as cryptography, they are useful. You can read the [Bitwise Operators](#) chapter on MDN when a need arises.

### Comparisons

Equality check `==` for values of different types converts them to a number (except `null` and `undefined` that equal each other and nothing else), so these are equal:

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

```
alert( 0 == '' ); // true
```

Other comparisons convert to a number as well.

The strict equality operator `===` doesn't do the conversion: different types always mean different values for it.

Values `null` and `undefined` are special: they equal `==` each other and don't equal anything else.

Greater/less comparisons compare strings character-by-character, other types are converted to a number.

=====

# Conditional branching: if, '?'

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.

## The “if” statement

```
let year = prompt('In which year was the ECMAScript-2015  
specification published?', '');
```

```
if (year < 2015) {
```

```
    alert( 'Too early...' );
```

```
} else if (year > 2015) {
```

```
    alert( 'Too late' );
```

```
} else {
```

```
    alert( 'Exactly!' );
```

```
}
```

## Conditional operator '?'

```
let result = condition ? value1 : value2;
```

```
=====
```

# The "switch" statement

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

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

## The syntax

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

It looks like this:

```
switch(x) {  
    case 'value1': // if (x === 'value1')  
        ...  
        [break]  
  
    case 'value2': // if (x === 'value2')  
        ...  
        [break]  
  
    default:  
        ...  
        [break]  
}
```

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

## Grouping of “case”

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

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

```
let a = 3;
```

```
switch (a) {
```

```
  case 4:
```

```
    alert('Right!');
```

```
    break;
```

```
  case 3: // (*) grouped two cases
```

```
  case 5:
```

```
    alert('Wrong!');
```

```
    alert("Why don't you take a math class?");
```

```
    break;
```

```
  default:
```

```
    alert('The result is strange. Really.');
```

```
}
```

Now both 3 and 5 show the same message.

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

## Type matters

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

For example, let’s consider the code:

```
let arg = prompt("Enter a value?");
```

```
switch (arg) {
```

```
  case '0':
```

```
  case '1':
```

```
    alert( 'One or zero' );
```

```
    break;
```

```
  case '2':
```

```
    alert( 'Two' );
```



```
break;
```

```
case 3:
```

```
alert( 'Never executes!' );
```

```
break;
```

```
default:
```

```
alert( 'An unknown value' );
```

```
}
```

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

=====

# Loops: while and for

## The “while” loop

The `while` loop has the following syntax:

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

## The “do...while” loop

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

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

## The “for” loop

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

It looks like this:

```
for (begin; condition; step) {
```

```
// ... loop body ...
```

```
}
```

## 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.

## Continue to the next iteration

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

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

```
=====
```

# Functions

We covered three ways to create a function in JavaScript:

1. Function Declaration: the function in the main code flow

```
function sum(a, b) {
```

```
  let result = a + b;
```

```
  return result;
```

```
}
```

2. Function Expression: the function in the context of an expression

```
let sum = function(a, b) {
```

```
  let result = a + b;
```

```
  return result;
```

```
};
```

3. Arrow functions:

`// expression on the right side`

```
let sum = (a, b) => a + b;
```

`// or multi-line syntax with { ... }, need return here:`

```
let sum = (a, b) => {
```

```
  // ...
```

```
  return a + b;
```

```
}
```

`// without arguments`

```
let sayHi = () => alert("Hello");
```

`// with a single argument`

```
let double = n => n * 2;
```

- Functions may have local variables: those declared inside its body or its parameter list. Such variables are only visible inside the function.
- Parameters can have default values: `function sum(a = 1, b = 2) {...}`.
- Functions always return something. If there's no `return` statement, then the result is `undefined`.

Small nuances in functions:

- **Outer Variables:** A function can access an outer variable as well. It can modify it as well. The outer variable is only used if there's no local one.

If a same-named variable is declared inside the function then it *shadows* the outer one. For instance, in the code below the function uses the local `userName`. The outer one is ignored:

```
let userName = 'John';
```

```
function showMessage() {
```

```
    let userName = "Bob"; // declare a local variable
```

```
    let message = 'Hello, ' + userName; // Bob
```

```
    alert(message);
```

```
}
```

```
// the function will create and use its own userName
```

```
showMessage();
```

```
alert( userName ); // John, unchanged, the function did not  
access the outer variable
```

- Initialized parameters supported (params with default values)
- **Never add a newline between `return` and the value**  
That doesn't work, because JavaScript assumes a semicolon after `return`.  
That'll work the same as:

```
    return;
```

```
    (some + long + expression + or + whatever * f(a) +  
f(b))
```

## Callback functions

We'll write a function `ask(question, yes, no)` with three parameters:

`question`

Text of the question

`yes`

Function to run if the answer is "Yes"

`no`

Function to run if the answer is "No"

The function should ask the `question` and, depending on the user's answer, call `yes()` or `no()`:

```
function ask(question, yes, no) {  
  if (confirm(question)) yes()  
  else no();  
}  
  
function showOk() {  
  alert( "You agreed." );  
}  
  
function showCancel() {  
  alert( "You canceled the execution." );  
}  
  
// usage: functions showOk, showCancel are passed as arguments  
// to ask  
  
ask("Do you agree?", showOk, showCancel);
```

```
=====
```

# Objects

Apart from the 7 primitive data types, Javascript has an Object data type, which is a non-primitive 8th data type.

Objects are used to store keyed collections of various data and more complex entities.

An empty object (“empty cabinet”) can be created using one of two syntaxes:

```
let user = new Object(); // "object constructor" syntax
```

```
let user = {}; // "object literal" syntax
```

Usually, the figure brackets `{ ... }` are used. That declaration is called an *object literal*.

## Literals and properties

We can immediately put some properties into `{ ... }` as “key: value” pairs:

```
let user = { // an object
```

```
  name: "John", // by key "name" store value "John"
```

```
  age: 30 // by key "age" store value 30
```

```
};
```



A property has a key (also known as “name” or “identifier”) before the colon “:” and a value to the right of it.

In the `user` object, there are two properties:

1. The first property has the name “name” and the value “John”.
2. The second one has the name “age” and the value 30.

The resulting `user` object can be imagined as a cabinet with two signed files labeled “name” and “age”.

We can add, remove and read files from it at any time.

Property values are accessible using the dot notation:

```
// get property values of the object:
```

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

```
alert( user.age ); // 30
```

The value can be of any type. Let’s add a boolean one:

```
user.isAdmin = true;
```

To remove a property, we can use the `delete` operator:

```
delete user.age;
```

We can also use multiword property names, but then they must be quoted:

```
let user = {
```

```
name: "John",
```

```
age: 30,
```

```
"likes birds": true // multiword property name must be quoted
```

```
};
```

## Square brackets

For multiword properties, the dot access doesn't work:

```
// this would give a syntax error
```

```
user.likes birds = true
```

There's an alternative "square bracket notation" that works with any string:

```
let user = {};
```

```
// set
```

```
user["likes birds"] = true;
```

```
// get
```

```
alert(user["likes birds"]); // true
```

```
// delete
```

```
delete user["likes birds"];
```

## Computed properties

We can use square brackets in an object literal, when creating an object. That's called *computed properties*.

For instance:

```
let fruit = prompt("Which fruit to buy?", "apple");
```

```
let bag = {
```

```
  [fruit]: 5, // the name of the property is taken from the  
  variable fruit
```

```
};
```

```
alert( bag.apple ); // 5 if fruit="apple"
```

The meaning of a computed property is simple: `[fruit]` means that the property name should be taken from `fruit`.

So, if a visitor enters "apple", `bag` will become `{apple: 5}`.

Essentially, that works the same as:

```
let fruit = prompt("Which fruit to buy?", "apple");
```

```
let bag = {};
```

```
// take property name from the fruit variable
```

```
bag[fruit] = 5;
```

...But looks nicer.

## Property value shorthand

In real code, we often use existing variables as values for property names.

For instance:

```
function makeUser(name, age) {  
  
  return {  
  
    name: name,  
  
    age: age,  
  
    // ...other properties  
  
  };  
}
```

```
let user = makeUser("John", 30);
```

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

In the example above, properties have the same names as variables. The use-case of making a property from a variable is so common, that there's a special *property value shorthand* to make it shorter.

Instead of `name:name` we can just write `name`, like this:

```
function makeUser(name, age) {  
  return {  
    name, // same as name: name  
    age,  // same as age: age  
    // ...  
  };  
}
```

We can use both normal properties and shorthands in the same object:

```
let user = {  
  name, // same as name:name  
  age: 30  
};
```

## Property existence test, "in" operator

A notable feature of objects in JavaScript, compared to many other languages, is that it's possible to access any property. There will be no error if the property doesn't exist!

Reading a non-existing property just returns `undefined`. So we can easily test whether the property exists:

```
let user = {};
```

```
alert( user.noSuchProperty === undefined ); // true means "no  
such property"
```

There's also a special operator `"in"` for that.

