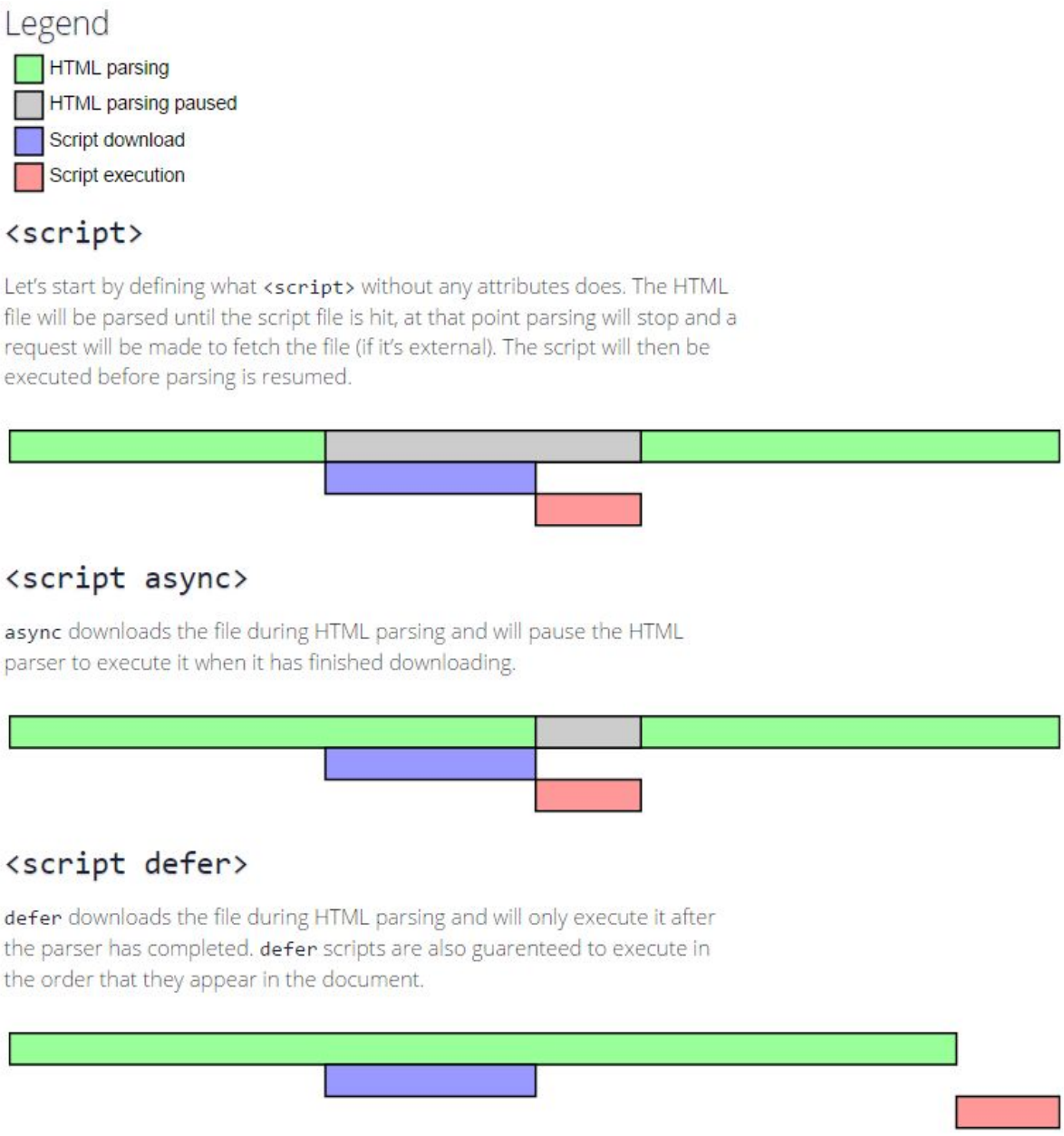
JS execution order in HTML



- Async scripts are executed as soon as the script is loaded, so it doesn't guarantee the order of execution (a script you included at the end may execute before the first script file)
- Defer scripts guarantees the order of execution in which they appear in the page.

To attach several scripts, use multiple tags:

```
<script src="/js/script1.js"></script>
<script src="/js/script2.js"></script>
```

As a rule, only the simplest scripts are put into HTML. More complex ones reside in separate files.

The benefit of a separate file is that the browser will download it and store it in its cache.
Other pages that reference the same script will take it from the cache instead of downloading it, so the file is actually downloaded only once.

That reduces traffic and makes pages faster.

Java script:

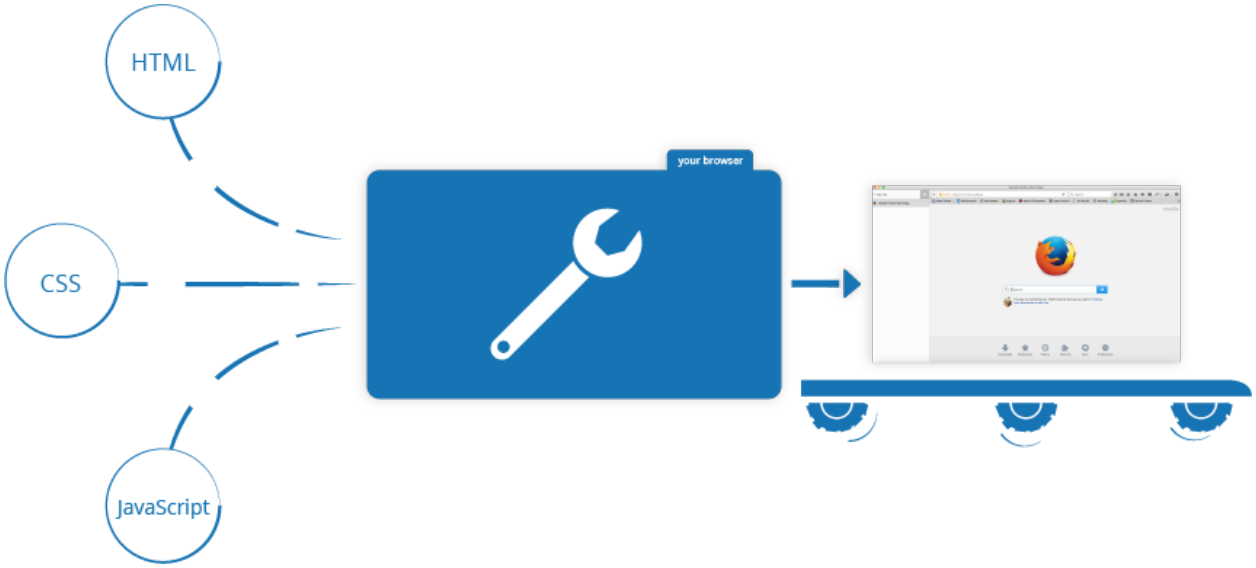
JavaScript® (often shortened to JS) is a lightweight, interpreted, object-oriented language with first-class functions, and is best known as the scripting language for Web pages, but it's used in many non-browser environments as well.

JavaScript runs on the client side of the web, which can be used to design / program how the web pages behave on the occurrence of an event. JavaScript is an easy
to learn and also powerful scripting language, widely used for controlling web page behaviour.

JavaScript is an object-based language **based on prototypes**, rather than being class-based.

JavaScript is a programming language that allows you to implement complex things on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, or interactive maps, or animated 2D/3D graphics, or scrolling video jukeboxes, etc. — you can bet that JavaScript is probably involved.

What happens behind the browser:



The JavaScript is executed by the **browser's JavaScript engine**, after the HTML and CSS have been assembled and put together into a web page. This ensures that the structure and style of the page are already in place by the time the JavaScript starts to run.

This is a good thing, as a very common use of JavaScript is to dynamically modify HTML and CSS to update a user interface, via the Document Object Model API (as mentioned above). If the JavaScript loaded and tried to run before the HTML and CSS was there to affect, then errors would occur.

What are browser developer tools:

Every modern web browser includes a powerful suite of developer tools. These tools do a range of things, from inspecting currently-loaded HTML, CSS and JavaScript to showing which assets the page has requested and how long they took to load.

JavaScript running order

When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in. For example, let's return to the block of JavaScript we saw in our first example:

Code:

```
var para = document.querySelector('p');
para.addEventListener('click', updateName);
function updateName() {
    var name = prompt('Enter a new name');
    para.textContent = 'Player 1: ' + name;
}
```

Here we are selecting a text paragraph (line 1), then attaching an event listener to it (line 3) so that when the paragraph is clicked, the updateName() code block (lines 5–8) is run.

If you swapped the order of the first two lines of code, it would no longer work — instead, you'd get an error returned in the browser developer console — TypeError: para is undefined. This means that the para object does not exist yet, so we can't add an event listener to it.

Interpreted versus compiled code:

You might hear the terms interpreted and compiled in the context of programming. JavaScript is an **interpreted language** — the code is run from top to bottom and the result of running the code is immediately returned. You don't have to transform the code into a different form before the browser runs it.

Compiled languages on the other hand are transformed (compiled) into another form before they are run by the computer. For example C/C++ are compiled into assembly language that is then run by the computer.

How do you add javascript into your page:

Internal:

```
<script>  </script>
```

External:

```
<script src=”script.js”> </script>
```

Inline javascript handlers:

```
function createParagraph() {
    var para = document.createElement('p');
    para.textContent = 'You clicked the button!';
    document.body.appendChild(para);
}

<button onclick="createParagraph()">Click me!</button>
```

Note : Please don't do this, however. It is bad practice to pollute your HTML with JavaScript, and it is inefficient — you'd have to include the onclick="createParagraph()" attribute on every button you wanted the JavaScript to apply to.

Comment:
Single Line : // , Multi - line comment : /* */

Java script error ref:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors

Javascript variables:

A variable is a container for a value, like a number we might use in a sum, or a string that we might use as part of a sentence. But one special thing about variables is that their contained values can change. Let's look at a simple example:

A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

Some examples of legal names are Number_hits, temp99, $credit, and _name.

Evaluating variables:

A variable declared using the var or let statement with no assigned value specified has the value of `undefined`.
An attempt to access an undeclared variable results in a ReferenceError exception being thrown:

```
var a;
console.log('The value of a is ' + a); // The value of a is undefined
console.log('The value of b is ' + b); // The value of b is undefined
var b;
console.log('The value of c is ' + c); // Uncaught ReferenceError: c is not defined
let x;
console.log('The value of x is ' + x); // The value of x is undefined
console.log('The value of y is ' + y); // Uncaught ReferenceError: y is not defined
let y;
```

You can use undefined to determine whether a variable has a value. In the following code, the variable input is not assigned a value, and the if statement evaluates to true.

```
var input;
if (typeof input === undefined) {
    doThis();
} else {
    doThat();
}
```

- Variables are case sensitive — so **myage is a different variable to myAge**.
- A safe convention to stick to is so-called "lower camel case"
- Avoid using JavaScript reserved words as your variable names — by this, we mean the words that make up the actual syntax of JavaScript! So you can't use words like var, function, let, and for as variable names. Browsers will recognize them as different code items, and so you'll get errors.

Scope of a variable:

Assigning a value to an undeclared variable implicitly creates it as a global variable (it becomes a property of the global object) when the assignment is executed. The differences between declared and undeclared variables are:

Declared variables are constrained in the execution context in which they are declared. Undeclared variables are always global.

```
function x() {
    y = 1;   // Throws a ReferenceError in strict mode
     var z = 2;
}
x();
console.log(y); // logs "1"
console.log(z); // Throws a ReferenceError: z is not defined outside x
```

Declared variables are a non-configurable property of their execution context (function or global). Undeclared variables are configurable (e.g. can be deleted).

```
var a = 1;
b = 2;
delete this.a; // Throws a TypeError in strict mode. Fails silently otherwise.
delete this.b;
console.log(a, b); // Throws a ReferenceError.
```

For instance,

```
var x= 10 ;
var y = 10;
{
    var x= 6;
    let y=6;
}
console.log(x) // prints 6
```

```
console.log(y) // prints 10
```

For instance,

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

Var hoisting

Because **variable declarations (and declarations in general) are processed before any code is executed**, declaring a variable anywhere in the code is equivalent to declaring it at the top. This also means that a variable can appear to be used before it's declared. This behavior is called "hoisting", as it appears that the variable declaration is moved to the top of the function or global code.

```
bla = 2;
var bla;
// ...

// is implicitly understood as:
var bla;
bla = 2;
```

For that reason, it is recommended to always declare variables at the top of their scope (the top of global code and the top of function code) so it's clear which variables are function scoped (local) and which are resolved on the scope chain.

It's important to point out that the hoisting will affect the variable declaration, but not its value's initialization. The value will be indeed assigned when the assignment statement is reached:

```
function do_something() {
    console.log(bar); // undefined
    var bar = 111;
    console.log(bar); // 111
}

// is implicitly understood as:
function do_something() {
    var bar;
    console.log(bar); // undefined
    bar = 111;
    console.log(bar); // 111
}
```

“var” has no block scope

Variables, declared with var, are either function-wide or global. 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); // Error: test is not defined
```

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

```
for (var i = 0; i < 10; i++) {
    // ...
}
alert(i); // 10, "i" 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); // Error: phrase is not defined (Check the Developer Console)
```

Global object

The global object provides variables and functions that are available anywhere. By default, those that are built into the language or the environment.

In a browser it is named window, for Node.js it is global, for other environments it may have another name.

Recently, globalThis was added to the language, as a standardized name for a global object, that should be supported across all environments. In some browsers, namely non-Chromium Edge, globalThis is not yet supported, but can be easily polyfilled.

We'll use window here, assuming that our environment is a browser. If your script may run in other environments, it's better to use globalThis instead.

All properties of the global object can be accessed directly:

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

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

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

Please don’t rely on that! This behavior exists for compatibility reasons. Modern scripts use JavaScript modules where such thing doesn’t happen.

If we used let instead, such thing wouldn’t happen:

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

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

```
// make current user information global, to let all scripts access it
window.currentUser = {
    name: "John"
};

// somewhere else in code
alert(currentUser.name);  // John

// or, if we have a local variable with the name "currentUser"
// get it from window explicitly (safe!)
alert(window.currentUser.name); // John
```

We use the global object to test for support of modern language features.

For instance, test if a built-in Promise object exists (it doesn’t in really old browsers):

```
if (!window.Promise) {
    alert("Your browser is really old!");
}
```

If there’s none (say, we’re in an old browser), we can create “polyfills”: add functions that are not supported by the environment, but exist in the modern standard.

```
if (!window.Promise) {
    window.Promise = ... // custom implementation of the modern language feature
}
```

Variable types:

Numbers

You can store numbers in variables, either whole numbers like 30 (also called integers) or decimal numbers like 2.456 (also called floats or floating point numbers). You don't need to declare variable types in JavaScript, unlike some other programming languages. When you give a variable a number value, you don't include quotes:

```
var myAge = 17;
```

Booleans

Booleans are true/false values — they can have two values, true or false. These are generally used to test a condition, after which code is run as appropriate. So for example, a simple case would be:

```
var iAmAlive = true;
```

Whereas in reality it would be used more like this:

```
var test = 6 < 3;
```

This is using the "less than" operator (<) to test whether 6 is less than 3. As you might expect, it will return false, because 6 is not less than 3! You will learn a lot more about such operators later on in the course.

The “undefined” value

The special value undefined also stands apart. It makes a type of its own, just like null.

The meaning of undefined “value is not assigned”.

If a variable is declared, but not assigned, then its value is undefined:

```
let x;
alert(x); // shows "undefined"
```

Data type conversion

JavaScript is a **dynamically typed language**. That means you don't have to specify the data type of a variable when you declare it, and data types are converted automatically as needed during script execution. So, for example, you could define a variable as follows:

```
var answer = 42;
```

And later, you could assign the same variable a string value, for example:

```
answer = 'Thanks for all the fish...';
```

Because JavaScript is dynamically typed, this assignment does not cause an error message.

In expressions involving numeric and string values with the + operator, JavaScript converts numeric values to strings. For example, consider the following statements:

```
x = 'The answer is ' + 42 // "The answer is 42"
y = 42 + ' is the answer' // "42 is the answer"
```

In statements involving other operators, JavaScript does not convert numeric values to strings. For example:

```
'37' - 7 // 30
'37' + 7 // "377"
```

String conversion

String conversion happens when we need the string form of a value.

For example, alert(value) does it to show the value.

We can also call the String(value) function to convert a value to a string:

```
let value = true;
alert(typeof value); // boolean

value = String(value); // now value is a string "true"
alert(typeof value); // string
```

String conversion is mostly obvious. A false becomes "false", null becomes "null", etc.

Numeric conversion

We can use the Number(value) function to explicitly convert a value to a number:

```
let str = "123";
alert(typeof str); // string

let num = Number(str); // becomes a number 123
alert(typeof num); // number
```

Explicit conversion is usually required when we read a value from a string-based source like a text form but expect a number to be entered.

If the string is not a valid number, the result of such a conversion is NaN

```
alert( Number("   123   ") ); // 123
```

```
alert( Number("123z") );      // NaN (error reading a number at "z")
alert( Number(true) );        // 1
alert( Number(false) );       // 0
```

Boolean conversion

Boolean conversion is the simplest one.

It happens in logical operations (later we'll meet condition tests and other similar things) but can also be performed explicitly with a call to Boolean(value).

The conversion rule:

- Values that are intuitively "empty", like 0, an empty string, null, undefined, and NaN, become false.
- Other values become true.

For instance:

```
alert( Boolean(1) ); // true
alert( Boolean(0) ); // false
alert( Boolean("hello") ); // true
alert( Boolean("") ); // false
```

Numbers versus string

So what happens when we try to add (or concatenate) a string and a number?

- `'Front ' + 242;`

You might expect this to throw an error, but it works just fine. Trying to represent a string as a number doesn't really make sense, but representing a number as a string does, so the browser rather cleverly converts the number to a string and concatenates the two strings together.

- You can even do this with two numbers — you can force a number to become a string by wrapping it in quote marks.

```
var myDate = '19' + '67';
typeof myDate;
```

If you have a numeric variable that you want to convert to a string but not change otherwise, or a string variable that you want to convert to a number but not change otherwise, you can use the following two constructs:

- The Number object will convert anything passed to it into a number, if it can. Try the following:
  ```
  var myString = '123';
  var myNum = Number(myString);
  typeof myNum; // number

  var myString = 'hello';
  var myNum = Number(myString);
  typeof myNum; // NaN
  ```

- On the other hand, every number has a method called toString() that will convert it to the equivalent string. Try this:
  ```
  var myNum = 123;
  var myString = myNum.toString();
  typeof myString;
  ```

Strings

Strings are pieces of text. When you give a variable a string value, you need to wrap it in single or double quote marks, otherwise JavaScript will try to interpret it as another variable name.

Strings as objects

We've said it before, and we'll say it again — ***everything* is an object in JavaScript**. When you create a string, for example by using

```
var string = 'This is my string';
```

your variable becomes a string object instance, and as a result has a large number of properties and methods available to it. You can see this if you go to the String object page and look down the list on the side of the page!

Long literal strings:

Sometimes, your code will include strings which are very long. Rather than having lines that go on endlessly, or wrap at the whim of your editor, you may wish to specifically break the string into multiple lines in the source code without affecting the actual string contents. There are two ways you can do this.

You can use the + operator to append multiple strings together, like this:

```
let longString = "This is a very long string which needs " +
                 "to wrap across multiple lines because " +
                 "otherwise my code is unreadable.";
```

Or you can use the backslash character ("\") at the end of each line to indicate that the string will continue on the next line. Make sure there is no space or any other character after the backslash (except for a line break), or as an indent; otherwise it will not work. That form looks like this:

```
let longString = "This is a very long string which needs \
to wrap across multiple lines because \
otherwise my code is unreadable.";
```

Character access

There are two ways to access an individual character in a string. The first is the charAt() method:

```
return 'cat'.charAt(1); // returns "a"
```

The other way (introduced in ECMAScript 5) is to treat the string as an array-like object, where individual characters correspond to a numerical index:

```
return 'cat'[1]; // returns "a"
```

Finding the length of a string :

This is easy — you simply use the length property. Try entering the following lines:

```
var browserType = 'mozilla';
browserType.length;
```

Comparing the length of a string :

C developers have the strcmp() function for comparing strings. In JavaScript, you just use the less-than and greater-than operators:

```
var a = 'a';
var b = 'b';
if (a < b) { // true
  console.log(a + ' is less than ' + b);
} else if (a > b) {
  console.log(a + ' is greater than ' + b);
} else {
  console.log(a + ' and ' + b + ' are equal.');
}
```

Distinction between string primitives and String objects

Note that JavaScript distinguishes between String objects and primitive string values. (The same is true of Boolean and Numbers.)

String literals (denoted by double or single quotes) and strings returned from String calls in a non-constructor context (i.e., without using the new keyword) are primitive strings. **JavaScript automatically converts primitives to String objects**, so that it's possible to use String object

methods for primitive strings. In contexts where a method is to be invoked on a primitive string or a property lookup occurs, JavaScript will automatically wrap the string primitive and call the method or perform the property lookup.

```
var s_prim = 'foo';
var s_obj = new String('foo');

console.log(s_prim) // 'foo'
console.log(s_obj) // {'0': 'f', '1': 'o', '2': 'o'}
console.log(typeof s_prim); // Logs "string"
console.log(typeof s_obj);  // Logs "object"
```

String primitives and String objects also give different results when using eval(). Primitives passed to eval are treated as source code; String objects are treated as all other objects are, by returning the object. For example:

```
var s1 = '2 + 2';             // creates a string primitive
var s2 = new String('2 + 2'); // creates a String object
console.log(eval(s1));        // returns the number 4
console.log(eval(s2));        // returns the string "2 + 2"
```

For these reasons, code may break when it encounters String objects when it expects a primitive string instead, although generally authors need not worry about the distinction.

A String object can always be converted to its primitive counterpart with the valueOf() method.

```
console.log(eval(s2.valueOf())); // returns the number 4
```

For more string methods ,
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Valid string examples :

```
var sgl = 'Single quotes.';
var dbl = "Double quotes";
var sglDbl = 'Would you eat a "fish supper"?';
var dblSgl = "I'm feeling blue.";
var bigmouth = 'I\' ve got no right to take my place...';
var dolphinGoodbye = 'So long and thanks for all the fish';
```

String interpolation:

The JavaScript term for inserting the data saved to a variable into a string is *string interpolation*.

```
let myPet= “dog”
console.log(‘I own a pet’ + myPet + ‘.’);
```

In the newest version of JavaScript (ES6) we can insert variables into strings with ease, by using ` symbol instead of quotes and with $ symbol.

```
let myName="Gautham";
let myCity="Chennai";
console.log(`My name is ${myName}. My favorite city is ${myCity}`);
```

Using special characters in strings

In addition to ordinary characters, you can also include special characters in strings, as shown in the following example.

\0	Null Byte
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return

\t	Tab
\v	Vertical tab
\'	Apostrophe or single quote
\"	Double quote
\\	Backslash character

Escaping characters

For characters not listed in the table, a preceding backslash is ignored, but this usage is deprecated and should be avoided.

You can insert a quotation mark inside a string by preceding it with a backslash. This is known as *escaping* the quotation mark. For example:

```
var quote = "He read \"The Cremation of Sam McGee\" by R.W. Service.";
console.log(quote);
```

The result of this would be:
```
He read "The Cremation of Sam McGee" by R.W. Service.
```

To include a literal backslash inside a string, you must escape the backslash character. For example, to assign the file path c:\temp to a string, use the following:
```
var home = 'c:\\temp';
```

Multi line strings:

Any new line characters inserted in the source are part of the template literal. Using normal strings, you would have to use the following syntax in order to get multi-line strings:

```
console.log('string text line 1\n\
string text line 2');
```

```
// "string text line 1
// string text line 2"
```

To get the same effect with multi-line strings, you can now write:

```
console.log(`string text line 1
string text line 2`);
// "string text line 1
// string text line 2"
```

Finding a substring inside a string and extracting it

Sometimes you'll want to find if a smaller string is present inside a larger one (we generally say *if a substring is present inside a string*). This can be done using the indexOf() method, which takes a single parameter — the substring you want to search for. Try this:

```
let browserType = ‘mozilla’
```

- `browserType.indexOf('zilla');` // return 2 (because substring “zilla” starts at position 2)
- `browserType.indexOf('vanilla');` // return -1 (not found).
- When you know where a substring starts inside a string, and you know at which character you want it to end, slice() can be used to extract it.
  `browserType.slice(0,3);` // returns moz

Changing case

The string methods toLowerCase() and toUpperCase() take a string and convert all the characters to lower- or uppercase, respectively.

```
var radData = 'My NaMe Is MuD';
radData.toLowerCase();
radData.toUpperCase();
```

Updating parts of a string

You can replace one substring inside a string with another substring using the replace() method. This works very simply at a basic level, although there are some advanced things you can do with it that we won't go into yet.

It takes two parameters — the string you want to replace, and the string you want to replace it with.

```
browserType.replace('moz','van');
```

Numbers:

There is no specific type for integers. In addition to being able to represent floating-point numbers, the number type has three symbolic values: +Infinity, -Infinity, and NaN (not-a-number). See also JavaScript data types and structures for context with other primitive types in JavaScript.

You can use four types of number literals: decimal, binary, octal, and hexadecimal.

Decimal numbers

```
1234567890
42
// Caution when using leading zeros:
0888 // 888 parsed as decimal
0777 // parsed as octal in non-strict mode (511 in decimal)
```

Note that decimal literals can start with a zero (0) followed by another decimal digit, but if every digit after the leading 0 is smaller than 8, the number gets parsed as an octal number.

Binary:

Binary number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "B" (0b or 0B). If the digits after the 0b are not 0 or 1, the following SyntaxError is thrown: "Missing binary digits after 0b".

```
var FLT_SIGNBIT  = 0b10000000000000000000000000000000; // 2147483648
var FLT_EXPONENT = 0b01111111100000000000000000000000; // 2139095040
var FLT_MANTISSA = 0B00000000011111111111111111111111; // 8388607
```

Octal numbers:

Octal number syntax uses a leading zero. If the digits after the 0 are outside the range 0 through 7, the number will be interpreted as a decimal number.

```
var n = 0755; // 493
var m = 0644; // 420
```

In ECMAScript 2015, octal numbers are supported if they are prefixed with 0o, e.g.:

```
var a = 0o10; // ES2015: 8
```

Hexadecimal numbers:

Hexadecimal number syntax uses a leading zero followed by a lowercase or uppercase Latin letter "X" (0x or 0X). If the digits after 0x are outside the range (0123456789ABCDEF), the following SyntaxError is thrown: "Identifier starts immediately after numeric literal".

