3/25/2022

Nagendra Mirajkar

Zensar AES Practices

ES6

ECMAScript 6

Table of Contents

[Introduction 1](#_Toc99093732)

[JavaScript let: Declaring Block-Scoped Variables 1](#_Toc99093733)

[Introduction to the JavaScript let keyword 1](#_Toc99093734)

[JavaScript let and global object 2](#_Toc99093735)

[JavaScript let and callback function in a for loop 2](#_Toc99093736)

[Redeclaration 4](#_Toc99093737)

[JavaScript let variables and hoisting 4](#_Toc99093738)

[Temporal death zone (TDZ) 5](#_Toc99093739)

[**Summary** 6](#_Toc99093740)

[Differences Between var and let 6](#_Toc99093741)

[#1: Variable scopes 6](#_Toc99093742)

[#2: Creating global properties 8](#_Toc99093743)

[#3: Redeclaration 8](#_Toc99093744)

[#4: The Temporal dead zone 8](#_Toc99093745)

[**The var variables** 8](#_Toc99093746)

[**The let variables** 9](#_Toc99093747)

[JavaScript const: Declaring Constants in ES6 9](#_Toc99093748)

[Introduction to the JavaScript const keyword 9](#_Toc99093749)

[JavaScript const and Objects 10](#_Toc99093750)

[JavaScript const and Arrays 11](#_Toc99093751)

[JavaScript const in a for loop 11](#_Toc99093752)

[Summary 12](#_Toc99093753)

[JavaScript Default Parameters 12](#_Toc99093754)

[Arguments vs. Parameters 13](#_Toc99093755)

[Setting JavaScript default parameters for a function 13](#_Toc99093756)

[More JavaScript default parameter examples 14](#_Toc99093757)

[**1) Passing undefined arguments** 14](#_Toc99093758)

[**2) Evaluating default parameters** 15](#_Toc99093759)

[**3) Using other parameters in default values** 16](#_Toc99093760)

[**Using functions** 17](#_Toc99093761)

[**The arguments object** 17](#_Toc99093762)

[JavaScript Rest Parameters 17](#_Toc99093763)

[Introduction to JavaScript rest parameters 17](#_Toc99093764)

[More JavaScript rest parameters examples 18](#_Toc99093765)

[JavaScript rest parameters and arrow function 20](#_Toc99093766)

[JavaScript rest parameter in a dynamic function 20](#_Toc99093767)

[JavaScript Spread Operator 21](#_Toc99093768)

[Introduction to the JavaScript spread operator 21](#_Toc99093769)

[JavaScript spread operator and apply() method 22](#_Toc99093770)

[A better way to use the Array’s push() method example 23](#_Toc99093771)

[JavaScript spread operator and array manipulation 24](#_Toc99093772)

[**1) Constructing array literal** 24](#_Toc99093773)

[**2) Concatenating arrays** 24](#_Toc99093774)

[**3) Copying an array** 24](#_Toc99093775)

[**JavaScript spread operator and strings** 24](#_Toc99093776)

[Summary 24](#_Toc99093777)

[Object Literal Syntax Extensions in ES6 25](#_Toc99093778)

[Object property initializer shorthand 25](#_Toc99093779)

[Computed property name 26](#_Toc99093780)

[Concise method syntax 27](#_Toc99093781)

[JavaScript for…of Loop 28](#_Toc99093782)

[Introduction to the JavaScript for…of loop 28](#_Toc99093783)

[**variable** 28](#_Toc99093784)

[**iterable** 29](#_Toc99093785)

[JavaScript for of loop examples 29](#_Toc99093786)

[**1) Iterating over arrays** 29](#_Toc99093787)

[**2) In-place object destructuring with for…of** 30](#_Toc99093788)

[**3) Iterating over strings** 31](#_Toc99093789)

[**3) Iterating over Map objects** 31](#_Toc99093790)

[**4) Iterating over set objects** 31](#_Toc99093791)

[for...of vs. for...in 32](#_Toc99093792)

[Summary 33](#_Toc99093793)

[A Quick Look at Octal and Binary Literals in ES6 33](#_Toc99093794)

[Octal literals 33](#_Toc99093795)

[Binary literals 34](#_Toc99093796)

[Summary 34](#_Toc99093797)

[JavaScript Template Literals In Depth 34](#_Toc99093798)

[The basic syntax of JavaScript template literals 35](#_Toc99093799)

[Multiline strings 35](#_Toc99093800)

[Variable and expression substitutions 37](#_Toc99093801)

[Tagged templates 38](#_Toc99093802)

[Summary 39](#_Toc99093803)