The syntax is:

```
"key" in object
```

For instance:

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

```
alert( "age" in user ); // true, user.age exists
```

```
alert( "blabla" in user ); // false, user.blabla doesn't exist
```

Please note that on the left side of `in` there must be a *property name*. That's usually a quoted string.

If we omit quotes, that means a variable should contain the actual name to be tested. For instance:

```
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!
```

## 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.

The syntax:

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

For instance, let's output all properties of `user`:

```
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  
  
}
```

```
=====
```



# 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, booleans, etc – are always copied “as a whole value”.

That’s easy to understand if we look a bit under the hood of what happens when we copy a value.

Let’s start with a primitive, such as a string.

Here we put a copy of message into phrase:

```
let message = "Hello!";
```

```
let phrase = message;
```

As a result we have two independent variables, each one storing the string "Hello!".

Quite an obvious result, right?

Objects are not like that.

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

Let's look at an example of such a variable:

```
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.

Now here's why it's important.

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

For instance:

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

Now we have two variables, each storing a reference to the same object:

We can use either variable to access the object and modify its contents:

```
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
```

It's as if we had a cabinet with two keys and used one of them (`admin`) to get into it and make changes. Then, if we later use another key (`user`), we are still opening the same cabinet and can access the changed contents.

## Comparison by reference

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

For instance, here `a` and `b` reference the same object, thus they are equal:

```
let a = {};
```

```
let b = a; // copy the reference
```

```
alert( a == b ); // true, both variables reference the same object
```

```
alert( a === b ); // true
```

And here two independent objects are not equal, even though they look alike (both are empty):

```
let a = {};
```

```
let b = {}; // two independent objects
```

```
alert( a == b ); // false
```

## Cloning and merging, Object.assign

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

But what if we need to duplicate an object?

To make a “real copy” (a clone) we can use `Object.assign` for the so-called “shallow copy” (nested objects are copied by reference) or a “deep cloning” function `structuredClone` or use a custom cloning implementation, such as `_.cloneDeep(obj)`.

```
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.

For example, we have `user` object, let's add a couple of permissions to it:

```
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
```

Here it copies all properties of `user` into the empty object and returns it.

There are also other methods of cloning an object, e.g. using the [spread syntax](#) `clone = { ...user }`, covered later in the tutorial.

## Nested cloning

### **structuredClone**

The call `structuredClone(object)` clones the `object` with all nested properties.

```
=====
=====
```

# Object methods, "this"

In javascript, objects can have functions inside them, like in classes in other languages.

## Method examples

For a start, let's teach the `user` to say hello:

```
let user = {
```

```
  name: "John",
```

```
  age: 30
```

```
};
```

```
user.sayHi = function() {
```

```
  alert("Hello!");
```

```
};
```

```
user.sayHi(); // Hello!
```

## Method shorthand

There exists a shorter syntax for methods in an object literal:

```
// these objects do the same
```

```
user = {  
  sayHi: function() {  
    alert("Hello");  
  }  
};
```

```
// method shorthand looks better, right?
```

```
user = {  
  sayHi() { // same as "sayHi: function(){...}"  
    alert("Hello");  
  }  
};
```



## “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`.

**To access the object, a method can use the `this` keyword.**

The value of `this` is the object “before dot”, the one used to call the method.

For instance:

```
let user = {
```

```
  name: "John",
```

```
  age: 30,
```

```
  sayHi() {
```

```
    // "this" is the "current object"
```

```
    alert(this.name);
```

```
  }
```

```
};
```

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

## 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.

```
=====
```

```
=====
```

# Optional chaining '?.'

## The “non-existing property” problem

In many practical cases we'd prefer to get `undefined` instead of an error here (meaning “no street”).

...and another example. In Web development, we can get an object that corresponds to a web page element using a special method call, such as `document.querySelector('.elem')`, and it returns `null` when there's no such element.

```
// document.querySelector('.elem') is null if there's no element
```

```
let html = document.querySelector('.elem').innerHTML; // error  
if it's null
```

Once again, if the element doesn't exist, we'll get an error accessing `.innerHTML` property of `null`. And in some cases, when the absence of the element is normal, we'd like to avoid the error and just accept `html = null` as the result.

## Optional chaining

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

In other words, `value?.prop`:

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

Here's the safe way to access `user.address.street` using `?.`:

```
let user = {}; // user has no address
```

```
alert( user?.address?.street ); // undefined (no error)
```

=====

=====

## Array Iteration Methods:

[The Array filter\(\) Method](#)

[The Array forEach\(\) Method](#)

[The Array map\(\) Method](#)

↳ forEach

```
const initialArray = [1, 2, 3];  
for (let i = 0; i < initialArray.length; i++) {  
  console.log(initialArray[i]);  
}
```

```
function logThing(str) {  
  console.log(str);  
}
```

```
initialArray.forEach(logThing)
```

↳ str.length↳ str.indexOf(target)↳ str.lastIndexOf(target)↳ str.slice(start, end)↳ str.substr(start, <sup>length</sup>end)↳ str.replace(~~str~~ "world", "javascript")↳ const value = "hi my name is bryson";const words = value.split(" ");words → ["hi", "my", "name", "is", "bryson"]↳ str.trim()↳ str.toUpperCase()↳ str.toLowerCase()



↳ `parseInt("42")`

$\Rightarrow 42$

`parseInt("42px")`

$\Rightarrow 42$

`parseInt("3.14")`

$\Rightarrow 3$

↳ `parseFloat("3.41xyz")`

↳ `initialArray = [1, 2, 3]`

`initialArray.push(4)`

`[1, 2, 3, 4]`

↳ `initialArray.pop()`

↳ `initialArray.shift()`

↳ `initialArray.unshift(0)`

↳ `initialArray.concat(secondArray)`



## ↳ Class

```
class Animal {
  constructor (name, legCount, speaks) {
    this.name = name;
    this.legCount = legCount;
    this.speaks = speaks;
  }
  speak () { console.log("hi there" + this.speaks); }
}

let dog = new Animal("dog", 4, "bhow bhow");
let cat = new Animal("cat", 4, "meow");

cat.speak();
```

Static props

→ belong to class & not object

↳ ~~const~~ `const currentDate = new Date();`

- `currentDate.getDate()`
- `currentDate.getMonth()`
- `currentDate.getFullYear()`
- `currentDate.getHours()`
- `currentDate.getMinutes()`
- `currentDate.getSeconds()`
- `currentDate.setFullYear(2022)`
- `currentDate.setMonth(3);`
- `currentDate.getTime()`

(1970)



## JSON

- ★ JSON.parse
- ★ JSON.stringify

```
const users = { "name": "aryan", "age": 24,  
                "gender": "male" }
```

```
const user = JSON.parse(users);  
console.log(user["gender"]);
```

## ↳ Math

- Math.round(value)
- Math.ceil(value)
- Math.floor(value)
- Math.random(value)
- Math.max(5, 10, 15)
- Math.min(5, 10, 15)
- Math.pow(value, power)
- Math.sqrt(value)

## ↳ Objects

- Object.keys(obj)
- Object.values(obj)
- Object.entries(obj)
- obj.hasOwnProperty("property")
- Object.assign(x, y, obj, {newProperty: "newValue"});



# Error handling, "try...catch"

## The "try...catch" syntax

The `try...catch` construct has two main blocks: `try`, and then `catch`:

```
try {
```

```
    // code...
```

```
} catch (err) {
```

```
    // error handling
```

```
}
```

## try...catch...finally

Wait, that's not all.

The `try...catch` construct may have one more code clause: `finally`.

If it exists, it runs in all cases:

- after `try`, if there were no errors,
- after `catch`, if there were errors.

The extended syntax looks like this:

```
try {
```

```
    ... try to execute the code ...
```

```
} catch (err) {
```

```
    ... handle errors ...
```

```
} finally {
```

```
    ... execute always ...
```

```
}
```

=====

# Rest parameters and spread syntax

Many JavaScript built-in functions support an arbitrary number of arguments.

For instance:

- `Math.max(arg1, arg2, ..., argN)` – returns the greatest of the arguments.
- `Object.assign(dest, src1, ..., srcN)` – copies properties from `src1..N` into `dest`.
- ...and so on.

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

Like here:

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

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

to gather all arguments into array `args`:

```
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
```

## Spread syntax

We've just seen how to get an array from the list of parameters.

But sometimes we need to do exactly the reverse.

```
let arr = [3, 5, 1];
```

```
alert( Math.max(...arr) ); // 5 (spread turns array into a list  
of arguments)
```

# Variable scope, closure

JavaScript is a very function-oriented language. It gives us a lot of freedom. A function can be created at any moment, passed as an argument to another function, and then called from a totally different place of code later.

We already know that a function can access variables outside of it (“outer” variables).

But what happens if outer variables change since a function is created? Will the function get newer values or the old ones?