```
0xFFFFFFFFFFFFFFFF // 295147905179352830000
0x123456789ABCDEF   // 81985529216486900
0XA                 // 10
```

Exponentiation

```
1E3   // 1000
2e6   // 2000000
0.1e2 // 10
```

Number object:

The built-in Number object has properties for numerical constants, such as maximum value, not-a-number, and infinity. You cannot change the values of these properties and you use them as follows:

```
var biggestNum = Number.MAX_VALUE;
var smallestNum = Number.MIN_VALUE;
var infiniteNum = Number.POSITIVE_INFINITY;
var negInfiniteNum = Number.NEGATIVE_INFINITY;
var notANum = Number.NaN;
```

Methods of Number:

```
Number.parseFloat(3.5444) // returns 3.5444
Number.parseInt(3.2) // returns 3
Number.isInteger(5) // returns true
Number.isInteger(5.2) // returns false
Number.isNaN()
```

Math Object:

The built-in Math object has properties and methods for mathematical constants and functions. For example, the Math object's PI property has the value of pi (3.141...), which you would use in an application as

```
console.log(Math.PI)
```

For Math methods ,
https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide/Numbers_and_dates

---

Operators

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

  ```
  let x = 1;
  x = -x;
  alert( x ); // -1, unary negation was applied
  ```
- An operator is *binary* if it has two operands. The same minus exists in binary form as well:

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

String concatenation, binary +

Now, let’s see special 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 one of the operands is a string, 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. The rule is simple: if either operand is a string, the other one is converted into a string as well.

However, note that operations run from left to right. If there are two numbers followed by a string, the numbers will be added before being converted to a string:

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

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
```

Operator precedence

If an expression has more than one operator, the execution order is defined by their *precedence*, or, in other words, the default priority order of operators.

Parentheses override any precedence, so if we're not satisfied with the default order, we can use them to change it. For example, write `(1 + 2) * 2`.

There are many operators in JavaScript. Every operator has a corresponding precedence number. The one with the larger number executes first. If the precedence is the same, the execution order is from left to right.

Here's an extract from the precedence table (you don't need to remember this, but note that unary operators are higher than corresponding binary ones):

Precedence	Name	Sign
…	…	…
16	unary plus	`+`
16	unary negation	`-`
14	multiplication	`*`
14	division	`/`
13	addition	`+`
13	subtraction	`-`
…	…	…
3	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 `3`.

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
```

It is possible to chain assignments:

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

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

Remainder %

The remainder operator `%`, despite its appearance, is not related to percents.

The result of `a % b` is the remainder of the integer division of `a` by `b`.

For instance:

```
alert( 5 % 2 ); // 1 is a remainder of 5 divided by 2
alert( 8 % 3 ); // 2 is a remainder of 8 divided by 3
alert( 6 % 3 ); // 0 is a remainder of 6 divided by 3
```

**Exponentiation **

The exponentiation operator `**` is a recent addition to the language.

For a natural number `b`, the result of `a ** b` is `a` multiplied by itself `b` times.

For instance:

```
alert( 2 ** 2 ); // 4  (2 * 2)
alert( 2 ** 3 ); // 8  (2 * 2 * 2)
alert( 2 ** 4 ); // 16 (2 * 2 * 2 * 2)
```

The operator works for non-integer numbers as well.

For instance:

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

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
```

To see the difference, here's an example:

```
let counter = 1;
let a = ++counter; // (*)
alert(a); // 2
```

Now, let's use the postfix form:

```
let counter = 1;
let a = counter++; // (*) changed ++counter to counter++
alert(a); // 1
```

Comma

The comma operator is one of the rarest and most unusual operators. Sometimes, it's used to write shorter code, so we need to know it in order to understand what's going on.

The comma operator allows us to evaluate several expressions, dividing them with a comma ,. Each of them is evaluated but only the result of the last one is returned.

For example:

```
let a = (1 + 2, 3 + 4)
alert( a ); // 7 (the result of 3 + 4)
```

Here, the first expression `1 + 2` is evaluated and its result is thrown away. Then, `3 + 4` is evaluated and returned as the result.

Comparison

String comparison

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

In other words, strings are compared letter-by-letter.

For example:

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

Comparison of different types

When comparing values of different types, JavaScript converts the values to numbers.

For example:

```
alert( '2' > 1 ); // true, string '2' becomes a number 2
alert( '01' == 1 ); // true, string '01' becomes a number 1
```

For boolean values, `true` becomes `1` and `false` becomes `0`.

For example:

```
alert( true == 1 ); // true
alert( false == 0 ); // true
```

Strict equality

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

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

The same thing happens with an empty string:

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

This happens because operands of different types are converted to numbers by the equality operator `==`. An empty string, just like `false`, becomes a zero.

What to do if we'd like to differentiate `0` from `false`?

A strict equality operator `===` checks the equality without type conversion.

In other words, if `a` and `b` are of different types, then `a === b` immediately returns `false` without an attempt to convert them.

Let's try it:

```
alert( 0 === false ); // false, because the types are different
```

There is also a "strict non-equality" operator `!==` analogous to `!=`.

The strict equality operator is a bit longer to write, but makes it obvious what's going on and leaves less room for errors.

Comparison with null and undefined

There's a non-intuitive behavior when `null` or `undefined` are compared to other values.

For a strict equality check `===`

These values are different, because each of them is a different type.

```
alert( null === undefined ); // false
```

For a non-strict check `==`

There's a special rule. These two are a "sweet couple": they equal each other (in the sense of `==`), but not any other value.

```
alert( null == undefined ); // true
```

Interaction : alert, prompt, confirm

- **Alert**

  `alert(message);`

  This shows a message and pauses script execution until the user presses “OK”.

- **Prompt**

  The function `prompt` accepts two arguments:

  `result = prompt(title, [default]);`

  It shows a modal window with a text message, an input field for the visitor, and the buttons OK/Cancel.

  title

  The text to show the visitor.

  default

  An optional second parameter, the initial value for the input field.

  The visitor may type something in the prompt input field and press OK. Or they can cancel the input by pressing Cancel or hitting the Esc key.

  The call to `prompt` returns the text from the input field or `null` if the input was canceled.

  For instance:

  ```
  let age = prompt('How old are you?', 100);
  alert(`You are ${age} years old!`); // You are 100 years old!
  ```

- **Confirm**

  `result = confirm(question);`

The function `confirm` shows a modal window with a `question` and two buttons: OK and Cancel.

The result is `true` if OK is pressed and `false` otherwise.

For example:

```
let isBoss = confirm("Are you the boss?");
alert( isBoss ); // true if OK is pressed
```

Objects:

An object is a collection of related data and/or functionality (which usually consists of several variables and functions — which are called **properties** and **methods** when they are inside objects.)

Creating an object often begins with defining and initializing a variable.

```
var person = {
    name: ['Bob', 'Smith'],
    age: 32,
    gender: 'male',
    interests: ['music', 'skiing'],
    bio: function() {
        alert(this.name[0] + ' ' + this.name[1] + ' is ' + this.age + ' years old. He likes ' +
this.interests[0] + ' and ' + this.interests[1] + '.');
    },
    greeting: function() {
        alert('Hi! I\'m ' + this.name[0] + '.');
    }
};
```

In our person object we've got a string, a number, two arrays, and two functions. The first four items are data items, and are referred to as the object's **properties**. The last two items are

functions that allow the object to do something with that data, and are referred to as the object's **methods**.

Multiword property names:

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
};
```

To work with multi word properties,

```
user.likes birds = true       // throws error
user["likes birds"] = true    //works
```

Computed properties

We can use square brackets in an object literal. 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}`.

We can use more complex expressions inside square brackets:

```
let fruit = 'apple';
let bag = {
    [fruit + 'Computers']: 5 // bag.appleComputers = 5
};
```

Square brackets are much more powerful than the dot notation. They allow any property names and variables. But they are also more cumbersome to write.

So most of the time, when property names are known and simple, the dot is used. And if we need something more complex, then we switch to square brackets.

Dot notation:

We can access the object's properties and methods using **dot notation**. The object name (person) acts as the **namespace** — it must be entered first to access anything **encapsulated** inside the object. Next you write a dot, then the item you want to access — this can be the name of a simple property, an item of an array property, or a call to one of the object's methods, for example:

```
person.age
person.interests[1]
person.bio()
```

```
var foo = {a: 'alpha', 2: 'two'};
console.log(foo.a);    // alpha
console.log(foo[2]);   // two
//console.log(foo.2);  // Error: Unexpected number
//console.log(foo[a]); // Error: a is not defined
console.log(foo['a']); // alpha
console.log(foo['2']); // two
```

Bracket Notation:

There is another way to access object properties — using bracket notation. Instead of using these:

`person.age`
`person.name.first`

You can use
`person['age']`
`person['name']['first']`

Setting object member:
So far we've only looked at retrieving (or **getting**) object members — you can also **set** (update) the value of object members by simply declaring the member you want to set (using dot or bracket notation), like this:

```
person.age = 45;
person['name']['last'] = 'Cratchit';
```

Try entering these lines, and then getting the members again to see how they've changed:

`person.age`
`person['name']['last']`

Setting members doesn't just stop at updating the values of existing properties and methods; you can also create completely new members. Try these:

```
person['eyes'] = 'hazel';
person.farewell = function() { alert("Bye everybody!"); }
```

Deleting object property

To remove a property, we can use `delete` operator:
`delete person.age;`

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
    };
}

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
        // ...
    };
}
```

Existence check

A notable objects feature is that it's possible to access any property. There will be no error if the property doesn't exist! Accessing a non-existing property just returns `undefined`. It provides a very common way to test whether the property exists – to get it and compare vs undefined:

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

There also exists a special operator "in" to check for the existence of a property.

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.

Enumerate the properties of an object

Starting with ECMAScript 5, there are three native ways to list/traverse object properties:

- for...in loops
  This method traverses all enumerable properties of an object.

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

  for (let key in user) {
      alert( key );  // name, age, isAdmin
      alert( user[key] ); // John, 30, true
  }
  ```

- Object.keys(o)

This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object o.

- Object.getOwnPropertyNames(o)

  This method returns an array containing all own properties' names (enumerable or not) of an object o.

Copying by reference

One of the fundamental differences of objects vs primitives is that they are stored and copied “by reference”.

Primitive values: strings, numbers, booleans – are assigned/copied “as a whole value”.

For instance:

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

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

Objects are not like that.

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

```
let user = { name: "John" };
let admin = user; // copy the reference
admin.name = 'Pete'; // changed by the "admin" reference
alert(user.name);
```

Comparison by reference

The equality `==` and strict equality `===` operators for objects work exactly the same.

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

For instance, if two variables reference the same object, 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 both are empty:

```
let a = {};
let b = {}; // two independent objects
alert( a == b ); // false
```

Const object

An object declared as const *can* be changed.

For instance:

```
const user = {
    name: "John"
};
user.age = 25; // (*)
alert(user.age); // 25
```

It might seem that the line (*) would cause an error, but no, there's totally no problem. That's because const fixes only value of user itself. And here user stores the reference to the same object all the time. The line (*) goes *inside* the object, it doesn't reassign user.

The const would give an error if we try to set user to something else, for instance:

```
const user = {
    name: "John"
};
// Error (can't reassign user)
```

```
user = {
    name: "Pete"
};
```

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? Create an independent copy, a clone?

That's also doable, but a little bit more difficult, because there's no built-in method for that in JavaScript. Actually, that's rarely needed. Copying by reference is good most of the time.

But if we really want that, then we need to create a new object and replicate the structure of the existing one by iterating over its properties and copying them on the primitive level.

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

let clone = {}; // the new empty object
// let's copy all user properties into it
for (let key in user) {
    clone[key] = user[key];
}

// now clone is a fully independent clone
clone.name = "Pete"; // changed the data in it
alert( user.name ); // still John in the original object
```

Also we can use the method `Object.assign` for that.

The syntax is:

`Object.assign(dest, [src1, src2, src3...])`

- Arguments dest, and src1, ..., srcN (can be as many as needed) are objects.
- It copies the properties of all objects src1, ..., srcN into dest. In other words, properties of all arguments starting from the 2nd are copied into the 1st. Then it returns dest.

For instance, we can use it to merge several objects into one:

```
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 }
```

If the receiving object (user) already has the same named property, it will be overwritten:

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

We also can use Object.assign to replace the loop for simple cloning:

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

It copies all properties of user into the empty object and returns it. Actually, the same as the loop, but shorter.

Object methods

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

Of course, we could use a pre-declared function as a method, like this:

```
let user = {
    // ...
};

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

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

Method shorthand

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

```
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
```

Here during the execution of user.sayHi(), the value of this will be user.

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

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

```
};
```

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

That's demonstrated below:

```
let user = {
    name: "John",
    age: 30,
    sayHi() {
        alert( user.name ); // leads to an error
    }
};
let admin = user;
admin.sayHi(); // works..
user = null; // overwrite to make things obvious
admin.sayHi(); // Whoops! inside sayHi(), the old name is used! Error!
```

Object.keys, values, entries

For plain objects, the following methods are available:

- Object.keys(obj) – returns an array of keys.
- Object.values(obj) – returns an array of values.
- Object.entries(obj) – returns an array of [key, value] pairs.

For instance:

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

- Object.keys(user) = ["name", "age"]
- Object.values(user) = ["John", 30]
- Object.entries(user) = [["name","John"], ["age",30]]

Here's an example of using Object.values to loop over property values:

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

Date object:

JavaScript does not have a date data type. However, you can use the Date object and its methods to work with dates and times in your applications. The Date object has a large number of methods for setting, getting, and manipulating dates. It does not have any properties.

Let's meet a new built-in object: Date. It stores the date, time and provides methods for date/time management.

Creation

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

Without arguments – create a Date object for the current date and time:

```
let now = new Date();
alert( now ); // Sat Nov 09 2019 18:02:13 GMT+0530 (India Standard Time)
```

new Date(milliseconds)

Create a Date object with the time equal to number of milliseconds (1/1000 of a second) passed after the Jan 1st of 1970 UTC+0.

```
// 0 means 01.01.1970 UTC+0
```

```
let Jan01_1970 = new Date(0);
alert( Jan01_1970 ); // Thu Jan 01 1970 05:30:00 GMT+0530 (India Standard Time)

// now add 24 hours, get 02.01.1970 UTC+0
let Jan02_1970 = new Date(24 * 3600 * 1000);
alert( Jan02_1970 ); //Fri Jan 02 1970 05:30:00 GMT+0530 (India Standard Time)
```

An integer number representing the number of milliseconds that has passed since the beginning of 1970 is called a *timestamp*.

It's a lightweight numeric representation of a date. We can always create a date from a timestamp using new Date(timestamp) and convert the existing Date object to a timestamp using the date.getTime() method (see below).

Dates before 01.01.1970 have negative timestamps, e.g.:

```
// 31 Dec 1969
let Dec31_1969 = new Date(-24 * 3600 * 1000);
alert( Dec31_1969 );
```

new Date(datestring)
If there is a single argument, and it's a string, then it is parsed automatically. The algorithm is the same as `Date.parse` uses, we'll cover it later.

```
let date = new Date("2017-01-26");
alert(date); //Thu Jan 26 2017 05:30:00 GMT+0530 (India Standard Time) {}
```

new Date(year, month, date, hours, minutes, seconds, ms)

Create the date with the given components in the local time zone. Only the first two arguments are necessary.

- The `year` must have 4 digits: `2013` is okay, `98` is not.
- The `month` count starts with `0` (Jan), up to `11` (Dec).
- The `date` parameter is actually the day of month, if absent then `1` is assumed.

- If hours/minutes/seconds/ms is absent, they are assumed to be equal 0.

For instance:

```
new Date(2011, 0, 1, 0, 0, 0, 0); // 1 Jan 2011, 00:00:00
new Date(2011, 0, 1); // Sat Jan 01 2011 00:00:00 GMT+0530 (India Standard Time)
```

Access date components

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

getFullYear() Get the year (4 digits)

getMonth() Get the month, **from 0 to 11**.

getDate() Get the day of month, from 1 to 31, the name of the method does look a little bit strange.

getHours(), getMinutes(), getSeconds(), getMilliseconds()

Additionally, we can get a day of week:

getDay()

Get the day of week, from 0 (Sunday) to 6 (Saturday). The first day is always Sunday, in some countries that's not so, but can't be changed.

All the methods above return the components relative to the local time zone.

There are also their UTC-counterparts, that return day, month, year and so on for the time zone UTC+0: getUTCFullYear(), getUTCMonth(), getUTCDay(). Just insert the "UTC" right after "get".

If your local time zone is shifted relative to UTC, then the code below shows different hours:

```
let date = new Date(); // current date
alert( date.getHours() ); // the hour in your current time zone
alert( date.getUTCHours() ); // the hour in UTC+0 time zone (London time without daylight savings)
```

Besides the given methods, there are two special ones that do not have a UTC-variant:

getTime()

Returns the timestamp for the date – a number of milliseconds passed from the January 1st of 1970 UTC+0.

getTimezoneOffset()

Returns the difference between the local time zone and UTC, in minutes:

```
// if you are in timezone UTC-1, outputs 60
// if you are in timezone UTC+3, outputs -180
alert( new Date().getTimezoneOffset() );
```

Setting date components

The following methods allow to set date/time components:

- setFullYear(year, [month], [date])
- setMonth(month, [date])
- setDate(date)
- setHours(hour, [min], [sec], [ms])
- setMinutes(min, [sec], [ms])
- setSeconds(sec, [ms])
- setMilliseconds(ms)
- setTime(milliseconds) (sets the whole date by milliseconds since 01.01.1970 UTC)

Every one of them except setTime() has a UTC-variant, for instance: setUTCHours().

As we can see, some methods can set multiple components at once, for example setHours. The components that are not mentioned are not modified.

For instance:

```
let today = new Date();
```

```
today.setHours(0);
alert(today); // still today, but the hour is changed to 0
today.setHours(0, 0, 0, 0);
alert(today); // still today, now 00:00:00 sharp.
```

Date to number, date diff

When a Date object is converted to number, it becomes the timestamp same as date.getTime():

```
let date = new Date();
alert(+date); // the number of milliseconds, same as date.getTime()
```

The important side effect: dates can be subtracted, the result is their difference in ms.

That can be used for time measurements:

```
let start = new Date(); // start measuring time
// do the job
for (let i = 0; i < 100000; i++) {
  let doSomething = i * i * i;
}
let end = new Date(); // end measuring time
alert( `The loop took ${end - start} ms` );
```

Date.now()

If we only want to measure time, we don’t need the Date object.

There’s a special method Date.now() that returns the current timestamp.

It is semantically equivalent to new Date().getTime(), but it doesn’t create an intermediate Date object. So it’s faster and doesn’t put pressure on garbage collection.

```
let start = Date.now(); // milliseconds count from 1 Jan 1970
```

Date.parse from a string

The method Date.parse(str) can read a date from a string.

The string format should be: YYYY-MM-DDTHH:mm:ss.sssZ, where:

- YYYY-MM-DD – is the date: year-month-day.
- The character "T" is used as the delimiter.
- HH:mm:ss.sss – is the time: hours, minutes, seconds and milliseconds.
- The optional 'Z' part denotes the time zone in the format +-hh:mm. A single letter Z that would mean UTC+0.

Shorter variants are also possible, like YYYY-MM-DD or YYYY-MM or even YYYY.

For instance:

```
let ms = Date.parse('2012-01-26T13:51:50.417-07:00');
alert(ms); // 1327611110417  (timestamp)
```

We can instantly create a new Date object from the timestamp:

```
let date = new Date( Date.parse('2012-01-26T13:51:50.417-07:00') );
alert(date);
```

For more , https://developer.mozilla.org/bm/docs/Web/JavaScript/Guide/Numbers_and_dates

---

JSON methods, toJSON

Let’s say we have a complex object, and we’d like to convert it into a string, to send it over a network, or just to output it for logging purposes.

We could implement the conversion like this:

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

```
    }
};
alert(user); // {name: "John", age: 30}
```

Luckily, there’s no need to write the code to handle all this. The task has been solved already.

JSON.stringify

The JSON (JavaScript Object Notation) is a general format to represent values and objects.

JavaScript provides methods:

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

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

```
let student = {
    name: 'John',
    age: 30,
    isAdmin: false,
    courses: ['html', 'css', 'js'],
    wife: null
};
let json = JSON.stringify(student);
alert(typeof json); // we've got a string!
alert(json);
/* JSON-encoded object:
{
    "name": "John",
    "age": 30,
    "isAdmin": false,
    "courses": ["html", "css", "js"],
    "wife": null
}
```

*/

The method JSON.stringify(student) takes the object and converts it into a string.
The resulting json string is called a *JSON-encoded* or *serialized* or *stringified* or *marshalled* object.

Please note that a JSON-encoded object has several important differences from the object literal:

- Strings use double quotes. No single quotes or backticks in JSON. So 'John' becomes "John".
- Object property names are double-quoted also. That's obligatory. So age:30 becomes "age":30.

JSON.stringify can be applied to primitives as well.

JSON supports following data types:

- Objects { ... }
- Arrays [...]
- Primitives:
    - strings,
    - numbers,
    - boolean values true/false,
    - null.

For instance:

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

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

For instance:

```
let meetup = {
```

```
    title: "Conference",
    room: {
        number: 23,
        participants: ["john", "ann"]
    }
};

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

The important limitation: there must be no circular references.

For instance:

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

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

Formatting: space

The third argument of JSON.stringify(value, replacer, space) is the number of spaces to use for pretty formatting.

Previously, all stringified objects had no indents and extra spaces. That's fine if we want to send an object over a network. The space argument is used exclusively for a nice output.

Here space = 2 tells JavaScript to show nested objects on multiple lines, with indentation of 2 spaces inside an object:

```
let user = {
    name: "John",
    age: 25,
    roles: {
        isAdmin: false,
        isEditor: true
    }
};
alert(JSON.stringify(user, null, 2));
/* two-space indents:
{
    "name": "John",
    "age": 25,
    "roles": {
        "isAdmin": false,
        "isEditor": true
    }
}
*/

/* for JSON.stringify(user, null, 4) the result would be more indented:

{
    "name": "John",
    "age": 25,
    "roles": {
```

```
        "isAdmin": false,
        "isEditor": true
    }
}
*/
```

The space parameter is used solely for logging and nice-output purposes.

JSON.parse

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

The syntax:

```
let value = JSON.parse(str, [reviver]);
```

str JSON-string to parse.

reviver

Optional function(key,value) that will be called for each (key, value) pair and can transform the value.

For instance:

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

Or for nested objects:

```
let userData = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';
let user = JSON.parse(userData);
alert( user.friends[1] ); // 1
```

The JSON may be as complex as necessary, objects and arrays can include other objects and arrays. But they must obey the same JSON format.

Here are typical mistakes in hand-written JSON (sometimes we have to write it for debugging purposes):

```
let json = `{
    name: "John",                     // mistake: property name without quotes
    "surname": 'Smith',               // mistake: single quotes in value (must be double)
    'isAdmin': false                  // mistake: single quotes in key (must be double)
    "birthday": new Date(2000, 2, 3), // mistake: no "new" is allowed, only bare values
    "friends": [0,1,2,3]              // here all fine
}`;
```

Besides, JSON does not support comments. Adding a comment to JSON makes it invalid.

Constructor, operator "new"

Constructor functions technically are regular functions. There are two conventions though:

1. They are named with capital letter first.
2. They should be executed only with "new" operator.

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

let user = new User("Jack");
alert(user.name); // Jack
alert(user.isAdmin); // false
```

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

1. A new empty object is created and assigned to this.
2. The function body executes. Usually it modifies this, adds new properties to it.
3. The value of this is returned

So let user = new User("Jack") gives the same result as:

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

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

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

Return from constructors

Usually, constructors do not have a return statement. Their task is to write all necessary stuff into this, and it automatically becomes the result.

But if there is a return statement, then the rule is simple:

- If return is called with an object, then the object is returned instead of this.
- If return is called with a primitive, it's ignored.

In other words, return with an object returns that object, in all other cases this is returned.

For instance, here return overrides this by returning an object:

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

alert( new BigUser().name );  // Godzilla, got that object
```

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

```
function SmallUser() {
```

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

Using constructor functions to create objects gives a great deal of flexibility. The constructor function may have parameters that define how to construct the object, and what to put in it.

Of course, we can add to this not only properties, but methods as well.

For instance, new User(name) below creates an object with the given name and the method sayHi:

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

Using the argument's object

The arguments of a function are maintained in an array-like object. Within a function, you can address the arguments passed to it as follows:

`arguments[i]` where i is the ordinal number of the argument, starting at zero. So, the first argument passed to a function would be `arguments[0]`. The total number of arguments is indicated by arguments.length.

Using the arguments object, you can call a function with more arguments than it is formally declared to accept. This is often useful if you don't know in advance how many arguments will

be passed to the function. You can use arguments.length to determine the number of arguments actually passed to the function, and then access each argument using the arguments object.

```
function myConcat(separator) {
    var result = ' ' ;   // initialize list
    var i;
    // iterate through arguments
    for (i = 1; i < arguments.length; i++) {
        result += arguments[i] + separator;
    }
    return result;
}
```

You can pass any number of arguments to this function, and it concatenates each argument into a string "list":

```
myConcat('; ', 'elephant', 'giraffe', 'lion', 'cheetah');
 // returns "elephant; giraffe; lion; cheetah; "
```

Arrow functions do not have "arguments"

If we access the arguments object from an arrow function, it takes them from the outer “normal” function.

Here’s an example:

```
function f() {
    let showArg = () => alert(arguments[0]);
    showArg();
}
f(1); // 1
```

Function parameter:

Starting with ECMAScript 2015, there are two new kinds of parameters: default parameters and rest parameters.

Default parameter:

In JavaScript, parameters of functions default to undefined. However, in some situations it might be useful to set a different default value. This is where default parameters can help.

In the past, the general strategy for setting defaults was to test parameter values in the body of the function and assign a value if they are undefined. If in the following example, no value is provided for b in the call, its value would be undefined when evaluating a*b and the call to multiply would have returned NaN. However, this is caught with the second line in this example:

```
function multiply(a, b) {
    b = typeof b !== 'undefined' ?  b : 1;
    return a * b;
}
multiply(5); // 5
```

With default parameters, the check in the function body is no longer necessary. Now, you can simply put 1 as the default value for b in the function head:

```
function multiply(a, b = 1) {
    return a * b;
}
multiply(5); // 5
multiply(5,6); // 30
```

Rest parameter:

The rest parameter syntax allows us to represent an indefinite number of arguments as an array. In the example, we use the rest parameters to collect arguments from the second one to the end.

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

```
    return a + b;
}
alert( sum(1, 2, 3, 4, 5) );
```

There will be no error because of “excessive” arguments. But of course in the result only the first two will be counted.

The rest of the parameters can be included in the function definition by using three dots ... followed by the name of the array that will contain them. The dots literally mean “gather the remaining parameters into an array”.