[ES6 Destructuring Assignment 39](#_Toc99093804)

[Introduction to JavaScript Array destructuring 40](#_Toc99093805)

[Array Destructuring Assignment and Rest syntax 41](#_Toc99093806)

[Setting default values 41](#_Toc99093807)

[Nested array destructuring 43](#_Toc99093808)

[Array Destructuring Assignment Applications 43](#_Toc99093809)

[**1) Swapping variables** 43](#_Toc99093810)

[**2) Functions that return multiple values** 44](#_Toc99093811)

[JavaScript Object Destructuring 44](#_Toc99093812)

[Introduction to the JavaScript object destructuring assignment 44](#_Toc99093813)

[Setting default values 46](#_Toc99093814)

[Destructuring a null object 47](#_Toc99093815)

[Nested object destructuring 47](#_Toc99093816)

[Destructuring function arguments 48](#_Toc99093817)

[Summary 49](#_Toc99093818)

[A Comprehensive Look at ES6 Modules 49](#_Toc99093819)

[Executing modules on web browsers 49](#_Toc99093820)

[Exporting 50](#_Toc99093821)

[Importing 51](#_Toc99093822)

[**Import a single binding** 52](#_Toc99093823)

[**Import multiple bindings** 53](#_Toc99093824)

[**Import an entire module as an object** 53](#_Toc99093825)

[Limitation of import and export statements 54](#_Toc99093826)

[Aliasing 54](#_Toc99093827)

[Re-exporting a binding 55](#_Toc99093828)

[Importing without bindings 56](#_Toc99093829)

[Default exports 56](#_Toc99093830)

[JavaScript Class 57](#_Toc99093831)

[Classes prior to ES6 revisited 57](#_Toc99093832)

[ES6 class declaration 58](#_Toc99093833)

[Class vs. Custom type 59](#_Toc99093834)

[Summary 60](#_Toc99093835)

[JavaScript Getters and Setters 60](#_Toc99093836)

[Introduction to the JavaScript getters and setters 60](#_Toc99093837)

[Using getter in an object literal 64](#_Toc99093838)

[Summary 64](#_Toc99093839)

[JavaScript Class Expressions 64](#_Toc99093840)

[Introduction to JavaScript class expressions 65](#_Toc99093841)

[First-class citizen 66](#_Toc99093842)

[Singleton 66](#_Toc99093843)

[Summary 68](#_Toc99093844)

[JavaScript Static Methods 68](#_Toc99093845)

[Introduction to the JavaScript static methods 68](#_Toc99093846)

[JavaScript static methods in ES6 69](#_Toc99093847)

[Calling a static method from the class constructor or an instance method 69](#_Toc99093848)

[Summary 70](#_Toc99093849)

[JavaScript Static Properties 70](#_Toc99093850)

[Introduction to the JavaScript static properties 70](#_Toc99093851)

[Summary 72](#_Toc99093852)

[JavaScript Computed Property 72](#_Toc99093853)

[Introduction to JavaScript Computed Property 72](#_Toc99093854)

[JavaScript Inheritance Using extends & super 73](#_Toc99093855)

[Implementing JavaScript inheritance using extends and super 73](#_Toc99093856)

[Shadowing methods 76](#_Toc99093857)

[Inheriting static members 77](#_Toc99093858)

[Inheriting from built-in types 78](#_Toc99093859)

[Summary 78](#_Toc99093860)

[Introduction to JavaScript new.target Metaproperty 79](#_Toc99093861)

[Introduction to JavaScript new.target 79](#_Toc99093862)

[JavaScript new.target in functions 79](#_Toc99093863)

[JavaScript new.target in constructors 80](#_Toc99093864)

[An Introduction to JavaScript Arrow Functions 81](#_Toc99093865)

[Introduction to JavaScript arrow functions 81](#_Toc99093866)

[**JavaScript arrow functions with multiple parameters** 82](#_Toc99093867)

[**JavaScript arrow functions with a single parameter** 83](#_Toc99093868)

[**JavaScript arrow functions with no parameter** 83](#_Toc99093869)

[Line break between parameter definition and arrow 83](#_Toc99093870)

[Statements & expressions in the arrow function body 84](#_Toc99093871)

[JavaScript arrow functions and object literal 84](#_Toc99093872)

[Arrow function vs. regular function 85](#_Toc99093873)

[**JavaScript arrow functions and this value** 85](#_Toc99093874)

[**JavaScript arrow functions and the arguments object** 87](#_Toc99093875)

[**JavaScript arrow functions and the prototype property** 87](#_Toc99093876)

[Summary 87](#_Toc99093877)

