Modern JS Cheatsheet

Cheatsheet for the JavaScript knowledge you will frequently encounter in modern projects.

View on GitHub

Modern JavaScript Cheatsheet

Image Credits: Ahmad Awais ⚡

> If you like this content, you can ping me or follow me on Twitter :+1:

Tweet @mbeaudru 378

Introduction

Motivation

This document is a cheatsheet for JavaScript you will frequently encounter in modern projects and most contemporary sample code.

This guide is not intended to teach you JavaScript from the ground up, but to help developers with basic knowledge who may struggle to get

familiar with modern codebases (or let's say to learn React for instance) because of the JavaScript concepts used.

Besides, I will sometimes provide personal tips that may be debatable but will take care to mention that it's a personal recommendation when I do so.

> **Note:** Most of the concepts introduced here are coming from a JavaScript language update (ES2015, often called ES6). You can find new features added by this update here; it's very well done.

Complementary Resources

When you struggle to understand a notion, I suggest you look for answers on the following resources:

- MDN (Mozilla Developer Network)
- You don't know JS (book)
- Eloquent JavaScript (book)
- Douglas Crockford's blog
- ES6 Features with examples
- Wes Bos blog (ES6)
- Javascript Basics for Beginners - a free Udacity course
- Reddit (JavaScript)
- Google to find specific blog and resources
- StackOverflow

Table of Contents

- Variable mutation

Notions

Variable declaration: var, const, let

In JavaScript, there are three keywords available to declare a variable, and each has its differences. Those are `var`, `let` and `const`.

Short explanation

Variables declared with `const` keyword can't be reassigned, while `let` and `var` can.

I recommend always declaring your variables with `const` by default, but with `let` if it is a variable that you need to *mutate* or reassign later.

	Scope	Reassignable	Mutable	Temporal Dead Zone
const	Block	No	Yes	Yes
let	Block	Yes	Yes	Yes
var	Function	Yes	Yes	No

Sample code

```
const person = "Nick";
person = "John" // Will raise an error, person can't be reassigned
```

```
let person = "Nick";
person = "John";
console.log(person) // "John", reassignment is allowed with let
```

Detailed explanation

The *scope* of a variable roughly means "where is this variable available in the code".

var

`var` declared variables are *function scoped*, meaning that when a variable is created in a function, everything in that function can access that variable. Besides, a *function scoped* variable created in a function can't be accessed outside this function.

I recommend you to picture it as if an *X scoped* variable meant that this variable was a property of X.

```
function myFunction() {
  var myVar = "Nick";
  console.log(myVar); // "Nick" - myVar is accessible inside the function
}
console.log(myVar); // Throws a ReferenceError, myVar is not accessible o
```

Still focusing on the variable scope, here is a more subtle example:

```
function myFunction() {
  var myVar = "Nick";
  if (true) {
    var myVar = "John";
    console.log(myVar); // "John"
    // actually, myVar being function scoped, we just erased the previous
  }
  console.log(myVar); // "John" - see how the instructions in the if bloc
}
console.log(myVar); // Throws a ReferenceError, myVar is not accessible o
```

Besides, *var* declared variables are moved to the top of the scope at execution. This is what we call var hoisting.

This portion of code:

```
console.log(myVar) // undefined -- no error raised
var myVar = 2;
```

is understood at execution like:

```
var myVar;
```

```
console.log(myVar) // undefined -- no error raised
myVar = 2;
```

let

`var` and `let` are about the same, but `let` declared variables

- are *block scoped*
- are **not** accessible before they are assigned
- can't be re-declared in the same scope

Let's see the impact of block-scoping taking our previous example:

```
function myFunction() {
  let myVar = "Nick";
  if (true) {
    let myVar = "John";
    console.log(myVar); // "John"
    // actually, myVar being block scoped, we just created a new variable
    // this variable is not accessible outside this block and totally ind
    // from the first myVar created !
  }
  console.log(myVar); // "Nick", see how the instructions in the if block
}
console.log(myVar); // Throws a ReferenceError, myVar is not accessible o
```

Now, what it means for *let* (and *const*) variables for not being accessible before being assigned:

```
console.log(myVar) // raises a ReferenceError !
let myVar = 2;
```

By contrast with *var* variables, if you try to read or write on a *let* or *const* variable before they are assigned an error will be raised. This phenomenon is often called *Temporal dead zone* or *TDZ*.

> **Note:** Technically, *let* and *const* variables declarations are being hoisted too, but not their assignation. Since they're made so that they can't be used before assignation, it intuitively feels like there is no hoisting, but there is. Find out more on this very detailed explanation here if you want to know more.

In addition, you can't re-declare a *let* variable:

```
let myVar = 2;
let myVar = 3; // Raises a SyntaxError
```

const

`const` declared variables behave like *let* variables, but also they can't be reassigned.

To sum it up, *const* variables:

- are *block scoped*
- are not accessible before being assigned
- can't be re-declared in the same scope
- can't be reassigned

```
const myVar = "Nick";
myVar = "John" // raises an error, reassignment is not allowed
```

```
const myVar = "Nick";
const myVar = "John" // raises an error, re-declaration is not allowed
```

But there is a subtlety : `const` variables are not **immutable** ! Concretely, it means that *object* and *array* `const` declared variables **can** be mutated.