For instance, 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
```

We can choose to get the first parameters as variables, and gather only the rest.

Here the first two arguments go into variables and the rest go into titles array:

```
function showName(firstName, lastName, ...titles) {
    alert( firstName + ' ' + lastName ); // Julius Caesar
    // the rest go into titles array
    // i.e. titles = ["Consul", "Imperator"]
    alert( titles[0] ); // Consul
    alert( titles[1] ); // Imperator
    alert( titles.length ); // 2
}
showName("Julius", "Caesar", "Consul", "Imperator");
```

The rest parameters must be at the end

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

```
function f(arg1, ...rest, arg2) { // arg2 after ...rest ?!
}
```

The ...rest must always be last.

Spread operator

We've just seen how to get an array from the list of parameters. But sometimes we need to do exactly the reverse.

For instance, there's a built-in function Math.max that returns the greatest number from a list:

```
alert( Math.max(3, 5, 1) ); // 5
```

Now let's say we have an array [3, 5, 1]. How do we call Math.max with it?

Passing it "as is" won't work, because Math.max expects a list of numeric arguments, not a single array:

```
let arr = [3, 5, 1];
alert( Math.max(arr) ); // NaN
```

And surely we can't manually list items in the code Math.max(arr[0], arr[1], arr[2]), because we may be unsure how many there are. As our script executes, there could be a lot, or there could be none. And that would get ugly.

Spread operator to the rescue! It looks similar to rest parameters, also using ..., but does quite the opposite.

When ...arr is used in the function call, it "expands" an iterable object arr into the list of arguments.

For Math.max:

```
let arr = [3, 5, 1];
alert( Math.max(...arr) ); // 5 (spread turns array into a list of arguments)
```

We also can pass multiple iterables this way:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];
alert( Math.max(...arr1, ...arr2) ); // 8
```

We can even combine the spread operator with normal values:

```
let arr1 = [1, -2, 3, 4];
let arr2 = [8, 3, -8, 1];
alert( Math.max(1, ...arr1, 2, ...arr2, 25) ); // 25
```

Also, the spread operator can be used to merge arrays:

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];
let merged = [0, ...arr, 2, ...arr2];
alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

In the examples above we used an array to demonstrate the spread operator, but any iterable will do.
For instance, here we use the spread operator to turn the string into array of characters:

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

Copy an array

```
var arr = [1, 2, 3];
var arr2 = [...arr]; // like arr.slice()
```

A better way to concatenate arrays

Array.concat is often used to concatenate an array to the end of an existing array. Without spread syntax this is done as:

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1 = arr1.concat(arr2); // Append all items from arr2 onto arr1
```

With spread syntax this becomes:

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1 = [...arr1, ...arr2]; // arr1 is now [0, 1, 2, 3, 4, 5]
```

Spread in Object literals

```
var obj1 = { foo: 'bar', x: 42 };
var obj2 = { foo: 'baz', y: 13 };
var clonedObj = { ...obj1 }; // Object { foo: "bar", x: 42 }
var mergedObj = { ...obj1, ...obj2 }; // Object { foo: "baz", x: 42, y: 13 }
```

Destructuring assignment

The two most used data structures in JavaScript are `Object` and `Array`.

Objects allow us to create a single entity that stores data items by key, and arrays allow us to gather data items into an ordered collection.

But when we pass those to a function, it may need not an object/array as a whole, but rather individual pieces.

Destructuring assignment is a special syntax that allows us to “unpack” arrays or objects into a bunch of variables, as sometimes that’s more convenient. Destructuring also works great with complex functions that have a lot of parameters, default values, and so on.

The destructuring assignment syntax is a JavaScript expression that makes it possible **to unpack values from arrays, or properties from objects, into distinct variables**.

Array destructuring

An example of how the array is destructured into variables:

```
// we have an array with the name and surname
let arr = ["Ilya", "Kantor"]

// destructuring assignment
// sets firstName = arr[0]
// and surname = arr[1]

let [firstName, surname] = arr;
alert(firstName); // Ilya
alert(surname);  // Kantor
```

Now we can work with variables instead of array members.

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

```
let [firstName, surname] = "Ilya Kantor".split(' ');
```

“Destructuring” does not mean “destructive”.

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

It’s just a shorter way to write:

```
// let [firstName, surname] = arr;
let firstName = arr[0];
let surname = arr[1];
```

Ignore elements using commas

Unwanted elements of the array can also be thrown away via an extra comma:

```
// second element is not needed
let [firstName, , title] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
alert( title ); // Consul
```

In the code above, the second element of the array is skipped, the third one is assigned to title, and the rest of the array items is also skipped (as there are no variables for them).

Works with any iterable on the right-side

Actually, we can use it with any iterable, not only arrays:

```
let [a, b, c] = "abc"; // ["a", "b", "c"]
let [one, two, three] = new Set([1, 2, 3]);
```

Assign to anything at the left-side

We can use any “assignables” at the left side.

For instance, an object property:

```
let user = {};
[user.name, user.surname] = "Ilya Kantor".split(' ');
alert(user.name); // Ilya
```

Looping with .entries()

In the previous chapter we saw the Object.entries(obj) method.

We can use it with destructuring to loop over keys-and-values of an object:

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

// loop over keys-and-values
for (let [key, value] of Object.entries(user)) {
    alert(`${key}:${value}`); // name:John, then age:30
}
```

And the same for a map:

```
let user = new Map();
user.set("name", "John");
user.set("age", "30");

for (let [key, value] of user) {
    alert(`${key}:${value}`); // name:John, then age:30
}
```

The rest '…'

If we want not just to get first values, but also to gather all that follows – we can add one more parameter that gets "the rest" using three dots "...":

```
let [name1, name2, ...rest] = ["Julius", "Caesar", "Consul", "of the Roman Republic"];
alert(name1); // Julius
alert(name2); // Caesar
// Note that type of `rest` is Array.
alert(rest[0]); // Consul
alert(rest[1]); // of the Roman Republic
alert(rest.length); // 2
```

The value of rest is the array of the remaining array elements. We can use any other variable name in place of rest, just make sure it has three dots before it and goes last in the destructuring assignment.

Default values

If there are fewer values in the array than variables in the assignment, there will be no error. Absent values are considered undefined:

```
let [firstName, surname] = [];
alert(firstName); // undefined
alert(surname); // undefined
```

If we want a “default” value to replace the missing one, we can provide it using =:

```
// default values
let [name = "Guest", surname = "Anonymous"] = ["Julius"];
alert(name);    // Julius (from array)
alert(surname); // Anonymous (default used)
```

Object destructuring

The destructuring assignment also works with objects.

The basic syntax is:
`let {var1, var2} = {var1:…, var2:…}`

We have an existing object at the right side, that we want to split into variables. The left side contains a “pattern” for corresponding properties. In the simple case, that’s a list of variable names in {...}.

For instance:

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};

let {title, width, height} = options;
alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

Properties options.title, options.width and options.height are assigned to the corresponding variables. The order does not matter. This works too:

```
// changed the order in let {...}
let {height, width, title} = { title: "Menu", height: 200, width: 100 }
```

If we want to assign a property to a variable with another name, for instance, options.width to go into the variable named w, then we can set it using a colon:

```
let options = {
    title: "Menu",
    width: 100,
    height: 200
};
// { sourceProperty: targetVariable }
let {width: w, height: h, title} = options;
// width -> w
// height -> h
// title -> title
alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

The colon shows “what : goes where”. In the example above the property width goes to w, property height goes to h, and title is assigned to the same name.

For potentially missing properties we can set default values using "=", like this:

```
let options = {
    title: "Menu"
};
let {width = 100, height = 200, title} = options;
alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
```

We also can combine both the colon and equality:

```
let options = {
    title: "Menu"
```

```
};
let {width: w = 100, height: h = 200, title} = options;
alert(title);  // Menu
alert(w);      // 100
alert(h);      // 200
```

What if the object has more properties than we have variables? Can we take some and then assign the “rest” somewhere?

```
let options = {
    title: "Menu",
    height: 200,
    width: 100
};
let {title, ...rest} = options;
// now title="Menu", rest={height: 200, width: 100}
alert(rest.height);  // 200
alert(rest.width);   // 100
```

Nested destructuring

If an object or an array contain other nested objects and arrays, we can use more complex left-side patterns to extract deeper portions.

In the code below options has another object in the property size and an array in the property items. The pattern at the left side of the assignment has the same structure to extract values from them:

```
let options = {
    size: {
        width: 100,
        height: 200
    },
    items: ["Cake", "Donut"],
```

```
    extra: true
};

// destructuring assignment split in multiple lines for clarity
let {
    size: { // put size here
        width,
        height
    },
    items: [item1, item2], // assign items here
    title = "Menu" // not present in the object (default value is used)
} = options;

alert(title);  // Menu
alert(width);  // 100
alert(height); // 200
alert(item1);  // Cake
alert(item2);  // Donut
```

Smart function parameters

There are times when a function has many parameters, most of which are optional. That's especially true for user interfaces. Imagine a function that creates a menu. It may have a width, a height, a title, items list and so on.

Here's a bad way to write such a function:

```
function showMenu(title = "Untitled", width = 200, height = 100, items = []) {
    // ...
}
```

In real-life, the problem is how to remember the order of arguments. Usually IDEs try to help us, especially if the code is well-documented, but still… Another problem is how to call a function when most parameters are ok by default.

Like this?

```
// undefined where default values are fine
showMenu("My Menu", undefined, undefined, ["Item1", "Item2"])
```

That’s ugly. And becomes unreadable when we deal with more parameters.

Destructuring comes to the rescue!

We can pass parameters as an object, and the function immediately destructurizes them into variables:

```
// we pass object to function
let options = {
    title: "My menu",
    items: ["Item1", "Item2"]
};

// ...and it immediately expands it to variables
function showMenu({title = "Untitled", width = 200, height = 100, items = []}) {
    // title, items – taken from options,
    // width, height – defaults used
    alert( `${title} ${width} ${height}` ); // My Menu 200 100
    alert( items ); // Item1, Item2
}
showMenu(options);
```

Please note that such destructuring assumes that showMenu() does have an argument. If we want all values by default, then we should specify an empty object:

```
showMenu({}); // ok, all values are default
showMenu(); // this would give an error
```

We can fix this by making {} the default value for the whole object of parameters:

```
function showMenu({ title = "Menu", width = 100, height = 200 } = {}) {
```

```
  alert( `${title} ${width} ${height}` );
}
showMenu(); // Menu 100 200
```

Arrays

An array is a single object that contains multiple values enclosed in square brackets and separated by commas.

Creating an array:
The following statements create equivalent arrays:

```
var arr = new Array(element0, element1, ..., elementN);
var arr = Array(element0, element1, ..., elementN);
var arr = [element0, element1, ..., elementN];
```

The bracket syntax is called an "array literal" or "array initializer." It's shorter than other forms of array creation, and so is generally preferred. See Array literals for details.

```
let fruits = ["Apple", "Orange", "Plum"];
alert( fruits[0] ); // Apple
alert( fruits[1] ); // Orange
```

To create an array with non-zero length, but without any items, either of the following can be used:

```
var arr = new Array(arrayLength);
var arr = Array(arrayLength);

// This has exactly the same effect
var arr = [];
arr.length = arrayLength;
```

If you wish to initialize an array with a single element, and the element happens to be a Number, you must use the bracket syntax. When a single Number value is passed to the Array() constructor or function, it is interpreted as an arrayLength, not as a single element.

```
var arr = [42];             // Creates an array with only one element: 42
var arr = Array(42);     // Creates an array with no elements & arr.length set to 42;
                                 // this isequivalent to:
var arr = [];
arr.length = 42;
```

Calling Array(N) results in a RangeError, if N is a non-whole number whose fractional portion is non-zero. The following example illustrates this behavior.

```
var arr = Array(9.3);  // RangeError: Invalid array length
```

Populating an array:

```
var emp = [];
emp[0] = 'Casey Jones';
emp[1] = 'Phil Lesh';
emp[2] = 'August West';
```

You can also populate an array when you create it:

```
var myArray = new Array('Hello', myVar, 3.14159);
var myArray = ['Mango', 'Apple', 'Orange'];
```

Understanding length:

The length property is special; it always returns the index of the last element plus one (in the following example, Dusty is indexed at 30, so cats.length returns 30 + 1).

```
var cats = [];
cats[30] = ['Dusty'];
console.log(cats.length); // 31 (creates empty values in between)

var cats = ['Dusty', 'Misty', 'Twiggy'];
console.log(cats.length); // 3

cats.length = 2;
console.log(cats); // logs "Dusty, Misty" - Twiggy has been removed

cats.length = 0;
console.log(cats); // logs []; the cats array is empty

cats.length = 3;
console.log(cats); // logs [ <3 empty items> ]
```

Array methods:

Methods that work with the end of the array:

pop

Extracts the last element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.pop() ); // remove "Pear" and alert it
alert( fruits ); // Apple, Orange
```

push

Append the element to the end of the array:

```
let fruits = ["Apple", "Orange"];
fruits.push("Pear");
alert( fruits ); // Apple, Orange, Pear
```

Methods that work with the beginning of the array:

shift

Extracts the first element of the array and returns it:

```
let fruits = ["Apple", "Orange", "Pear"];
alert( fruits.shift() ); // remove Apple and alert it
alert( fruits ); // Orange, Pear
```

unshift

Add the element to the beginning of the array:

```
let fruits = ["Orange", "Pear"];
fruits.unshift('Apple');
alert( fruits ); // Apple, Orange, Pear
```

splice

How to delete an element from the array?

The arrays are objects, so we can try to use delete:

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

The element was removed, but the array still has 3 elements, we can see that arr.length == 3.

That's natural, because delete obj.key removes a value by the key. It's all it does. Fine for objects. But for arrays we usually want the rest of elements to shift and occupy the freed place. We expect to have a shorter array now.

So, special methods should be used.

The arr.splice(start) method is a swiss army knife for arrays. It can do everything: insert, remove and replace elements.

Let’s start with the deletion:

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

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

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

Here we can see that splice returns the array of removed elements:

```
let arr = ["I", "study", "JavaScript", "right", "now"];
let removed = arr.splice(0, 2); // remove 2 first elements
alert( removed ); // "I", "study" <-- array of removed elements
```

The splice method is also able to insert the elements without any removals. For that we need to set deleteCount to 0:

```
let arr = ["I", "study", "JavaScript"];
// from index 2, delete 0
// then insert "complex" and "language"
arr.splice(2, 0, "complex", "language");
alert( arr ); // "I", "study", "complex", "language", "JavaScript"
```

slice

```
arr.slice([start], [end])
```

It returns a new array copying to it all items from index start to end (not including end). Both start and end can be negative, in that case position from array end is assumed.

It's similar to a string method str.slice, but instead of substrings it makes subarrays.

For instance:

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

We can also call it without arguments: arr.slice() creates a copy of arr. That's often used to obtain a copy for further transformations that should not affect the original array.

concat

The method arr.concat creates a new array that includes values from other arrays and additional items.

```
arr.concat(arg1, arg2...)
```

It accepts any number of arguments – either arrays or values.

The result is a new array containing items from arr, then arg1, arg2 etc.

If an argument argN is an array, then all its elements are copied. Otherwise, the argument itself is copied.

For instance:

```
let arr = [1, 2];
// create an array from: arr and [3,4]
alert( arr.concat([3, 4]) ); // 1,2,3,4
// create an array from: arr and [3,4] and [5,6]
alert( arr.concat([3, 4], [5, 6]) ); // 1,2,3,4,5,6

// create an array from: arr and [3,4], then add values 5 and 6
```

```
alert( arr.concat([3, 4], 5, 6) ); // 1,2,3,4,5,6
```

Normally, it only copies elements from arrays. Other objects, even if they look like arrays, are added as a whole:

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

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

Iterate: forEach

The arr.forEach method allows to run a function for every element of the array.

The syntax:

```
arr.forEach(function(item, index, array) {
    // ... do something with item
});
```

For instance, this shows each element of the array:

```
// for each element call alert
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
```

And this code is more elaborate about their positions in the target array:

```
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {
    alert(`${item} is at index ${index} in ${array}`);
});
```

The result of the function (if it returns any) is thrown away and ignored.

Searching in array

Now let’s cover methods that search in an array.

indexOf/lastIndexOf and includes

The methods arr.indexOf, arr.lastIndexOf and arr.includes have the same syntax and do essentially the same as their string counterparts, but operate on items instead of characters:

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

For instance:

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

Note that the methods use `===` comparison. So, if we look for false, it finds exactly false and not the zero.

If we want to check for inclusion, and don’t want to know the exact index, then arr.includes is preferred.

Also, a very minor difference of includes is that it correctly handles NaN, unlike indexOf/lastIndexOf:

```
const arr = [NaN];
alert( arr.indexOf(NaN) ); // -1 (should be 0, but === equality doesn't work for NaN)
alert( arr.includes(NaN) );// true (correct)
```

find and findIndex

Imagine we have an array of objects. How do we find an object with the specific condition?

Here the arr.find(fn) method comes in handy.

The syntax is:

```
let result = arr.find(function(item, index, array) {
    // if true is returned, item is returned and iteration is stopped
    // for falsy scenario returns undefined
});
```

The function is called for elements of the array, one after another:

- item is the element.
- index is its index.
- array is the array itself.

If it returns true, the search is stopped, the item is returned. If nothing found, undefined is returned.

For example, we have an array of users, each with the fields id and name. Let's find the one with id == 1:

```
let users = [
    {id: 1, name: "John"},
    {id: 2, name: "Pete"},
    {id: 3, name: "Mary"}
];

let user = users.find(item => item.id == 1);
alert(user.name); // John
```

In real life arrays of objects is a common thing, so the find method is very useful.

Note that in the example we provide to find the function item => item.id == 1 with one argument. That's typical, other arguments of this function are rarely used.

The arr.findIndex method is essentially the same, but it returns the index where the element was found instead of the element itself and -1 is returned when nothing is found.

filter

Another useful iterator method is .filter(). Like .map(), .filter() returns a new array. However, .filter() returns certain elements from the original array that evaluate to truthy based on conditions written in the block of the method.

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

If there may be many, we can use arr.filter(fn).

The syntax is similar to find, but filter returns an array of all matching elements:

```
let results = arr.filter(function(item, index, array) {
    // if true item is pushed to results and the iteration continues
    // returns empty array if nothing found
});
```

For instance:

```
let users = [
    {id: 1, name: "John"},
    {id: 2, name: "Pete"},
    {id: 3, name: "Mary"}
];

// returns array of the first two users

let someUsers = users.filter(item => item.id < 3);
alert(someUsers.length); // 2
```

Transform an array

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

map

The arr.map method is one of the most useful and often used.

It calls the function for each element of the array and returns the array of results.

The syntax is:

```
let result = arr.map(function(item, index, array) {
    // returns the new value instead of item
});
```

For instance, here we transform each element into its length:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

reverse

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

For instance:

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

It also returns the array arr after the reversal.

split and join

Here's the situation from real life. We are writing a messaging app, and the person enters the comma-delimited list of receivers: John, Pete, Mary. But for us an array of names would be much more comfortable than a single string. How to get it?

The str.split(delim) method does exactly that. It splits the string into an array by the given delimiter delim.

In the example below, we split by a comma followed by space:

```
let names = 'Bilbo, Gandalf, Nazgul';
```

```
let arr = names.split(', ');
for (let name of arr) {
    alert( `A message to ${name}.` ); // A message to Bilbo  (and other names)
}
```

The split method has an optional second numeric argument – a limit on the array length. If it is provided, then the extra elements are ignored. In practice it is rarely used though:

```
let arr = 'Bilbo, Gandalf, Nazgul, Saruman'.split(', ', 2);
alert(arr); // Bilbo, Gandalf
```

Split into letters

The call to split(s) with an empty s would split the string into an array of letters:

```
let str = "test";
alert( str.split('') ); // t,e,s,t
```

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

For instance:

```
let arr = ['Bilbo', 'Gandalf', 'Nazgul'];
let str = arr.join(';'); // glue the array into a string using ;
alert( str ); // Bilbo;Gandalf;Nazgul
```

Array.isArray

Arrays do not form a separate language type. They are based on objects.

So typeof does not help to distinguish a plain object from an array:

```
alert(typeof {}); // object
alert(typeof []); // same
```

But arrays are used so often that there's a special method for that: Array.isArray(value). It returns true if the value is an array, and false otherwise.

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

Internals

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

Remember, there are only 7 basic types in JavaScript. Array is an object and thus behaves like an object.

For instance, it is copied by reference:

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

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

Performance

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

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

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

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

The `shift` operation must do 3 things:

1. Remove the element with the index `0`.
2. Move all elements to the left, renumber them from the index `1` to `0`, from `2` to `1` and so on.
3. Update the `length` property.

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

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

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

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

Loops

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

```
let arr = ["Apple", "Orange", "Pear"];
for (let i = 0; i < arr.length; i++) {
    alert( arr[i] );
}
```

But for arrays there is another form of loop, `for..of`:

```
let fruits = ["Apple", "Orange", "Plum"];
// iterates over array elements
for (let fruit of fruits) {
    alert( fruit );
```

```
}
```

The for..of doesn't give access to the number of the current element, just its value, but in most cases that's enough. And it's shorter.

Multidimensional arrays

Arrays can have items that are also arrays. We can use it for multidimensional arrays, for example to store matrices:

```
let matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];
alert( matrix[1][1] ); // 5, the central element
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Indexed_collections

---

Iterables:

An object is **iterable** if it defines its iteration behavior, such as what values are looped over in a for...of construct. Some built-in types, such as Array or Map, have a default iteration behavior, while other types (such as Object) do not.

Arrays and strings are most widely used built-in iterables.

For a string, for..of loops over its characters:

```
for (let char of "test") {
    // triggers 4 times: once for each character
    alert( char ); // t, then e, then s, then t
}
```

In order to be **iterable**, an object must implement the **@@iterator** method, meaning that the object (or one of the objects up its prototype chain) must have a property with a Symbol.iterator key.

Arrays, String, etc (and Typed Arrays) are iterables over their elements:

```
for (const x of ['a', 'b']) {
    console.log(x);
}
// Output:
// 'a'
// 'b'
```

Getting one by one value from a built-in array

```
var arr= [‘one’,’two’,’three’]
var itr = arr[Symbol.iterator]()
console.log( itr.next() ) // { value: ‘one’, done: false }
```

Iterators:

An object is considered *iterable* if it has a method whose key is the symbol Symbol.iterator that returns a so-called *iterator*. The iterator is an object that returns values via its method next(). We say: it *iterates over* the *items* (the content) of the iterable, one per method call.

An iterator is any object which implements the Iterator protocol by having a next() method which returns an object with two properties: value, the next value in the sequence; and done, which is true if the last value in the sequence has already been consumed.

The most common iterator in Javascript is the Array iterator, which simply returns each value in the associated array in sequence.

Purpose of user defined Iterators:

Imagine that you have this array —

```
const myFavouriteAuthors = [
    'Neal Stephenson',
    'Arthur Clarke',
    'Isaac Asimov',
    'Robert Heinlein'
];
```

At some point, you will want to get back all the individual values in the array for printing them on the screen, manipulating them, or for performing some action on them. If I ask you how would you do that? You'll say — *it's easy. I'll just loop over them using for, while, for-of or one of these looping methods.*

Now, imagine that instead of the previous array, you had a custom data structure to hold all your authors. Like this —

```
const myFavouriteAuthors = {
    allAuthors: {
        fiction: ['J.k.Rowling', 'Sujatha', 'Krishnan'  ],
        scienceFiction: ['Isaac','Author','Neal'],
        fantasy: ['J.k.Rowling','Terry Pretchatt', 'J.R.R.Towrien']
    }
}
```

Now, myFavouriteAuthors is an object which contains another object allAuthors. allAuthors contains three arrays with keys fiction, scienceFiction, and fantasy. Now, if I ask you to loop over myFavouriteAuthors to get all the authors, what would your approach be? You can go ahead and try some combination of loops to get all the data.
However if you do this,

```
for (let author of myFavouriteAuthors) {
    console.log(author)
}
```

```
// TypeError: {} is not iterable
```

You would get a `TypeError` saying that the object is not *iterable*. Let's see what iterables are and how we can make an object iterable. At the end of this article, you'll know how to use `for-of` loop on custom objects, and in this case, on `myFavouriteAuthors`.

Making objects iterable

So as we learnt in the previous section, we need to implement a method called Symbol.iterator.

```
const myFavouriteAuthors = {
    allAuthors: {
        fiction           : [ 'Agatha Christie', 'J. K. Rowling', 'Dr. Seuss'],
        scienceFiction : [ 'Neal Stephenson', 'Arthur Clarke', 'Isaac Asimov', 'Robert Heinlein' ],
        Fantasy           : [ 'J. R. R. Tolkien', 'J. K. Rowling', 'Terry Pratchett' ],
    },
    [Symbol.iterator] () {
        const genres = Object.values(this.allAuthors); // Get all the authors in an array
        let currentGenreIndex = 0;
        let currentAuthorIndex = 0;
        return {
            next() {
                const authors = genres[currentGenreIndex];
                const doNothaveMoreAuthors = !(currentAuthorIndex < authors.length);
                if (doNothaveMoreAuthors) {
                    // When that happens, we move the genre index to the next genre
                    currentGenreIndex++;
                    currentAuthorIndex = 0;
                }
                // if all genres are over, then we need tell the iterator that we
                const doNotHaveMoreGenres = !(currentGenreIndex < genres.length);
                if (doNotHaveMoreGenres) {
                    // Hence, we return done as true.
```

```
                return {
                    value: undefined,
                    done: true
                };
            }

            // if everything is correct, return the author from the
            return {
                value: genres[currentGenreIndex][currentAuthorIndex++],
                done: false
            }
        }
    };
  }
};

for (const author of myFavouriteAuthors) {
  console.log(author);
}
console.log(...myFavouriteAuthors)
```

Keyed collection:

Map:

ECMAScript 2015 introduces a new data structure to map values to values. A Map object is a simple key/value map and can iterate its elements in insertion order.

Methods and properties are:

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.

- map.get(key) – returns the value by the key, undefined if key doesn't exist in map.
- map.has(key) – returns true if the key exists, false otherwise.
- map.delete(key) – removes the value by the key.
- map.clear() – removes everything from the map.
- map.size – returns the current element count

```
map.set('1', 'str1');   // a string key
map.set(1, 'num1');     // a numeric key
map.set(true, 'bool1'); // a boolean key

// remember the regular Object? it would convert keys to string
// Map keeps the type, so these two are different:

alert( map.get(1)   ); // 'num1'
alert( map.get('1') ); // 'str1'
alert( map.size ); // 3
```

As we can see, unlike objects, keys are not converted to strings. Any type of key is possible.

Map can also use objects as keys.

For instance:

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

Chaining

Every map.set call returns the map itself, so we can "chain" the calls:

```
map.set('1', 'str1')
    .set(1, 'num1')
```

```
    .set(true, 'bool1');
```

Iteration over Map

For looping over a `map`, there are 3 methods:

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

For instance:

```
let recipeMap = new Map([
    ['cucumber', 500],
    ['tomatoes', 350],
    ['onion',    50]
]);

// iterate over keys (vegetables)
for (let vegetable of recipeMap.keys()) {
    alert(vegetable); // cucumber, tomatoes, onion
}

// iterate over values (amounts)
for (let amount of recipeMap.values()) {
    alert(amount); // 500, 350, 50
}


// iterate over [key, value] entries
for (let entry of recipeMap) { // the same as of recipeMap.entries()
    alert(entry); // cucumber,500 (and so on)
}
```

The insertion order is used

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

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

```
// runs the function for each (key, value) pair
recipeMap.forEach( (value, key, map) => {
    alert(`${key}: ${value}`); // cucumber: 500 etc
});
```

Object.entries: Map from Object

When a Map is created, we can pass an array (or another iterable) with key/value pairs for initialization, like this:

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

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

So we can create a map from an object like this:

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

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

Object.fromEntries: Object from Map

We've just seen how to create Map from a plain object with Object.entries(obj).

There's Object.fromEntries method that does the reverse: given an array of [key, value] pairs, it creates an object from them:

```
let prices = Object.fromEntries([
    ['banana', 1],
    ['orange', 2],
    ['meat', 4]
]);

// now prices = { banana: 1, orange: 2, meat: 4 }
alert(prices.orange); // 2
```

We can use Object.fromEntries to get a plain object from Map.

E.g. We store the data in a Map, but we need to pass it to a 3rd-party code that expects a plain object.

Here we go:

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

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

Traditionally, objects have been used to map strings to values. Objects allow you to set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. Map objects, however, have a few more advantages that make them better maps.

- The keys of an Object are Strings, where they can be of any value for a Map.
- You can get the size of a Map easily while you have to manually keep track of size for an Object.
- The iteration of maps is in insertion order of the elements.
- An Object has a prototype, so there are default keys in the map. (this can be bypassed using map = Object.create(null))

These three tips can help you to decide whether to use a Map or an Object:

- Use maps over objects when keys are unknown until run time, and when all keys are the same type and all values are the same type.
- Use maps if there is a need to store primitive values as keys because object treats each key as a string whether it's a number value, boolean value or any other primitive value.
- Use objects when there is logic that operates on individual elements.