[When You Should Not Use Arrow Functions 88](#_Toc99093878)

[1) Event handlers 88](#_Toc99093879)

[2) Object methods 89](#_Toc99093880)

[3) Prototype methods 90](#_Toc99093881)

[4) Functions that use the arguments object 90](#_Toc99093882)

[Summary 91](#_Toc99093883)

[The Ultimate Guide to JavaScript Symbol 91](#_Toc99093884)

[Creating symbols 91](#_Toc99093885)

[Sharing symbols 92](#_Toc99093886)

[Symbol usages 93](#_Toc99093887)

[**A) Using symbols as unique values** 93](#_Toc99093888)

[**B) Using symbol as the computed property name of an object** 93](#_Toc99093889)

[Well-known symbols 94](#_Toc99093890)

[**Symbol.hasInstance** 94](#_Toc99093891)

[**Symbol.iterator** 95](#_Toc99093892)

[**Symbol.isConcatSpreadable** 96](#_Toc99093893)

[**Symbol.toPrimitive** 98](#_Toc99093894)

[The Essential Guide to JavaScript Iterators 99](#_Toc99093895)

[The for loop issues 99](#_Toc99093896)

[Iteration protocols 100](#_Toc99093897)

[**Iterator protocol** 100](#_Toc99093898)

[**Iterable protocol** 100](#_Toc99093899)

[Iterators 101](#_Toc99093900)

[Cleaning up 102](#_Toc99093901)

[JavaScript Generators 103](#_Toc99093902)

[Introduction to JavaScript Generators 103](#_Toc99093903)

[More generator examples 106](#_Toc99093904)

[Using generators to implement iterators 106](#_Toc99093905)

[Using a generator to implement the Bag data structure 108](#_Toc99093906)

[Summary 109](#_Toc99093907)

[JavaScript yield 109](#_Toc99093908)

[Introduction to the JavaScript yield keyword 109](#_Toc99093909)

[JavaScript yield examples 109](#_Toc99093910)

[**A) Returning a value** 109](#_Toc99093911)

[**B) Returning undefined** 110](#_Toc99093912)

[**C) Passing a value to the next() method** 110](#_Toc99093913)

[**D) Using yield in an array** 111](#_Toc99093914)

[**E) Using yield to return an array** 112](#_Toc99093915)

[**F) Using the yield to return individual elements of an array** 112](#_Toc99093916)

[JavaScript Promises 113](#_Toc99093917)

[Why JavaScript promises 113](#_Toc99093918)

[**Using callbacks to deal with an asynchronous operation** 115](#_Toc99093919)

[Understanding JavaScript Promises 116](#_Toc99093920)

[Creating a promise 117](#_Toc99093921)

[Consuming a Promise: then, catch, finally 118](#_Toc99093922)

[**1) The then() method** 118](#_Toc99093923)

[**2) The catch() method** 121](#_Toc99093924)

[**3) The finally() method** 121](#_Toc99093925)

[A practical JavaScript Promise example 122](#_Toc99093926)

[Summary 125](#_Toc99093927)

[Promise Chaining 125](#_Toc99093928)

[Introduction to the JavaScript promise chaining 125](#_Toc99093929)

[Multiple handlers for a promise 127](#_Toc99093930)

[Returning a Promise 129](#_Toc99093931)

[**Promise chaining syntax** 130](#_Toc99093932)

[JavaScript Promise.all() 132](#_Toc99093933)

[Introduction to the JavaScript Promise.all() method 132](#_Toc99093934)

[JavaScript Promise.all() method examples 132](#_Toc99093935)

[**1) Resolved promises example** 133](#_Toc99093936)

[**2) Rejected promises example** 134](#_Toc99093937)

[Summary 134](#_Toc99093938)

[JavaScript Promise.race() 135](#_Toc99093939)

[Introduction to JavaScript Promise.race() static method 135](#_Toc99093940)

[JavaScript Promise.race() examples 136](#_Toc99093941)

[**1) Simple JavaScript Promise.race() examples** 136](#_Toc99093942)

[**2) Practical JavaScript Promise.race() example** 137](#_Toc99093943)

[Summary 141](#_Toc99093944)

[Promise Error Handling 141](#_Toc99093945)

[Normal error 141](#_Toc99093946)

[Errors inside the Promises 142](#_Toc99093947)

[Calling reject() function 143](#_Toc99093948)

[Missing the catch() method 144](#_Toc99093949)

[Summary 145](#_Toc99093950)

[JavaScript Map Object 145](#_Toc99093951)

[Introduction to JavaScript Map object 145](#_Toc99093952)

[Useful JavaScript Map methods 145](#_Toc99093953)