For objects:

```
const person = {
  name: 'Nick'
};
person.name = 'John' // this will work ! person variable is not completel
console.log(person.name) // "John"
person = "Sandra" // raises an error, because reassignment is not allowed
```

For arrays:

```
const person = [];
```

```
person.push('John'); // this will work ! person variable is not completel
console.log(person[0]) // "John"
person = ["Nick"] // raises an error, because reassignment is not allowed
```

External resource

- How let and const are scoped in JavaScript - WesBos
- Temporal Dead Zone (TDZ) Demystified

Arrow function

The ES6 JavaScript update has introduced *arrow functions*, which is another way to declare and use functions. Here are the benefits they bring:

- More concise
- *this* is picked up from surroundings
- implicit return

Sample code

- Concision and implicit return

```
function double(x) { return x * 2; } // Traditional way
console.log(double(2)) // 4
```

```
const double = x => x * 2; // Same function written as an arrow function
console.log(double(2)) // 4
```

- *this* reference

In an arrow function, *this* is equal to the *this* value of the enclosing execution context. Basically, with arrow functions, you don't have to do the "that = this" trick before calling a function inside a function anymore.

```
function myFunc() {
  this.myVar = 0;
  setTimeout(() => {
    this.myVar++;
```

```
    console.log(this.myVar) // 1
  }, 0);
}
```

Detailed explanation

Concision

Arrow functions are more concise than traditional functions in many ways. Let's review all the possible cases:

- Implicit VS Explicit return

An **explicit return** is a function where the *return* keyword is used in its body.

```
function double(x) {
  return x * 2; // this function explicitly returns x * 2, *return* key
}
```

In the traditional way of writing functions, the return was always explicit. But with arrow functions, you can do *implicit return* which means that you don't need to use the keyword *return* to return a value.

```
const double = (x) => {
  return x * 2; // Explicit return here
}
```

Since this function only returns something (no instructions before the *return* keyword) we can do an implicit return.

```
const double = (x) => x * 2; // Correct, returns x*2
```

To do so, we only need to **remove the brackets** and the **return** keyword. That's why it's called an *implicit* return, the *return* keyword is not there, but this function will indeed return `x * 2`.

> **Note:** If your function does not return a value (with *side effects*), it doesn't do an explicit nor an implicit return.

Besides, if you want to implicitly return an *object* you **must have parentheses around it** since it will conflict with the block braces:

```
const getPerson = () => ({ name: "Nick", age: 24 })
console.log(getPerson()) // { name: "Nick", age: 24 } -- object implicitl
```

- Only one argument

If your function only takes one parameter, you can omit the parentheses around it. If we take back the above *double* code:

```
const double = (x) => x * 2; // this arrow function only takes one para
```

Parentheses around the parameter can be avoided:

```
const double = x => x * 2; // this arrow function only takes one parame
```

- No arguments

When there is no argument provided to an arrow function, you need to provide parentheses, or it won't be valid syntax.

```
() => { // parentheses are provided, everything is fine
  const x = 2;
  return x;
}
```

```
=> { // No parentheses, this won't work!
  const x = 2;
  return x;
}
```

this reference

To understand this subtlety introduced with arrow functions, you must know how this behaves in JavaScript.

In an arrow function, *this* is equal to the *this* value of the enclosing

execution context. What it means is that an arrow function doesn't create a new *this*, it grabs it from its surrounding instead.

Without arrow function, if you wanted to access a variable from *this* in a function inside a function, you had to use the *that = this* or *self = this* trick.

For instance, using setTimeout function inside myFunc:

```
function myFunc() {
  this.myVar = 0;
  var that = this; // that = this trick
  setTimeout(
    function() { // A new *this* is created in this function scope
      that.myVar++;
      console.log(that.myVar) // 1

      console.log(this.myVar) // undefined -- see function declaration ab
    },
    0
  );
}
```

But with arrow function, *this* is taken from its surrounding:

```
function myFunc() {
  this.myVar = 0;
  setTimeout(
    () => { // this taken from surrounding, meaning myFunc here
      this.myVar++;
      console.log(this.myVar) // 1
    },
    0
  );
}
```

Useful resources

- Arrow functions introduction - WesBos
- JavaScript arrow function - MDN
- Arrow function and lexical *this*

Function default parameter value

Starting from ES2015 JavaScript update, you can set default value to your function parameters using the following syntax:

```
function myFunc(x = 10) {
  return x;
}
console.log(myFunc()) // 10 -- no value is provided so x default value 10
console.log(myFunc(5)) // 5 -- a value is provided so x is equal to 5 in

console.log(myFunc(undefined)) // 10 -- undefined value is provided so de
console.log(myFunc(null)) // null -- a value (null) is provided, see belo
```

The default parameter is applied in two and only two situations:

- No parameter provided
- *undefined* parameter provided

In other words, if you pass in *null* the default parameter **won't be applied**.

> **Note:** Default value assignment can be used with destructured parameters as well (see next notion to see an example)

External resource

- Default parameter value - ES6 Features
- Default parameters - MDN

Destructuring objects and arrays

Destructuring is a convenient way of creating new variables by extracting some values from data stored in objects or arrays.

To name a few use cases, *destructuring* can be used to destructure function parameters or *this.props* in React projects for instance.

Explanation with sample code

- Object

Let's consider the following object for all the samples:

```
const person = {
  firstName: "Nick",
  lastName: "Anderson",
  age: 35,
  sex: "M"
}
```

Without destructuring

```
const first = person.firstName;
const age = person.age;
const city = person.city || "Paris";
```

With destructuring, all in one line:

```
const { firstName: first, age, city = "Paris" } = person; // That's it !

console.log(age) // 35 -- A new variable age is created and is equal to p
console.log(first) // "Nick" -- A new variable first is created and is eq
console.log(firstName) // ReferenceError -- person.firstName exists BUT t
console.log(city) // "Paris" -- A new variable city is created and since
```

Note : In `const { age } = person;`, the brackets after *const* keyword are not used to declare an object nor a block but is the *destructuring* syntax.

- Function parameters

Destructuring is often used to destructure objects parameters in functions.