Set:

A Set is a special type of collection – “set of values” (without keys), where each value may occur only once.

Its main methods are:

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

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

For example, we have visitors coming, and we'd like to remember everyone. But repeated visits should not lead to duplicates. A visitor must be "counted" only once.

Set is just the right thing for that:

```
let set = new Set();
let john = { name: "John" };
let pete = { name: "Pete" };
let mary = { name: "Mary" };

// visits, some users come multiple times
set.add(john);
set.add(pete);
set.add(mary);
set.add(john);
set.add(mary);

// set keeps only unique values
alert( set.size ); // 3
for (let user of set) {
    alert(user.name); // John (then Pete and Mary)
}
```

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

Iteration over Set

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

```
let set = new Set(["oranges", "apples", "bananas"]);
for (let value of set) alert(value);
```

```
// the same with forEach:
set.forEach((value, valueAgain, set) => {
    alert(value);
});
```

The same methods `Map` has for iterators are also supported:

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

Converting between Arrays & Sets

You can create an Array from a Set using Array.from or the spread operator. Also, the Set constructor accepts an Array to convert in the other direction. Note again that Set objects store unique values, so any duplicate elements from an Array are deleted when converting.

```
Array.from(mySet);
mySet2 = new Set([1, 2, 3, 4]);
```

Array & Set compared

Traditionally, a set of elements has been stored in arrays in JavaScript in a lot of situations. The new Set object, however, has some advantages:

- Check whether an element exists in a collection using `indexOf` for arrays is slow.
- Set objects let you delete elements by their value. With an array you would have to `splice` based on an element's index.
- The value `NaN` cannot be found with `indexOf` in an array.
- Set objects store unique values; you don't have to keep track of duplicates by yourself.

---

this:

A function's this keyword behaves a little differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode.

In most cases, the value of **this is determined by how a function is called.** It can't be set by assignment during execution, and it may be different each time the function is called. **ES5 introduced the bind method** to set the value of a function's this regardless of how it's called, and **ES2015 introduced arrow functions** which don't provide their own this binding (it retains the this value of the enclosing lexical context).

```
var test = {
    prop: 42,
    func: function() {
        return this.prop;
    },
};
console.log(test.func()); // output: 42
```

Global context:

In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true

a = 37;
console.log(window.a); // 37

this.b = "MDN";
```

```
console.log(window.b)  // "MDN"
console.log(b)         // "MDN"
```

Function context:

Inside a function, the value of this depends on how the function is called.

Since the following code is not in strict mode, and because the value of this is not set by the call, this will default to the global object, which is window in a browser.

```
function f1() {
    return this;
}

// In a browser:
f1() === window; // true

// In Node:
f1() === global; // true
```

In strict mode, however, the value of this remains at whatever it was set to when entering the execution context, so, in the following case, this will default to undefined**:**

```
function f2() {
    'use strict'; // see strict mode
    return this;
}
f2() === undefined; // true
```

So, in **strict mode,** if this was not defined by the execution context, it remains undefined.

bind vs call vs apply

Use .bind() when you want that function to later be called with a certain context, useful in events. Use .call() or .apply() when you want to call the function immediately, and modify the context.

An object can be passed as the first argument to call or apply and this will be bound to it.

```
var obj = {a: 'Custom'};
var a = 'Global';

function whatsThis() {
    return this.a;  // The value of this is dependent on how the function is called
}

whatsThis();          // 'Global'
whatsThis.call(obj);  // 'Custom'
whatsThis.apply(obj); // 'Custom'
```

Ex: 2

```
function add(c, d) {
    return this.a + this.b + c + d;
}
var o = {a: 1, b: 3};
add.call(o, 5, 7); // 16
add.apply(o, [10, 20]); // 34
```

The bind method:

ECMAScript 5 introduced Function.prototype.bind. Calling f.bind(someObject)creates a new function with the same body and scope as f, but where this occurs in the original function, in the new function it is permanently bound to the first argument of bind, regardless of how the function is being used.

```
function f() {
    return this.a;
}
```

```
var g = f.bind({a: 'azerty'});
console.log(g()); // azerty


var h = g.bind({a: 'yoo'}); // bind only works once!
console.log(h()); // azerty


var o = {a: 37, f: f, g: g, h: h};
console.log(o.a, o.f(), o.g(), o.h()); // 37,37, azerty, azerty
```

When a function is called as a method of an object, its this is set to the object the method is called on.

In the following example, when o.f() is invoked, inside the function this is bound to the o object.

```
var o = {
    prop: 37,
    f: function() {
        return this.prop;
    }
};
console.log(o.f()); // 37
```

Note that this behavior is not at all affected by how or where the function was defined. In the previous example, we defined the function inline as the f member during the definition of o. However, we could have just as easily defined the function first and later attached it to o.f. Doing so results in the same behavior:

```
var o = {prop: 37};
function independent() {
    return this.prop;
}
o.f = independent;
console.log(o.f()); // 37
```

```
o.b = {g: independent, prop: 42};
console.log(o.b.g()); // 42
```

As a constructor:

When a function is used as a constructor (with the new keyword), its this is bound to the new object being constructed.

```
function C() {
    this.a = 37;
}
var o = new C();
console.log(o.a); // 37

function C2() {
    this.a = 37;
    return {a: 38};
}
o = new C2();
console.log(o.a); // 38
```

Ref: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

---

Recursion and stack

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

Two ways of thinking

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

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

pow(2, 4) = 16

There are two ways to implement it.

1. Iterative thinking: the for loop:

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

alert( pow(2, 3) ); // 8
```

2. Recursive thinking: simplify the task and call self:

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
    return x * pow(x, n - 1);
  }
}
alert( pow(2, 3) ); // 8
```

Please note how the recursive variant is fundamentally different.

1. If n == 1, then everything is trivial. It is called *the base* of recursion, because it immediately produces the obvious result: pow(x, 1) equals x.
2. Otherwise, we can represent pow(x, n) as x * pow(x, n - 1). In maths, one would write $x^n = x * x^{n-1}$. This is called *a recursive step*: we transform the task into a simpler action

(multiplication by `x`) and a simpler call of the same task (`pow` with lower `n`). Next steps simplify it further and further until `n` reaches `1`.

We can also say that `pow` *recursively calls itself* till `n == 1`.

For example, to calculate `pow(2, 4)` the recursive variant does these steps:

1. `pow(2, 4) = 2 * pow(2, 3)`
2. `pow(2, 3) = 2 * pow(2, 2)`
3. `pow(2, 2) = 2 * pow(2, 1)`
4. `pow(2, 1) = 2`

So, the recursion reduces a function call to a simpler one, and then – to even more simpler, and so on, until the result becomes obvious.

Recursion is usually shorter

A recursive solution is usually shorter than an iterative one.

Here we can rewrite the same using the conditional operator `?` instead of `if` to make `pow(x, n)` more terse and still very readable:

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

The maximal number of nested calls (including the first one) is called *recursion depth*. In our case, it will be exactly `n`.

The maximal recursion depth is limited by JavaScript engine. We can rely on it being 10000, some engines allow more, but 100000 is probably out of limit for the majority of them. There are automatic optimizations that help alleviate this (“tail calls optimizations”), but they are not yet supported everywhere and work only in simple cases.

That limits the application of recursion, but it still remains very wide. There are many tasks where recursive way of thinking gives simpler code, easier to maintain.

The execution context and stack

Now let's examine how recursive calls work. For that we'll look under the hood of functions.

The information about the process of execution of a running function is stored in its ***execution context***.

The execution context is an internal data structure that contains details about the execution of a function: where the control flow is now, the current variables, the value of this (we don't use it here) and few other internal details.

One function call has exactly one execution context associated with it.

When a function makes a nested call, the following happens:

- The current function is paused.
- The execution context associated with it is remembered in a special data structure called *execution context stack*.
- The nested call executes.
- After it ends, the old execution context is retrieved from the stack, and the outer function is resumed from where it stopped.

Let's see what happens during the pow(2, 3) call.
pow(2, 3)
In the beginning of the call pow(2, 3) the execution context will store variables: x = 2, n = 3, the execution flow is at line 1 of the function.

We can sketch it as:

- **Context: { x: 2, n: 3, at line 1 } pow(2, 3)**

That's when the function starts to execute. The condition n == 1 is false, so the flow continues into the second branch of if:

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

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

The variables are same, but the line changes, so the context is now:

- **Context: { x: 2, n: 3, at line 5 } pow(2, 3)**

To calculate `x * pow(x, n - 1)`, we need to make a subcall of `pow` with new arguments `pow(2, 2)`.

To do a nested call, JavaScript remembers the current execution context in the *execution context stack*.

Here we call the same function `pow`, but it absolutely doesn't matter. The process is the same for all functions:

1. The current context is "remembered" on top of the stack.
2. The new context is created for the subcall.
3. When the subcall is finished – the previous context is popped from the stack, and its execution continues.

Here's the context stack when we entered the subcall `pow(2, 2)`:

- **Context: { x: 2, n: 2, at line 1 } pow(2, 2)**
- Context: { x: 2, n: 3, at line 5 } pow(2, 3)

The new current execution context is on top (and bold), and previous remembered contexts are below.

When we finish the subcall – it is easy to resume the previous context, because it keeps both variables and the exact place of the code where it stopped.

The process repeats: a new subcall is made at line `5`, now with arguments `x=2`, `n=1`.

A new execution context is created, the previous one is pushed on top of the stack:

- **Context: { x: 2, n: 1, at line 1 } pow(2, 1)**
- Context: { x: 2, n: 2, at line 5 } pow(2, 2)

- Context: { x: 2, n: 3, at line 5 } pow(2, 3)

There are 2 old contexts now and 1 currently running for `pow(2, 1)`.

During the execution of `pow(2, 1)`, unlike before, the condition `n == 1` is truthy, so the first branch of `if` works

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

There are no more nested calls, so the function finishes, returning `2`.

As the function finishes, its execution context is not needed anymore, so it's removed from the memory. The previous one is restored off the top of the stack:

- **Context: { x: 2, n: 2, at line 5 } pow(2, 2)**
- Context: { x: 2, n: 3, at line 5 } pow(2, 3)

The execution of `pow(2, 2)` is resumed. It has the result of the subcall `pow(2, 1)`, so it also can finish the evaluation of `x * pow(x, n - 1)`, returning `4`.

Then the previous context is restored:

- **Context: { x: 2, n: 3, at line 5 } pow(2, 3)**

When it finishes, we have a result of `pow(2, 3) = 8`.

The recursion depth in this case was: **3**.

As we can see from the illustrations above, recursion depth equals the maximal number of context in the stack.

Note the memory requirements. Contexts take memory. In our case, raising to the power of `n` actually requires the memory for `n` contexts, for all lower values of `n`.

A loop-based algorithm is more memory-saving:

```
function pow(x, n) {
    let result = 1;
    for (let i = 0; i < n; i++) {
        result *= x;
    }
    return result;
}
```

The iterative `pow` uses a single context changing `i` and `result` in the process. Its memory requirements are small, fixed and do not depend on `n`.

Any recursion can be rewritten as a loop. The loop variant usually can be made more effective.

Function object, NFE

As we already know, a function in JavaScript is a value. Every value in JavaScript has a type. What type is a function?

In JavaScript, functions are objects.

A good way to imagine functions is as callable “action objects”. We can not only call them, but also treat them as objects: add/remove properties, pass by reference etc.

The “name” property

Function objects contain some useable properties.

For instance, a function’s name is accessible as the “name” property:

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

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

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

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

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

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

Object methods have names too:

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

The "length" property

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

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

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

Here we can see that rest parameters are not counted.

Custom properties

We can also add properties of our own.

Here we add the `counter` property to track the total calls count:

```
function sayHi() {
    alert("Hi");
    // let's count how many times we run
    sayHi.counter++;
}
sayHi.counter = 0; // initial value
sayHi(); // Hi
sayHi(); // Hi
alert( `Called ${sayHi.counter} times` ); // Called 2 times
```

Named Function Expression

Named Function Expression, or NFE, is a term for Function Expressions that have a name.

For instance, let’s take an ordinary Function Expression:

```
let sayHi = function(who) {
    alert(`Hello, ${who}`);
};
```

And add a name to it:

```
let sayHi = function func(who) {
    alert(`Hello, ${who}`);
};
```

Did we achieve anything here? What's the purpose of that additional "func" name?

First let's note, that we still have a Function Expression. Adding the name "func" after function did not make it a Function Declaration, because it is still created as a part of an assignment expression.

Adding such a name also did not break anything.

The function is still available as sayHi():

```
let sayHi = function func(who) {
    alert(`Hello, ${who}`);
};
sayHi("John"); // Hello, John
```

There are two special things about the name func, that are the reasons for it:

1. It allows the function to reference itself internally.
2. It is not visible outside of the function.

For instance, the function sayHi below calls itself again with "Guest" if no who is provided:

```
let sayHi = function func(who) {
    if (who) {
        alert(`Hello, ${who}`);
    } else {
        func("Guest"); // use func to re-call itself
    }
};
sayHi(); // Hello, Guest
```

```
// But this won't work:
func(); // Error, func is not defined (not visible outside of the function)
```

Why do we use func? Maybe just use sayHi for the nested call?

Actually, in most cases we can:

```
let sayHi = function(who) {
    if (who) {
        alert(`Hello, ${who}`);
    } else {
        sayHi("Guest");
    }
};
```

The problem with that code is that sayHi may change in the outer code. If the function gets assigned to another variable instead, the code will start to give errors:

```
let sayHi = function(who) {
    if (who) {
        alert(`Hello, ${who}`);
    } else {
        sayHi("Guest"); // Error: sayHi is not a function
    }
};
let welcome = sayHi;
sayHi = null;
welcome(); // Error, the nested sayHi call doesn't work any more!
```

That happens because the function takes sayHi from its outer lexical environment. There’s no local sayHi, so the outer variable is used. And at the moment of the call that outer sayHi is null.

The optional name which we can put into the Function Expression is meant to solve exactly these kinds of problems.

Let’s use it to fix our code:

```
let sayHi = function func(who) {
    if (who) {
        alert(`Hello, ${who}`);
    } else {
        func("Guest"); // Now all fine
    }
};
let welcome = sayHi;
sayHi = null;
welcome(); // Hello, Guest (nested call works)
```

Now it works, because the name "func" is function-local.

Scheduling: setTimeout and setInterval

We may decide to execute a function not right now, but at a certain time later. That’s called “scheduling a call”.
There are two methods for it:

- `setTimeout` allows us to run a function once after the interval of time.
- `setInterval` allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.

These methods are not a part of JavaScript specification. But most environments have the internal scheduler and provide these methods. In particular, they are supported in all browsers and Node.js.

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 (not supported in IE9-)

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
```

Canceling with clearTimeout

A call to setTimeout returns a "timer identifier" timerId that we can use to cancel the execution.

The syntax to cancel:

```
let timerId = setTimeout(...);
clearTimeout(timerId);
```

In the code below, we schedule the function and then cancel it (changed our mind). As a result, nothing happens:

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier
clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

As we can see from alert output, in a browser the timer identifier is a number. In other environments, this can be something else. For instance, Node.js returns a timer object with additional methods.

Again, there is no universal specification for these methods, so that’s fine.

For browsers, timers are described in the timers section of HTML5 standard.

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.

To stop further calls, we should call `clearInterval(timerId)`.

The following example will show the message every 2 seconds. After 5 seconds, the output is stopped:

```
// repeat with the interval of 2 seconds
let timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop
setTimeout(() => { clearInterval(timerId); alert('stop'); }, 5000);
```

Time goes on while alert is shown

In most browsers, including Chrome and Firefox the internal timer continues “ticking” while showing alert/confirm/prompt.

So if you run the code above and don’t dismiss the alert window for some time, then in the next alert will be shown immediately as you do it. The actual interval between alerts will be shorter than 2 seconds.

Nested setTimeout

There are two ways of running something regularly.

One is setInterval. The other one is a nested setTimeout, like this:

```
/** instead of:
let timerId = setInterval(() => alert('tick'), 2000);
*/

let timerId = setTimeout(function tick() {
    alert('tick');
    timerId = setTimeout(tick, 2000); // (*)
}, 2000);
```

Zero delay setTimeout

There’s a special use case: setTimeout(func, 0), or just setTimeout(func).

This schedules the execution of func as soon as possible. But the scheduler will invoke it only after the currently executing script is complete.

So the function is scheduled to run “right after” the current script.

For instance, this outputs “Hello”, then immediately “World”:

```
setTimeout(() => alert("World"));
alert("Hello");
```

The first line “puts the call into calendar after 0ms”. But the scheduler will only “check the calendar” after the current script is complete, so "Hello" is first, and "World" – after it.

Decorators and forwarding, call/apply

JavaScript gives exceptional flexibility when dealing with functions. They can be passed around, used as objects, and now we’ll see how to *forward* calls between them and *decorate* them.

Transparent caching

Let’s say we have a function `slow(x)` which is CPU-heavy, but its results are stable. In other words, for the same `x` it always returns the same result.

If the function is called often, we may want to cache (remember) the results to avoid spending extra-time on recalculations.

But instead of adding that functionality into `slow()` we’ll create a wrapper function, that adds caching. As we’ll see, there are many benefits of doing so.

Here’s the code, and explanations follow:

```
function slow(x) {
  // there can be a heavy CPU-intensive job here
  alert(`Called with ${x}`);
  return x;
}
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {    // if there's such key in cache
      return cache.get(x); // read the result from it
    }
    let result = func(x);  // otherwise call func
    cache.set(x, result);  // and cache (remember) the result
    return result;
```

```
    };
}
slow = cachingDecorator(slow);
alert( slow(1) ); // slow(1) is cached
alert( "Again: " + slow(1) ); // the same
alert( slow(2) ); // slow(2) is cached
alert( "Again: " + slow(2) ); // the same as the previous line
```

Using “func.call” for the context

The caching decorator mentioned above is not suited to work with object methods.

For instance, in the code below worker.slow() stops working after the decoration:

```
// we'll make worker.slow caching
let worker = {
    someMethod() {
        return 1;
    },
    slow(x) {
        // scary CPU-heavy task here
        alert("Called with " + x);
        return x * this.someMethod(); // (*)
    }
};

// same code as before
function cachingDecorator(func) {
    let cache = new Map();
    return function(x) {
        if (cache.has(x)) {
            return cache.get(x);
        }
        let result = func(x); // (**)
```

```
        cache.set(x, result);
        return result;
    };
}
alert( worker.slow(1) ); // the original method works
worker.slow = cachingDecorator(worker.slow); // now make it caching
alert( worker.slow(2) ); // Whoops! Error: Cannot read property 'someMethod' of undefined
```

The error occurs in the line (*) that tries to access this.someMethod and fails. Can you see why?

The reason is that the wrapper calls the original function as func(x) in the line (**). And, when called like that, the function gets this = undefined.

So, the wrapper passes the call to the original method, but without the context this. Hence the error.

Let's fix it.

There's a special built-in function method func.call(context, …args) that allows to call a function explicitly setting this.

The syntax is:

```
func.call(context, arg1, arg2, ...)
```

It runs func providing the first argument as this, and the next as the arguments.

To put it simply, these two calls do almost the same:

```
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

They both call func with arguments 1, 2 and 3. The only difference is that func.call also sets this to obj.

As an example, in the code below we call sayHi in the context of different objects: sayHi.call(user) runs sayHi providing this=user, and the next line sets this=admin:

```
function sayHi() {
```

```
    alert(this.name);
}
let user = { name: "John" };
let admin = { name: "Admin" };
// use call to pass different objects as "this"
sayHi.call( user ); // this = John
sayHi.call( admin ); // this = Admin
```

And here we use call to call say with the given context and phrase:

```
function say(phrase) {
    alert(this.name + ': ' + phrase);
}
let user = { name: "John" };
// user becomes this, and "Hello" becomes the first argument
say.call( user, "Hello" ); // John: Hello
```

In our case, we can use call in the wrapper to pass the context to the original function:

```
let worker = {
    someMethod() {
        return 1;
    },
    slow(x) {
        alert("Called with " + x);
        return x * this.someMethod(); // (*)
    }
};

function cachingDecorator(func) {
    let cache = new Map();
    return function(x) {
        if (cache.has(x)) {
            return cache.get(x);
```

```
        }
        let result = func.call(this, x); // "this" is passed correctly now
        cache.set(x, result);
        return result;
    };
}
worker.slow = cachingDecorator(worker.slow); // now make it caching
alert( worker.slow(2) ); // works
alert( worker.slow(2) ); // works, doesn't call the original (cached)
```

Now everything is fine.

To make it all clear, let's see more deeply how this is passed along:

1. After the decoration `worker.slow` is now the wrapper function `(x) { ... }`.
2. So when `worker.slow(2)` is executed, the wrapper gets 2 as an argument and `this=worker` (it's the object before dot).
3. Inside the wrapper, assuming the result is not yet cached, `func.call(this, x)` passes the current this (=worker) and the current argument (=2) to the original method.

Functions vs Methods:

One thing we need to clear up before we move on — technically speaking, **built in browser functions are not functions — they are methods.** This sounds a bit scary and confusing, but don't worry — the words function and method are largely interchangeable, at least for our purposes, at this stage in your learning.

The distinction is that methods are functions defined inside objects. Built-in browser functions (**methods**) and variables (which are called **properties**) are stored inside structured objects, to make the code more efficient and easier to handle.

Invoking functions:

```
function square(number) {
    return number * number;
}
square(5)
```

Primitive parameters (such as a number) are passed to functions **by value**; the value is passed to the function, but if the function changes the value of the parameter, **this change is not reflected globally or in the calling function**.

If you pass an object (i.e. a non-primitive value, such as Array or a user-defined object) as a parameter and the function changes the object's properties, that change is visible outside the function, as shown in the following example:

```
function myFunc(theObject) {
    theObject.make = 'Toyota';
}
var mycar = {make: 'Honda', model: 'Accord', year: 1998};
var x, y;
x = mycar.make; // x gets the value "Honda"
myFunc(mycar); // calling function
y = mycar.make; // y gets the value "Toyota"
```

Anonymous function:

We can also create function without a name.

```
function() {
    alert('hello');
}
var square = function(number) { return number * number; };
var x = square(4); // x gets the value 16
```

However, a name can be provided with a function expression and can be used inside the function to refer to itself, or in a debugger to identify the function in stack traces:

```
var factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1); };
console.log(factorial(3));
```

This is called an **anonymous function** — it has no name! It also won't do anything on its own. You generally use an anonymous function along with an event handler, for example the following would run the code inside the function whenever the associated button is clicked:

```
var myButton = document.querySelector('button');
myButton.onclick = function() {
    alert('hello');
}
```

You can also assign an anonymous function to be the value of a variable, for example:

```
var myGreeting = function() {
    alert('hello');
}
myGreeting;
```

Function scope and conflicts:

- Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.
- Also, a function defined in the global scope can access all variables defined in the global scope. A function defined inside another function can also access all variables defined in its parent function and any other variable to which the parent function has access.

Example 1:

```
// The following variables are defined in the global scope
var num1 = 20, num2 = 3,  name = 'Chamahk';

// This function is defined in the global scope
function multiply() {
    return num1 * num2;
```

```
}
multiply(); // Returns 60

// A nested function example
function getScore() {
    var num1 = 2, num2 = 3;
    function add() {
        return name + ' scored ' + (num1 + num2);
    }
    return add();
}
getScore(); // Returns "Chamahk scored 5"
```

Example 2:

```
function myBigFunction() {
    var myValue = 1;
    subFunction(myValue);
}

function subFunction(value) {
    console.log(value);
}
```

A function with an empty return or without it returns undefined

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

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

An empty return is also the same as return undefined:
function doNothing() {
    return;
```

```
}
alert( doNothing() === undefined ); // true
```

Never add a newline between return and the value

For a long expression in return, it might be tempting to put it on a separate line, like this:

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

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))
```

So, it effectively becomes an empty return.
If we want the returned expression to wrap across multiple lines, we should start it at the same line as return. Or at least put the opening parentheses there as follows:

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

And it will work just as we expect it to.

Why is there a semicolon at the end?

You might wonder, why does Function Expression have a semicolon ; at the end, but Function Declaration does not:

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

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

The answer is simple:

- There's no need for ; at the end of code blocks and syntax structures that use them like if { ... }, for { }, function f { } etc.
- A Function Expression is used inside the statement: let sayHi = ...;, as a value. It's not a code block, but rather an assignment. The semicolon ; is recommended at the end of statements, no matter what the value is. So the semicolon here is not related to the Function Expression itself, it just terminates the statement.

Callback function

Let's look at more examples of passing functions as values and using function expressions.

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

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}

function showOk() {
     alert( "You agreed." );
}
function showCancel() {
     alert( "You canceled the execution." );
}
ask("Do you agree?", showOk, showCancel);
```

In practice, such functions are quite useful. The major difference between a real-life `ask` and the example above is that real-life functions use more complex ways to interact with the user than a simple `confirm`. In the browser, such function usually draws a nice-looking question window. But that's another story.

The arguments showOk and showCancel of ask are called *callback functions* or just *callbacks*.

The idea is that we pass a function and expect it to be “called back” later if necessary. In our case, showOk becomes the callback for “yes” answer, and showCancel for “no” answer.

We can use Function Expressions to write the same function much shorter:

```
function ask(question, yes, no) {
    if (confirm(question)) yes()
    else no();
}
ask(
    "Do you agree?",
    function() { alert("You agreed."); },
    function() { alert("You canceled the execution."); }
);
```

Here, functions are declared right inside the ask(...) call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of ask (because they are not assigned to variables), but that’s just what we want here.

Such code appears in our scripts very naturally, it’s in the spirit of JavaScript.

Function declaration vs Function expression

The more subtle difference is *when* a function is created by the JavaScript engine.

A Function Expression is created when the execution reaches it and is usable only from that moment.

Once the execution flow passes to the right side of the assignment let sum = function… – here we go, the function is created and can be used (assigned, called, etc.) from now on.

Function Declarations are different.

A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in the whole script, no matter where it is.

That's due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an "initialization stage".

And after all Function Declarations are processed, the code is executed. So it has access to these functions.

For example, this works:

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

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

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

…If it were a Function Expression, then it wouldn't work:

```
sayHi("John"); // error!
let sayHi = function(name) {  // (*) no magic any more
  alert( `Hello, ${name}` );
};
```

Function Expressions are created when the execution reaches them. That would happen only in the line (*). Too late.

Another special feature of Function Declarations is their block scope.

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

For instance, let's imagine that we need to declare a function welcome() depending on the age variable that we get during runtime. And then we plan to use it some time later.

If we use Function Declaration, it won't work as intended:

```
let age = prompt("What is your age?", 18);

// conditionally declare a function

if (age < 18) {
    function welcome() {
        alert("Hello!");
    }
} else {
    function welcome() {
        alert("Greetings!");
    }
}

// ...use it later
welcome(); // Error: welcome is not defined
```

That’s because a Function Declaration is only visible inside the code block in which it resides.

Here’s another example:

```
let age = 16; // take 16 as an example
if (age < 18) {
    welcome();
    function welcome() {
        alert("Hello!");
    }
    welcome();
} else {
    function welcome() {
        alert("Greetings!");
    }
}
// Here we're out of curly braces,
// so we can not see Function Declarations made inside of them.
```

```
welcome(); // Error: welcome is not defined
```

What can we do to make `welcome` visible outside of `if`?