[JavaScript Map examples 146](#_Toc99093954)

[**Create a new Map object** 146](#_Toc99093955)

[**Add elements to a Map** 146](#_Toc99093956)

[**Initialize a map with an iterable object** 147](#_Toc99093957)

[**Get an element from a map by key** 147](#_Toc99093958)

[**Check the existence of an element by key** 147](#_Toc99093959)

[**Get the number of elements in the map** 147](#_Toc99093960)

[**Iterate over map keys** 147](#_Toc99093961)

[**Iterate over map values** 148](#_Toc99093962)

[**Iterate over map elements** 148](#_Toc99093963)

[**Convert map keys or values to a array** 149](#_Toc99093964)

[**Delete an element by key** 150](#_Toc99093965)

[**Delete all elements in the map** 150](#_Toc99093966)

[WeakMap 150](#_Toc99093967)

[The Beginner’s Guide to JavaScript Set Type in ES6 151](#_Toc99093968)

[Introduction to the JavaScript Set object 151](#_Toc99093969)

[Useful Set methods 151](#_Toc99093970)

[JavaScript Set examples 152](#_Toc99093971)

[**Create a new Set from an Array** 152](#_Toc99093972)

[**Get the size of a Set** 152](#_Toc99093973)

[**Add elements to a Set** 153](#_Toc99093974)

[**Check if a value is in the Set** 153](#_Toc99093975)

[**Remove elements from a set** 153](#_Toc99093976)

[**Looping the elements of a JavaScript Set** 154](#_Toc99093977)

[**Invoke a callback function on each element of a set** 154](#_Toc99093978)

[WeakSets 155](#_Toc99093979)

[JavaScript Array.of() 155](#_Toc99093980)

[Introduction to the JavaScript Array.of() method 155](#_Toc99093981)

[JavaScript Array.of() examples 156](#_Toc99093982)

[JavaScript Array.of() polyfill 156](#_Toc99093983)

[JavaScript Array.from() 157](#_Toc99093984)

[Introduction to JavaScript Array Array.from() method 157](#_Toc99093985)

[JavaScript Array.from() method examples 158](#_Toc99093986)

[**A) Create an array from an array-like object** 158](#_Toc99093987)

[**B) JavaScript Array Array.from() with a mapping function** 158](#_Toc99093988)

[**C) JavaScript Array Array.from() with a this value** 159](#_Toc99093989)

[**D) Create an array from an iterable object** 159](#_Toc99093990)

[JavaScript Array find() Method 160](#_Toc99093991)

[Introduction to the Array find() method 160](#_Toc99093992)

[**Arguments** 160](#_Toc99093993)

[**1) callback** 160](#_Toc99093994)

[**2) thisArg** 161](#_Toc99093995)

[**Return value** 161](#_Toc99093996)

[JavaScript find() examples 161](#_Toc99093997)

[JavaScript Array findIndex() Method 162](#_Toc99093998)

[Introduction to the JavaScript Array findIndex() Method 162](#_Toc99093999)

[**1) testFn** 162](#_Toc99094000)

[**2) thisArg** 162](#_Toc99094001)

[JavaScript Array findIndex() examples 163](#_Toc99094002)

[**1) Using the Array findIndex() method with a simple array example** 163](#_Toc99094003)

[**2) Using the Array findIndex() method with a more complex condition** 163](#_Toc99094004)

[**3) Using the Array findIndex() method with an array of objects** 163](#_Toc99094005)

[JavaScript Object.assign() 164](#_Toc99094006)

[Using JavaScript Object.assign() to clone an object 164](#_Toc99094007)

[Using JavaScript Object.assign() to merge objects 165](#_Toc99094008)

[Summary 166](#_Toc99094009)

[JavaScript Object.is() 166](#_Toc99094010)

[Negative zero 166](#_Toc99094011)

[NaN 167](#_Toc99094012)

[JavaScript String startsWith 168](#_Toc99094013)

[**Arguments** 169](#_Toc99094014)

[JavaScript String startsWith() examples 169](#_Toc99094015)

[JavaScript String endsWith 170](#_Toc99094016)

[**Arguments** 170](#_Toc99094017)

[JavaScript String endsWith() examples 170](#_Toc99094018)

[JavaScript String includes() Method 171](#_Toc99094019)

[Introduction to JavaScript String includes() method 172](#_Toc99094020)

[JavaScript String includes() examples 172](#_Toc99094021)

[JavaScript Proxy 173](#_Toc99094022)

[What is a JavaScript Proxy object 173](#_Toc99094023)

[Creating a proxy object 173](#_Toc99094024)