Without destructuring

```
function joinFirstLastName(person) {
  const firstName = person.firstName;
  const lastName = person.lastName;
  return firstName + '-' + lastName;
}

joinFirstLastName(person); // "Nick-Anderson"
```

In destructuring the object parameter *person*, we get a more concise

function:

```
function joinFirstLastName({ firstName, lastName }) { // we create firstN
  return firstName + '-' + lastName;
}

joinFirstLastName(person); // "Nick-Anderson"
```

Destructuring is even more pleasant to use with arrow functions:

```
const joinFirstLastName = ({ firstName, lastName }) => firstName + '-' +

joinFirstLastName(person); // "Nick-Anderson"
```

- Array

Let's consider the following array:

```
const myArray = ["a", "b", "c"];
```

Without destructuring

```
const x = myArray[0];
const y = myArray[1];
```

With destructuring

```
const [x, y] = myArray; // That's it !

console.log(x) // "a"
console.log(y) // "b"
```

Useful resources

- ES6 Features - Destructuring Assignment
- Destructuring Objects - WesBos
- ExploringJS - Destructuring

Array methods - map / filter / reduce / find

Map, *filter*, *reduce* and *find* are array methods that are coming from a programming paradigm named *functional programming*.

To sum it up:

- **Array.prototype.map()** takes an array, does something on its elements and returns an array with the transformed elements.
- **Array.prototype.filter()** takes an array, decides element by element if it should keep it or not and returns an array with the kept elements only
- **Array.prototype.reduce()** takes an array and aggregates the elements into a single value (which is returned)
- **Array.prototype.find()** takes an array, and returns the first element that satisfies the provided condition.

I recommend to use them as much as possible in following the principles of functional programming because they are composable, concise and elegant.

With those four methods, you can avoid the use of *for* and *forEach* loops in most situations. When you are tempted to do a *for* loop, try to do it with *map*, *filter*, *reduce* and *find* composed. You might struggle to do it at first because it requires you to learn a new way of thinking, but once you've got it things get easier.

Sample code

```
const numbers = [0, 1, 2, 3, 4, 5, 6];
const doubledNumbers = numbers.map(n => n * 2); // [0, 2, 4, 6, 8, 10, 12
const evenNumbers = numbers.filter(n => n % 2 === 0); // [0, 2, 4, 6]
const sum = numbers.reduce((prev, next) => prev + next, 0); // 21
const greaterThanFour = numbers.find((n) => n>4); // 5
```

Compute total grade sum for students with grades 10 or above by composing map, filter and reduce:

```
const students = [
  { name: "Nick", grade: 10 },
  { name: "John", grade: 15 },
  { name: "Julia", grade: 19 },
```

```
  { name: "Nathalie", grade: 9 },
];

const aboveTenSum = students
  .map(student => student.grade) // we map the students array to an array
  .filter(grade => grade >= 10) // we filter the grades array to keep tho
  .reduce((prev, next) => prev + next, 0); // we sum all the grades 10 or

console.log(aboveTenSum) // 44 -- 10 (Nick) + 15 (John) + 19 (Julia), Nat
```

Explanation

Let's consider the following array of numbers for our examples:

```
const numbers = [0, 1, 2, 3, 4, 5, 6];
```

Array.prototype.map()

```
const doubledNumbers = numbers.map(function(n) {
  return n * 2;
});
console.log(doubledNumbers); // [0, 2, 4, 6, 8, 10, 12]
```

What's happening here? We are using .map on the *numbers* array, the map is iterating on each element of the array and passes it to our function. The goal of the function is to produce and return a new value from the one passed so that map can replace it.

Let's extract this function to make it more clear, just for this once:

```
const doubleN = function(n) { return n * 2; };
const doubledNumbers = numbers.map(doubleN);
console.log(doubledNumbers); // [0, 2, 4, 6, 8, 10, 12]
```

Note : You will frequently encounter this method used in combination with arrow functions

```
const doubledNumbers = numbers.map(n => n * 2);
console.log(doubledNumbers); // [0, 2, 4, 6, 8, 10, 12]
```

`numbers.map(doubleN)` produces `[doubleN(0), doubleN(1), doubleN(2), doubleN(3), doubleN(4), doubleN(5), doubleN(6)]` which is equal to `[0, 2, 4, 6, 8, 10, 12]`.

> **Note:** If you do not need to return a new array and just want to do a loop that has side effects, you might just want to use a for / forEach loop instead of a map.

Array.prototype.filter()

```
const evenNumbers = numbers.filter(function(n) {
  return n % 2 === 0; // true if "n" is par, false if "n" isn't
});
console.log(evenNumbers); // [0, 2, 4, 6]
```

Note : You will frequently encounter this method used in combination with arrow functions

```
const evenNumbers = numbers.filter(n => n % 2 === 0);
console.log(evenNumbers); // [0, 2, 4, 6]
```

We are using .filter on the *numbers* array, filter is iterating on each element of the array and passes it to our function. The goal of the function is to return a boolean that will determine whether the current value will be kept or not. Filter then returns the array with only the kept values.

Array.prototype.reduce()

The reduce method goal is to *reduce* all elements of the array it iterates on into a single value. How it aggregates those elements is up to you.

```
const sum = numbers.reduce(
  function(acc, n) {
    return acc + n;
  },
  0 // accumulator variable value at first iteration step
);

console.log(sum) // 21
```

Note : You will frequently encounter this method used in combination with arrow functions

```
const sum = numbers.reduce((acc, n) => acc + n, 0);
console.log(sum) // 21
```

Just like for .map and .filter methods, .reduce is applied on an array and takes a function as the first parameter.

This time though, there are changes:

- .reduce takes two parameters

The first parameter is a function that will be called at each iteration step.

The second parameter is the value of the accumulator variable (*acc* here) at the first iteration step (read next point to understand).

- Function parameters

The function you pass as the first parameter of .reduce takes two parameters. The first one (*acc* here) is the accumulator variable, whereas the second parameter (*n*) is the current element.