The correct approach would be to use a Function Expression and assign `welcome` to the variable that is declared outside of `if` and has the proper visibility.

This code works as intended:

```
let age = prompt("What is your age?", 18);
let welcome;
if (age < 18) {
    welcome = function() {
        alert("Hello!");
    };
} else {
    welcome = function() {
        alert("Greetings!");
    };
}

welcome(); // this works
```

Or we could simplify it even further using a question mark operator `?`:

```
let age = prompt("What is your age?", 18);
let welcome = (age < 18) ?
    function() { alert("Hello!"); } :
    function() { alert("Greetings!"); };
welcome(); // ok now
```

Arrow functions

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

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

…This creates a function func that has arguments arg1..argN, evaluates the expression on the right side with their use and returns its result.

In other words, it's roughly the same as:

```
let func = function(arg1, arg2, ...argN) {
    return expression;
};
```

Let's look at an example:

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

/* The arrow function is a shorter form of:
let sum = function(a, b) {
    return a + b;
};
*/
alert( sum(1, 2) ); // 3
```

If we have only one argument, then parentheses around parameters can be omitted, making that even shorter:

```
// let double = function(n) { return n * 2 }
let double = n => n * 2;
alert( double(3) ); // 6
```

If there are no arguments, parentheses should be empty

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

Arrow functions can be used in the same way as Function Expressions.
For instance, here's the rewritten example with welcome():

```
let age = prompt("What is your age?", 18);
let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");
welcome();
```

Arrow functions may appear unfamiliar and not very readable at first, but that quickly changes as the eyes get used to the structure.

They are very convenient for simple one-line actions, when we're just too lazy to write many words.

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.

For instance, here arrow() uses this from the outer user.sayHi() method:

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

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

That's a special feature of arrow functions, it's useful when we actually do not want to have a separate this, but rather to take it from the outer context

For instance, while giving function expression,

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

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

For instance,

```
function Person() {
    this.age = 0;
    setInterval(function growUp() {
        // In nonstrict mode, the growUp() function defines `this` as the global object,
        // which is different from the `this defined by the Person() constructor.
        this.age++;
    }, 1000);
}
var p = new Person();
```

In ECMAScript 3/5, this issue was fixed by assigning the value in this to a variable that could be closed over.

```
function Person() {
    var self = this;
    self.age = 0;

    setInterval(function growUp() {
        // The callback refers to the `self` variable of which
        // the value is the expected object.
        self.age++;
```

```
    }, 1000);
}
```

An arrow function does not have its own this; this **value of the enclosing execution context is used.** Thus, in the following code, this within the function that is passed to setInterval has the same value as this in the enclosing function:

```
function Person() {
    this.age = 0;
    setInterval(() => {
        this.age++; // |this| properly refers to the person object
    }, 1000);
}
var p = new Person();
```

Nested functions & closure :

Closures are one of the most powerful features of JavaScript. JavaScript allows for the nesting of functions and grants the inner function full access to all the variables and functions defined inside the outer function (and all other variables and functions that the outer function has access to). However, the outer function does not have access to the variables and functions defined inside the inner function. This provides a sort of encapsulation for the variables of the inner function.

You can nest a function within a function. The nested (inner) function is private to its containing (outer) function. It also forms a *closure*. A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).

Since a nested function is a closure, this means that a nested function can "inherit" the arguments and variables of its containing function. In other words, the inner function contains the scope of the outer function.

To summarize:

- The inner function can be accessed only from statements in the outer function.
- The inner function forms a closure: the inner function can use the arguments and variables of the outer function, while the outer function cannot use the arguments and variables of the inner function.

The following example shows nested functions:

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}
a = addSquares(2, 3); // returns 13
b = addSquares(3, 4); // returns 25
c = addSquares(4, 5); // returns 41
```

Since the inner function forms a closure, you can call the outer function and specify arguments for both the outer and inner function:

```
function outside(x) {
  function inside(y) {
    return x + y;
  }
  return inside;
}
fn_inside = outside(3); // Think of it like: give me a function that adds 3 to whatever you give
```

```
                    // it
result = fn_inside(5); // returns 8 becoz, x=3 and y =5
result1 = outside(3)(5); // returns 8
```

Preservation of variables:

Notice how x is preserved when inside is returned. A closure must preserve the arguments and variables in all scopes it references. Since each call provides potentially different arguments, a new closure is created for each call to outside. The memory can be freed only when the returned inside is no longer accessible.

Logic behind nested function :

```
function A (x) {
   function B(y){
      function C (z) {
         return x + y + z ;
      }
      return C;
   }
   return B;
}
```

Here, In this example, C accesses B's y and A's x. This can be done because:

1. B forms a closure including A, i.e. B can access A's arguments and variables.
2. C forms a closure including B.
3. Because B's closure includes A, C's closure includes A, C can access both B*and* A's arguments and variables. In other words, C *chains* the scopes of Band A in that order.

Easy access to assign values to x, y, z is

```
result_Var = A(1)(2)(3) // returns 6
```

Or else , step by step

```
assign_y_value = A(100)
```

```
// returns
ƒ B(y){
      function C (z) {
         return 100 + y + z ;
      }
      return C;
   }

Step 2:
assign_z_value = assign_y_value(200)
//returns
ƒ C(z) {
         return 100 + 200 + z ;
      }

Step 3:
result = assign_z_value(300)
// returns 600
```

Name conflicts :

When two arguments or variables in the scope of a closure have the same name, there is a *name conflict*. More inner scopes take precedence, so the inner-most scope takes the highest precedence, while the outer-most scope takes the lowest. This is the scope chain. The first on the chain is the inner-most scope, and the last is the outer-most scope. Consider the following:

```
Ex: 1
function outside() {
    var x = 5;
    function inside(x) {
        return x * 2;
    }
    return inside;
```

```
}
outside()(10); // returns 20 instead of 10
```

Ex:2

```
function outside() {
    var x = 5;
    function inside(y) {
        return x * 2;
    }
    return inside;
}
outside()(10); // returns 10
```

The name conflict happens at the statement return x and is between inside's parameter x and outside's variable x. The scope chain here is {inside, outside, global object}. Therefore inside's x takes precedences over outside's x, and 20 (inside's x) is returned instead of 10 (outside's x).

Emulating private methods with closures

Languages such as Java provide the ability to declare methods private, meaning that they can only be called by other methods in the same class.

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code: they also provide a powerful way of managing your global namespace, keeping non-essential methods from cluttering up the public interface to your code.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Using closures in this way is known as the module pattern.

```
var counter = (function() {
    var privateCounter = 0;
    function changeBy(val) {
        privateCounter += val;
```

```
    }
    return {
        increment: function() {
            changeBy(1);
        },
        decrement: function() {
            changeBy(-1);
        },
        value: function() {
            return privateCounter;
        }
    };
})();

console.log(counter.value()); // logs 0
counter.increment();
counter.increment();
console.log(counter.value()); // logs 2
counter.decrement();
console.log(counter.value()); // logs 1
```

Here, though, we create a single lexical environment that is shared by three functions: counter.increment, counter.decrement, and counter.value.

The shared lexical environment is created in the body of an anonymous function, which is executed as soon as it has been defined. The lexical environment contains two private items: a variable called `privateCounter` and a function called `changeBy`. Neither of these private items can be accessed directly from outside the anonymous function. Instead, they must be accessed by the three public functions that are returned from the anonymous wrapper.

Those three public functions are closures that share the same environment. Thanks to JavaScript's lexical scoping, they each have access to the privateCounter variable and changeBy function.

You'll notice we're defining an anonymous function that creates a counter, and then we call it immediately and assign the result to the counter variable. We could store this function in a separate variable makeCounter and use it to create several counters.

```
var makeCounter = function() {
    var privateCounter = 0;
    function changeBy(val) {
        privateCounter += val;
    }
    return {
        increment: function() {
            changeBy(1);
        },
        decrement: function() {
            changeBy(-1);
        },
        value: function() {
            return privateCounter;
        }
    }
};

var counter1 = makeCounter();
var counter2 = makeCounter();
alert(counter1.value()); /* Alerts 0 */
counter1.increment();
counter1.increment();
alert(counter1.value()); /* Alerts 2 */
counter1.decrement();
```

```
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

Notice how each of the two counters, counter1 and counter2, maintains its independence from the other. Each closure references a different version of the privateCounter variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable; however changes to the variable value in one closure do not affect the value in the other closure.

Closure Scope Chain

For every closure we have three scopes

- Local Scope (Own scope)
- Outer Functions Scope
- Global Scope

So, we have access to all three scopes for a closure but often make a common mistake when we have nested inner functions. Consider the following example:

```
// global scope
var e = 10;
function sum(a){
    return function(b){
        return function(c){
            // outer functions scope
            return function(d){
                // local scope
                return a + b + c + d + e;
            }
        }
    }
}
```

```
console.log(sum(1)(2)(3)(4)); // log 20

// We can also write without anonymous functions:

// global scope
var e = 10;
function sum(a) {
    return function sum2(b){
        return function sum3(c){
            // outer functions scope
            return function sum4(d){
                // local scope
                return a + b + c + d + e;
            }
        }
    }
}

var s = sum(1);
var s1 = s(2);
var s2 = s1(3);
var s3 = s2(4);
console.log(s3) //log 20
```

Property flags and descriptors

As we know, objects can store properties.

Until now, a property was a simple “key-value” pair to us. But an object property is actually a more flexible and powerful thing.

Property flags

Object properties, besides a **value**, have three special attributes (so-called “flags”):

- **writable** – if `true`, the value can be changed, otherwise it’s read-only.
- **enumerable** – if `true`, then listed in loops, otherwise not listed.
- **configurable** – if `true`, the property can be deleted and these attributes can be modified, otherwise not.

We didn’t see them yet, because generally they do not show up. When we create a property “the usual way”, all of them are `true`. But we also can change them anytime.

First, let’s see how to get those flags.

The method Object.getOwnPropertyDescriptor allows to query the *full* information about a property.

The syntax is:
`let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);`

obj

The object to get information from.

propertyName
The name of the property.

The returned value is a so-called “property descriptor” object: it contains the value and all the flags.

For instance:

```
let user = {
    name: "John"
};
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
alert( JSON.stringify(descriptor, null, 2 ) );
/* property descriptor:
{
```

```
    "value": "John",
    "writable": true,
    "enumerable": true,
    "configurable": true
}
*/
```

To change the flags, we can use Object.defineProperty. The syntax is:

```
Object.defineProperty(obj, propertyName, descriptor)
```

obj, propertyName
The object and its property to apply the descriptor.

descriptor
Property descriptor to apply.

If the property exists, defineProperty updates its flags. Otherwise, it creates the property with the given value and flags; in that case, if a flag is not supplied, it is assumed false.

For instance, here a property name is created with all falsy flags:

```
let user = {};
Object.defineProperty(user, "name", {
    value: "John"
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
    "value": "John",
    "writable": false,
    "enumerable": false,
    "configurable": false
```

```
}
 */
```

Compare it with “normally created” `user.name` above: now all flags are falsy. If that’s not what we want then we’d better set them to `true` in `descriptor`.

Now let’s see the effects of the flags by example.

Non-writable

Let’s make `user.name` non-writable (can’t be reassigned) by changing `writable` flag:

```
let user = {
    name: "John"
};
Object.defineProperty(user, "name", {
    writable: false
});
user.name = "Pete"; // Error: Cannot assign to read only property 'name'
```

Now no one can change the name of our user, unless they apply their own `defineProperty` to override ours.

Non-enumerable

Now let’s add a custom `toString` to `user`.

Normally, a built-in `toString` for objects is non-enumerable, it does not show up in `for..in`. But if we add a `toString` of our own, then by default it shows up in `for..in`, like this:

```
let user = {
    name: "John",
    toString() {
        return this.name;
    }
};
```

```
// By default, both our properties are listed:
for (let key in user) alert(key); // name, toString
```

If we don't like it, then we can set `enumerable:false`. Then it won't appear in a `for..in` loop, just like the built-in one:

```
let user = {
    name: "John",
    toString() {
        return this.name;
    }
};
Object.defineProperty(user, "toString", {
    enumerable: false
});
// Now our toString disappears:
for (let key in user) alert(key); // name
```

Non-enumerable properties are also excluded from `Object.keys`:

```
alert(Object.keys(user)); // name
```

Non-configurable

The non-configurable flag (`configurable:false`) is sometimes preset for built-in objects and properties.

A non-configurable property can not be deleted.

For instance, `Math.PI` is non-writable, non-enumerable and non-configurable:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');
alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
    "value": 3.141592653589793,
```

```
    "writable": false,
    "enumerable": false,
    "configurable": false
}
*/
```

So, a programmer is unable to change the value of Math.PI or overwrite it.

```
Math.PI = 3; // Error
// delete Math.PI won't work either
```

Making a property non-configurable is a one-way road. We cannot change it back with defineProperty.

To be precise, non-configurability imposes several restrictions on defineProperty:

1. Can't change configurable flag.
2. Can't change the enumerable flag.
3. Can't change writable: false to true (the other way round works).
4. Can't change get/set for an accessor property (but can assign them if absent).

Here we are making user.name a "forever sealed" constant:

"Non-configurable" doesn't mean "non-writable"

Notable exception: a value of non-configurable, but writable property can be changed.

The idea of configurable: false is to prevent changes to property flags and its deletion, not changes to its value.

Object.defineProperties

There's a method Object.defineProperties(obj, descriptors) that allows to define many properties at once.

The syntax is:

```
Object.defineProperties(obj, {
```

```
    prop1: descriptor1,
    prop2: descriptor2
    // ...
});
```

For instance:

```
Object.defineProperties(user, {
    name: { value: "John", writable: false },
    surname: { value: "Smith", writable: false },
    // ...
});
```

So, we can set many properties at once.

Take a look into below code

```
let user = {
    name: 'Alex',
    age : 25,
    dob : '3rd, Oct',
    profession: 'Software Engineer'
}

console.log(Object.getOwnPropertyDescriptor(user, 'dob'))
Object.defineProperty(user, 'test', { value: 'testValue' })
console.log(Object.getOwnPropertyDescriptor(user, 'test'))
```

O/p

```
{ value: '3rd, Oct', writable: true, enumerable: true, configurable: true }
{ value: 'testValue', writable: false, enumerable: false, configurable: false }
```

Getters and Setters:

Getter and setter methods get and set the properties inside of an object. There are a couple of advantages to using these methods for getting and setting properties directly:

- You can check if new data is valid before setting a property.
- You can perform an action on the data while you are getting or setting a property.
- You can control which properties can be set and retrieved.

Example:

```
let person = {
    _name: 'Lu Xun',
    _age: 137,
    set age(ageIn) {
        if (typeof ageIn === 'number') {
            this._age = ageIn;
        } else {
            console.log('Invalid input');
            return 'Invalid input';
        }
    }
};
```

- We prepended the property names with underscores (_). Developers use an underscore before a property name to indicate a property or value should not be modified directly by other code. We recommend prepending all properties with an underscore, and creating setters for all attributes you want to access later in your code.
- The `set age()` setter method accepts `ageIn` as a variable. The `ageIn` variable holds the new value that we will store in `_age`.
- Inside of the `.age()` setter we use a conditional statement to check if the `ageIn` variable (our new value) is a number.
- If the input value is a number (valid input), then we use `this._age` to change the value assigned to `_age`. If it is not valid, then we output a message to the user.

How to call a setter method?

Now that you know how to create a setter method, you may be wondering how we use it. We call setter methods the same way we edited properties.

```
person.age='Thirty-nine'
```

Getter:

Getters are used to get the property values inside of an object.

```
let person = {
    _name: 'Lu Xun',
    _age: 137,
    set age(ageIn) {
        if (typeof ageIn === 'number') {
            this._age = ageIn;
        } else {
            console.log('Invalid input');
            return 'Invalid input';
        }
    },
    get age() {
        console.log(`${this._name} is ${this._age} years old.`);
        return this._age;
    }
};

person.age = 'Thirty-nine';
person.age = 39;
console.log(person.age);
```

Note: _ should be placed before property name while using getter & setter, otherwise code runs to infinite.

Accessor descriptors

Descriptors for accessor properties are different from those for data properties.

For accessor properties, there is no `value` or `writable`, but instead there are `get` and `set` functions.

That is, an accessor descriptor may have:

- **get** – a function without arguments, that works when a property is read,
- **set** – a function with one argument, that is called when the property is set,
- **enumerable** – same as for data properties,
- **configurable** – same as for data properties.

For instance, to create an accessor `fullName` with `defineProperty`, we can pass a descriptor with `get` and `set`:

```
let user = {
    name: "John",
    surname: "Smith"
};
Object.defineProperty(user, 'fullName', {
    get() {
        return `${this.name} ${this.surname}`;
    },
    set(value) {
        [this.name, this.surname] = value.split(" ");
    }
});
alert(user.fullName); // John Smith
for(let key in user) alert(key); // name, surname
```

Prototypal inheritance

In programming, we often want to take something and extend it.

For instance, we have a `user` object with its properties and methods, and want to make `admin` and `guest` as slightly modified variants of it. We'd like to reuse what we have in `user`, not copy/reimplement its methods, just build a new object on top of it.

Prototypal inheritance is a language feature that helps in that.

[[Prototype]]

In JavaScript, objects have a special hidden property `[[Prototype]]` (as named in the specification), that is either `null` or references another object. That object is called "a prototype":

The property `[[Prototype]]` is internal and hidden, but there are many ways to set it.

One of them is to use `__proto__`, like this:

```
let animal = {
    eats: true
};
let rabbit = {
    jumps: true
};
rabbit.__proto__ = animal;
```

If we look for a property in rabbit, and it's missing, JavaScript automatically takes it from animal.

For instance:

```
let animal = {
    eats: true
};
let rabbit = {
    jumps: true
};
rabbit.__proto__ = animal; // (*)
// we can find both properties in rabbit now:
alert( rabbit.eats ); // true (**)
alert( rabbit.jumps ); // true
```

Here the line (*) sets animal to be a prototype of rabbit.

Then, when alert tries to read property rabbit.eats (**), it’s not in rabbit, so JavaScript follows the [[Prototype]] reference and finds it in animal

Here we can say that "animal is the prototype of rabbit" or "rabbit prototypically inherits from animal".

So if animal has a lot of useful properties and methods, then they become automatically available in rabbit. Such properties are called “inherited”.

If we have a method in animal, it can be called on rabbit:

```
let animal = {
    eats: true,
    walk() {
        alert("Animal walk");
    }
};
let rabbit = {
    jumps: true,
    __proto__: animal
};
// walk is taken from the prototype
rabbit.walk(); // Animal walk
```

The prototype chain can be longer:

```
let animal = {
    eats: true,
    walk() {
        alert("Animal walk");
    }
};
let rabbit = {
```

```
    jumps: true,
    __proto__: animal
};
let longEar = {
    earLength: 10,
    __proto__: rabbit
};
// walk is taken from the prototype chain
longEar.walk(); // Animal walk
alert(longEar.jumps); // true (from rabbit)
```

There are only two limitations:

1. The references can't go in circles. JavaScript will throw an error if we try to assign `__proto__` in a circle.
2. The value of `__proto__` can be either an object or null. Other types are ignored.

Writing doesn't use prototype

The prototype is only used for reading properties.

Write/delete operations work directly with the object.

In the example below, we assign its own walk method to rabbit:

```
let animal = {
    eats: true,
    walk() {
      /* this method won't be used by rabbit */
    }
};
let rabbit = {
    __proto__: animal
};
rabbit.walk = function() {
```

```
    alert("Rabbit! Bounce-bounce!");
};
rabbit.walk(); // Rabbit! Bounce-bounce!
```

From now on, `rabbit.walk()` call finds the method immediately in the object and executes it, without using the prototype:

Accessor properties are an exception, as assignment is handled by a setter function. So writing to such a property is actually the same as calling a function.

For that reason admin.fullName works correctly in the code below:

```
let user = {
    name: "John",
    surname: "Smith",
    set fullName(value) {
        [this.name, this.surname] = value.split(" ");
    },
    get fullName() {
        return `${this.name} ${this.surname}`;
    }
};

let admin = {
    __proto__: user,
    isAdmin: true
};
alert(admin.fullName); // John Smith (*)

// setter triggers!
admin.fullName = "Alice Cooper"; // (**)
```

Here in the line (*) the property admin.fullName has a getter in the prototype user, so it is called. And in the line (**) the property has a setter in the prototype, so it is called.

The value of “this”

An interesting question may arise in the example above: what’s the value of this inside set fullName(value)? Where are the properties this.name and this.surname written: into user or admin?

The answer is simple: this is not affected by prototypes at all.

No matter where the method is found: in an object or its prototype. In a method call, this is always the object before the dot.

So, the setter call admin.fullName= uses admin as this, not user.

For instance, here animal represents a “method storage”, and rabbit makes use of it.

The call rabbit.sleep() sets this.isSleeping on the rabbit object:

```
// animal has methods
let animal = {
    walk() {
        if (!this.isSleeping) {
            alert(`I walk`);
        }
    },
    sleep() {
        this.isSleeping = true;
    }
};
let rabbit = {
    name: "White Rabbit",
    __proto__: animal
};
// modifies rabbit.isSleeping
```

```
rabbit.sleep();
alert(rabbit.isSleeping); // true
alert(animal.isSleeping); // undefined (no such property in the prototype)
```

for…in loop

The **for..in loop iterates over inherited properties too**.

For instance:

```
let animal = {
    eats: true
};
let rabbit = {
    jumps: true,
    __proto__: animal
};

// Object.keys only returns own keys
alert(Object.keys(rabbit)); // jumps
// for..in loops over both own and inherited keys
for(let prop in rabbit) alert(prop); // jumps, then eats
```

If that's not what we want, and we'd like to exclude inherited properties, there's a built-in method **obj.hasOwnProperty(key): it returns true if obj has its own** (not inherited) property named key.

So we can filter out inherited properties (or do something else with them):

```
let animal = {e
    eats: true
};
let rabbit = {
    jumps: true,
    __proto__: animal
```

```
};
for (let prop in rabbit) {
    let isOwn = rabbit.hasOwnProperty(prop);
    if (isOwn) {
        alert(`Our: ${prop}`); // Our: jumps
    } else {
        alert(`Inherited: ${prop}`); // Inherited: eats
   }
}
```

F.prototype

Remember, new objects can be created with a constructor function, like `new F()`.

If `F.prototype` is an object, then the `new` operator uses it to set `[[Prototype]]` for the new object.

Here's the example:

```
let animal = {
    eats: true
};
function Rabbit(name) {
    this.name = name;
}
Rabbit.prototype = animal;
let rabbit = new Rabbit("White Rabbit"); //  rabbit.__proto__ == animal
alert( rabbit.eats ); // true
```

Setting Rabbit.prototype = animal literally states the following: "When a new Rabbit is created, assign its [[Prototype]] to animal".

Default F.prototype, constructor property

Every function has the `"prototype"` property even if we don't supply it.

The default `"prototype"` is an object with the only property `constructor` that points back to the function itself.

Like this:

```
function Rabbit() {}
/* default prototype
Rabbit.prototype = { constructor: Rabbit };
*/
```

We can check it:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }
alert( Rabbit.prototype.constructor == Rabbit ); // true
```

Naturally, if we do nothing, the constructor property is available to all rabbits through [[Prototype]]:

```
function Rabbit() {}
// by default:
// Rabbit.prototype = { constructor: Rabbit }
let rabbit = new Rabbit(); // inherits from {constructor: Rabbit}
alert(rabbit.constructor == Rabbit); // true (from prototype)
```

We can use `constructor` property to create a new object using the same constructor as the existing one.

Like here:

```
function Rabbit(name) {
    this.name = name;
    alert(name);
}
let rabbit = new Rabbit("White Rabbit");
let rabbit2 = new rabbit.constructor("Black Rabbit");
```

That’s handy when we have an object, we don't know which constructor was used for it (e.g. it comes from a 3rd party library), and we need to create another one of the same kind.

For instance:

```
function Rabbit() {}

Rabbit.prototype = {
    jumps: true
};
let rabbit = new Rabbit();
alert(rabbit.constructor === Rabbit); // false
```

So, to keep the right "constructor" we can choose to add/remove properties to the default "prototype" instead of overwriting it as a whole:

```
function Rabbit() {}
// Not overwrite Rabbit.prototype totally, just add to it
Rabbit.prototype.jumps = true
// the default Rabbit.prototype.constructor is preserved
```

Or, alternatively, recreate the constructor property manually:

```
Rabbit.prototype = {
    jumps: true,
    constructor: Rabbit
};
// now constructor is also correct, because we added it
```

Native prototypes

The "prototype" property is widely used by the core of JavaScript itself. All built-in constructor functions use it.

For instance

```
let obj = {};
alert(obj.__proto__ === Object.prototype); // true
// obj.toString === obj.__proto__.toString == Object.prototype.toString
```

Please note that there is no more [[Prototype]] in the chain above Object.prototype:

```
alert(Object.prototype.__proto__); // null
```

Other built-in prototypes

Other built-in objects such as Array, Date, Function and others also keep methods in prototypes.
For instance, when we create an array [1, 2, 3], the default new Array() constructor is used internally. So Array.prototype becomes its prototype and provides methods. That's very memory-efficient.

By specification, all of the built-in prototypes have Object.prototype on the top. That's why some people say that "everything inherits from objects".

Let's check the prototypes manually:

```
let arr = [1, 2, 3];
// it inherits from Array.prototype?
alert( arr.__proto__ === Array.prototype ); // true

// then from Object.prototype?
alert( arr.__proto__.__proto__ === Object.prototype ); // true

// and null on the top.
alert( arr.__proto__.__proto__.__proto__ ); // null
```

Primitives

The most intricate thing happens with strings, numbers and booleans.

As we remember, they are not objects. But if we try to access their properties, temporary wrapper objects are created using built-in constructors String, Number and Boolean. They provide the methods and disappear.

These objects are created invisibly to us and most engines optimize them out, but the specification describes it exactly this way. Methods of these objects also reside in prototypes, available as String.prototype, Number.prototype and Boolean.prototype.

Class

In practice, we often need to create many objects of the same kind, like users, or goods or whatever.

As we already know from the chapter Constructor, operator "new", `new function` can help with that.

But in the modern JavaScript, there's a more advanced "class" construct, that introduces great new features which are useful for object-oriented programming.

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();
```

In JavaScript, a class is a kind of function.

Here, take a look:

```
class User {
    constructor(name) { this.name = name; }
    sayHi() { alert(this.name); }
}

// proof: User is a function
alert(typeof User); // function
```

Class Expression

Just like functions, classes can be defined inside another expression, passed around, returned, assigned, etc.

Here's an example of a class expression:

```
let User = class {
    sayHi() {
        alert("Hello");
```

```
    }
};
```

Similar to Named Function Expressions, class expressions may have a name.

If a class expression has a name, it’s visible inside the class only:

```
// "Named Class Expression"
// (no such term in the spec, but that's similar to Named Function Expression)
let User = class MyClass {
    sayHi() {
        alert(MyClass); // MyClass name is visible only inside the class
    }
};

new User().sayHi(); // works, shows MyClass definition
alert(MyClass); // error, MyClass name isn't visible outside of the class
```

We can even make classes dynamically “on-demand”, like this:

```
function makeClass(phrase) {
    // declare a class and return it
    return class {
        sayHi() {
            alert(phrase);
        };
    };
}

// Create a new class
let User = makeClass("Hello");
new User().sayHi(); // Hello
```

Getters/setters, other shorthands

Just like literal objects, classes may include getters/setters, computed properties etc.