[A simple proxy example 174](#_Toc99094025)

[Proxy Traps 175](#_Toc99094026)

[**The get() trap** 175](#_Toc99094027)

[**The set() trap** 176](#_Toc99094028)

[The apply() trap 177](#_Toc99094029)

[More traps 178](#_Toc99094030)

[JavaScript Reflection 178](#_Toc99094031)

[What is reflection 178](#_Toc99094032)

[Reflect API 179](#_Toc99094033)

[**Creating objects: Reflect.construct()** 179](#_Toc99094034)

[**Calling a function: Reflect.apply()** 180](#_Toc99094035)

[**Defining a property: Reflect.defineProperty()** 181](#_Toc99094036)

Introduction

ECMAScript 2015 or ES2015 is a significant update to the JavaScript programming language. It is the first major update to the language since ES5 which was standardized in 2009. Therefore, ES2015 is often called ES6.

JavaScript let: Declaring Block-Scoped Variables

**Summary**: in this tutorial, you will learn how to use the JavaScript let keyword to declare block-scoped variables.

## **Introduction to the JavaScript let keyword**

In ES5, when you [declare a variable](https://www.javascripttutorial.net/javascript-variables/) using the var keyword, the scope of the variable is either global or local. If you declare a variable outside of a function, the scope of the variable is global. When you declare a variable inside a function, the scope of the variable is local.

ES6 provides a new way of declaring a variable by using the let keyword. The let keyword is similar to the var keyword, except that these variables are blocked-scope. For example:

let variable\_name;

Code language: JavaScript (javascript)

In JavaScript, blocks are denoted by curly braces {} , for example, the [if else](https://www.javascripttutorial.net/javascript-if-else/), [for](https://www.javascripttutorial.net/javascript-for-loop/), [do while](https://www.javascripttutorial.net/javascript-do-while/), [while](https://www.javascripttutorial.net/javascript-while-loop/), [try catch](https://www.javascripttutorial.net/javascript-try-catch/) and so on:

if(condition) {

*// inside a block*

}

Code language: JavaScript (javascript)

See the following example:

let x = 10;

if (x == 10) {

let x = 20;

console.log(x); *// 20: reference x inside the block*

}

console.log(x); *// 10: reference at the begining of the script*

Code language: JavaScript (javascript)

How the script works:

* First, declare a variable x and initialize its value to 10.
* Second, declare a new variable with the same name x inside the if block but with an initial value of 20.
* Third, output the value of the variable x inside and after the  [if](https://www.javascripttutorial.net/javascript-if-else/) block.

Because the let keyword declares a block-scoped variable, the x variable inside the if block is a **new variable**and it shadows the x variable declared at the top of the script. Therefore, the value of x in the console is 20.

When the JavaScript engine completes executing the if block, the x variable inside the if block is out of scope. Therefore, the value of the x variable that following the if block is 10.

## **JavaScript let and global object**

When you declare a global variable using the var keyword, you add that variable to the property list of the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/). In the case of the web browser, the global object is the window. For example:

var a = 10;

console.log(window.a); *// 10*

Code language: JavaScript (javascript)

However, when you use the let keyword to declare a variable, that variable is **not** attached to the global object as a property. For example:

let b = 20;

console.log(window.b); *// undefined*

Code language: JavaScript (javascript)

## **JavaScript let and callback function in a for loop**

See the following example.

for (var i = 0; i < 5; i++) {

setTimeout(function () {

console.log(i);

}, 1000);

}

Code language: JavaScript (javascript)

The intention of the code is to output numbers from 0 to 4 to the console every second. However, it outputs the number 5 five times:

5

5

5

5

5

In this example, the variable i is a global variable. After the loop, its value is 5. When the callback functions are passed to the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function executes, they reference the same variable i with the value 5.

In ES5, you can fix this issue by creating another scope so that each callback function references a new variable. And to create a new scope, you need to create a function. Typically, you use the [IIFE](https://www.javascripttutorial.net/javascript-immediately-invoked-function-expression-iife/) pattern as follows:

for (var i = 0; i < 5; i++) {

(function (j) {

setTimeout(function () {

console.log(j);

}, 1000);

})(i);

}

Code language: JavaScript (javascript)

Output:

0

1

2

3

4

In ES6, the let keyword declares a new variable in each loop iteration. Therefore, you just need to replace the var keyword with the let keyword to fix the issue:

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

setTimeout(function () {

console.log(i);

}, 1000);

}

Code language: JavaScript (javascript)

To make the code completely ES6 style, you can use an [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) as follows:

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

setTimeout(() => console.log(i), 1000);

}

Code language: JavaScript (javascript)

Note that you’ll learn more about the [arrow functions in the later tutorial](https://www.javascripttutorial.net/es6/javascript-arrow-function/).