The accumulator variable is equal to the return value of your function at the **previous** iteration step. At the first step of the iteration, *acc* is equal to the value you passed as .reduce second parameter.

At first iteration step

`acc = 0` because we passed in 0 as the second parameter for reduce

`n = 0` first element of the *number* array

Function returns *acc* + *n* –> 0 + 0 –> 0

At second iteration step

`acc = 0` because it's the value the function returned at the previous iteration step

`n = 1` second element of the *number* array

Function returns *acc* + *n* –> 0 + 1 –> 1

At third iteration step

`acc = 1` because it's the value the function returned at the previous iteration step

`n = 2` third element of the *number* array

Function returns *acc* + *n* –> 1 + 2 –> 3

At fourth iteration step

`acc = 3` because it's the value the function returned at the previous iteration step

`n = 3` fourth element of the *number* array

Function returns *acc* + *n* –> 3 + 3 –> 6

[...] At last iteration step

`acc = 15` because it's the value the function returned at the previous iteration step

`n = 6` last element of the *number* array

Function returns *acc* + *n* –> 15 + 6 –> 21

As it is the last iteration step, **.reduce** returns 21.

Array.prototype.find()

```
const greaterThanZero = numbers.find(function(n) {
  return n > 0; // return number just greater than 0 is present
});
console.log(greaterThanZero); // 1
```

Note : You will frequently encounter this method used in combination with arrow functions

We are using .find on the *numbers* array, .find is iterating on each element of the array and passes it to our function, until the condition is

met. The goal of the function is to return the element that satisfies the current testing function. The .find method executes the callback function once for each index of the array until the callback returns a truthy value.

Note : It immediately returns the value of that element (that satisfies the condition) if found. Otherwise, returns undefined.

External Resource

- Understanding map / filter / reduce in JS

Spread operator "..."

The spread operator `...` has been introduced with ES2015 and is used to expand elements of an iterable (like an array) into places where multiple elements can fit.

Sample code

```
const arr1 = ["a", "b", "c"];
const arr2 = [...arr1, "d", "e", "f"]; // ["a", "b", "c", "d", "e", "f"]
```

```
function myFunc(x, y, ...params) {
  console.log(x);
  console.log(y);
  console.log(params)
}

myFunc("a", "b", "c", "d", "e", "f")
// "a"
// "b"
// ["c", "d", "e", "f"]
```

```
const { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };
console.log(x); // 1
console.log(y); // 2
console.log(z); // { a: 3, b: 4 }

const n = { x, y, ...z };
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }
```

Explanation

In iterables (like arrays)

If we have the two following arrays:

```
const arr1 = ["a", "b", "c"];
const arr2 = [arr1, "d", "e", "f"]; // [["a", "b", "c"], "d", "e", "f"]
```

arr2 the first element is an array because *arr1* is injected as is into *arr2*. But what we want is *arr2* to be an array of letters. To do so, we can *spread* the elements of *arr1* into *arr2*.

With spread operator

```
const arr1 = ["a", "b", "c"];
const arr2 = [...arr1, "d", "e", "f"]; // ["a", "b", "c", "d", "e", "f"]
```

Function rest parameter

In function parameters, we can use the rest operator to inject parameters into an array we can loop in. There is already an **arguments** object bound to every function that is equal to an array of all the parameters passed into the function.

```
function myFunc() {
  for (var i = 0; i < arguments.length; i++) {
    console.log(arguments[i]);
  }
}

myFunc("Nick", "Anderson", 10, 12, 6);
// "Nick"
// "Anderson"
// 10
// 12
// 6
```

But let's say that we want this function to create a new student with its grades and with its average grade. Wouldn't it be more convenient to

extract the first two parameters into two separate variables, and then have all the grades in an array we can iterate over?

That's exactly what the rest operator allows us to do!

```
function createStudent(firstName, lastName, ...grades) {
  // firstName = "Nick"
  // lastName = "Anderson"
  // [10, 12, 6] -- "..." takes all other parameters passed and creates a

  const avgGrade = grades.reduce((acc, curr) => acc + curr, 0) / grades.l

  return {
    firstName: firstName,
    lastName: lastName,
    grades: grades,
    avgGrade: avgGrade
  }
}

const student = createStudent("Nick", "Anderson", 10, 12, 6);
console.log(student);
// {
//   firstName: "Nick",
//   lastName: "Anderson",
//   grades: [10, 12, 6],
//   avgGrade: 9,33
// }
```

> **Note:** createStudent function is bad because we don't check if grades.length exists or is different from 0. But it's easier to read this way, so I didn't handle this case.

Object properties spreading

For this one, I recommend you read previous explanations about the rest operator on iterables and function parameters.

```
const myObj = { x: 1, y: 2, a: 3, b: 4 };
const { x, y, ...z } = myObj; // object destructuring here
console.log(x); // 1
console.log(y); // 2
console.log(z); // { a: 3, b: 4 }

// z is the rest of the object destructured: myObj object minus x and y p
```

```
const n = { x, y, ...z };
console.log(n); // { x: 1, y: 2, a: 3, b: 4 }

// Here z object properties are spread into n
```

External resources

- TC39 - Object rest/spread
- Spread operator introduction - WesBos
- JavaScript & the spread operator
- 6 Great uses of the spread operator

Object property shorthand

When assigning a variable to an object property, if the variable name is equal to the property name, you can do the following:

```
const x = 10;
const myObj = { x };
console.log(myObj.x) // 10
```

Explanation

Usually (pre-ES2015) when you declare a new *object literal* and want to use variables as object properties values, you would write this kind of code:

```
const x = 10;
const y = 20;

const myObj = {
  x: x, // assigning x variable value to myObj.x
  y: y // assigning y variable value to myObj.y
};

console.log(myObj.x) // 10
console.log(myObj.y) // 20
```

As you can see, this is quite repetitive because the properties name of myObj are the same as the variable names you want to assign to those

properties.