Here's an example for user.name implemented using get/set:

```
class User {
    constructor(name) {
        // invokes the setter
        this.name = name;
    }

    get name() {
        return this._name;
    }


    set name(value) {
        if (value.length < 4) {
            alert("Name is too short.");
            return;
        }
        this._name = value;
    }
}
let user = new User("John");
alert(user.name); // John
user = new User(""); // Name too short.
```

Class properties

In the example above, `User` only had methods. Let's add a property:

Note: Class-level properties are a recent addition to the language.

```
class User {
```

```
    name = "Anonymous";
    sayHi() {
        alert(`Hello, ${this.name}!`);
    }
}
new User().sayHi();
alert(User.prototype.sayHi); // placed in User.prototype
alert(User.prototype.name); // undefined, not placed in User.prototype
```

The property name is not placed into User.prototype. Instead, it is created by new before calling the constructor, it's a property of the object itself.

Class inheritance

Class inheritance is a way for one class to extend another class. So we can create new functionality on top of the existing.

The "extends" keyword

Let's say we have class Animal:

```
class Animal {
    constructor(name) {
        this.speed = 0;
        this.name = name;
    }

    run(speed) {
        this.speed += speed;
       alert(`${this.name} runs with speed ${this.speed}.`);
   }

   stop() {
       this.speed = 0;
       alert(`${this.name} stands still.`);
```

```
    }
}

let animal = new Animal("My animal");
```

As rabbits are animals, Rabbit class should be based on Animal, have access to animal methods, so that rabbits can do what “generic” animals can do.

The syntax to extend another class is: class Child extends Parent.

Let’s create class Rabbit that inherits from Animal:

```
class Rabbit extends Animal {
    hide() {
        alert(`${this.name} hides!`);
    }
}

let rabbit = new Rabbit("White Rabbit");
rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

Overriding a method

Now let’s move forward and override a method. By default, all methods that are not specified in class Rabbit are taken directly “as is” from class Animal.

But if we specify our own method in Rabbit, such as stop() then it will be used instead:

```
class Rabbit extends Animal {
    stop() {
        // ...now this will be used for rabbit.stop()
        // instead of stop() from class Animal
    }
}
```

Usually we don’t want to totally replace a parent method, but rather to build on top of it to tweak or extend its functionality. We do something in our method, but call the parent method before/after it or in the process.

Classes provide "super" keyword for that.

- super.method(...) to call a parent method.
- super(...) to call a parent constructor (inside our constructor only).

For instance, let our rabbit auto hide when stopped:

```
class Animal {
    constructor(name) {
        this.speed = 0;
        this.name = name;
    }

    run(speed) {
        this.speed += speed;
        alert(`${this.name} runs with speed ${this.speed}.`);
    }
    stop() {
        this.speed = 0;
        alert(`${this.name} stands still.`);
    }
}

class Rabbit extends Animal {
    hide() {
        alert(`${this.name} hides!`);
    }
    stop() {
        super.stop(); // call parent stop
        this.hide(); // and then hide
    }
}
```

```
let rabbit = new Rabbit("White Rabbit");
rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.stop(); // White Rabbit stands still. White habbit hides!
```

Now `Rabbit` has the `stop` method that calls the parent `super.stop()` in the process.

Arrow functions have no `super`

As was mentioned in the chapter Arrow functions revisited, arrow functions do not have `super`.

If accessed, it's taken from the outer function. For instance:

```
class Rabbit extends Animal {
    stop() {
        setTimeout(() => super.stop(), 1000); // call parent stop after 1sec
    }
}
```

The `super` in the arrow function is the same as in `stop()`, so it works as intended. If we specified a "regular" function here, there would be an error:

```
// Unexpected super
setTimeout(function() { super.stop() }, 1000);
```

Overriding constructor

Until now, Rabbit did not have its own constructor.

According to the specification, if a class extends another class and has no constructor, then the following "empty" constructor is generated:

```
class Rabbit extends Animal {
    // generated for extending classes without own constructors
    constructor(...args) {
        super(...args);
    }
```

```
}
```

As we can see, it basically calls the parent constructor passing it all the arguments. That happens if we don't write a constructor of our own.

Now let's add a custom constructor to Rabbit. It will specify the earLength in addition to name:

```
class Animal {
    constructor(name) {
        this.speed = 0;
        this.name = name;
    }
    // ...
}

class Rabbit extends Animal {
    constructor(name, earLength) {
        super(name);
        this.earLength = earLength;
    }
    // ...
}

// now fine
let rabbit = new Rabbit("White Rabbit", 10);
alert(rabbit.name); // White Rabbit
alert(rabbit.earLength); // 10
```

Static properties and methods

We can also assign a method to the class function itself, not to its "prototype". Such methods are called *static*.

In a class, they are prepended by static keyword, like this:

```
class User {
```

```
  static staticMethod() {
    alert(this === User);
  }
}

User.staticMethod(); // true
```

That actually does the same as assigning it as a property directly:

```
class User { }
User.staticMethod = function() {
  alert(this === User);
};

User.staticMethod(); // true
```

The value of this in User.staticMethod() call is the class constructor User itself (the “object before dot” rule).

Usually, static methods are used to implement functions that belong to the class, but not to any particular object of it.

For instance, we have Article objects and need a function to compare them. A natural solution would be to add Article.compare method, like this:

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }

  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}

// usage
let articles = [
```

```
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];

articles.sort(Article.compare);
alert( articles[0].title ); // CSS
```

Here Article.compare stands “above” articles, as a means to compare them. It’s not a method of an article, but rather of the whole class.

Another example would be a so-called “factory” method. Imagine, we need few ways to create an article:

1. Create by given parameters (title, date etc).
2. Create an empty article with today’s date.
3. …or else somehow.

The first way can be implemented by the constructor. And for the second one we can make a static method of the class.

Like Article.createTodays() here:

```
class Article {
    constructor(title, date) {
        this.title = title;
        this.date = date;
    }
    static createTodays() {
        // remember, this = Article
        return new this("Today's digest", new Date());
    }
}
let article = Article.createTodays();
alert( article.title ); // Today's digest
```

Now every time we need to create today's digest, we can call Article.createTodays(). Once again, that’s not a method of an article, but a method of the whole class.

Static properties

Static properties are also possible, they look like regular class properties, but prepended by static:

```
class Article {
    static publisher = "Ilya Kantor";
}

alert( Article.publisher ); // Ilya Kantor
```

That is the same as a direct assignment to Article:
`Article.publisher = "Ilya Kantor";`

Inheritance of static properties and methods

Static properties and methods are inherited.

For instance, Animal.compare and Animal.planet in the code below are inherited and accessible as Rabbit.compare and Rabbit.planet:

```
class Animal {
    static planet = "Earth";
    constructor(name, speed) {
        this.speed = speed;
        this.name = name;
    }

    run(speed = 0) {
        this.speed += speed;
        alert(`${this.name} runs with speed ${this.speed}.`);
    }

    static compare(animalA, animalB) {
        return animalA.speed - animalB.speed;
```

```
    }
}

// Inherit from Animal

class Rabbit extends Animal {
    hide() {
        alert(`${this.name} hides!`);
    }
}

let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];

rabbits.sort(Rabbit.compare);
rabbits[0].run(); // Black Rabbit runs with speed 5.
alert(Rabbit.planet); // Earth
```

Now when we call Rabbit.compare, the inherited Animal.compare will be called.

Extending built-in classes

Built-in classes like Array, Map and others are extendable also.

For instance, here PowerArray inherits from the native Array:

```
// add one more method to it (can do more)

class PowerArray extends Array {
    isEmpty() {
        return this.length === 0;
    }
}
let arr = new PowerArray(1, 2, 5, 10, 50);
```

```
alert(arr.isEmpty()); // false

let filteredArr = arr.filter(item => item >= 10);
alert(filteredArr); // 10, 50
alert(filteredArr.isEmpty()); // false
```

Please note a very interesting thing. Built-in methods like `filter`, `map` and others – return new objects of exactly the inherited type `PowerArray`. Their internal implementation uses the object's `constructor` property for that.

In the example above,
`arr.constructor === PowerArray`

When `arr.filter()` is called, it internally creates the new array of results using exactly `arr.constructor`, not basic `Array`. That's actually very cool, because we can keep using `PowerArray` methods further on the result.

Class checking: "instanceof"

The `instanceof` operator allows to check whether an object belongs to a certain class. It also takes inheritance into account.

Such a check may be necessary in many cases. Here we'll use it for building a *polymorphic* function, the one that treats arguments differently depending on their type.

The instanceof operator

The syntax is:

`obj instanceof Class`
It returns `true` if `obj` belongs to the `Class` or a class inheriting from it.

```
class Rabbit {}
let rabbit = new Rabbit();
// is it an object of Rabbit class?
alert( rabbit instanceof Rabbit ); // true
```

It also works with constructor functions:

```
// instead of class
function Rabbit() {}
alert( new Rabbit() instanceof Rabbit ); // true
```

…And with built-in classes like Array:

```
let arr = [1, 2, 3];
alert( arr instanceof Array ); // true
alert( arr instanceof Object ); // true
```

Please note that arr also belongs to the Object class. That's because Array prototypically inherits from Object.

Error handling, "try..catch"

No matter how great we are at programming, sometimes our scripts have errors. They may occur because of our mistakes, an unexpected user input, an erroneous server response, and for a thousand other reasons.

Usually, a script "dies" (immediately stops) in case of an error, printing it to console.

But there's a syntax construct try..catch that allows to "catch" errors and, instead of dying, do something more reasonable.

The "try…catch" syntax

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

```
try {
    // code...
} catch (err) {
   // error handling
}
```

It works like this:

1. First, the code in try {...} is executed.

2. If there were no errors, then catch(err) is ignored: the execution reaches the end of try and goes on, skipping catch.
3. If an error occurs, then try execution is stopped, and the control flows to the beginning of catch(err). The err variable (can use any name for it) will contain an error object with details about what happened.

So, an error inside the try {…} block does not kill the script: we have a chance to handle it in catch.

Let's see examples. An errorless example: shows alert (1) and (2):

```
try {
    alert('Start of try runs');  // (1)
    // ...no errors here
    alert('End of try runs');   // (2)
} catch(err) {
    alert('Catch is ignored, because there are no errors'); // (3)
}
```

An example with an error: shows (1) and (3):

```
try {
    alert('Start of try runs');  // (1)
    lalala; // error, variable is not defined!
    alert('End of try (never reached)');  // (2)
} catch(err) {
    alert(`Error has occurred!`); // (3)
}
```

try..catch only works for runtime errors

For try..catch to work, the code must be runnable. In other words, it should be valid JavaScript.

It won't work if the code is syntactically wrong, for instance it has unmatched curly braces:

```
try {
    {{{{{{{{{{{{
```

```
} catch(e) {
    alert("The engine can't understand this code, it's invalid");
}
```

The JavaScript engine first reads the code, and then runs it. The errors that occur on the reading phase are called “parse-time” errors and are unrecoverable (from inside that code). That’s because the engine can’t understand the code.

So, try..catch can only handle errors that occur in the valid code. Such errors are called “runtime errors” or, sometimes, “exceptions”.

try..catch works synchronously

If an exception happens in “scheduled” code, like in setTimeout, then try..catch won’t catch it:

```
try {
    setTimeout(function() {
    noSuchVariable; // script will die here
  }, 1000);
} catch (e) {
    alert( "won't work" );
}
```

That’s because the function itself is executed later, when the engine has already left the try..catch construct.

To catch an exception inside a scheduled function, try..catch must be inside that function:

```
setTimeout(function() {
    try {
        noSuchVariable; // try..catch handles the error!
    } catch {
        alert( "error is caught here!" );
    }
```

```
}, 1000);
```

Error object

When an error occurs, JavaScript generates an object containing the details about it. The object is then passed as an argument to `catch`:

```
try {
    // ...
} catch(err) { //  the "error object", could use another word instead of err
    // ...
}
```

For all built-in errors, the error object has two main properties:

name

Error name. For instance, for an undefined variable that's `"ReferenceError"`.

message

Textual message about error details.

There are other non-standard properties available in most environments. One of most widely used and supported is:

stack

Current call stack: a string with information about the sequence of nested calls that led to the error. Used for debugging purposes.

For instance:

```
try {
    lalala; // error, variable is not defined!
} catch(err) {
    alert(err.name); // ReferenceError
    alert(err.message); // lalala is not defined
    alert(err.stack); // ReferenceError: lalala is not defined at (...call stack)
```

```
    // Can also show an error as a whole
    // The error is converted to string as "name: message"
    alert(err); // ReferenceError: lalala is not defined
}
```

Let’s explore a real-life use case of try..catch.

As we already know, JavaScript supports the JSON.parse(str) method to read JSON-encoded values.

Usually it’s used to decode data received over the network, from the server or another source.

We receive it and call JSON.parse like this:

```
let json = '{"name":"John", "age": 30}'; // data from the server
let user = JSON.parse(json); // convert the text representation to JS object
// now user is an object with properties from the string
alert( user.name ); // John
alert( user.age );  // 30
```

You can find more detailed information about JSON in the JSON methods, toJSON chapter.

If json is malformed, JSON.parse generates an error, so the script “dies”.

Should we be satisfied with that? Of course not!

This way, if something’s wrong with the data, the visitor will never know that (unless they open the developer console). And people really don’t like when something “just dies” without any error message.

Let’s use try..catch to handle the error:

```
let json = "{ bad json }";
try {
    let user = JSON.parse(json); // <-- when an error occurs...
    alert( user.name ); // doesn't work
} catch (e) {
    // ...the execution jumps here
```

```
    alert( "Our apologies, the data has errors, we'll try to request it one more time." );
    alert( e.name );
    alert( e.message );
}
```

Throwing our own errors

What if json is syntactically correct, but doesn't have a required name property?

```
let json = '{ "age": 30 }'; // incomplete data
try {
    let user = JSON.parse(json); // <-- no errors
    alert( user.name ); // no name!
} catch (e) {
    alert( "doesn't execute" );
}
```

Here JSON.parse runs normally, but the absence of name is actually an error for us. To unify error handling, we'll use the throw operator.

"Throw" operator

The throw operator generates an error. The syntax is:

```
throw <error object>
```

Technically, we can use anything as an error object. That may be even a primitive, like a number or a string, but it's better to use objects, preferably with name and message properties (to stay somewhat compatible with built-in errors).

JavaScript has many built-in constructors for standard errors: Error, SyntaxError, ReferenceError, TypeError and others. We can use them to create error objects as well.

Their syntax is:

```
let error = new Error(message);
// or
let error = new SyntaxError(message);
```

```
let error = new ReferenceError(message);
// …
```

For built-in errors (not for any objects, just for errors), the name property is exactly the name of the constructor. And message is taken from the argument.

For instance:

```
let error = new Error("Things happen o_O");
alert(error.name); // Error
alert(error.message); // Things happen o_O
```

Let’s see what kind of error JSON.parse generates:

```
try {
    JSON.parse("{ bad json o_O }");
} catch(e) {
    alert(e.name); // SyntaxError
    alert(e.message); // Unexpected token b in JSON at position 2
}
```

As we can see, that’s a SyntaxError.

And in our case, the absence of name is an error, as users must have a name.

So let’s throw it:

```
let json = '{ "age": 30 }'; // incomplete data
try {
    let user = JSON.parse(json); // <-- no errors
    if (!user.name) {
        throw new SyntaxError("Incomplete data: no name"); // (*)
    }
    alert( user.name );
} catch(e) {
    alert( "JSON Error: " + e.message ); // JSON Error: Incomplete data: no name
```

```
}
```

In the line (*), the throw operator generates a SyntaxError with the given message, the same way as JavaScript would generate it itself. The execution of try immediately stops and the control flow jumps into catch.

Now catch became a single place for all error handling: both for JSON.parse and other cases.

Rethrowing

In the example above we use `try..catch` to handle incorrect data. But is it possible that *another unexpected error* occurs within the `try {...}` block? Like a programming error (variable is not defined) or something else, not just this “incorrect data” thing.

For example:

```
let json = '{ "age": 30 }'; // incomplete data
try {
    user = JSON.parse(json); // <-- forgot to put "let" before user
    // ...
} catch(err) {
    alert("JSON Error: " + err); // JSON Error: ReferenceError: user is not defined
    // (no JSON Error actually)
}
```

Of course, everything’s possible! Programmers do make mistakes. Even in open-source utilities used by millions for decades – suddenly a bug may be discovered that leads to terrible hacks.

In our case, `try..catch` is meant to catch “incorrect data” errors. But by its nature, `catch` gets *all* errors from `try`. Here it gets an unexpected error, but still shows the same `"JSON Error"` message. That’s wrong and also makes the code more difficult to debug.

Fortunately, we can find out which error we get, for instance from its `name`:

```
try {
    user = { /*...*/ };
} catch(e) {
    alert(e.name); // "ReferenceError" for accessing an undefined variable
```

}

The rule is simple:

Catch should only process errors that it knows and “rethrow” all others.

The “rethrowing” technique can be explained in more detail as:

1. Catch gets all errors.
2. In the `catch(err) {...}` block we analyze the error object `err`.
3. If we don’t know how to handle it, we do `throw err`.

In the code below, we use rethrowing so that `catch` only handles `SyntaxError`:

```
let json = '{ "age": 30 }'; // incomplete data
try {
    let user = JSON.parse(json);
    if (!user.name) {
        throw new SyntaxError("Incomplete data: no name");
    }
    blabla(); // unexpected error
    alert( user.name );
} catch(e) {
    if (e.name == "SyntaxError") {
        alert( "JSON Error: " + e.message );
    } else {
        throw e; // rethrow (*)
    }
}
```

The error throwing on line (*) from inside `catch` block “falls out” of `try..catch` and can be either caught by an outer `try..catch` construct (if it exists), or it kills the script.

So the `catch` block actually handles only errors that it knows how to deal with and “skips” all others.

The example below demonstrates how such errors can be caught by one more level of `try..catch`:

```
function readData() {
```

```
    let json = '{ "age": 30 }';
    try {
        // ...
        blabla(); // error!
    } catch (e) {
        // ...
        if (e.name != 'SyntaxError') {
            throw e; // rethrow (don't know how to deal with it)
        }
    }
}

try {
    readData();
} catch (e) {
    alert( "External catch got: " + e ); // caught it!
}
```

Here readData only knows how to handle SyntaxError, while the outer try..catch knows how to handle everything.

try…catch…finally

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(e) {
    ... handle errors ...
```

```
} finally {
    ... execute always ...
}
```

Try running this code:

```
try {
    alert( 'try' );
    if (confirm('Make an error?')) BAD_CODE();
} catch (e) {
    alert( 'catch' );
} finally {
    alert( 'finally' );
}
```

The code has two ways of execution:

1. If you answer “Yes” to “Make an error?”, then `try -> catch -> finally`.
2. If you say “No”, then `try -> finally`.

The `finally` clause is often used when we start doing something and want to finalize it in any case of outcome.

finally and return

The `finally` clause works for *any* exit from `try..catch`. That includes an explicit return.

In the example below, there’s a `return` in `try`. In this case, finally is executed just before the control returns to the outer code.

```
function func() {
    try {
        return 1;
    } catch (e) {
        /* ... */
    } finally {
        alert( 'finally' );
```

```
    }
}
alert( func() ); // first works alert from finally, and then this one
```

try..finally

The try..finally construct, without catch clause, is also useful. We apply it when we don't want to handle errors here (let them fall through), but want to be sure that processes that we started are finalized.

```
function func() {
    // start doing something that needs completion (like measurements)
    try {
        // ...
    } finally {
        // complete that thing even if all dies
    }
}
```

Global catch

Environment-specific

The information from this section is not a part of the core JavaScript.

Let's imagine we've got a fatal error outside of try..catch, and the script died. Like a programming error or some other terrible thing.

Is there a way to react on such occurrences? We may want to log the error, show something to the user (normally they don't see error messages), etc.

There is none in the specification, but environments usually provide it, because it's really useful. For instance, Node.js has `process.on("uncaughtException")` for that. And in the browser we can assign a function to the special window.onerror property, that will run in case of an uncaught error.

The syntax:

```
window.onerror = function(message, url, line, col, error) {
    // ...
};
```

message Error message.

url URL of the script where error happened.

line, col Line and column numbers where error happened.

error Error object.

For instance

```
<script>
    window.onerror = function(message, url, line, col, error) {
        alert(`${message}\n At ${line}:${col} of ${url}`);
    };

    function readData() {
        badFunc(); // Whoops, something went wrong!
    }
    readData();
</script>
```

The role of the global handler window.onerror is usually not to recover the script execution – that's probably impossible in case of programming errors, but to send the error message to developers.

There are also web-services that provide error-logging for such cases, like https://errorception.com or http://www.muscula.com.

They work like this:

1. We register at the service and get a piece of JS (or a script URL) from them to insert on pages.

2. That JS script sets a custom window.onerror function.
3. When an error occurs, it sends a network request about it to the service.
4. We can log in to the service web interface and see errors.

Custom errors, extending Error

When we develop something, we often need our own error classes to reflect specific things that may go wrong in our tasks. For errors in network operations we may need HttpError, for database operations DbError, for searching operations NotFoundError and so on.

Our errors should support basic error properties like message, name and, preferably, stack. But they also may have other properties of their own, e.g. HttpError objects may have a statusCode property with a value like 404 or 403 or 500.

JavaScript allows to use throw with any argument, so technically our custom error classes don't need to inherit from Error. But if we inherit, then it becomes possible to use obj instanceof Error to identify error objects. So it's better to inherit from it.

As the application grows, our own errors naturally form a hierarchy. For instance, HttpTimeoutError may inherit from HttpError, and so on.

Extending Error

As an example, let's consider a function readUser(json) that should read JSON with user data.
Here's an example of how a valid json may look:

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

Internally, we'll use JSON.parse. If it receives malformed json, then it throws SyntaxError. But even if json is syntactically correct, that doesn't mean that it's a valid user, right? It may miss the necessary data. For instance, it may not have name and age properties that are essential for our users.

Our function readUser(json) will not only read JSON, but check ("validate") the data. If there are no required fields, or the format is wrong, then that's an error. And that's not a SyntaxError, because the data is syntactically correct, but another kind of error. We'll call it ValidationError

and create a class for it. An error of that kind should also carry the information about the offending field.

Our `ValidationError` class should inherit from the built-in `Error` class.

That class is built-in, but here's its approximate code so we can understand what we're extending:

```
// The "pseudocode" for the built-in Error class defined by JavaScript itself
class Error {
    constructor(message) {
        this.message = message;
        this.name = "Error"; // (different names for different built-in error classes)
        this.stack = <call stack>; // non-standard, but most environments support it
    }
}
```

Now let's inherit `ValidationError` from it and try it in action:

```
class ValidationError extends Error {
    constructor(message) {
        super(message); // (1)
        this.name = "ValidationError"; // (2)
    }
}

function test() {
    throw new ValidationError("Whoops!");
}

try {
    test();
} catch(err) {
    alert(err.message); // Whoops!
    alert(err.name); // ValidationError
```

```
    alert(err.stack); // a list of nested calls with line numbers for each
}
```

Please note: in the line (1) we call the parent constructor. JavaScript requires us to call `super` in the child constructor, so that's obligatory. The parent constructor sets the `message` property.

The parent constructor also sets the `name` property to `"Error"`, so in the line (2) we reset it to the right value.

Let's try to use it in `readUser(json)`:

```
class ValidationError extends Error {
    constructor(message) {
        super(message);
        this.name = "ValidationError";
    }
}

// Usage
function readUser(json) {
    let user = JSON.parse(json);
    if (!user.age) {
        throw new ValidationError("No field: age");
    }
    if (!user.name) {
        throw new ValidationError("No field: name");
    }
   return user;
}

// Working example with try..catch
try {
    let user = readUser('{ "age": 25 }');
} catch (err) {
```

```
    if (err instanceof ValidationError) {
        alert("Invalid data: " + err.message); // Invalid data: No field: name
    } else if (err instanceof SyntaxError) { // (*)
        alert("JSON Syntax Error: " + err.message);
    } else {
        throw err; // unknown error, rethrow it (**)
    }
}
```

The `try..catch` block in the code above handles both our `ValidationError` and the built-in `SyntaxError` from `JSON.parse`.

Please take a look at how we use `instanceof` to check for the specific error type in the line `(*)`.

We could also look at `err.name`, like this:

```
// ...
// instead of (err instanceof SyntaxError)
} else if (err.name == "SyntaxError") { // (*)
// ...
```

Introduction : Callbacks

Many actions in JavaScript are *asynchronous*. In other words, we initiate them now, but they finish later.

For instance, we can schedule such actions using `setTimeout`.

There are other real-world examples of asynchronous actions, e.g. loading scripts and modules (we'll cover them in later chapters).

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);
}
```

It appends to the document the new, dynamically created, tag <script src="…"> with given src. The browser automatically starts loading it and executes when complete.

We can use this function like this:

```
// load and execute the script at the given path
loadScript('/my/script.js');
```

The script is executed “asynchronously”, as it starts loading now, but runs later, when the function has already finished.

If there’s any code below loadScript(…), it doesn’t wait until the script loading finishes.

```
loadScript('/my/script.js');
// the code below loadScript
// doesn't wait for the script loading to finish
// ...
```

Let’s say we need to use the new script as soon as it loads. It declares new functions, and we want to run them.

But if we do that immediately after the loadScript(…) call, that wouldn’t work:

```
loadScript('/my/script.js'); // the script has "function newFunction() {…}"
newFunction(); // no such function!
```

Naturally, the browser probably didn’t have time to load the script. As of now, the loadScript function doesn’t provide a way to track the load completion. The script loads and eventually runs, that’s all. But we’d like to know when it happens, to use new functions and variables from that 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);
}
```

Now if we want to call new functions from the script, we should write that in the callback:

```
loadScript('/my/script.js', function() {
    // the callback runs after the script is loaded
    newFunction(); // so now it works
    ...
});
```

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

Here’s a runnable example with a real script:

```
function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(script);
    document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script => {
    alert(`Cool, the script ${script.src} is loaded`);
    alert( _ ); // function declared in the loaded script
});
```

That's called a "callback-based" style of asynchronous programming. A function that does something asynchronously should provide a `callback` argument where we put the function to run after it's complete.

Here we did it in `loadScript`, but of course it's a general approach.

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`);
    });
});
```

After the outer `loadScript` is complete, the callback initiates the inner one.

What if we want one more script…?

```
loadScript('/my/script.js', function(script) {
    loadScript('/my/script2.js', function(script) {
        loadScript('/my/script3.js', function(script) {
            // ...continue after all scripts are loaded
        });
    })
});
```

So, every new action is inside a callback. That's fine for few actions, but not good for many, so we'll see other variants soon.

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.

Pyramid of Doom

From the first look, it's a viable way of asynchronous coding. And indeed it is. For one or maybe two nested calls it looks fine.

But for multiple asynchronous actions that follow one after another we'll have code like this:

```
loadScript('1.js', function(error, script) {
    if (error) {
        handleError(error);
    } else {
        // ...
        loadScript('2.js', function(error, script) {
            if (error) {
                handleError(error);
            } else {
                // ...
                loadScript('3.js', function(error, script) {
                    if (error) {
                        handleError(error);
                    } else {
                        // ...continue after all scripts are loaded (*)
                    }
                });
            }
        })
    }
});
```

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of `...` that may include more loops, conditional statements and so on.

That's sometimes called "callback hell" or "pyramid of doom."

The "pyramid" of nested calls grows to the right with every asynchronous action. Soon it spirals out of control.

So this way of coding isn't very good.