Following things are only valid for **let/const**, **var** is a different and old beast

## Code blocks

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

For example:

```
{  
  // 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
```

## Nested functions

A function is called “nested” when it is created inside another function.

It is easily possible to do this with JavaScript.

We can use it to organize our code, like this:

```
function sayHiBye(firstName, lastName) {
```

```
    // helper nested function to use below
```

```
    function getFullName() {
```

```
        return firstName + " " + lastName;
```

```
    }
```

```
    alert( "Hello, " + getFullName() );
```

```
    alert( "Bye, " + getFullName() );
```

```
}
```

## 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

[https://www.w3schools.com/js/js\\_function\\_closures.asp](https://www.w3schools.com/js/js_function_closures.asp)

# Scheduling: setTimeout and setInterval

## setTimeout

The syntax:

```
let timerId = setTimeout(func|code, [delay], [arg1], [arg2], ...)
```

Parameters:

**func | code**

Function or a string of code to execute. Usually, that’s a function. For historical reasons, a string of code can be passed, but that’s not recommended.

**delay**

The delay before run, in milliseconds (1000 ms = 1 second), by default 0.

`arg1, arg2...`

Arguments for the function

For instance, this code calls `sayHi()` after one second:

```
function sayHi() {
```

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

```
}
```

```
setTimeout(sayHi, 1000);
```

With arguments:

```
function sayHi(phrase, who) {
```

```
    alert( phrase + ', ' + who );
```

```
}
```

```
setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```



Use Case: Debouncing

## setInterval

The `setInterval` method has the same syntax as `setTimeout`:

```
let timerId = setInterval(func|code, [delay], [arg1], [arg2], ...)
```

All arguments have the same meaning. But unlike `setTimeout` it runs the function not only once, but regularly after the given interval of time.

## Function binding

When passing object methods as callbacks, for instance to `setTimeout`, there's a known problem: "losing `this`".

### Losing "this"

Once a method is passed somewhere separately from the object – `this` is lost.

Here's how it may happen with `setTimeout`:

```
let user = {
```

```
  firstName: "John",
```

```
  sayHi () {
```

```
    alert(`Hello, ${this.firstName}!`);
```

```
  }
```

```
};
```

```
setTimeout(user.sayHi, 1000); // Hello, undefined!
```

As we can see, the output shows not “John” as `this.firstName`, but `undefined`!

That’s because `setTimeout` got the function `user.sayHi`, separately from the object.

The simplest solution is to use a wrapping function:

```
let user = {
```

```
  firstName: "John",
```

```
  sayHi() {
```

```
    alert(`Hello, ${this.firstName}!`);
```

```
  }
```

```
};
```

```
setTimeout(function() {
```

```
  user.sayHi(); // Hello, John!
```

```
}, 1000);
```

Now it works, because it receives `user` from the outer lexical environment, and then calls the method normally.

The same, but shorter:

```
setTimeout(() => user.sayHi(), 1000); // Hello, John!
```

Looks fine, but a slight vulnerability appears in our code structure.

Functions provide a built-in method `bind` that allows to fix `this`.

The basic syntax is:

```
// more complex syntax will come a little later
```

```
let boundFunc = func.bind(context);
```

The result of `func.bind(context)` is a special function-like “exotic object”, that is callable as function and transparently passes the call to `func` setting `this=context`.

In other words, calling `boundFunc` is like `func` with fixed `this`.

Example:

```
let user = {  
  firstName: "John",  
  sayHi() {  
    alert(`Hello, ${this.firstName}!`);  
  }  
};
```

```
let sayHi = user.sayHi.bind(user); // (*)
```

```
// can run it without an object
```

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

```
setTimeout(sayHi, 1000); // Hello, John!
```

```
// even if the value of user changes within 1 second
```

```
// sayHi uses the pre-bound value which is reference to the old  
user object
```

```
user = {
```

```
  sayHi() { alert("Another user in setTimeout!"); }
```

```
};
```

```
=====
```

# Class basic syntax

## The “class” syntax

The basic syntax is:

```
class MyClass {  
    // class methods  
    constructor() { ... }  
    method1() { ... }  
    method2() { ... }  
    method3() { ... }  
    ...  
}
```

Then use `new MyClass()` to create a new object with all the listed methods.

The `constructor()` method is called automatically by `new`, so we can initialize the object there.

For example:

```
class User {  
  
    constructor(name) {  
        this.name = name;  
    }  
  
    sayHi() {  
        alert(this.name);  
    }  
}  
  
// Usage:  
let user = new User("John");
```

```
user.sayHi () ;
```

When `new User ("John")` is called:

1. A new object is created.
2. The `constructor` runs with the given argument and assigns it to `this.name`.

...Then we can call object methods, such as `user.sayHi ()`

There are many more topics related to class, such as:

- Getters/Setters
- Inheritance
- Static Properties and methods
- Private/Protected properties

But we shall learn those on the go, not required as of now.

# Introduction: callbacks

FUNCTIONS ARE FIRST CLASS CITIZENS IN JAVASCRIPT

Take a look at the function `loadScript(src)`, that loads a script with the given `src`:

```
function loadScript(src) {  
  
    // creates a <script> tag and append it to the page  
  
    // this causes the script with given src to start loading and  
    run when complete  
  
    let script = document.createElement('script');  
  
    script.src = src;  
  
    document.head.append(script);  
  
}
```

Let's add a `callback` function as a second argument to `loadScript` that should execute when the script loads:

```
function loadScript(src, callback) {
```

```
  let script = document.createElement('script');
```

```
  script.src = src;
```

```
  script.onload = () => callback(script);
```

```
  document.head.append(script);
```

```
}
```

That's the idea: the second argument is a function (usually anonymous) that runs when the action is completed.



## Callback in callback

How can we load two scripts sequentially: the first one, and then the second one after it?

The natural solution would be to put the second `loadScript` call inside the callback, like this:

```
loadScript('/my/script.js', function(script) {
```

```
    alert(`Cool, the ${script.src} is loaded, let's load one  
more`);
```

```
    loadScript('/my/script2.js', function(script) {
```

```
        alert(`Cool, the second script is loaded`);
```

```
    });
```

```
});
```

IF WE GO ON LIKE THIS - CALLBACK INSIDE CALLBACK, IT WILL BE A HUGE PROBLEM FOR US ONLY - POOR READABILITY AND MANAGEMENT OF CODE - THIS IS **THE CALLBACK HELL**

## Handling errors

In the above examples we didn't consider errors. What if the script loading fails? Our callback should be able to react on that.

Here's an improved version of `loadScript` that tracks loading errors:

```
function loadScript(src, callback) {
```

```
  let script = document.createElement('script');
```

```
  script.src = src;
```

```
  script.onload = () => callback(null, script);
```

```
  script.onerror = () => callback(new Error(`Script load error  
for ${src}`));
```

```
document.head.append(script);
```

```
}
```

It calls `callback(null, script)` for successful load and `callback(error)` otherwise.

The usage:

```
loadScript('/my/script.js', function(error, script) {
```

```
  if (error) {
```

```
    // handle error
```

```
  } else {
```

```
    // script loaded successfully
```

```
  }
```

```
});
```

Once again, the recipe that we used for `loadScript` is actually quite common. It's called the "error-first callback" style.

The convention is:

1. The first argument of the `callback` is reserved for an error if it occurs. Then `callback(err)` is called.
2. The second argument (and the next ones if needed) are for the successful result. Then `callback(null, result1, result2...)` is called.

So the single `callback` function is used both for reporting errors and passing back results.

Example:

```
fs.readFile('demofile1.html', function(err, data) {  
    console.log(data)  
});
```

```
=====
```

# Promise

Imagine that you're a top singer, and fans ask day and night for your upcoming song.

To get some relief, you promise to send it to them when it's published. You give your fans a list. They can fill in their email addresses, so that when the song becomes available, all subscribed parties instantly receive it. And even if something goes very wrong, say, a fire in the studio, so that you can't publish the song, they will still be notified.

Everyone is happy: you, because the people don't crowd you anymore, and fans, because they won't miss the song.

This is a real-life analogy for things we often have in programming:

1. A "producing code" that does something and takes time. For instance, some code that loads the data over a network. That's a "singer".
2. A "consuming code" that wants the result of the "producing code" once it's ready. Many functions may need that result. These are the "fans".
3. A *promise* is a special JavaScript object that links the "producing code" and the "consuming code" together. In terms of our analogy: this is the "subscription list". The "producing code" takes whatever time it needs to produce the promised result, and the "promise" makes that result available to all of the subscribed code when it's ready.

The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {  
  
    // executor (the producing code, "singer")  
  
});
```

The function passed to `new Promise` is called the *executor*. When `new Promise` is created, the executor runs automatically. It contains the producing code which should eventually produce the result. In terms of the analogy above: the executor is the “singer”.

Its arguments `resolve` and `reject` are callbacks provided by JavaScript itself. Our code is only inside the executor.