With ES2015, when the variable name is the same as the property name, you can do this shorthand:

```
const x = 10;
const y = 20;

const myObj = {
  x,
  y
};

console.log(myObj.x) // 10
console.log(myObj.y) // 20
```

External resources

- Property shorthand - ES6 Features

Promises

A promise is an object which can be returned synchronously from an asynchronous function (ref).

Promises can be used to avoid callback hell, and they are more and more frequently encountered in modern JavaScript projects.

Sample code

```
const fetchingPosts = new Promise((res, rej) => {
  $.get("/posts")
    .done(posts => res(posts))
    .fail(err => rej(err));
});

fetchingPosts
  .then(posts => console.log(posts))
  .catch(err => console.log(err));
```

Explanation

When you do an *Ajax request* the response is not synchronous because you want a resource that takes some time to come. It even may never come if the resource you have requested is unavailable for some reason (404).

To handle that kind of situation, ES2015 has given us *promises*. Promises can have three different states:

- Pending
- Fulfilled
- Rejected

Let's say we want to use promises to handle an Ajax request to fetch the resource X.

Create the promise

We firstly are going to create a promise. We will use the jQuery get method to do our Ajax request to X.

```
const xFetcherPromise = new Promise( // Create promise using "new" keywor
  function(resolve, reject) { // Promise constructor takes a function par
    $.get("X") // Launch the Ajax request
      .done(function(X) { // Once the request is done...
        resolve(X); // ... resolve the promise with the X value as parame
      })
      .fail(function(error) { // If the request has failed...
        reject(error); // ... reject the promise with the error as parame
      });
  }
)
```

As seen in the above sample, the Promise object takes an *executor* function which takes two parameters **resolve** and **reject**. Those parameters are functions which when called are going to move the promise *pending* state to respectively a *fulfilled* and *rejected* state.

The promise is in pending state after instance creation and its *executor* function is executed immediately. Once one of the function *resolve* or *reject* is called in the *executor* function, the promise will call its associated handlers.

Promise handlers usage

To get the promise result (or error), we must attach to it handlers by doing the following:

```
xFetcherPromise
  .then(function(X) {
    console.log(X);
  })
  .catch(function(err) {
    console.log(err)
  })
```

If the promise succeeds, *resolve* is executed and the function passed as `.then` parameter is executed.

If it fails, *reject* is executed and the function passed as `.catch` parameter is executed.

> **Note :** If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached. (Ref: MDN)

External Resources

- JavaScript Promises for dummies - Jecelyn Yeen
- JavaScript Promise API - David Walsh
- Using promises - MDN
- What is a promise - Eric Elliott
- JavaScript Promises: an Introduction - Jake Archibald
- Promise documentation - MDN

Template literals

Template literals is an *expression interpolation* for single and multiple-line strings.

In other words, it is a new string syntax in which you can conveniently use any JavaScript expressions (variables for instance).

Sample code

```
const name = "Nick";
`Hello ${name}, the following expression is equal to four : ${2+2}`;

// Hello Nick, the following expression is equal to four: 4
```

External resources

- String interpolation - ES6 Features
- ES6 Template Strings - Addy Osmani

Tagged template literals

Template tags are *functions that can be prefixed to a template literal*. When a function is called this way, the first parameter is an array of the *strings* that appear between the template's interpolated variables, and the subsequent parameters are the interpolated values. Use a spread operator `...` to capture all of them. (Ref: MDN).

> **Note :** A famous library named styled-components heavily relies on this feature.

Below is a toy example on how they work.

```
function highlight(strings, ...values) {
  const interpolation = strings.reduce((prev, current) => {
    return prev + current + (values.length ? "<mark>" + values.shift() +
  }, "");

  return interpolation;
}

const condiment = "jam";
const meal = "toast";

highlight`I like ${condiment} on ${meal}.`;
// "I like <mark>jam</mark> on <mark>toast</mark>."
```

A more interesting example:

```
function comma(strings, ...values) {
  return strings.reduce((prev, next) => {
    let value = values.shift() || [];
    value = value.join(", ");
    return prev + next + value;
  }, "");
}

const snacks = ['apples', 'bananas', 'cherries'];
comma`I like ${snacks} to snack on.`;
// "I like apples, bananas, cherries to snack on."
```

External resources

- Wes Bos on Tagged Template Literals
- Library of common template tags

Imports / Exports

ES6 modules are used to access variables or functions in a module explicitly exported by the modules it imports.

I highly recommend to take a look at MDN resources on import/export (see external resources below), it is both straightforward and complete.

Explanation with sample code

Named exports

Named exports are used to export several values from a module.

> **Note :** You can only name-export first-class citizens that have a name.

```
// mathConstants.js
export const pi = 3.14;
export const exp = 2.7;
export const alpha = 0.35;

// -------------

// myFile.js
import { pi, exp } from './mathConstants.js'; // Named import -- destruct
console.log(pi) // 3.14
```

```
console.log(exp) // 2.7

// -------------

// mySecondFile.js
import * as constants from './mathConstants.js'; // Inject all exported v
console.log(constants.pi) // 3.14
console.log(constants.exp) // 2.7
```

While named imports looks like *destructuring*, they have a different syntax and are not the same. They don't support default values nor *deep* destructuring.