## **Redeclaration**

The var keyword allows you to redeclare a variable without any issue:

var counter = 0;

var counter;

console.log(counter); *// 0*

Code language: JavaScript (javascript)

However, redeclaring a variable using the let keyword will result in an error:

let counter = 0;

let counter;

console.log(counter);

Code language: JavaScript (javascript)

Here’s the error message:

Uncaught SyntaxError: Identifier 'counter' has already been declared

Code language: JavaScript (javascript)

## **JavaScript let variables and hoisting**

Let’s examine the following example:

{

console.log(counter); *//*

let counter = 10;

}

Code language: JavaScript (javascript)

This code causes an error:

Uncaught ReferenceError: Cannot access 'counter' before initialization

Code language: JavaScript (javascript)

In this example, accessing the counter variable before declaring it causes a ReferenceError. You may think that a variable declaration using the let keyword does not **hoist,**but it does**.**

In fact, the JavaScript engine will hoist a variable declared by the let keyword to the top of the block. However, the JavaScript engine does not initialize the variable. Therefore, when you reference an uninitialized variable, you’ll get a ReferenceError.

## **Temporal death zone (TDZ)**

A variable declared by the let keyword has a so-called temporal dead zone (TDZ). The TDZ is the time from the start of the block until the variable declaration is processed.

The following example illustrates that the temporal dead zone is time-based, not location-based.

{ *// enter new scope, TDZ starts*

let log = function () {

console.log(message); *// messagedeclared later*

};

*// This is the TDZ and accessing log*

*// would cause a ReferenceError*

let message= 'Hello'; *// TDZ ends*

log(); *// called outside TDZ*

}

Code language: JavaScript (javascript)

In this example:

First, the curly brace starts a new block scope, therefore, the TDZ starts.

Second, the log() function expression accesses the message variable. However, the log() function has not been executed yet.

Third, declare the message variable and initialize its value to 10. The time from the start of the block scope to the time that the message variable is accessed is called a temporal death zone. When the JavaScript engine processes the declaration, the TDZ ends.

Finally, call the log() function that accesses the message variable outside of the TDZ.

Note that if you access a variable declared by the let keyword in the TDZ, you’ll get a ReferenceError as illustrated in the following example.

{ *// TDZ starts*

console.log(typeof myVar); *// undefined*

console.log(typeof message); *// ReferenceError*

let message; *// TDZ ends*

}

Code language: JavaScript (javascript)

Notice that myVar variable is a non-existing variable, therefore, its type is [undefined](https://www.javascripttutorial.net/javascript-data-types/#undefined).

The temporal death zone prevents you from accidentally referencing a variable before its declaration.

### **Summary**

* Variables are declared using the let keyword are block-scoped, are not initialized to any value, and are not attached to the global object.
* Redeclaring a variable using the let keyword will cause an error.
* A temporal dead zone of a variable declared using the let keyword starts from the block until the initialization is evaluated.

# Differences Between var and let

**Summary**: in this tutorial, you will learn about the differences between the var and let keywords.

## **#1: Variable scopes**

The var variables belong to the global scope when you define them outside a function. For example:

var counter;

Code language: JavaScript (javascript)

In this example, the counter is a global variable. It means that the counter variable is accessible by any functions.

When you declare a variable inside a function using the var keyword, the scope of the variable is local. For example:

function increase() {

var counter = 10;

}

*// cannot access the counter variable here*

Code language: JavaScript (javascript)

In this example, the counter variable is local to the increase() function. It cannot be accessible outside of the function.

The following example displays four numbers from 0 to 4 inside the loop and the number 5 outside the loop.

for (var i = 0; i < 5; i++) {

console.log("Inside the loop:", i);

}

console.log("Outside the loop:", i);

Code language: JavaScript (javascript)

Output:

Inside the loop: 0

Inside the loop: 1

Inside the loop: 2

Inside the loop: 3

Inside the loop: 4

Outside the loop: 5

Code language: Shell Session (shell)

In this example, the i variable is a global variable. Therefore, it can be accessed from both inside and after the [for](https://www.javascripttutorial.net/javascript-for-loop/) loop.

The following example uses the let keyword instead of the var keyword:

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

console.log("Inside the loop:", i);

}

console.log("Outside the loop:", i);

Code language: JavaScript (javascript)

In this case, the code shows four numbers from 0 to 4 inside a loop and a reference error:

Inside the loop: 0

Inside the loop: 1

Inside the loop: 2

Inside the loop: 3

Inside the loop: 4

The error:

Uncaught ReferenceError: i is not defined

Code language: JavaScript (javascript)

Since this example uses the let keyword, the variable i is blocked scope. It means that the variable i only exists and can be accessible inside the for loop block.