When the executor obtains the result, be it soon or late, doesn't matter, it should call one of these callbacks:

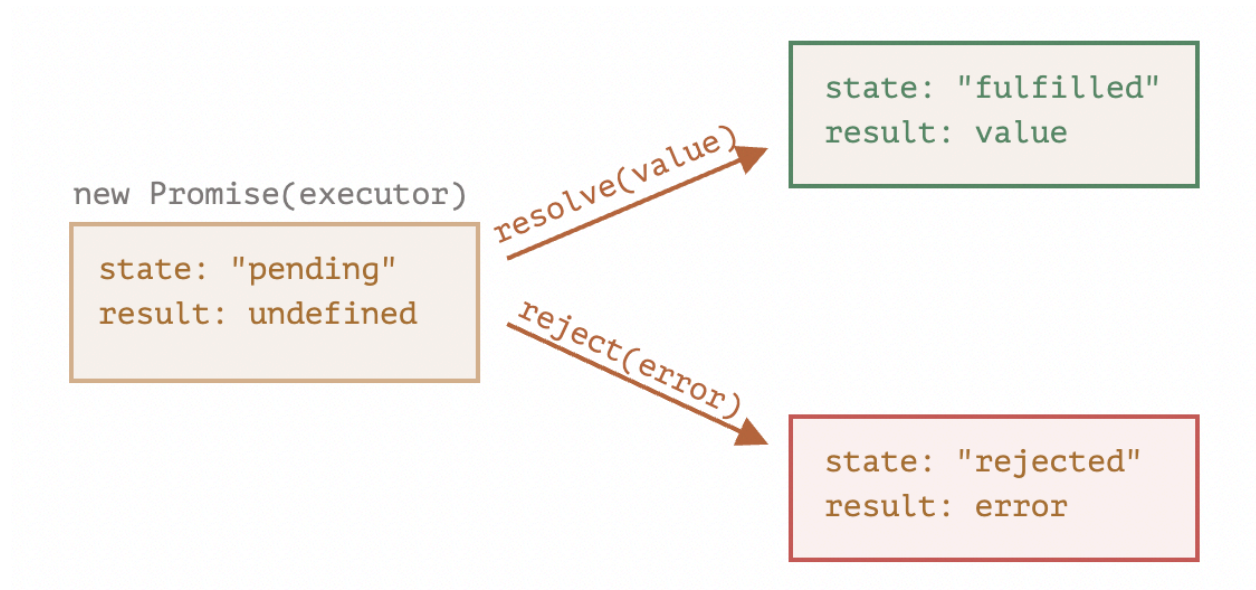
`resolve(value)` — if the job is finished successfully, with result `value`.  
`reject(error)` — if an error has occurred, `error` is the error object.

So to summarize: the executor runs automatically and attempts to perform a job. When it is finished with the attempt, it calls `resolve` if it was successful or `reject` if there was an error.

The `promise` object returned by the `new Promise` constructor has these internal properties:

`state` — initially `"pending"`, then changes to either `"fulfilled"` when `resolve` is called or `"rejected"` when `reject` is called.  
`result` — initially `undefined`, then changes to `value` when `resolve(value)` is called or `error` when `reject(error)` is called.

So the executor eventually moves `promise` to one of these states:



Syntax:

```
let promise = new Promise(function(resolve, reject) {  
    // Make an asynchronous call and either resolve or reject  
  
});
```

```
const myPromise = new Promise((resolve, reject) => {
```

```
    setTimeout(() => {
```

```
        resolve("foo");
```

```
    }, 300);
```

```
});
```

**THIS SYNTAX IS FOR CREATING PROMISES, BUT WE AS DEVELOPERS RARELY NEED TO WRITE THESE, WE SHOULD MAINLY FOCUS ON LEARNING HOW TO CONSUME THEM.**

```
myPromise.finally(() => {  
  
  loading = false;  
  
  console.log(`Promise Settled and loading is ${loading}`);  
  
}).then((result) => {  
  
  console.log({result});  
  
}).catch((error) => {  
  
  console.log(error)  
  
});
```

## Promise Object Properties

A JavaScript Promise object can be:

- Pending
- Fulfilled
- Rejected

**Promise concurrency**



The Promise class offers four static methods to facilitate async task [concurrency](#):

#### [Promise.all\(\)](#)

Fulfills when all of the promises fulfill; rejects when any of the promises rejects.

#### [Promise.allSettled\(\)](#)

Fulfills when all promises settle.

#### [Promise.any\(\)](#)

Fulfills when any of the promises fulfills; rejects when all of the promises reject.

#### [Promise.race\(\)](#)

Settles when any of the promises settles. In other words, fulfills when any of the promises fulfills; rejects when any of the promises rejects.

Example using : <https://pokeapi.co/api/v2/pokemon/1/>=

Example using: <https://api.github.com/users/aryansindhi18>

## Async/await

There's a special syntax to work with promises in a more comfortable fashion, called "async/await". It's surprisingly easy to understand and use.

## Async functions

Let's start with the `async` keyword. It can be placed before a function, like this:

```
async function f() {  
  
    return 1;  
  
}
```

The word "async" before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

For instance, this function returns a resolved promise with the result of `1`; let's test it:

```
async function f() {  
  
    return 1;  
  
}  
  
f().then(alert); // 1
```

...We could explicitly return a promise, which would be the same:

```
async function f() {  
  
    return Promise.resolve(1);  
  
}
```

```
f().then(alert); // 1
```

So, `async` ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There's another keyword, `await`, that works only inside `async` functions, and it's pretty cool.

## Await

The syntax:

```
// works only inside async functions
```

```
let value = await promise;
```

The keyword `await` makes JavaScript wait until that promise settles and returns its result.

Here's an example with a promise that resolves in 1 second:

```
async function f() {
```

```
  let promise = new Promise((resolve, reject) => {
```

```
    setTimeout(() => resolve("done!"), 1000)
```

```
  });
```

```
  let result = await promise; // wait until the promise resolves  
  (*)
```

```
  alert(result); // "done!"
```

```
}
```

```
f();
```

The function execution “pauses” at the line `(*)` and resumes when the promise settles, with `result` becoming its result. So the code above shows “done!” in one second.

Let’s emphasize: `await` literally suspends the function execution until the promise settles, and then resumes it with the promise result. That doesn’t cost any CPU resources, because the JavaScript engine can do other jobs in the meantime: execute other scripts, handle events, etc.

It’s just a more elegant syntax of getting the promise result than `promise.then`. And, it’s easier to read and write.

#### Repl.it links:

<https://replit.com/@aryansindhi18/SquareOutgoingDemand#index.js>

<https://replit.com/@aryansindhi18/RemarkableMixedBases>

<https://replit.com/@aryansindhi18/OtherBrownModel#index.js>

```
=====
=====
```

# JavaScript Prototypes and Inheritance – and Why They Say Everything in JS is an Object

## Intro

Have you ever wondered how strings, arrays or objects “know” the methods each of them have? How does a string know it can `.toUpperCase()` or an array know that it can `.sort()`? We never defined these methods manually, right?

The answer is that these methods come built-in within each type of data structure thanks to something called **prototype inheritance**.

In JavaScript, an object can inherit properties of another object. The object from where the properties are inherited is called the prototype. In short, objects can inherit properties from other objects — the prototypes.

You’re probably wondering: why the need for inheritance in the first place? Well, **inheritance solves the problem of data and logic duplication**. By inheriting, objects can share properties and methods without the need of manually setting those properties and methods on each object.

# How to Access a Prototype's Properties and Methods in JavaScript

When we try to access a property of an object, the property is not only searched in the object itself. It's also searched in the prototype of the object, in the prototype of the prototype, and so on – until a property is found that matches the name or the end of the **prototype chain** is reached.

If the property or method isn't found anywhere in the prototype chain, only then will JavaScript return `undefined`.

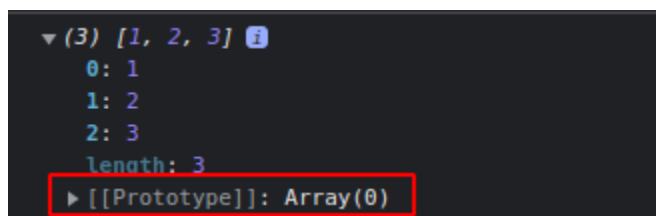
Every object in JavaScript has an internal property called `[[Prototype]]`.