Besides, you can do aliases but the syntax is different from the one used in destructuring:

```
import { foo as bar } from 'myFile.js'; // foo is imported and injected i
```

Default import / export

Concerning the default export, there is only a single default export per module. A default export can be a function, a class, an object or anything else. This value is considered the "main" exported value since it will be the simplest to import. Ref: MDN

```
// coolNumber.js
const ultimateNumber = 42;
export default ultimateNumber;

// ------------

// myFile.js
import number from './coolNumber.js';
// Default export, independently from its name, is automatically injected
console.log(number) // 42
```

Function exporting:

```
// sum.js
export default function sum(x, y) {
  return x + y;
```

```
}
// -------------

// myFile.js
import sum from './sum.js';
const result = sum(1, 2);
console.log(result) // 3
```

External resources

- ES6 Modules in bulletpoints
- Export - MDN
- Import - MDN
- Understanding ES6 Modules
- Destructuring special case - import statements
- Misunderstanding ES6 Modules - Kent C. Dodds
- Modules in JavaScript

JavaScript *this*

this operator behaves differently than in other languages and is in most cases determined by how a function is called. (Ref: MDN).

This notion is having many subtleties and being quite hard, I highly suggest you to deep dive in the external resources below. Thus, I will provide what I personally have in mind to determine what *this* is equal to. I have learned this tip from this article written by Yehuda Katz.

```
function myFunc() {
  ...
}

// After each statement, you find the value of *this* in myFunc

myFunc.call("myString", "hello") // "myString" -- first .call parameter v

// In non-strict-mode
myFunc("hello") // window -- myFunc() is syntax sugar for myFunc.call(win

// In strict-mode
myFunc("hello") // undefined -- myFunc() is syntax sugar for myFunc.call(
```

```
var person = {
  myFunc: function() { ... }
}

person.myFunc.call(person, "test") // person Object -- first call paramet
person.myFunc("test") // person Object -- person.myFunc() is syntax sugar

var myBoundFunc = person.myFunc.bind("hello") // Creates a new function i
person.myFunc("test") // person Object -- The bind method has no effect o
myBoundFunc("test") // "hello" -- myBoundFunc is person.myFunc with "hell
```

External resources

- Understanding JavaScript Function Invocation and “this” - Yehuda Katz
- JavaScript this - MDN

Class

JavaScript is a prototype-based language (whereas Java is class-based language, for instance). ES6 has introduced JavaScript classes which are meant to be a syntactic sugar for prototype-based inheritance and **not** a new class-based inheritance model (ref).

The word *class* is indeed error prone if you are familiar with classes in other languages. If you do, avoid assuming how JavaScript classes work on this basis and consider it an entirely different notion.

Since this document is not an attempt to teach you the language from the ground up, I will assume you know what prototypes are and how they behave. If you do not, see the external resources listed below the sample code.

Samples

Before ES6, prototype syntax:

```
var Person = function(name, age) {
  this.name = name;
  this.age = age;
}
Person.prototype.stringSentence = function() {
```

```
  return "Hello, my name is " + this.name + " and I'm " + this.age;
}
```

With ES6 class syntax:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  stringSentence() {
    return `Hello, my name is ${this.name} and I am ${this.age}`;
  }
}

const myPerson = new Person("Manu", 23);
console.log(myPerson.age) // 23
console.log(myPerson.stringSentence()) // "Hello, my name is Manu and I'm
```

External resources

For prototype understanding:

- Understanding Prototypes in JS - Yehuda Katz
- A plain English guide to JS prototypes - Sebastian Porto
- Inheritance and the prototype chain - MDN

For classes understanding:

- ES6 Classes in Depth - Nicolas Bevacqua
- ES6 Features - Classes
- JavaScript Classes - MDN

`Extends` and `super` keywords

The `extends` keyword is used in class declarations or class expressions to create a class which is a child of another class (Ref: MDN). The subclass inherits all the properties of the superclass and additionally can add new properties or modify the inherited ones.

The `super` keyword is used to call functions on an object's parent,

including its constructor.

- `super` keyword must be used before the `this` keyword is used in constructor
- Invoking `super()` calls the parent class constructor. If you want to pass some arguments in a class's constructor to its parent's constructor, you call it with `super(arguments)`.
- If the parent class have a method (even static) called `X`, you can use `super.X()` to call it in a child class.

Sample Code

```
class Polygon {
  constructor(height, width) {
    this.name = 'Polygon';
    this.height = height;
    this.width = width;
  }

  getHelloPhrase() {
    return `Hi, I am a ${this.name}`;
  }
}

class Square extends Polygon {
  constructor(length) {
    // Here, it calls the parent class' constructor with lengths
    // provided for the Polygon's width and height
    super(length, length);
    // Note: In derived classes, super() must be called before you
    // can use 'this'. Leaving this out will cause a reference error.
    this.name = 'Square';
    this.length = length;
  }

  getCustomHelloPhrase() {
    const polygonPhrase = super.getHelloPhrase(); // accessing parent met
    return `${polygonPhrase} with a length of ${this.length}`;
  }

  get area() {
    return this.height * this.width;
  }
}

const mySquare = new Square(10);
console.log(mySquare.area) // 100
```

```
console.log(mySquare.getHelloPhrase()) // 'Hi, I am a Square' -- Square i
console.log(mySquare.getCustomHelloPhrase()) // 'Hi, I am a Square with a
```

Note : If we had tried to use `this` before calling `super()` in Square class, a ReferenceError would have been raised:

```
class Square extends Polygon {
  constructor(length) {
    this.height; // ReferenceError, super needs to be called first!