In JavaScript, a block is delimited by a pair of curly braces {} like in the if...else and for statements:

if(condition) {

*// inside a block*

}

for(...) {

*// inside a block*

}

Code language: JavaScript (javascript)

## **#2: Creating global properties**

The global var variables are added to the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) as [properties](https://www.javascripttutorial.net/javascript-object-properties/). The global object is window on the web browser and global on Node.js:

var counter = 0;

console.log(window.counter); *// 0*

Code language: JavaScript (javascript)

However, the let variables are not added to the global object:

let counter = 0;

console.log(window.counter); *// undefined*

Code language: JavaScript (javascript)

## **#3: Redeclaration**

The var keyword allows you to redeclare a variable without any issue:

var counter = 10;

var counter;

console.log(counter); *// 10*

Code language: JavaScript (javascript)

However, if you redeclare a variable with the let keyword, you will get an error:

let counter = 10;

let counter; *// error*

Code language: JavaScript (javascript)

## **#4: The Temporal dead zone**

The let variables have temporal dead zones while the var variables don’t. To understand the temporal dead zone, let’s examine the life cycles of both var and let variables, which have two steps: creation and execution.

### **The var variables**

* In the creation phase, the JavaScript engine assigns storage spaces to var variables and immediately initializes them to undefined.
* In the execution phase, the JavaScript engine assigns the var variables the values specified by the assignments if there are ones. Otherwise, the var variables remain undefined.

See the [execution context](https://www.javascripttutorial.net/javascript-execution-context/) for more information.

### **The let variables**

* In the creation phase, the JavaScript engine assigns storage spaces to the let variables but does not initialize the variables. Referencing uninitialized variables will cause a ReferenceError.
* The let variables have the same execution phase as the var variables.

The temporal dead zone starts from the block until the let variable declaration is processed. In other words, it is the location where you cannot access the let variables before they are defined.

In this tutorial, you have learned about the differences between var and let keywords.

# JavaScript const: Declaring Constants in ES6

**Summary**: in this tutorial, you’ll learn how to define constants by using the JavaScript const keyword.

## **Introduction to the JavaScript const keyword**

ES6 provides a new way of declaring a constant by using the const keyword. The const keyword creates a read-only reference to a value.

const CONSTANT\_NAME = value;

Code language: JavaScript (javascript)

By convention, the constant identifiers are in uppercase.

Like the [let](https://www.javascripttutorial.net/es6/javascript-let/) keyword, the const keyword declares blocked-scope variables. However, the block-scoped variables declared by the const keyword can’t be **reassigned**.

The variables declared by the let keyword are mutable. It means that you can change their values anytime you want as shown in the following example:

let a = 10;

a = 20;

a = a + 5;

console.log(a); *// 25*

Code language: JavaScript (javascript)

However, variables created by the const keyword are “immutable”. In other words, you can’t reassign them to different values.

If you attempt to reassign a variable declared by the const keyword, you’ll get a TypeError like this:

const RATE = 0.1;

RATE = 0.2; *// TypeError*

Code language: JavaScript (javascript)

Unlike the let keyword, you need to initialize the value to the variable declared by the const keyword.

The following example causes a SyntaxError due to missing the initializer in the const variable declaration:

const RED; *// SyntaxError*

Code language: JavaScript (javascript)

## **JavaScript const and Objects**

The const keyword ensures that the variable it creates is read-only. However, it doesn’t mean that the actual value to which the const variable reference is immutable. For example:

const person = { age: 20 };

person.age = 30; *// OK*

console.log(person.age); *// 30*

Code language: JavaScript (javascript)

Even though the person variable is a constant, you can change the value of its property.

However, you cannot reassign a different value to the person constant like this:

person = { age: 40 }; *// TypeError*

Code language: JavaScript (javascript)

If you want the value of the person object to be immutable, you have to freeze it by using the Object.freeze() method:

const person = Object.freeze({age: 20});

person.age = 30; *// TypeError*

Code language: JavaScript (javascript)

Note that Object.freeze() is shallow, meaning that it can freeze the properties of the object, not the objects referenced by the properties.