If we create an array and log it to the console like this:

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

```
console.log(arr)
```

We will see this:



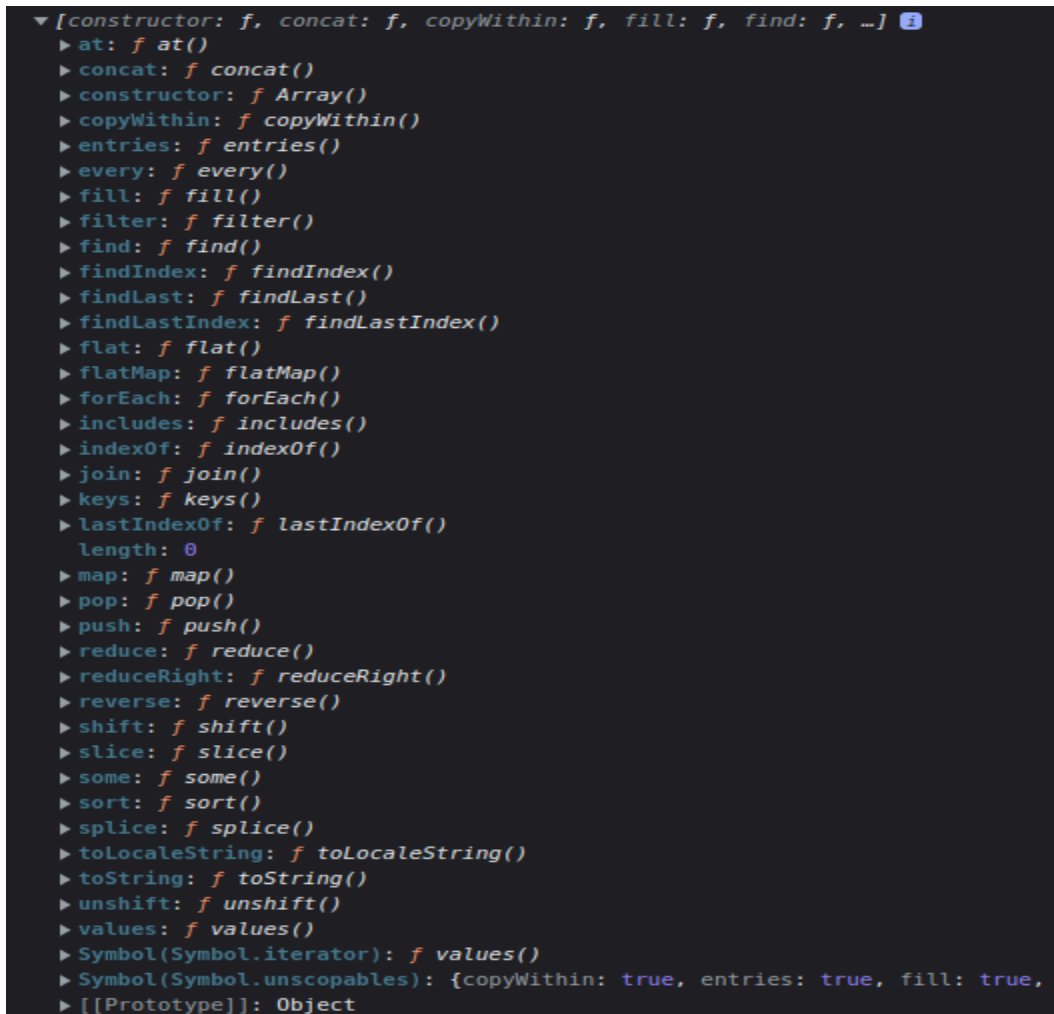
The double square brackets that enclose `[[Prototype]]` signify that it is an internal property, and cannot be accessed directly in code.

To find the `[[Prototype]]` of an object, we will use the `Object.getPrototypeOf()` method.

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

```
console.log(Object.getPrototypeOf(arr))
```

The output will consist of several built-in properties and methods:



```
▼ [constructor: f, concat: f, copyWithin: f, fill: f, find: f, ...] ⓘ
  ► at: f at()
  ► concat: f concat()
  ► constructor: f Array()
  ► copyWithin: f copyWithin()
  ► entries: f entries()
  ► every: f every()
  ► fill: f fill()
  ► filter: f filter()
  ► find: f find()
  ► findIndex: f findIndex()
  ► findLast: f findLast()
  ► findLastIndex: f findLastIndex()
  ► flat: f flat()
  ► flatMap: f flatMap()
  ► forEach: f forEach()
  ► includes: f includes()
  ► indexOf: f indexOf()
  ► join: f join()
  ► keys: f keys()
  ► lastIndexOf: f lastIndexOf()
  ► length: 0
  ► map: f map()
  ► pop: f pop()
  ► push: f push()
  ► reduce: f reduce()
  ► reduceRight: f reduceRight()
  ► reverse: f reverse()
  ► shift: f shift()
  ► slice: f slice()
  ► some: f some()
  ► sort: f sort()
  ► splice: f splice()
  ► toLocaleString: f toLocaleString()
  ► toString: f toString()
  ► unshift: f unshift()
  ► values: f values()
  ► Symbol(Symbol.iterator): f values()
  ► Symbol(Symbol.unscopables): {copyWithin: true, entries: true, fill: true,
  ► [[Prototype]]: Object
```

Keep in mind that prototypes can also be changed and modified through different methods.

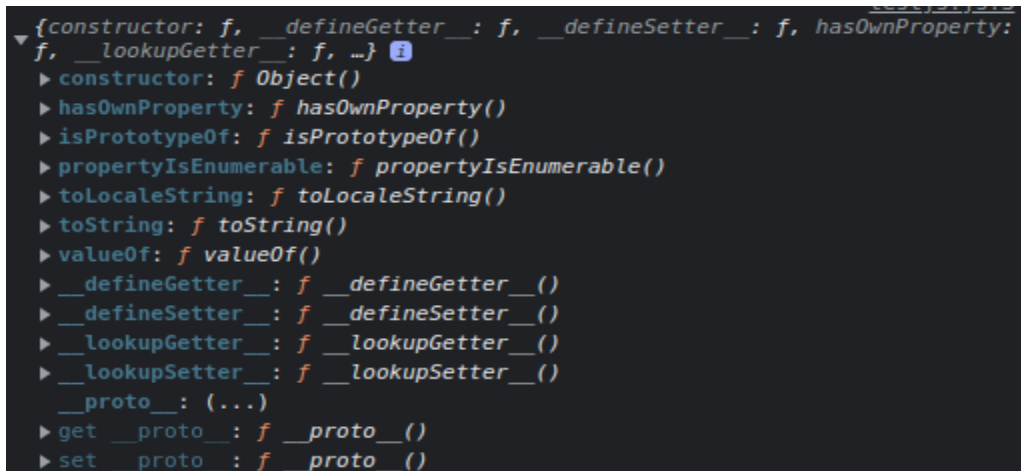
## The Prototype Chain

At the end of the prototype chain is `Object.prototype`. All objects inherit the properties and methods of `Object`. Any attempt to search beyond the end of the chain results in `null`.

If you look for the prototype of the prototype of an array, a function, or a string, you'll see it's an object. And that's because in JavaScript all objects are descendants or instances of `Object.prototype`, which is an object that sets properties and methods to all other JavaScript data types.

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

```
const arrProto = Object.getPrototypeOf(arr)
```



```
{constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, __lookupSetter__: f, __proto__: (...)}  
  ▶ constructor: f Object()  
  ▶ hasOwnProperty: f hasOwnProperty()  
  ▶ isPrototypeOf: f isPrototypeOf()  
  ▶ propertyIsEnumerable: f propertyIsEnumerable()  
  ▶ toLocaleString: f toLocaleString()  
  ▶ toString: f toString()  
  ▶ valueOf: f valueOf()  
  ▶ __defineGetter__: f __defineGetter__()  
  ▶ __defineSetter__: f __defineSetter__()  
  ▶ __lookupGetter__: f __lookupGetter__()  
  ▶ __lookupSetter__: f __lookupSetter__()  
  ▶ __proto__: (...)  
  ▶ get __proto__: f __proto__()  
  ▶ set __proto__: f __proto__()
```



Each type of prototype (for example array prototype) defines its own methods and properties, and in some cases overrides the `Object.prototype` methods and properties (that's why arrays have methods that objects don't).

But under the hood and going up the ladder of the prototype chain, **everything in JavaScript is built upon the** `Object.prototype`.

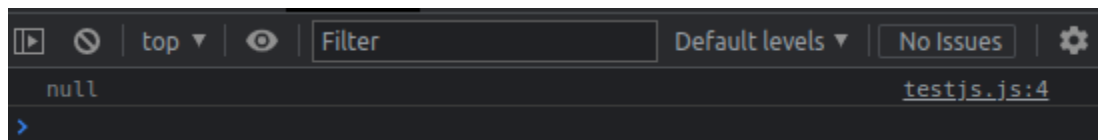
If we try to look into the prototype of `Object.prototype` we get `null`.

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

```
const arrProto = Object.getPrototypeOf(arr)
```

```
const objectProto = Object.getPrototypeOf(arrProto)
```

```
console.log(Object.getPrototypeOf(objectProto))
```



# A Prototype-Based Language

JavaScript is a **prototype-based language**, meaning object properties and methods can be shared through generalized objects that have the ability to be cloned and extended.