We can try to alleviate the problem by making every action a standalone function, like this:

```
loadScript('1.js', step1);
function step1(error, script) {
    if (error) {
        handleError(error);
    } else {
        // ...
        loadScript('2.js', step2);
    }
}

function step2(error, script) {
    if (error) {
        handleError(error);
    } else {
        // ...
        loadScript('3.js', step3);
    }
}


function step3(error, script) {
    if (error) {
        handleError(error);
    } else {
        // ...continue after all scripts are loaded (*)
    }
};
```

See? It does the same, and there's no deep nesting now because we made every action a separate top-level function.

It works, but the code looks like a torn apart spreadsheet. It's difficult to read, and you probably noticed that one needs to eye-jump between pieces while reading it. That's inconvenient, especially if the reader is not familiar with the code and doesn't know where to eye-jump.

Also, the functions named `step*` are all of single use, they are created only to avoid the "pyramid of doom." No one is going to reuse them outside of the action chain. So there's a bit of namespace cluttering here.

We'd like to have something better.

Luckily, there are other ways to avoid such pyramids. One of the best ways is to use "promises," described in the next chapter.

Promise

A Promise is an object representing the eventual completion or failure of an asynchronous operation. Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

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.
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 finished successfully, with result value.
- reject(error) — if an error 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) called or error when reject(error) is called.

Here's an example of a promise constructor and a simple executor function with "producing code" that takes time (via setTimeout):

```
let promise = new Promise(function(resolve, reject) {
    // the function is executed automatically when the promise is constructed
    // after 1 second signal that the job is done with the result "done"
    setTimeout(() => resolve("done"), 1000);
});
```

We can see two things by running the code above:

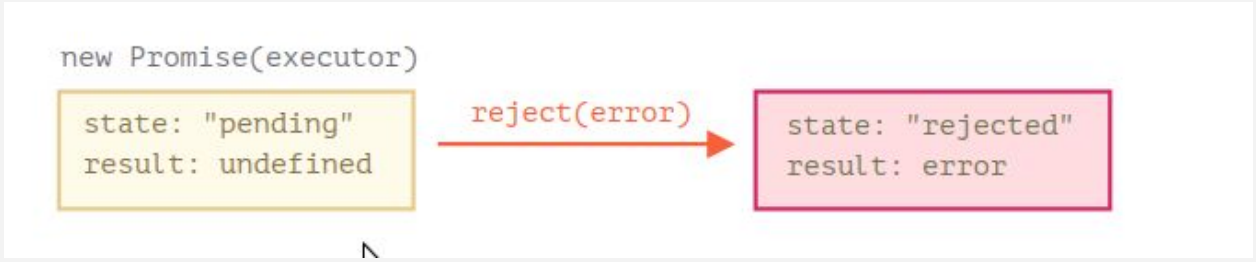
1. The executor is called automatically and immediately (by new Promise).
2. The executor receives two arguments: resolve and reject. These functions are pre-defined by the JavaScript engine, so we don't need to create them. We should only call one of them when ready.
   After one second of "processing" the executor calls resolve("done") to produce the result. This changes the state of the promise object:

That was an example of a successful job completion, a “fulfilled promise”.

And now an example of the executor rejecting the promise with an error:

```
let promise = new Promise(function(resolve, reject) {
    // after 1 second signal that the job is finished with an error
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

The call to reject(...) moves the promise object to "rejected" state:



To summarize, the executor should perform a job (usually something that takes time) and then call resolve or reject to change the state of the corresponding promise object.

A promise that is either resolved or rejected is called “settled”, as opposed to an initially “pending” promise.

There can be only a single result or an error

The executor should call only one resolve or one reject. Any state change is final. All further calls of resolve and reject are ignored:

```
let promise = new Promise(function(resolve, reject) {
    resolve("done");
```

```
    reject(new Error("…")); // ignored
    setTimeout(() => resolve("…")); // ignored
});
```

The idea is that a job done by the executor may have only one result or an error. Also, resolve/reject expect only one argument (or none) and will ignore additional arguments.

Immediately calling resolve/reject

> In practice, an executor usually does something asynchronously and calls resolve/reject after some time, but it doesn't have to. We also can call resolve or reject immediately, like this:

```
let promise = new Promise(function(resolve, reject) {
    // not taking our time to do the job
    resolve(123); // immediately give the result: 123
});
```

For instance, this might happen when we start to do a job but then see that everything has already been completed and cached.

That's fine. We immediately have a resolved promise.

Consumers: then, catch, finally

A Promise object serves as a link between the executor (the "producing code" or "singer") and the consuming functions (the "fans"), which will receive the result or error. Consuming functions can be registered (subscribed) using methods .then, .catch and .finally.

then

The most important, fundamental one is .then. The syntax is:

```
promise.then(
    function(result) { /* handle a successful result */ },
```

```
    function(error) { /* handle an error */ }
);
```

The first argument of .then is a function that runs when the promise is resolved, and receives the result.

The second argument of .then is a function that runs when the promise is rejected, and receives the error.

For instance, here's a reaction to a successfully resolved promise:

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
    result => alert(result), // shows "done!" after 1 second
    error => alert(error) // doesn't run
);
```

The first function was executed.

And in the case of a rejection, the second one:

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then (
    result => alert(result), // doesn't run
    error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

If we're interested only in successful completions, then we can provide only one function argument to .then:

```
let promise = new Promise(resolve => {
    setTimeout(() => resolve("done!"), 1000);
});
promise.then(alert); // shows "done!" after 1 second
```

catch

If we're only interested in errors, then we can use null as the first argument: .then(null, errorHandlingFunction). Or we can use .catch(errorHandlingFunction), which is exactly the same:

```
let promise = new Promise((resolve, reject) => {
    setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

The call .catch(f) is a complete analog of .then(null, f), it's just a shorthand.

finally

Just like there's a finally clause in a regular try {...} catch {...}, there's finally in promises.

The call .finally(f) is similar to .then(f, f) in the sense that f always runs when the promise is settled: be it resolve or reject.

finally is a good handler for performing cleanup, e.g. stopping our loading indicators, as they are not needed anymore, no matter what the outcome is.

```
new Promise((resolve, reject) => {
    /* do something that takes time, and then call resolve/reject */
})

// runs when the promise is settled, doesn't matter successfully or not
```

```
.finally(() => stop loading indicator)
.then(result => show result, err => show error)
```

It's not exactly an alias of then(f,f) though. There are several important differences:

1. A finally handler has no arguments. In finally we don't know whether the promise is successful or not. That's all right, as our task is usually to perform "general" finalizing procedures.
2. A finally handler passes through results and errors to the next handler.
   For instance, here the result is passed through finally to then:

```
new Promise((resolve, reject) => {
    setTimeout(() => resolve("result"), 2000)
})
.finally(() => alert("Promise ready"))
.then(result => alert(result)); //  .then handles the result
```

And here there's an error in the promise, passed through finally to catch:

```
new Promise((resolve, reject) => {
    throw new Error("error");
})
 .finally(() => alert("Promise ready"))
.catch(err => alert(err));  // <-- .catch handles the error object
```

That's very convenient, because finally is not meant to process a promise result. So it passes it through. Last, but not least, .finally(f) is a more convenient syntax than .then(f, f): no need to duplicate the function f.

On settled promises handlers run immediately

If a promise is pending, .then/catch/finally handlers wait for it. Otherwise, if a promise has already settled, they execute immediately:

Example: loadScript

We've got the loadScript function for loading a script from the previous chapter.

Here's the callback-based variant, just to remind us of it:

```
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);
}
```

Let's rewrite it using Promises.

The new function loadScript will not require a callback. Instead, it will create and return a Promise object that resolves when the loading is complete. The outer code can add handlers (subscribing functions) to it using .then:

```
function loadScript(src) {
    return new Promise(function(resolve, reject) {
        let script = document.createElement('script');
        script.src = src;
        script.onload = () => resolve(script);
        script.onerror = () => reject(new Error(`Script load error for ${src}`));
        document.head.append(script);
    });
}
```

Usage:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.11/lodash.js");
promise.then(
    script => alert(`${script.src} is loaded!`),
    error => alert(`Error: ${error.message}`)
);
promise.then(script => alert('Another handler...'));
```

We can immediately see a few benefits over the callback-based pattern:

Promises	Callbacks
Promises allow us to do things in the natural order. First, we run `loadScript(script)`, and `.then` we write what to do with the result.	We must have a `callback` function at our disposal when calling `loadScript(script, callback)`. In other words, we must know what to do with the result *before* `loadScript` is called.
We can call `.then` on a Promise as many times as we want. Each time, we're adding a new "fan", a new subscribing function, to the "subscription list". More about this in the next chapter: Promises chaining.	There can be only one callback.

Promises chaining

Promises provide a couple of recipes to do that. It looks like this:

```
new Promise(function(resolve, reject) {
    setTimeout(() => resolve(1), 1000); // (*)
}).then(function(result) { // (**)
    alert(result); // 1
    return result * 2;
}).then(function(result) { // (***)
    alert(result); // 2
    return result * 2;
}).then(function(result) {
    alert(result); // 4
```

```
    return result * 2;
});
```

The idea is that the result is passed through the chain of .then handlers.

When a handler returns a value, it becomes the result of that promise, so the next .then is called with it.

A classic newbie error: technically we can also add many .then to a single promise. This is not chaining.

For example:

```
let promise = new Promise(function(resolve, reject) {
    setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
    alert(result); // 1
    return result * 2;
});

promise.then(function(result) {
    alert(result); // 1
    return result * 2;
});

promise.then(function(result) {
    alert(result); // 1
    return result * 2;
});
```

What we did here is just several handlers to one promise. They don’t pass the result to each other; instead they process it independently.

All .then on the same promise get the same result – the result of that promise. So in the code above all alert show the same: 1.

Returning promises

A handler, used in .then(handler) may create and return a promise.

In that case further handlers wait until it settles, and then get its result.

For instance:

```
new Promise(function(resolve, reject) {
    setTimeout(() => resolve(1), 1000);
}).then(function(result) {
    alert(result); // 1
    return new Promise((resolve, reject) => { // (*)
        setTimeout(() => resolve(result * 2), 1000);
    });
}).then(function(result) { // (**)
    alert(result); // 2
    return new Promise((resolve, reject) => {
        setTimeout(() => resolve(result * 2), 1000);
    });
}).then(function(result) {
    alert(result); // 4
});
```

Here the first .then shows 1 and returns new Promise(…) in the line (*). After one second it resolves, and the result (the argument of resolve, here it's result * 2) is passed on to handler of the second .then. That handler is in the line (**), it shows 2 and does the same thing.

So the output is the same as in the previous example: 1 → 2 → 4, but now with 1 second delay between alert calls.

Returning promises allows us to build chains of asynchronous actions.

Example: loadScript

Let’s use this feature with the promisified loadScript, defined in the previous chapter, to load scripts one by one, in sequence:

```
function loadScript(src) {
    return new Promise(function(resolve, reject) {
        let script = document.createElement('script');
        script.src = src;
        script.onload = () => resolve(script);
        script.onerror = () => reject(new Error(`Script load error for ${src}`));
        document.head.append(script);
    });
}

loadScript("/article/promise-chaining/one.js")
    .then(function(script) {
        return loadScript("/article/promise-chaining/two.js");
    })
    .then(function(script) {
        return loadScript("/article/promise-chaining/three.js");
    })
    .then(function(script) {
        // use functions declared in scripts
        // to show that they indeed loaded
        one();
        two();
        three();
    });
```

This code can be made bit shorter with arrow functions:

```
loadScript("/article/promise-chaining/one.js")
    .then(script => loadScript("/article/promise-chaining/two.js"))
```

```
  .then(script => loadScript("/article/promise-chaining/three.js"))
  .then(script => {
    // scripts are loaded, we can use functions declared there
    one();
    two();
    three();
  });
```

Bigger example: fetch

In frontend programming promises are often used for network requests. So let's see an extended example of that.

We'll use the fetch method to load the information about the user from the remote server. It has a lot of optional parameters covered in separate chapters, but the basic syntax is quite simple:

```
let promise = fetch(url);
```

This makes a network request to the `url` and returns a promise. The promise resolves with a response object when the remote server responds with headers, but *before the full response is downloaded*.

To read the full response, we should call the method `response.text()`: it returns a promise that resolves when the full text is downloaded from the remote server, with that text as a result.

The code below makes a request to `user.json` and loads its text from the server:

```
fetch('/article/promise-chaining/user.json')
  // .then below runs when the remote server responds
  .then(function(response) {
    // response.text() returns a new promise that resolves with the full response text
    // when it loads
    return response.text();
  })
  .then(function(text) {
    // ...and here's the content of the remote file
```

```
        alert(text); // {"name": "iliakan", isAdmin: true}
    });
```

There is also a method response.json() that reads the remote data and parses it as JSON. In our case that's even more convenient, so let's switch to it.

We'll also use arrow functions for brevity:

```
// same as above, but response.json() parses the remote content as JSON
fetch('/article/promise-chaining/user.json')
    .then(response => response.json())
    .then(user => alert(user.name)); // iliakan, got user name
```

Now let's do something with the loaded user.

For instance, we can make one more requests to GitHub, load the user profile and show the avatar:

```
// Make a request for user.json
fetch('/article/promise-chaining/user.json')
    .then(response => response.json())
    // Make a request to GitHub
    .then(user => fetch(`https://api.github.com/users/${user.name}`))
    .then(response => response.json())
    .then(githubUser => {
        let img = document.createElement('img');
        img.src = githubUser.avatar_url;
        img.className = "promise-avatar-example";
        document.body.append(img);
        setTimeout(() => img.remove(), 3000); // (*)
    });
```

The code works; see comments about the details. However, there's a potential problem in it, a typical error for those who begin to use promises.

Look at the line (*): how can we do something *after* the avatar has finished showing and gets removed? For instance, we'd like to show a form for editing that user or something else. As of now, there's no way.

To make the chain extendable, we need to return a promise that resolves when the avatar finishes showing.

```
fetch('/article/promise-chaining/user.json')
    .then(response => response.json())
    .then(user => fetch(`https://api.github.com/users/${user.name}`))
    .then(response => response.json())
     .then(githubUser => new Promise(function(resolve, reject) { // (*)
          let img = document.createElement('img');
          img.src = githubUser.avatar_url;
          img.className = "promise-avatar-example";
          document.body.append(img);
          setTimeout(() => {
              img.remove();
              resolve(githubUser); // (**)
          }, 3000);
     }))

     // triggers after 3 seconds
     .then(githubUser => alert(`Finished showing ${githubUser.name}`));
```

That is, the .then handler in line (*) now returns new Promise, that becomes settled only after the call of resolve(githubUser) in setTimeout (**). The next .then in the chain will wait for that.

As a good practice, an asynchronous action should always return a promise. That makes it possible to plan actions after it; even if we don't plan to extend the chain now, we may need it later.

Error handling with promises

Promise chains are great at error handling. When a promise rejects, the control jumps to the closest rejection handler. That's very convenient in practice.

For instance, in the code below the URL to fetch is wrong (no such site) and .catch handles the error:

```
fetch('https://no-such-server.blabla') // rejects
    .then(response => response.json())
    .catch(err => alert(err)) // TypeError: failed to fetch (the text may vary)
```

As you can see, the .catch doesn't have to be immediate. It may appear after one or maybe several .then.

Or, maybe, everything is all right with the site, but the response is not valid JSON. The easiest way to catch all errors is to append .catch to the end of chain:

```
fetch('/article/promise-chaining/user.json')
    .then(response => response.json())
    .then(user => fetch(`https://api.github.com/users/${user.name}`))
    .then(response => response.json())
    .then(githubUser => new Promise((resolve, reject) => {
        let img = document.createElement('img');
        img.src = githubUser.avatar_url;
        img.className = "promise-avatar-example";
        document.body.append(img);
        setTimeout(() => {
            img.remove();
            resolve(githubUser);
        }, 3000);
    }))
    .catch(error => alert(error.message));
```

Normally, such .catch doesn't trigger at all. But if any of the promises above rejects (a network problem or invalid json or whatever), then it would catch it.

Implicit try…catch

The code of a promise executor and promise handlers has an "invisible try..catch" around it. If an exception happens, it gets caught and treated as a rejection.

For instance, this code:

```
new Promise((resolve, reject) => {
    throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

…Works exactly the same as this:

```
new Promise((resolve, reject) => {
    reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

The "invisible try..catch" around the executor automatically catches the error and turns it into rejected promise.

This happens not only in the executor function, but in its handlers as well. If we throw inside a .then handler, that means a rejected promise, so the control jumps to the nearest error handler.

Here's an example:

```
new Promise((resolve, reject) => {
    resolve("ok");
}).then((result) => {
    throw new Error("Whoops!"); // rejects the promise
}).catch(alert); // Error: Whoops!
```

This happens for all errors, not just those caused by the throw statement. For example, a programming error:

```
new Promise((resolve, reject) => {
    resolve("ok");
}).then((result) => {
    blabla(); // no such function
}).catch(alert); // ReferenceError: blabla is not defined
```

The final .catch not only catches explicit rejections, but also occasional errors in the handlers above.

Rethrowing

As we already noticed, .catch at the end of the chain is similar to try..catch. We may have as many .then handlers as we want, and then use a single .catch at the end to handle errors in all of them.

In a regular try..catch we can analyze the error and maybe rethrow it if it can't be handled. The same thing is possible for promises.

If we throw inside .catch, then the control goes to the next closest error handler. And if we handle the error and finish normally, then it continues to the closest successful .then handler.

In the example below the .catch successfully handles the error:

```
// the execution: catch -> then
new Promise((resolve, reject) => {
    throw new Error("Whoops!");
}).catch(function(error) {
    alert("The error is handled, continue normally");
}).then(() => alert("Next successful handler runs"));
```

Here the .catch block finishes normally. So the next successful .then handler is called.

In the example below we see the other situation with .catch. The handler (*) catches the error and just can't handle it (e.g. it only knows how to handle URIError), so it throws it again:

```
// the execution: catch -> catch -> then

new Promise((resolve, reject) => {
    throw new Error("Whoops!");
}).catch(function(error) { // (*)
    if (error instanceof URIError) {
        // handle it
    } else {
```

```
        alert("Can't handle such error");
        throw error; // throwing this or another error jumps to the next catch
    }
}).then(function() {
    /* doesn't run here */
}).catch(error => { // (**)
    alert(`The unknown error has occurred: ${error}`);
    // don't return anything => execution goes the normal way
});
```

The execution jumps from the first `.catch` (*) to the next one (**) down the chain.

Unhandled rejections

What happens when an error is not handled? For instance, we forgot to append `.catch` to the end of the chain, like here:

```
new Promise(function() {
    noSuchFunction(); // Error here (no such function)
}).then(() => {
}); // without .catch at the end!
```

In case of an error, the promise becomes rejected, and the execution should jump to the closest rejection handler. But there is none. So the error gets “stuck”. There’s no code to handle it.

In practice, just like with regular unhandled errors in code, it means that something has gone terribly wrong.

What happens when a regular error occurs and is not caught by `try..catch`? The script dies with a message in the console. A similar thing happens with unhandled promise rejections.

The JavaScript engine tracks such rejections and generates a global error in that case. You can see it in the console if you run the example above.

In the browser we can catch such errors using the event `unhandledrejection`:

```
window.addEventListener('unhandledrejection', function(event) {
```

```
    // the event object has two special properties:
    alert(event.promise); // [object Promise] - the promise that generated the error
    alert(event.reason); // Error: Whoops! - the unhandled error object
});
new Promise(function() {
    throw new Error("Whoops!");
}); // no catch to handle the error
```

The event is the part of the HTML standard.

If an error occurs, and there's no `.catch`, the `unhandledrejection` handler triggers, and gets the `event` object with the information about the error, so we can do something.

Usually such errors are unrecoverable, so our best way out is to inform the user about the problem and probably report the incident to the server.

In non-browser environments like Node.js there are other ways to track unhandled errors.

Promise API

There are 5 static methods in the `Promise` class. We'll quickly cover their use cases here.

Promise.all

Let's say we want many promises to execute in parallel and wait until all of them are ready.

For instance, download several URLs in parallel and process the content once they are all done.

That's what `Promise.all` is for. The syntax is:

```
let promise = Promise.all([...promises...]);
```

`Promise.all` takes an array of promises (it technically can be any iterable, but is usually an array) and returns a new promise.

The new promise resolves when all listed promises are settled, and the array of their results becomes its result.

For instance, the Promise.all below settles after 3 seconds, and then its result is an array [1, 2, 3]:

```
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 3000)), // 1
  new Promise(resolve => setTimeout(() => resolve(2), 2000)), // 2
  new Promise(resolve => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 when promises are ready: each promise contributes an array member
```

Please note that the order of the resulting array members is the same as in its source promises. Even though the first promise takes the longest time to resolve, it's still first in the array of results.

If any of the promises is rejected, the promise returned by Promise.all immediately rejects with that error.

For instance:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Here the second promise rejects in two seconds. That leads to an immediate rejection of Promise.all, so .catch executes: the rejection error becomes the outcome of the entire Promise.all.

In case of an error, other promises are ignored

> If one promise rejects, Promise.all immediately rejects, completely forgetting about the other ones in the list. Their results are ignored.
>
> For example, if there are multiple fetch calls, like in the example above, and one fails, the others will still continue to execute, but Promise.all won't watch them anymore. They will probably settle, but their results will be ignored.

Promise.all does nothing to cancel them, as there's no concept of "cancellation" in promises. In another chapter we'll cover AbortController that can help with that, but it's not a part of the Promise API.

Promise.all(iterable) allows non-promise "regular" values initerable

Normally, Promise.all(...) accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's passed to the resulting array "as is".

For instance, here the results are [1, 2, 3]:

```
Promise.all([
    new Promise((resolve, reject) => {
        setTimeout(() => resolve(1), 1000)
    }),
    2,
    3
]).then(alert); // 1, 2, 3
```

So we are able to pass ready values to Promise.all where convenient.

Promise.allSettled

Promise.all rejects as a whole if any promise rejects. That's good for "all or nothing" cases, when we need *all* results successful to proceed:

```
Promise.all([
    fetch('/template.html'),
    fetch('/style.css'),
    fetch('/data.json')
]).then(render); // render method needs results of all fetches
```

Promise.allSettled just waits for all promises to settle, regardless of the result. The resulting array has:

1. {status:"fulfilled", value:result} for successful responses,
2. {status:"rejected", reason:error} for errors.

For example, we'd like to fetch the information about multiple users. Even if one request fails, we're still interested in the others.

Let's use Promise.allSettled:

```
let urls = [
    'https://api.github.com/users/iliakan',
    'https://api.github.com/users/remy',
    'https://no-such-url'
];


Promise.allSettled(urls.map(url => fetch(url)))
    .then(results => { // (*)
        results.forEach((result, num) => {
            if (result.status == "fulfilled") {
                alert(`${urls[num]}: ${result.value.status}`);
            }
            if (result.status == "rejected") {
                alert(`${urls[num]}: ${result.reason}`);
            }
        });
    });
```

The results in the line (*) above will be:

```
[
    {status: 'fulfilled', value: ...response...},
    {status: 'fulfilled', value: ...response...},
    {status: 'rejected', reason: ...error object...}
]
```

So for each promise we get its status and value/error.

Promise.race

Similar to Promise.all, but waits only for the first settled promise and gets its result (or error).

The syntax is:
let promise = Promise.race(iterable);

For instance, here the result will be 1:

```
Promise.race([
    new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
    new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000)),
    new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

The first promise here was fastest, so it became the result. After the first settled promise “wins the race”, all further results/errors are ignored.

Promise.resolve/reject

Methods Promise.resolve and Promise.reject are rarely needed in modern code, because async/await syntax (we’ll cover it a bit later) makes them somewhat obsolete.

We cover them here for completeness and for those who can’t use async/await for some reason.

Promise.resolve

Promise.resolve(value) creates a resolved promise with the result value.

Same as:
let promise = new Promise(resolve => resolve(value));

The method is used for compatibility, when a function is expected to return a promise.

For example, the loadCached function below fetches a URL and remembers (caches) its content. For future calls with the same URL it immediately gets the previous content from cache, but uses Promise.resolve to make a promise of it, so the returned value is always a promise:

```
let cache = new Map();
function loadCached(url) {
    if (cache.has(url)) {
        return Promise.resolve(cache.get(url)); // (*)
    }
    return fetch(url)
        .then(response => response.text())
        .then(text => {
            cache.set(url,text);
            return text;
        });
}
```

We can write loadCached(url).then(…), because the function is guaranteed to return a promise. We can always use .then after loadCached. That's the purpose of Promise.resolve in the line (*).

Promise.reject

Promise.reject(error) creates a rejected promise with error.

Same as:

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

In practice, this method is almost never used.

Microtasks

Promise handlers .then/.catch/.finally are always asynchronous.

Even when a Promise is immediately resolved, the code on the lines *below* .then/.catch/.finally will still execute before these handlers.

Here's a demo:

```
let promise = Promise.resolve();
promise.then(() => alert("promise done!"));
alert("code finished"); // this alert shows first
```

If you run it, you see code finished first, and then promise done!.

That's strange, because the promise is definitely done from the beginning.

Why did the .then trigger afterwards? What's going on?

Microtasks queue

Asynchronous tasks need proper management. For that, the Ecma standard specifies an internal queue PromiseJobs, more often referred to as the "microtask queue" (ES8 term).

As stated in the specification:

- The queue is first-in-first-out: tasks enqueued first are run first.
- Execution of a task is initiated only when nothing else is running.

Or, to say more simply, when a promise is ready, its .then/catch/finally handlers are put into the queue; they are not executed yet. When the JavaScript engine becomes free from the current code, it takes a task from the queue and executes it.
That's why "code finished" in the example above shows first.

Promise handlers always go through this internal queue.

If there's a chain with multiple .then/catch/finally, then every one of them is executed asynchronously. That is, it first gets queued, then executed when the current code is complete and previously queued handlers are finished.

What if the order matters for us? How can we make code finished run after promise done?

Easy, just put it into the queue with .then:

```
Promise.resolve()
  .then(() => alert("promise done!"))
  .then(() => alert("code finished"));
```

Now the order is as intended.

Unhandled rejection

Remember the unhandledrejection event from the chapter Error handling with promises?

Now we can see exactly how JavaScript finds out that there was an unhandled rejection.

An “unhandled rejection” occurs when a promise error is not handled at the end of the microtask queue.

Normally, if we expect an error, we add .catch to the promise chain to handle it:

```
let promise = Promise.reject(new Error("Promise Failed!"));
promise.catch(err => alert('caught'));
// doesn't run: error handled
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

But if we forget to add .catch, then, after the microtask queue is empty, the engine triggers the event:

```
let promise = Promise.reject(new Error("Promise Failed!")); // Promise Failed!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

What if we handle the error later? Like this:

```
let promise = Promise.reject(new Error("Promise Failed!"));
setTimeout(() => promise.catch(err => alert('caught')), 1000);
// Error: Promise Failed!
window.addEventListener('unhandledrejection', event => alert(event.reason));
```

Now, if we run it, we'll see Promise Failed! first and then caught.

If we didn't know about the microtasks queue, we could wonder: "Why did unhandledrejection handler run? We did catch and handle the error!"

But now we understand that unhandledrejection is generated when the microtask queue is complete: the engine examines promises and, if any of them is in the "rejected" state, then the event triggers.

In the example above, .catch added by setTimeout also triggers. But it does so later, after unhandledrejection has already occurred, so it doesn't change anything.

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 makes JavaScript wait until the promise settles, and then go on with the result. That doesn't cost any CPU resources, because the 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, easier to read and write.

Can’t use await in regular functions

If we try to use await in non-async function, there would be a syntax error:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // Syntax error
}
```

We will get this error if we do not put async before a function. As said, await only works inside an async function.