    // Here, it calls the parent class' constructor with lengths
    // provided for the Polygon's width and height
    super(length, length);

    // Note: In derived classes, super() must be called before you
    // can use 'this'. Leaving this out will cause a reference error.
    this.name = 'Square';
  }
}
```

External Resources

- Extends - MDN
- Super operator - MDN
- Inheritance - MDN

Async Await

In addition to Promises, there is a new syntax you might encounter to handle asynchronous code named *async / await*.

The purpose of async/await functions is to simplify the behavior of using promises synchronously and to perform some behavior on a group of Promises. Just as Promises are similar to structured callbacks, async/await is similar to combining generators and promises. Async functions *always* return a Promise. (Ref: MDN)

> **Note :** You must understand what promises are and how they work before trying to understand async / await since they rely on it.

> **Note 2:** *await* must be used in an *async* function, which means

> that you can't use await in the top level of our code since that is not inside an async function.

Sample code

```
async function getGithubUser(username) { // async keyword allows usage of
  const response = await fetch(`https://api.github.com/users/${username}`
  return response.json();
}

getGithubUser('mbeaudru')
  .then(user => console.log(user)) // logging user response - cannot use
  .catch(err => console.log(err)); // if an error is thrown in our async
```

Explanation with sample code

Async / Await is built on promises but they allow a more imperative style of code.

The *async* operator marks a function as asynchronous and will always return a *Promise*. You can use the *await* operator in an *async* function to pause execution on that line until the returned Promise from the expression either resolves or rejects.

```
async function myFunc() {
  // we can use await operator because this function is async
  return "hello world";
}

myFunc().then(msg => console.log(msg)) // "hello world" -- myFunc's retur
```

When the *return* statement of an async function is reached, the Promise is fulfilled with the value returned. If an error is thrown inside an async function, the Promise state will turn to *rejected*. If no value is returned from an async function, a Promise is still returned and resolves with no value when execution of the async function is complete.

await operator is used to wait for a *Promise* to be fulfilled and can only be used inside an *async* function body. When encountered, the code execution is paused until the promise is fulfilled.

> **Note :** *fetch* is a function that returns a Promise that allows to do an AJAX request

Let's see how we could fetch a github user with promises first:

```
function getGithubUser(username) {
  return fetch(`https://api.github.com/users/${username}`).then(response
}

getGithubUser('mbeaudru')
  .then(user => console.log(user))
  .catch(err => console.log(err));
```

Here's the *async / await* equivalent:

```
async function getGithubUser(username) { // promise + await keyword usage
  const response = await fetch(`https://api.github.com/users/${username}`
  return response.json();
}

getGithubUser('mbeaudru')
  .then(user => console.log(user))
  .catch(err => console.log(err));
```

async / await syntax is particularly convenient when you need to chain promises that are interdependent.

For instance, if you need to get a token in order to be able to fetch a blog post on a database and then the author informations:

> **Note :** *await* expressions needs to be wrapped in parentheses to call its resolved value's methods and properties on the same line.

```
async function fetchPostById(postId) {
  const token = (await fetch('token_url')).json().token;
  const post = (await fetch(`/posts/${postId}?token=${token}`)).json();
  const author = (await fetch(`/users/${post.authorId}`)).json();

  post.author = author;
  return post;
}
```

```
fetchPostById('gzIrzeo64')
  .then(post => console.log(post))
  .catch(err => console.log(err));
```

Error handling

Unless we add *try / catch* blocks around *await* expressions, uncaught exceptions – regardless of whether they were thrown in the body of your *async* function or while it's suspended during *await* – will reject the promise returned by the *async* function. Using the `throw` statement in an async function is the same as returning a Promise that rejects. (Ref: PonyFoo).

> **Note :** Promises behave the same!

With promises, here is how you would handle the error chain:

```
function getUser() { // This promise will be rejected!
  return new Promise((res, rej) => rej("User not found !"));
}

function getAvatarByUsername(userId) {
  return getUser(userId).then(user => user.avatar);
}

function getUserAvatar(username) {
  return getAvatarByUsername(username).then(avatar => ({ username, avatar
}

getUserAvatar('mbeaudru')
  .then(res => console.log(res))
  .catch(err => console.log(err)); // "User not found !"
```

The equivalent with *async / await*:

```
async function getUser() { // The returned promise will be rejected!
  throw "User not found !";
}

async function getAvatarByUsername(userId) => {
  const user = await getUser(userId);
  return user.avatar;
}
```

```
async function getUserAvatar(username) {
  var avatar = await getAvatarByUsername(username);
  return { username, avatar };
}

getUserAvatar('mbeaudru')
  .then(res => console.log(res))
  .catch(err => console.log(err)); // "User not found !"
```

External resources

- Async/Await - JavaScript.Info
- ES7 Async/Await
- 6 Reasons Why JavaScript's Async/Await Blows Promises Away
- JavaScript awaits
- Using Async Await in Express with Node 8
- Async Function
- Await
- Using async / await in express with node 8

Truthy / Falsy

In JavaScript, a truthy or falsy value is a value that is being casted into a boolean when evaluated in a boolean context. An example of boolean context would be the evaluation of an `if` condition:

Every value will be casted to `true` unless they are equal to:

- `false`
- `0`
- `""` (empty string)
- `null`
- `undefined`
- `NaN`

Here are examples of *boolean context*:

- `if` condition evaluation

```
if (myVar) {}
```

`myVar` can be any first-class citizen (variable, function, boolean) but it will be casted into a boolean because it's evaluated in a boolean context.

- After logical **NOT** `!` operator

This operator returns false if its single operand can be converted to true; otherwise, returns true.

```
!0 // true -- 0 is falsy so it returns true
!!0 // false -- 0 is falsy so !0 returns true so !(!0) returns false
!!"" // false -- empty string is falsy so NOT (NOT false) equals false
```

- With the *Boolean* object constructor

```
new Boolean(0) // false
new Boolean(1) // true
```

- In a ternary evaluation

```
myVar ? "truthy" : "falsy"
```

myVar is evaluated in a boolean context.

Be careful when comparing 2 values. The object values (that should be cast to true) is **not** being casted to Boolean but it forced to convert into a primitive value one using ToPrimitives specification. Internally, when an object is compared to Boolean value like `[] == true`, it does `[].toString() == true` so...