When it comes to inheritance, JavaScript has only one structure: objects.

Each object has a private property (referred to as its `[[Prototype]]`) that maintains a link to another object called its prototype. That prototype object has its own prototype, and so on until an object whose prototype is `null` is reached.

By definition, `null` has no prototype, and acts as the final link in this chain of prototypes.

This is known as prototypical inheritance and differs from class inheritance. Among popular object-oriented programming languages, JavaScript is relatively unique, as other prominent languages such as PHP, Python, and Java are class-based languages, which instead define classes as blueprints for objects.

At this point you may be thinking "But we CAN implement classes on JavaScript!". And yes, we can, but as syntactic sugar. 🤔🤔

## Javascript Classes

Classes are a way to set a blueprint to create objects with predefined properties and methods. By creating a class with specific properties and methods, you can later on instantiate objects from that class, that will inherit all the properties and methods that that class has.

In JavaScript, we can create classes in the following way:

```
class Alien {  
  
  constructor (name, phrase) {  
  
    this.name = name  
  
    this.phrase = phrase  
  
    this.species = "alien"  
  
  }  
  
  fly = () => console.log("Zzzzzziiiiiinnnnngggg!!")  
  
  sayPhrase = () => console.log(this.phrase)  
  
}
```

And then we can instantiate an object from that class like this:

```
const alien1 = new Alien("Ali", "I'm Ali the alien!")
```

```
console.log(alien1.name) // output: "Ali"
```

Classes are used as a way to make code more modular, organized, and understandable and are heavily used in OOP programming.

But keep in mind that JavaScript doesn't really support classes like other languages. The `class` keyword was introduced with ES6 as syntactic sugar that facilitates this way of organizing code.

To visualize this, see that the same thing we did by previously defining a `class`, we can do it by defining a function and editing the prototype in the following way:

```
function Alien(name, phrase) {
```

```
    this.name = name
```

```
    this.phrase = phrase
```

```
    this.species = "alien"
```

```
}
```

```
Alien.prototype.fly = () => console.log("Zzzzzzziiiiinnnnngggg!!")
```

```
Alien.prototype.sayPhrase = () => console.log(this.phrase)
```

```
const alien1 = new Alien("Ali", "I'm Ali the alien!")
```

```
console.log(alien1.name) // output "Ali"
```

```
console.log(alien1.phrase) // output "I'm Ali the alien!"
```

```
alien1.fly() // output "Zzzzzziiiiiinnnnngggg"
```

Any function can be invoked as a constructor with the keyword `new` and the `prototype` property of that function is used for the object to inherit methods from. In JavaScript, “class” is only used conceptually to describe the above practice – technically they’re just functions. 😊

Although this doesn't necessarily make a lot of difference (we can still perfectly implement OOP and use classes like in most other programming languages), it's important to remember that JavaScript is built with prototype inheritance at its core.

# How to Create Objects – Classes

So any video game needs characters, right? And all characters have certain **characteristics** (properties) like color, height, name, and so on and **abilities** (methods) like jumping, running, punching, and so on. Objects are the perfect data structure to use to store this kind of information. 🔥

Say we have 3 different character "species" available, and we want to create 6 different characters, 2 of each species.

A way of creating our characters could be to just manually create the objects using [object literals](#), in this way:

```
const alien1 = {  
  
  name: "Ali",  
  
  species: "alien",  
  
  phrase: () => console.log("I'm Ali the alien!"),  
  
  fly: () => console.log("Zzzzzziiiiiinnnnngggg!!")  
  
}  
  
const alien2 = {  
  
  name: "Lien",
```

```
    species: "alien",

    sayPhrase: () => console.log("Run for your lives!"),

    fly: () => console.log("Zzzzzziiiiiinnnnngggg!!")

}

const bug1 = {

    name: "Buggy",

    species: "bug",

    sayPhrase: () => console.log("Your debugger doesn't work with
me!"),

    hide: () => console.log("You can't catch me now!")

}

const bug2 = {

    name: "Erik",

    species: "bug",

    sayPhrase: () => console.log("I drink decaf!"),
```

```
      hide: () => console.log("You can't catch me now!")
    }

const Robot1 = {

  name: "Tito",

  species: "robot",

  sayPhrase: () => console.log("I can cook, swim and dance!"),

  transform: () => console.log("Optimus prime!")

}

const Robot2 = {

  name: "Terminator",

  species: "robot",

  sayPhrase: () => console.log("Hasta la vista, baby!"),

  transform: () => console.log("Optimus prime!")

}
```



See that all characters have the `name` and `species` properties and also the `sayPhrase` method. Moreover, each species has a method that belongs only to that species (for example, aliens have the `fly` method).

As you can see, some data is shared by all characters, some data is shared by each species, and some data is unique to each individual character.

This approach works. See that we can perfectly access properties and methods like this:

```
console.log(alien1.name) // output: "Ali"
```

```
console.log(bug2.species) // output: "bug"
```

```
Robot1.sayPhrase() // output: "I can cook, swim and dance!"
```

```
Robot2.transform() // output: "Optimus prime!"
```

The problem with this is that it doesn't scale well at all and it's error prone. Imagine that our game could have hundreds of characters. We would need to manually set the properties and methods for each of them!

To solve this problem we need a programmatic way of creating objects and setting different properties and methods given a set of conditions. And that's what **classes** are good for. 😊

Classes set a blueprint to create objects with predefined properties and methods. By creating a class, you can later on **instantiate** (create) objects from that class, that will inherit all the properties and methods that class has.

Refactoring our previous code, we can create a class for each of our character species, like this:

```
class Alien { // Name of the class

    // The constructor method will take a number of parameters and
    assign those parameters as properties to the created object.

    constructor (name, phrase) {

        this.name = name

        this.phrase = phrase

        this.species = "alien"

    }

    // These will be the object's methods.

    fly = () => console.log("Zzzzzziinnnnngggg!!")
```

```
    sayPhrase = () => console.log(this.phrase)

}

class Bug {

    constructor (name, phrase) {

        this.name = name

        this.phrase = phrase

        this.species = "bug"

    }

    hide = () => console.log("You can't catch me now!")

    sayPhrase = () => console.log(this.phrase)

}

class Robot {
```

```
    constructor (name, phrase) {  
  
        this.name = name  
  
        this.phrase = phrase  
  
        this.species = "robot"  
  
    }  
  
    transform = () => console.log("Optimus prime!")  
  
    sayPhrase = () => console.log(this.phrase)  
  
}
```

And then we can instantiate our characters from those classes like this:

```
const alien1 = new Alien("Ali", "I'm Ali the alien!")  
  
// We use the "new" keyword followed by the corresponding class name  
  
// and pass it the corresponding parameters according to what was declared  
in the class constructor function  
  
const alien2 = new Alien("Lien", "Run for your lives!")
```

```
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me!")
```

```
const bug2 = new Bug("Erik", "I drink decaf!")
```

```
const Robot1 = new Robot("Tito", "I can cook, swim and dance!")
```

```
const Robot2 = new Robot("Terminator", "Hasta la vista, baby!")
```

Then again, we can access each object properties and methods like this:

```
console.log(alien1.name) // output: "Ali"
```

```
console.log(bug2.species) // output: "bug"
```

```
Robot1.sayPhrase() // output: "I can cook, swim and dance!"
```

```
Robot2.transform() // output: "Optimus prime!"
```

What is nice about this approach and the use of classes in general is that we can use those "blueprints" to create new objects quicker and more securely than if we did it "manually".

Also, our code is better organized as we can clearly identify where each object properties and methods are defined (in the class). And this makes future changes or adaptations much easier to implement.

## Some things to keep in mind about classes:

Following [this definition](#), put in more formal terms,

*"a class in a program is a definition of a "type" of custom data structure that includes both data and behaviors that operate on that data. Classes define how such a data structure works, but classes are not themselves concrete values. To get a concrete value that you can use in the program, a class must be instantiated (with the "new" keyword) one or more times."*

*Remember that classes aren't actual entities or objects. Classes are the blueprints or molds that we're going to use to create the actual objects.*

- *Class names are declared with a capital first letter and camelCase by convention. The class keyword creates a constant, so it cannot be redefined afterwards.*
- *Classes must always have a constructor method that will later on be used to instantiate that class. A constructor in JavaScript is just a plain old function that returns an object. The only thing special about it is that, when invoked with the "new" keyword, it assigns its prototype as the prototype of the returned object.*
- *The "this" keyword points to the class itself and is used to define the class properties within the constructor method.*
- *Methods can be added by simply defining the function name and its execution code.*
- *JavaScript is a prototype-based language, and within JavaScript classes are used only as syntactic sugar. This doesn't make a huge difference here, but it's good to know and keep in mind. You can read [this article if you'd like to know more about this topic](#).*

# The Four Principles of OOP

OOP is normally explained with 4 key principles that dictate how OOP programs work. These are **inheritance**, **encapsulation**, **abstraction** and **polymorphism**. Let's review each of them.