Let’s take the showAvatar() example from the chapter Promises chaining and rewrite it using async/await:

1. We’ll need to replace .then calls with await.
2. Also we should make the function async for them to work.

```
async function showAvatar() {
  // read our JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // read github user
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // show the avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
```

```
    img.className = "promise-avatar-example";
    document.body.append(img);


    // wait 3 seconds
    await new Promise((resolve, reject) => setTimeout(resolve, 3000));
    img.remove();
    return githubUser;
}
showAvatar();
```

Pretty clean and easy to read, right? Much better than before.

await won’t work in the top-level code

People who are just starting to use `await` tend to forget the fact that we can’t use `await` in top-level code. For example, this will not work:

```
// syntax error in top-level code
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
```

But we can wrap it into an anonymous async function, like this:

```
(async () => {
    let response = await fetch('/article/promise-chaining/user.json');
    let user = await response.json();
    ...
})();
```

Async class methods

To declare an async class method, just prepend it with `async`:

```
class Waiter {
```

```
    async wait() {
        return await Promise.resolve(1);
    }
}
new Waiter()
    .wait()
    .then(alert); // 1
```

The meaning is the same: it ensures that the returned value is a promise and enables `await`.

Error handling

If a promise resolves normally, then `await promise` returns the result. But in the case of a rejection, it throws the error, just as if there were a `throw` statement at that line.

This code:

```
async function f() {
    await Promise.reject(new Error("Whoops!"));
}
```

…is the same as this:

```
async function f() {
    throw new Error("Whoops!");
}
```

In real situations, the promise may take some time before it rejects. In that case there will be a delay before `await` throws an error.

We can catch that error using `try..catch`, the same way as a regular `throw`:

```
async function f() {
    try {
        let response = await fetch('http://no-such-url');
    } catch(err) {
        alert(err); // TypeError: failed to fetch
    }
```

```
}
f();
```

In the case of an error, the control jumps to the `catch` block. We can also wrap multiple lines:

```
async function f() {
    try {
        let response = await fetch('/no-user-here');
        let user = await response.json();
    } catch(err) {
        // catches errors both in fetch and response.json
        alert(err);
    }
}
f();
```

If we don't have `try..catch`, then the promise generated by the call of the async function `f()` becomes rejected. We can append `.catch` to handle it:

```
async function f() {
    let response = await fetch('http://no-such-url');
}

// f() becomes a rejected promise
f().catch(alert); // TypeError: failed to fetch // (*)
```

If we forget to add `.catch` there, then we get an unhandled promise error (viewable in the console). We can catch such errors using a global `unhandledrejection` event handler as described in the chapter Error handling with promises.

async/await and promise.then/catch

When we use async/await, we rarely need .then, because await handles the waiting for us. And we can use a regular try..catch instead of .catch. That's usually (but not always) more convenient.

But at the top level of the code, when we're outside any async function, we're syntactically unable to use await, so it's a normal practice to add .then/catch to handle the final result or falling-through error, like in the line (*) of the example above.

async/await works well with Promise.all

When we need to wait for multiple promises, we can wrap them in Promise.all and then await:

```
// wait for the array of results
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

In the case of an error, it propagates as usual, from the failed promise to Promise.all, and then becomes an exception that we can catch using try..catch around the call.

Generators

Regular functions return only one, single value (or nothing).

Generators can return ("yield") multiple values, one after another, on-demand. They work great with iterables, allowing to create data streams with ease.

Generator functions

To create a generator, we need a special syntax construct: `function*`, so-called “generator function”.

It looks like this:

```
function* generateSequence() {
    yield 1;
    yield 2;
    return 3;
}
```

Generator functions behave differently from regular ones. When such a function is called, it doesn’t run its code. Instead it returns a special object, called “generator object”, to manage the execution.

Here, take a look:

```
function* generateSequence() {
    yield 1;
    yield 2;
    return 3;
}
// "generator function" creates "generator object"
let generator = generateSequence();
alert(generator); // [object Generator]
```

The function code execution hasn’t started yet:

The main method of a generator is `next()`. When called, it runs the execution until the nearest `yield <value>` statement (`value` can be omitted, then it’s `undefined`). Then the function execution pauses, and the yielded `value` is returned to the outer code.

The result of `next()` is always an object with two properties:

- `value`: the yielded value.
- `done`: `true` if the function code has finished, otherwise `false`.

For instance, here we create the generator and get its first yielded value:

```
function* generateSequence() {
    yield 1;
    yield 2;
    return 3;
}
let generator = generateSequence();
let one = generator.next();
alert(JSON.stringify(one)); // {value: 1, done: false}
```

As of now, we got the first value only, and the function execution is on the second line:

Let's call `generator.next()` again. It resumes the code execution and returns the next `yield`:

```
let two = generator.next();
alert(JSON.stringify(two)); // {value: 2, done: false}
```

And, if we call it a third time, the execution reaches the `return` statement that finishes the function:

```
let three = generator.next();
alert(JSON.stringify(three)); // {value: 3, done: true}
```

Now the generator is done. We should see it from `done:true` and process `value:3` as the final result.

New calls to `generator.next()` don't make sense any more. If we do them, they return the same object: `{done: true}`.

`function* f(…)` or `function *f(…)`?

Both syntaxes are correct.

But usually the first syntax is preferred, as the star * denotes that it's a generator function, it describes the kind, not the name, so it should stick with the function keyword.

Generators are iterable

As you probably already guessed looking at the next() method, generators are iterable.

We can loop over their values using for..of:

```
function* generateSequence() {
    yield 1;
    yield 2;
    return 3;
}

let generator = generateSequence();
for(let value of generator) {
    alert(value); // 1, then 2
}
```

Looks a lot nicer than calling .next().value, right?

…But please note: the example above shows 1, then 2, and that's all. It doesn't show 3!

It's because for..of iteration ignores the last value, when done: true. So, if we want all results to be shown by for..of, we must return them with yield:

```
function* generateSequence() {
    yield 1;
    yield 2;
    yield 3;
}
let generator = generateSequence();
for(let value of generator) {
    alert(value); // 1, then 2, then 3
}
```

As generators are iterable, we can call all related functionality, e.g. the spread syntax ...:

```
function* generateSequence() {
    yield 1;
    yield 2;
    yield 3;
}
let sequence = [0, ...generateSequence()];
alert(sequence); // 0, 1, 2, 3
```

In the code above, ...generateSequence() turns the iterable generator object into an array of items.

Using generators for iterables

Some time ago, in the chapter Iterables we created an iterable range object that returns values from..to.

Here, let's remember the code:

```
let range = {
    from: 1,
    to: 5,
    // for..of range calls this method once in the very beginning
    [Symbol.iterator]() {
        // ...it returns the iterator object:
        // onward, for..of works only with that object, asking it for next values
        return {
            current: this.from,
            last: this.to,
            // next() is called on each iteration by the for..of loop
            next() {
                // it should return the value as an object {done:.., value :...}
                if (this.current <= this.last) {
                    return { done: false, value: this.current++ };
```

```
                } else {
                    return { done: true };
                }
            }
        };
    }
};

// iteration over range returns numbers from range.from to range.to
alert([...range]); // 1,2,3,4,5
```

We can use a generator function for iteration by providing it as Symbol.iterator.

Here's the same range, but much more compact:

```
let range = {
    from: 1,
    to: 5,
    *[Symbol.iterator]() {
        for(let value = this.from; value <= this.to; value++) {
            yield value;
        }
    }
};
alert( [...range] ); // 1,2,3,4,5
```

That works, because range[Symbol.iterator]() now returns a generator, and generator methods are exactly what for..of expects:

- it has a .next() method
- that returns values in the form {value: ..., done: true/false}

That's not a coincidence, of course. Generators were added to JavaScript language with iterators in mind, to implement them easily.

Generator composition

Generator composition is a special feature of generators that allows transparently “embed” generators in each other.

For instance, we have a function that generates a sequence of numbers:

```
function* generateSequence(start, end) {
    for (let i = start; i <= end; i++) yield i;
}
```

Now we’d like to reuse it to generate a more complex sequence:

- first, digits `0..9` (with character codes 48…57),
- followed by uppercase alphabet letters `A..Z` (character codes 65…90)
- followed by lowercase alphabet letters `a..z` (character codes 97…122)

We can use this sequence e.g. to create passwords by selecting characters from it (could add syntax characters as well), but let’s generate it first.

In a regular function, to combine results from multiple other functions, we call them, store the results, and then join at the end.

For generators, there’s a special `yield*` syntax to “embed” (compose) one generator into another.

The composed generator:

```
function* generateSequence(start, end) {
    for (let i = start; i <= end; i++) yield i;
}
function* generatePasswordCodes() {
    // 0..9
    yield* generateSequence(48, 57);
    // A..Z
    yield* generateSequence(65, 90);
    // a..z
    yield* generateSequence(97, 122);
}
let str = '';
```

```
for(let code of generatePasswordCodes()) {
    str += String.fromCharCode(code);
}
alert(str); // 0..9A..Za..z
```

“yield” is a two-way road

Until this moment, generators were similar to iterable objects, with a special syntax to generate values. But in fact they are much more powerful and flexible.

That’s because `yield` is a two-way road: it not only returns the result outside, but also can pass the value inside the generator.

To do so, we should call `generator.next(arg)`, with an argument. That argument becomes the result of `yield`.

Let’s see an example:

```
function* gen() {
    // Pass a question to the outer code and wait for an answer
    let result = yield "2 + 2 = ?"; // (*)
    alert(result);
}
let generator = gen();
let question = generator.next().value; // <-- yield returns the value
generator.next(4); // --> pass the result into the generator
```

1. The first call `generator.next()` should be always made without an argument (the argument is ignored if passed). It starts the execution and returns the result of the first `yield "2+2=?"`. At this point the generator pauses the execution, while staying on the line `(*)`.
2. Then, as shown at the picture above, the result of `yield` gets into the `question` variable in the calling code.

3. On generator.next(4), the generator resumes, and 4 gets in as the result: let result = 4.

Please note, the outer code does not have to immediately call next(4). It may take time. That's not a problem: the generator will wait.

For instance:

```
// resume the generator after some time
setTimeout(() => generator.next(4), 1000);
```

As we can see, unlike regular functions, a generator and the calling code can exchange results by passing values in next/yield.

To make things more obvious, here's another example, with more calls:

```
function* gen() {
  let ask1 = yield "2 + 2 = ?";
  alert(ask1); // 4
  let ask2 = yield "3 * 3 = ?"
  alert(ask2); // 9
}
let generator = gen();
alert( generator.next().value ); // "2 + 2 = ?"
alert( generator.next(4).value ); // "3 * 3 = ?"
alert( generator.next(9).done ); // true
```

generator.throw

As we observed in the examples above, the outer code may pass a value into the generator, as the result of yield.

…But it can also initiate (throw) an error there. That's natural, as an error is a kind of result.

To pass an error into a yield, we should call generator.throw(err). In that case, the err is thrown in the line with that yield.

For instance, here the yield of "2 + 2 = ?" leads to an error:

```
function* gen() {
    try {
        let result = yield "2 + 2 = ?"; // (1)
        alert("The execution does not reach here, because the exception is thrown above");
    } catch(e) {
        alert(e); // shows the error
    }
}
let generator = gen();
let question = generator.next().value;
generator.throw(new Error("The answer is not found in my database")); // (2)
```

The error, thrown into the generator at line (2) leads to an exception in line (1) with yield. In the example above, try..catch catches it and shows it.

If we don't catch it, then just like any exception, it "falls out" the generator into the calling code.

The current line of the calling code is the line with generator.throw, labelled as (2). So we can catch it here, like this:

```
function* generate() {
    let result = yield "2 + 2 = ?"; // Error in this line
}
let generator = generate();
let question = generator.next().value;
try {
    generator.throw(new Error("The answer is not found in my database"));
} catch(e) {
    alert(e); // shows the error
}
```

If we don't catch the error there, then, as usual, it falls through to the outer calling code (if any) and, if uncaught, kills the script.

Modules, introduction

As our application grows bigger, we want to split it into multiple files, so called “modules”. A module usually contains a class or a library of functions.

For a long time, JavaScript existed without a language-level module syntax. That wasn’t a problem, because initially scripts were small and simple, so there was no need.

But eventually scripts became more and more complex, so the community invented a variety of ways to organize code into modules, special libraries to load modules on demand.

For instance:

- **AMD** – one of the most ancient module systems, initially implemented by the library require.js.
- **CommonJS** – the module system created for Node.js server.
- **UMD** – one more module system, suggested as a universal one, compatible with AMD and CommonJS.

Now all these slowly become a part of history, but we still can find them in old scripts.

The language-level module system appeared in the standard in 2015, gradually evolved since then, and is now supported by all major browsers and in Node.js. So we’ll study it from now on.

What is a module?

A module is just a file. One script is one module.

Modules can load each other and use special directives `export` and `import` to interchange functionality, call functions of one module from another one:

- `export` keyword labels variables and functions that should be accessible from outside the current module.
- `import` allows the import of functionality from other modules.

For instance, if we have a file sayHi.js exporting a function:

```
// 📁 sayHi.js
export function sayHi(user) {
    alert(`Hello, ${user}!`);
}
```

…Then another file may import and use it:

```
// 📁 main.js
import {sayHi} from './sayHi.js';
alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

The import directive loads the module by path ./sayHi.js relative to the current file, and assigns exported function sayHi to the corresponding variable.

Let’s run the example in-browser.

As modules support special keywords and features, we must tell the browser that a script should be treated as a module, by using the attribute <script type="module">

```
//say.js
export function sayHi(user) {
    return `Hello, ${user}!`;
}
```

```
//index.html
<!doctype html>
<script type="module">
    import {sayHi} from './say.js';
    document.body.innerHTML = sayHi('John');
</script>
```

The browser automatically fetches and evaluates the imported module (and its imports if needed), and then runs the script.

Core module features

What's different in modules, compared to "regular" scripts?

There are core features, valid both for browser and server-side JavaScript.

Always "use strict"

Modules always `use strict`, by default. E.g. assigning to an undeclared variable will give an error.

```
<script type="module">
    a = 5; // error
</script>
```

Module-level scope

Each module has its own top-level scope. In other words, top-level variables and functions from a module are not seen in other scripts.

In the example below, two scripts are imported, and `hello.js` tries to use `user` variable declared in `user.js`, and fails:

```
//user.js
let user = "John";

//hello.js
alert(user); // no such variable (each module has independent variables)

//index.html
<!doctype html>
<script type="module" src="user.js"></script>
<script type="module" src="hello.js"></script>
```

Modules are expected to export what they want to be accessible from outside and import what they need.

So we should import user.js into hello.js and get the required functionality from it instead of relying on global variables.

This is the correct variant:

```
//user.js

export let user = "John";

//hello.js
import {user} from './user.js';
document.body.innerHTML = user; // John

//index.html
<!doctype html>
<script type="module" src="hello.js"></script>
```

In the browser, independent top-level scope also exists for each <script type="module">:

```
<script type="module">
    // The variable is only visible in this module script
    let user = "John";
</script>

<script type="module">
    alert(user); // Error: user is not defined
</script>
```

If we really need to make a window-level global variable, we can explicitly assign it to window and access as window.user. But that's an exception requiring a good reason.

A module code is evaluated only the first time when imported

If the same module is imported into multiple other places, its code is executed only the first time, then exports are given to all importers.

That has important consequences. Let's look at them using examples:

First, if executing a module code brings side-effects, like showing a message, then importing it multiple times will trigger it only once – the first time:

```
// 📁 alert.js
alert("Module is evaluated!");


// Import the same module from different files

// 📁 1.js
import `./alert.js`; // Module is evaluated!

// 📁 2.js
import `./alert.js`; // (shows nothing)
```

In practice, top-level module code is mostly used for initialization, creation of internal data structures, and if we want something to be reusable – export it.

Now, a more advanced example.

Let's say, a module exports an object:

```
// 📁 admin.js
export let admin = {
    name: "John"
};
```

If this module is imported from multiple files, the module is only evaluated the first time, `admin` object is created, and then passed to all further importers.

All importers get exactly the one and only `admin` object:

```
// 📁 1.js
import {admin} from './admin.js';
```

```
admin.name = "Pete";

// 📁 2.js
import {admin} from './admin.js';
alert(admin.name); // Pete

// Both 1.js and 2.js imported the same object
// Changes made in 1.js are visible in 2.js
```

So, let's reiterate – the module is executed only once. Exports are generated, and then they are shared between importers, so if something changes the `admin` object, other modules will see that.

Such behavior allows us to *configure* modules on first import. We can setup its properties once, and then in further imports it's ready.

For instance, the `admin.js` module may provide certain functionality, but expect the credentials to come into the `admin` object from outside:

```
// 📁 admin.js
export let admin = { };
export function sayHi() {
    alert(`Ready to serve, ${admin.name}!`);
}
```

In `init.js`, the first script of our app, we set `admin.name`. Then everyone will see it, including calls made from inside `admin.js` itself:

```
// 📁 init.js
import {admin} from './admin.js';
admin.name = "Pete";
```

Another module can also see `admin.name`:

```
// 📁 other.js
import {admin, sayHi} from './admin.js';
```

```
alert(admin.name); // Pete
sayHi(); // Ready to serve, Pete!
```

In a module, “this” is undefined

That’s kind of a minor feature, but for completeness we should mention it.

In a module, top-level `this` is undefined.

Compare it to non-module scripts, where `this` is a global object:

```
<script>
    alert(this); // window
</script>

<script type="module">
    alert(this); // undefined
</script>
```

Module scripts are deferred

Module scripts are *always* deferred, same effect as `defer` attribute (described in the chapter Scripts: async, defer), for both external and inline scripts.

In other words:

- downloading external module scripts `<script type="module" src="...">` doesn’t block HTML processing, they load in parallel with other resources.
- module scripts wait until the HTML document is fully ready (even if they are tiny and load faster than HTML), and then run.
- relative order of scripts is maintained: scripts that go first in the document, execute first.

Async works on inline scripts

For non-module scripts, the async attribute only works on external scripts. Async scripts run immediately when ready, independently of other scripts or the HTML document.

For module scripts, it works on inline scripts as well.

For example, the inline script below has async, so it doesn't wait for anything.

It performs the import (fetches ./analytics.js) and runs when ready, even if the HTML document is not finished yet, or if other scripts are still pending.

That's good for functionality that doesn't depend on anything, like counters, ads, document-level event listeners.

```
<!-- all dependencies are fetched (analytics.js), and the script runs -->
<!-- doesn't wait for the document or other <script> tags -->
<script async type="module">
  import {counter} from './analytics.js';
  counter.count();
</script>
```

External scripts

External scripts that have type="module" are different in two aspects:

External scripts with the same src run only once:

```
<!-- the script my.js is fetched and executed only once -->
<script type="module" src="my.js"></script>
```

1. <script type="module" src="my.js"></script>
2. External scripts that are fetched from another origin (e.g. another site) require CORS headers, as described in the chapter Fetch: Cross-Origin Requests. In other words, if a module script is fetched from another origin, the remote server must supply a header Access-Control-Allow-Origin allowing the fetch.

   ```
   <!-- another-site.com must supply Access-Control-Allow-Origin →
   <!-- otherwise, the script won't execute →
   ```

`<script type="module" src="http://another-site.com/their.js"></script>`

That ensures better security by default.

Note: export & import statements allowed only inside module scripts.

Build tools

In real-life, browser modules are rarely used in their “raw” form. Usually, we bundle them together with a special tool such as Webpack and deploy to the production server.

One of the benefits of using bundlers – they give more control over how modules are resolved, allowing bare modules and much more, like CSS/HTML modules.

Build tools do the following:

1. Take a “main” module, the one intended to be put in `<script type="module">` in HTML.
2. Analyze its dependencies: imports and then imports of imports etc.
3. Build a single file with all modules (or multiple files, that’s tunable), replacing native `import` calls with bundler functions, so that it works. “Special” module types like HTML/CSS modules are also supported.
4. In the process, other transformations and optimizations may be applied:
    - Unreachable code removed.
    - Unused exports removed (“tree-shaking”).
    - Development-specific statements like `console` and `debugger` removed.
    - Modern, bleeding-edge JavaScript syntax may be transformed to older one with similar functionality using Babel.
    - The resulting file is minified (spaces removed, variables replaced with shorter names, etc).

If we use bundle tools, then as scripts are bundled together into a single file (or few files), `import/export` statements inside those scripts are replaced by special bundler functions. So the resulting “bundled” script does not contain any `import/export`, it doesn’t require `type="module"`, and we can put it into a regular script:

```
<!-- Assuming we got bundle.js from a tool like Webpack -->
```

```
<script src="bundle.js"></script>
```

Export and Import

Export and import directives have several syntax variants.

In the previous chapter we saw a simple use, now let's explore more examples.

Export before declarations

We can label any declaration as exported by placing `export` before it, be it a variable, function or a class.

For instance, here all exports are valid:

```
// export an array
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// export a constant
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// export a class
export class User {
    constructor(name) {
        this.name = name;
    }
}
```

Export apart from declarations

Also, we can export separately.

Here we first declare, and then export:

```
// 📁 say.js
function sayHi(user) {
    alert(`Hello, ${user}!`);
```

```
}

function sayBye(user) {
    alert(`Bye, ${user}!`);
}
export {sayHi, sayBye}; // a list of exported variables
```

…Or, technically we could put `export` above functions as well.

Import *

Usually, we put a list of what to import in curly braces `import {...}`, like this:

```
// 📁 main.js
import {sayHi, sayBye} from './say.js';
sayHi('John'); // Hello, John!
sayBye('John'); // Bye, John!
```

But if there's a lot to import, we can import everything as an object using `import * as <obj>`, for instance:

```
// 📁 main.js
import * as say from './say.js';`
say.sayHi('John');
say.sayBye('John');
```

At first sight, "import everything" seems such a cool thing, short to write, why should we ever explicitly list what we need to import?

Well, there are few reasons.

1. Modern build tools (webpack and others) bundle modules together and optimize them to speedup loading and remove unused stuff.
   Let's say, we added a 3rd-party library `say.js` to our project with many functions:

   ```
   // 📁 say.js
   export function sayHi() { ... }
   ```

export function sayBye() { ... }

export function becomeSilent() { ... }

Now if we only use one of say.js functions in our project:

```
// 📁 main.js
import {sayHi} from './say.js';
```

…Then the optimizer will see that and remove the other functions from the bundled code, thus making the build smaller. That is called “tree-shaking”.

2. Explicitly listing what to import gives shorter names: sayHi() instead of say.sayHi().
3. Explicit list of imports gives better overview of the code structure: what is used and where. It makes code support and refactoring easier.

Import “as”

We can also use it to import under different names.

For instance, let’s import sayHi into the local variable hi for brevity, and import sayBye as bye:

```
// 📁 main.js
import {sayHi as hi, sayBye as bye} from './say.js';
hi('John'); // Hello, John!
bye('John'); // Bye, John!
```

Export “as”

The similar syntax exists for export.

Let’s export functions as hi and bye:

```
// 📁 say.js
...
export {sayHi as hi, sayBye as bye};
```

Now hi and bye are official names for outsiders, to be used in imports:

```
// 📁 main.js
import * as say from './say.js';
```

```
say.hi('John'); // Hello, John!
say.bye('John'); // Bye, John!
```

Export default

In practice, there are mainly two kinds of modules.

1. Modules that contain a library, pack of functions, like say.js above.
2. Modules that declare a single entity, e.g. a module user.js exports only class User.

Mostly, the second approach is preferred, so that every “thing” resides in its own module.

Naturally, that requires a lot of files, as everything wants its own module, but that’s not a problem at all. Actually, code navigation becomes easier if files are well-named and structured into folders.

Modules provide special export default (“the default export”) syntax to make the “one thing per module” way look better.

Put export default before the entity to export:

```
// 📁 user.js
export default class User { // just add "default"
    constructor(name) {
        this.name = name;
    }
}
```

There may be only one export default per file.

…And then import it without curly braces:

```
// 📁 main.js
import User from './user.js'; // not {User}, just User
new User('John');
```

Imports without curly braces look nicer. A common mistake when starting to use modules is to forget curly braces at all. So, remember, import needs curly braces for named exports and doesn't need them for the default one.

Technically, we may have both `default` and `named exports` in a single module, but in practice people usually don't mix them. A module has either named exports or the default one.

As there may be at most one default export per file, the exported entity may have no name.

For instance, these are all perfectly valid `default exports`:

```
export default class { // no class name
    constructor() { ... }
}
export default function(user) { // no function name
    alert(`Hello, ${user}!`);
}
// export a single value, without making a variable
export default ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
```

Not giving a name is fine, because export default is only one per file, so import without curly braces knows what to import.

Without default, such export would give an error:

```
export class { // Error! (non-default export needs a name)
    constructor() {}
}
```

The "default" name

In some situations the `default` keyword is used to reference the `default export`.

For example, to export a function separately from its definition:

```
function sayHi(user) {
```

```
    alert(`Hello, ${user}!`);
}
// same as if we added "export default" before the function
export {sayHi as default};
```

Or, another situation, let's say a module user.js exports one main "default" thing and a few named ones (rarely the case, but happens):

```
// 📁 user.js
export default class User {
    constructor(name) {
        this.name = name;
    }
}
export function sayHi(user) {
    alert(`Hello, ${user}!`);
}
```

Here's how to import the default export along with a named one:

```
// 📁 main.js
import {default as User, sayHi} from './user.js';
new User('John');
```

Dynamic imports

Export and import statements that we covered in previous chapters are called "static". The syntax is very simple and strict.

First, we can't dynamically generate any parameters of import.

The module path must be a primitive string, can't be a function call. This won't work:

```
import ... from getModuleName(); // Error, only from "string" is allowed
```

Second, we can't import conditionally or at run-time:

```
if(...) {
```

```
    import ...; // Error, not allowed!
}
{
    import ...; // Error, we can't put import in any block
}
```

That's because import/export aim to provide a backbone for the code structure. That's a good thing, as code structure can be analyzed, modules can be gathered and bundled into one file by special tools, unused exports can be removed ("tree-shaken"). That's possible only because the structure of imports/exports is simple and fixed.

But how can we import a module dynamically, on-demand?

The import() expression

The import(module) expression loads the module and returns a promise that resolves into a module object that contains all its exports. It can be called from any place in the code.

We can use it dynamically in any place of the code, for instance:

```
let modulePath = prompt("Which module to load?");

import(modulePath)
    .then(obj => <module object>)
    .catch(err => <loading error, e.g. if no such module>)
```

Or, we could use let module = await import(modulePath) if inside an async function.

For instance, if we have the following module say.js:

```
// 📁 say.js
export function hi() {
    alert(`Hello`);
}
export function bye() {
    alert(`Bye`);
}
```

…Then dynamic import can be like this:

```
let {hi, bye} = await import('./say.js');
hi();
bye();
```

Or, if say.js has the default export:

```
// say.js
export default function() {
    alert("Module loaded (export default)!");
}
```

…Then, in order to access it, we can use default property of the module object:

```
let obj = await import('./say.js');
let say = obj.default;
// or, in one line: let {default: say} = await import('./say.js');
say();
```

Here's the full example:

```
//say.js

export function hi() {
    alert(`Hello`);
}
export function bye() {
    alert(`Bye`);
}
export default function() {
    alert("Module loaded (export default)!");
}
```

Please note:

Dynamic imports work in regular scripts, they don't require script type="module".