```
let a = [] == true // a is false since [].toString() give "" back.
let b = [1] == true // b is true since [1].toString() give "1" back.
let c = [2] == true // c is false since [2].toString() give "2" back.
```

External resources

- Truthy (MDN)
- Falsy (MDN)
- Truthy and Falsy values in JS - Josh Clanton

Anamorphisms and Catamorphisms

Anamorphisms

Anamorphisms are functions that map from some object to a more complex structure containing the type of the object. It is the process of *unfolding* a simple structure into a more complex one. Consider unfolding an integer to a list of integers. The integer is our initial object and the list of integers is the more complex structure.

Sample code

```
function downToOne(n) {
  const list = [];

  for (let i = n; i > 0; --i) {
    list.push(i);
  }

  return list;
}

downToOne(5)
  //=> [ 5, 4, 3, 2, 1 ]
```

Catamorphisms

Catamorphisms are the opposite of Anamorphisms, in that they take objects of more complex structure and *fold* them into simpler structures. Take the following example `product` which take a list of integers and returns a single integer.

Sample code

```
function product(list) {
  let product = 1;

  for (const n of list) {
    product = product * n;
  }

  return product;
}

product(downToOne(5)) // 120
```

External resources

- Anamorphisms in JavaScript
- Anamorphism
- Catamorphism

Generators

Another way to write the `downToOne` function is to use a Generator. To instantiate a `Generator` object, one must use the `function *` declaration. Generators are functions that can be exited and later re-entered with its context (variable bindings) saved across re-entrances.

For example, the `downToOne` function above can be rewritten as:

```
function * downToOne(n) {
  for (let i = n; i > 0; --i) {
    yield i;
  }
}

[...downToOne(5)] // [ 5, 4, 3, 2, 1 ]
```

Generators return an iterable object. When the iterator's `next()` function is called, it is executed until the first `yield` expression, which specifies the value to be returned from the iterator or with `yield*`, which delegates to another generator function. When a `return` expression is called in the generator, it will mark the generator as done and pass back as the return value. Further calls to `next()` will not return any new values.

Sample code

```
// Yield Example
function * idMaker() {
  var index = 0;
  while (index < 2) {
    yield index;
    index = index + 1;
  }
}
```

```
var gen = idMaker();

gen.next().value; // 0
gen.next().value; // 1
gen.next().value; // undefined
```

The `yield*` expression enables a generator to call another generator function during iteration.

```
// Yield * Example
function * genB(i) {
  yield i + 1;
  yield i + 2;
  yield i + 3;
}

function * genA(i) {
  yield i;
  yield* genB(i);
  yield i + 10;
}

var gen = genA(10);

gen.next().value; // 10
gen.next().value; // 11
gen.next().value; // 12
gen.next().value; // 13
gen.next().value; // 20
```

```
// Generator Return Example
function* yieldAndReturn() {
  yield "Y";
  return "R";
  yield "unreachable";
}

var gen = yieldAndReturn()
gen.next(); // { value: "Y", done: false }
gen.next(); // { value: "R", done: true }
gen.next(); // { value: undefined, done: true }
```

External resources

- Mozilla MDN Web Docs, Iterators and Generators

Static Methods

Short explanation

The `static` keyword is used in classes to declare static methods. Static methods are functions in a class that belongs to the class object and are not available to any instance of that class.

Sample code

```
class Repo {
  static getName() {
    return "Repo name is modern-js-cheatsheet"
  }
}

// Note that we did not have to create an instance of the Repo class
console.log(Repo.getName()) // Repo name is modern-js-cheatsheet

let r = new Repo();
console.log(r.getName()) // Uncaught TypeError: r.getName is not a functi
```

Detailed explanation

Static methods can be called within another static method by using the `this` keyword, this doesn't work for non-static methods though. Non-static methods cannot directly access static methods using the `this` keyword.

Calling other static methods from a static method.

To call a static method from another static method, the `this` keyword can be used like so;

```
class Repo {
  static getName() {
    return "Repo name is modern-js-cheatsheet"
  }

  static modifyName() {
```

```
    return this.getName() + '-added-this'
  }
}

console.log(Repo.modifyName()) // Repo name is modern-js-cheatsheet-added
```

Calling static methods from non-static methods.

Non-static methods can call static methods in 2 ways;

1. Using the class name.

To get access to a static method from a non-static method we use the class name and call the static method like a property. e.g `ClassName.StaticMethodName`

```
class Repo {
  static getName() {
    return "Repo name is modern-js-cheatsheet"
  }

  useName() {
    return Repo.getName() + ' and it contains some really important stuff
  }
}

// we need to instantiate the class to use non-static methods
let r = new Repo()
console.log(r.useName()) // Repo name is modern-js-cheatsheet and it cont
```

1. Using the constructor

Static methods can be called as properties on the constructor object.

```
class Repo {
  static getName() {
    return "Repo name is modern-js-cheatsheet"
  }

  useName() {
    // Calls the static method as a property of the constructor
    return this.constructor.getName() + ' and it contains some really imp
  }
}
```

```
// we need to instantiate the class to use non-static methods
let r = new Repo()
console.log(r.useName()) // Repo name is modern-js-cheatsheet and it cont
```

External resources

- static keyword- MDN
- Static Methods- Javascript.info
- Static Members in ES6- OdeToCode

Glossary

Scope

The context in which values and expressions are “visible,” or can be referenced. If a variable or other expression is not “in the current scope,” then it is unavailable for use.

Source: MDN

Variable mutation

A variable is said to have been mutated when its initial value has changed afterward.

```
var myArray = [];
myArray.push("firstEl") // myArray is being mutated
```

A variable is said to be *immutable* if it can’t be mutated.

Check MDN Mutable article for more details.