## Inheritance

Inheritance is the ability to **create classes based on other classes**. With inheritance, we can define a **parent class** (with certain properties and methods), and then **children classes** that will inherit from the parent class all the properties and methods that it has.

Let's see this with an example. Imagine all the characters we defined before will be the enemies of our main character. And as enemies, they will all have the "power" property and the "attack" method.

One way to implement that would be just to add the same properties and methods to all the classes we had, like this:

```
class Bug {  
  
    constructor (name, phrase, power) {  
  
        this.name = name  
  
        this.phrase = phrase
```

```
    this.power = power

    this.species = "bug"

  }

  hide = () => console.log("You can't catch me now!")

  sayPhrase = () => console.log(this.phrase)

  attack = () => console.log(`I'm attacking with a power of
${this.power}!`)

}
```

```
class Robot {

  constructor (name, phrase, power) {

    this.name = name

    this.phrase = phrase

    this.power = power

    this.species = "robot"

  }

}
```



```

    }

    transform = () => console.log("Optimus prime!")

    sayPhrase = () => console.log(this.phrase)

    attack = () => console.log(`I'm attacking with a power of
    ${this.power}!`)

  }

const bug1 = new Bug("Buggy", "Your debugger doesn't work with me!",
10)

const Robot1 = new Robot("Tito", "I can cook, swim and dance!", 15)

console.log(bug1.power) //output: 10

Robot1.attack() // output: "I'm attacking with a power of 15!"

```

But you can see we're repeating code, and that's not optimal.

A better way would be to declare a parent "Enemy" class which is then extended by all enemy species, like this:

```
class Enemy {  
  
    constructor(power) {  
  
        this.power = power  
  
    }  
  
    attack = () => console.log(`I'm attacking with a power of  
${this.power}!`)  
  
}
```

```
class Alien extends Enemy {  
  
    constructor(name, phrase, power) {  
  
        super(power)  
  
        this.name = name  
  
    }  
  
}
```

```

        this.phrase = phrase

        this.species = "alien"

    }

    fly = () => console.log("Zzzzzzziiiiinnnnngggg!!")

    sayPhrase = () => console.log(this.phrase)

}

...

```

See that the enemy class looks just like any other. We use the constructor method to receive parameters and assign them as properties, and methods are declared like simple functions.

On the children class, we use the **extends** keyword to declare the parent class we want to inherit from. Then on the constructor method, we have to declare the "power" parameter and use the **super** function to indicate that property is declared on the parent class.

When we instantiate new objects, we just pass the parameters as they were declared in the corresponding constructor function and *voilà!* We can now access the properties and methods declared in the parent class. 😎

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10)
```

```
const alien2 = new Alien("Lien", "Run for your lives!", 15)
```

```
alien1.attack() // output: I'm attacking with a power of 10!
```

```
console.log(alien2.power) // output: 15
```

Now let's say we want to add a new parent class that groups all our characters (no matter if they're enemies or not), and we want to set a property of "speed" and a "move" method. We can do that like this:

```
class Character {  
  
  constructor (speed) {  
  
    this.speed = speed  
  
  }  
}
```

```
    move = () => console.log(`I'm moving at the speed of  
${this.speed}!`)  
  
}
```

```
class Enemy extends Character {  
  
    constructor(power, speed) {  
  
        super(speed)  
  
        this.power = power  
  
    }
```

```
    attack = () => console.log(`I'm attacking with a power of  
${this.power}!`)  
  
}
```

```
class Alien extends Enemy {  
  
  constructor (name, phrase, power, speed) {  
  
    super(power, speed)  
  
    this.name = name  
  
    this.phrase = phrase  
  
    this.species = "alien"  
  
  }  
  
  fly = () => console.log("Zzzzzziinnnnngggg!!")  
  
  sayPhrase = () => console.log(this.phrase)  
  
}
```

First we declare the new "Character" parent class. Then we extend it on the Enemy class. And finally we add the new "speed" parameter to the constructor and super functions in our Alien class.

We instantiate passing the parameters as always, and *voilà* again, we can access properties and methods from the "grandparent" class. 😊

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
```

```
const alien2 = new Alien("Lien", "Run for your lives!", 15, 60)
```

```
alien1.move() // output: "I'm moving at the speed of 50!"
```

```
console.log(alien2.speed) // output: 60
```

Now that we know more about inheritance, let's refactor our code so we avoid code repetition as much as possible:

```
class Character {  
  
  constructor (speed) {  
  
    this.speed = speed  
  
  }  
  
  move = () => console.log(`I'm moving at the speed of  
${this.speed}!`)  
  
}
```

```
class Enemy extends Character {  
  
    constructor(name, phrase, power, speed) {  
  
        super(speed)  
  
        this.name = name  
  
        this.phrase = phrase  
  
        this.power = power  
  
    }  
  
    sayPhrase = () => console.log(this.phrase)  
  
    attack = () => console.log(`I'm attacking with a power of  
    ${this.power}!`)  
  
}
```

```
class Alien extends Enemy {
```



```
    constructor (name, phrase, power, speed) {

        super(name, phrase, power, speed)

        this.species = "alien"

    }

    fly = () => console.log("Zzzzzzziiiiinnnnnggggg!!")

}
```

```
class Bug extends Enemy {

    constructor (name, phrase, power, speed) {

        super(name, phrase, power, speed)

        this.species = "bug"

    }

    hide = () => console.log("You can't catch me now!")

}
```

```
class Robot extends Enemy {  
  
    constructor (name, phrase, power, speed) {  
  
        super(name, phrase, power, speed)  
  
        this.species = "robot"  
  
    }  
  
    transform = () => console.log("Optimus prime!")  
  
}
```

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
```

```
const alien2 = new Alien("Lien", "Run for your lives!", 15, 60)
```

```
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me!",  
25, 100)
```

```
const bug2 = new Bug("Erik", "I drink decaf!", 5, 120)
```

```
const Robot1 = new Robot("Tito", "I can cook, swim and dance!", 125,  
30)
```

```
const Robot2 = new Robot("Terminator", "Hasta la vista, baby!", 155,  
40)
```

See that our species classes look much smaller now, thanks to the fact that we moved all shared properties and methods to a common parent class. That's the kind of efficiency inheritance can help us with. 😊

### Some things to keep in mind about inheritance:

- A class can only have one parent class to inherit from. You can't extend multiple classes, though there're are hacks and ways around this.
- You can extend the inheritance chain as much as you want, setting parent, grandparent, great grandparent classes and so on.

If a child class inherits any properties from a parent class, it must first assign the parent properties calling the `super()` function before assigning its own properties.

## An example:

// This works:

```
class Alien extends Enemy {  
  
  constructor (name, phrase, power, speed) {  
  
    super(name, phrase, power, speed)  
  
    this.species = "alien"  
  
  }  
  
  fly = () => console.log("Zzzzzziiiiinnnnnggggg!!")  
  
}
```

// This throws an error:

```
class Alien extends Enemy {
```

```

    constructor (name, phrase, power, speed) {

        this.species = "alien" // ReferenceError: Must call super
        constructor in derived class before accessing 'this' or returning from
        derived constructor

        super(name, phrase, power, speed)

    }

    fly = () => console.log("Zzzzzzziiiiinnnnnggggg!!")

}

```

- When inheriting, all parent methods and properties will be inherited by the children. We can't decide what to inherit from a parent class (same as we can't choose what virtues and defects we inherit from our parents. 😊 We'll get back to this when we talk about composition).

Children classes can override the parent's properties and methods.

To give an example, in our previous code, the Alien class extends the Enemy class and it inherits the `attack` method which logs `I'm attacking with a power of ${this.power}!!`:

```
class Enemy extends Character {  
  
  constructor(name, phrase, power, speed) {  
  
    super(speed)  
  
    this.name = name  
  
    this.phrase = phrase  
  
    this.power = power  
  
  }  
  
  sayPhrase = () => console.log(this.phrase)
```

```
        attack = () => console.log(`I'm attacking with a power of  
${this.power}!`)  
  
    }
```

```
class Alien extends Enemy {  
  
    constructor (name, phrase, power, speed) {  
  
        super(name, phrase, power, speed)  
  
        this.species = "alien"  
  
    }  
  
    fly = () => console.log("Zzzzzziiiiinnnnngggg!!")  
  
}
```

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
```

```
alien1.attack() // output: I'm attacking with a power of 10!
```

Let's say we want the `attack` method to do a different thing in our `Alien` class. We can override it by declaring it again, like this:

```
class Enemy extends Character {  
  
  constructor(name, phrase, power, speed) {  
  
    super(speed)  
  
    this.name = name  
  
    this.phrase = phrase  
  
    this.power = power  
  
  }  
  
  sayPhrase = () => console.log(this.phrase)  
  
  attack = () => console.log(`I'm attacking with a power of  
    ${this.power}!`)
```



```
}
```

```
class Alien extends Enemy {
```

```
  constructor (name, phrase, power, speed) {
```

```
    super(name, phrase, power, speed)
```

```
    this.species = "alien"
```

```
  }
```

```
  fly = () => console.log("Zzzzzziiiiinnnnngggg!!")
```

```
  attack = () => console.log("Now I'm doing a different thing,  
HA!") // Override the parent method.
```

```
}
```

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
```

```
alien1.attack() // output: "Now I'm doing a different thing, HA!"
```

## Encapsulation

Encapsulation is another key concept in OOP, and it stands for an object's capacity to "decide" which information it exposes to "the outside" and which it doesn't. Encapsulation is implemented through **public and private properties and methods**.

In JavaScript, all objects' properties and methods are public by default. "Public" just means we can access an object's property/method from outside its own body:

```
// Here's our class
```

```
class Alien extends Enemy {  
  
    constructor (name, phrase, power, speed) {  
  
        super(name, phrase, power, speed)  
  
        this.species = "alien"  
  
    }  
  
    fly = () => console.log("Zzzzzziinnnnngggg!!")  
  
}
```

```
// Here's our object
```

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)
```

```
// Here we're accessing our public properties and methods
```

```
console.log(alien1.name) // output: Ali
```

```
alien1.sayPhrase() // output: "I'm Ali the alien!"
```

To make this clearer, let's see how private properties and methods look like.

Let's say we want our Alien class to have a `birthYear` property, and use that property to execute a `howOld` method, but we don't want that property to be accessible from anywhere else other than the object itself. We could implement that like this:

```
class Alien extends Enemy {  
  
    #birthYear // We first need to declare the private property,  
    always using the '#' symbol as the start of its name.
```

```
    constructor (name, phrase, power, speed, birthYear) {
```

```
        super(name, phrase, power, speed)
```

```
this.species = "alien"
```

```
this.#birthYear = birthYear // Then we assign its value  
within the constructor function
```

```
}
```

```
fly = () => console.log("Zzzzzziinnnnngggg!!")
```

```
howOld = () => console.log(`I was born in ${this.#birthYear}`)  
// and use it in the corresponding method.
```

```
}
```

```
// We instantiate the same way we always do
```

```
const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50, 10000)
```

Then we can access the `howOld` method, like this:

```
alien1.howOld() // output: "I was born in 10000"
```

But if we try to access the property directly, we'll get an error. And the private property won't show up if we log the object.

```
console.log(alien1.#birthYear) // This throws an error
```

```
console.log(alien1)
```

```
// output:
```

```
// Alien {
```

```
//   move: [Function: move],
```

```
//   speed: 50,
```

```
//   sayPhrase: [Function: sayPhrase],
```

```
//   attack: [Function: attack],
```

```
//   name: 'Ali',
```

```
//   phrase: "I'm Ali the alien!",
```

```
//   power: 10,
```

```
//   fly: [Function: fly],
```

```
//   howOld: [Function: howOld],
```

```
//   species: 'alien'
```

```
// }
```

Encapsulation is useful in cases where we need certain properties or methods for the inner working of the object, but we don't want to expose that to the exterior. Having private properties/methods ensures we don't "accidentally" expose information we don't want.

## **Abstraction**

Abstraction is a principle that says that a class should only represent information that is relevant to the problem's context. In plain English, only expose to the outside the properties and methods that you're going to use. If it's not needed, don't expose it.

This principle is closely related to encapsulation, as we can use public and private properties/methods to decide what gets exposed and what doesn't.

## **Polymorphism**

Then there is polymorphism (sounds really sophisticated, doesn't it? OOP names are the coolest... 🙄). Polymorphism means "many forms" and is actually a simple concept. It's the ability of one method to return different values according to certain conditions.

For example, we saw that the Enemy class has the `sayPhrase` method. And all our species classes inherit from the Enemy class, which means they all have the `sayPhrase` method as well.

But we can see that when we call the method on different species, we get different results:

```
const alien2 = new Alien("Lien", "Run for your lives!", 15, 60)
```

```
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me!",  
25, 100)
```

```
alien2.sayPhrase() // output: "Run for your lives!"
```

```
bug1.sayPhrase() // output: "Your debugger doesn't work with me!"
```

And that's because we passed each class a different parameter at instantiation. That's one kind of polymorphism, **parameter-based**. 🔥

Another kind of polymorphism is **inheritance-based**, and that refers to when we have a parent class that sets a method and the child overrides that method to modify it in some way.

The example we saw previously applies perfectly here as well:

```
class Enemy extends Character {

    constructor(name, phrase, power, speed) {

        super(speed)

        this.name = name

        this.phrase = phrase

        this.power = power

    }

    sayPhrase = () => console.log(this.phrase)

    attack = () => console.log(`I'm attacking with a power of
    ${this.power}!`)

}

class Alien extends Enemy {

    constructor (name, phrase, power, speed) {

        super(name, phrase, power, speed)
```



```

    this.species = "alien"

}

fly = () => console.log("Zzzzzzziiiiinnnnngggg!!")

attack = () => console.log("Now I'm doing a different thing,
HA!") // Override the parent method.

}

const alien1 = new Alien("Ali", "I'm Ali the alien!", 10, 50)

alien1.attack() // output: "Now I'm doing a different thing, HA!"

```

This implementation is polymorphic because if we commented out the `attack` method in the `Alien` class, we would still be able to call it on the object:

```
alien1.attack() // output: "I'm attacking with a power of 10!"
```

We got the same method that can do one thing or another depending if it was overridden or not. Polymorphic. 🔥 🔥

# Object Composition

[Object composition](#) is a technique that works as an alternative to inheritance.

When we talked about inheritance we mentioned that child classes always inherit all parent methods and properties. Well, by using composition we can assign properties and methods to objects in a more flexible way than inheritance allows, so objects only get what they need and nothing else.

We can implement this quite simply, by using functions that receive the object as a parameter and assign it the desired property/method. Let's see it in an example.

Say now we want to add the flying ability to our bug characters. As we've seen in our code, only aliens have the `fly` method. So one option could be to duplicate the exact same method in the `Bug` class:

```
class Alien extends Enemy {  
  
    constructor (name, phrase, power, speed) {  
  
        super(name, phrase, power, speed)  
  
        this.species = "alien"  
  
    }  
}
```

```

    fly = () => console.log("Zzzzzziinnnnngggg!!")

}

class Bug extends Enemy {

    constructor (name, phrase, power, speed) {

        super(name, phrase, power, speed)

        this.species = "bug"

    }

    hide = () => console.log("You can't catch me now!")

    fly = () => console.log("Zzzzzziinnnnngggg!!") // We're
duplicating code =(

}

```

Another option would be to move the `fly` method up to the `Enemy` class, so it can be inherited by both the `Alien` and `Bug` classes. But that also makes the method available to classes that don't need it, like `Robot`.

```
class Enemy extends Character {

    constructor(name, phrase, power, speed) {

        super(speed)

        this.name = name

        this.phrase = phrase

        this.power = power

    }

    sayPhrase = () => console.log(this.phrase)

    attack = () => console.log(`I'm attacking with a power of
    ${this.power}!`)

    fly = () => console.log("Zzzzzziiiiiinnnnnggggg!!")

}
```

```
class Alien extends Enemy {
```

```
    constructor (name, phrase, power, speed) {  
  
        super(name, phrase, power, speed)  
  
        this.species = "alien"  
  
    }  
  
}  
  
class Bug extends Enemy {  
  
    constructor (name, phrase, power, speed) {  
  
        super(name, phrase, power, speed)  
  
        this.species = "bug"  
  
    }  
  
    hide = () => console.log("You can't catch me now!")  
  
}
```

```
class Robot extends Enemy {  
  
  constructor (name, phrase, power, speed) {  
  
    super(name, phrase, power, speed)  
  
    this.species = "robot"  
  
  }  
  
  transform = () => console.log("Optimus prime!")  
  
  // I don't need the fly method =(  
  
}
```

As you can see, inheritance causes problems when the starting plan we had for our classes changes (which in the real world is pretty much always). Object composition proposes an approach in which objects get properties and methods assigned only as they need them.

In our example, we could create a function and its only responsibility would be to add the flying method to any object that receives as parameter:

```
const bug1 = new Bug("Buggy", "Your debugger doesn't work with me!",  
25, 100)
```

```
const addFlyingAbility = obj => {  
  
  obj.fly = () => console.log(`Now ${obj.name} can fly!`)  
  
}
```

```
addFlyingAbility(bug1)
```

```
bug1.fly() // output: "Now Buggy can fly!"
```

And we could have very similar functions for each power or ability we may want our monsters to have.

As you can surely see, this approach is a lot more flexible than having parent classes with fixed properties and methods to inherit. Whenever an object needs a method, we just call the corresponding function and that's it. 🔥