For example, the company object is constant and frozen.

const company = Object.freeze({

name: 'ABC corp',

address: {

street: 'North 1st street',

city: 'San Jose',

state: 'CA',

zipcode: 95134

}

});

Code language: JavaScript (javascript)

But the company.address object is not immutable, you can add a new property to the company.address object as follows:

company.address.country = 'USA'; *// OK*

Code language: JavaScript (javascript)

## **JavaScript const and Arrays**

Consider the following example:

const colors = ['red'];

colors.push('green');

console.log(colors); *// ["red", "green"]*

colors.pop();

colors.pop();

console.log(colors); *// []*

colors = []; *// TypeError*

Code language: JavaScript (javascript)

In this example, we declare an array colors that has one element using the const keyword. Then, we can change the array’s elements by adding the green color. However, we cannot reassign the array colors to another array.

## **JavaScript const in a for loop**

ES6 provides a new construct called [for...of](https://www.javascripttutorial.net/es6/javascript-for-of/) that allows you to create a loop iterating over iterable objects such as [arrays](https://www.javascripttutorial.net/javascript-array/), [maps](https://www.javascripttutorial.net/es6/javascript-map/), and [sets](https://www.javascripttutorial.net/es6/javascript-set/).

let scores = [75, 80, 95];

for (let score of scores) {

console.log(score);

}

Code language: JavaScript (javascript)

If you don’t intend to modify the score variable inside the loop, you can use the const keyword instead:

let scores = [75, 80, 95];

for (const score of scores) {

console.log(score);

}

Code language: JavaScript (javascript)

In this example, the for...of   creates a new binding for the const keyword in each loop iteration. In other words, a new score constant is created in each iteration.

Notice that the const will not work in an imperative [for](https://www.javascripttutorial.net/javascript-for-loop/)loop. Trying to use the const keyword to declare a variable in the imperative for loop will result in a TypeError:

for (const i = 0; i < scores.length; i++) { *// TypeError*

console.log(scores[i]);

}

Code language: JavaScript (javascript)

The reason is that the declaration is only evaluated once before the loop body starts.

## **Summary**

* The const keyword creates a read-only reference to a value. The readonly reference cannot be reassigned but the value can be change.
* The variables declared by the const keyword are blocked-scope and cannot be redeclared.

# JavaScript Default Parameters

**Summary**: in this tutorial, you will learn how to handle JavaScript default parameters in ES6.

function say(message='Hi') {

console.log(message);

}

say(); *// 'Hi'*

say('Hello') *// 'Hello'*

Code language: JavaScript (javascript)

The default value of the message paramater in the say() function is 'Hi'.

In JavaScript, default function parameters allow you to initialize named parameters with default values if no values or undefined are passed into the function.

## **Arguments vs. Parameters**

Sometimes, you can use the terms argument and parameter interchangeably. However, by definition, parameters are what you specify in the [function declaration](https://www.javascripttutorial.net/javascript-function/) whereas the arguments are what you pass into the function.

Consider the following add() function:

function add(x, y) {

return x + y;

}

add(100,200);

Code language: JavaScript (javascript)

In this example, the x and y are the parameters of the add() function, and the values passed to the add() function 100 and 200 are the arguments.

## **Setting JavaScript default parameters for a function**

In JavaScript, a parameter has a default value of [undefined](https://www.javascripttutorial.net/javascript-data-types/#undefined). It means that if you don’t pass the arguments into the [function](https://www.javascripttutorial.net/javascript-function/), its parameters will have the default values of undefined.

See the following example:

function say(message) {

console.log(message);

}

say(); *// undefined*

Code language: JavaScript (javascript)

The say() function takes the message parameter. Because we didn’t pass any argument into the say() function, the value of the message parameter is undefined.

Suppose that you want to give the message parameter a default value 10.

A typical way for achieving this is to test parameter value and assign a default value if it is undefined using a [ternary operator](https://www.javascripttutorial.net/javascript-ternary-operator/):

function say(message) {

message = typeof message !== 'undefined' ? message : 'Hi';

console.log(message);

}

say(); *// 'Hi'*

Code language: JavaScript (javascript)

In this example, we didn’t pass any value into the say() function. Therefore, the default value of the message argument is undefined. Inside the function, we reassigned the message variable the Hi string.

ES6 provides you with an easier way to set the default values for the function parameters like this:

function fn(param1=default1, param2=default2,..) {

}

Code language: JavaScript (javascript)

In the syntax above, you use the [assignment operator](https://www.javascripttutorial.net/javascript-assignment-operators/) (=) and the default value after the parameter name to set a default value for that parameter. For example:

function say(message='Hi') {

console.log(message);

}

say(); *// 'Hi'*

say(undefined); *// 'Hi'*

say('Hello'); *// 'Hello'*

Code language: JavaScript (javascript)

How it works.

* In the first function call, we didn’t pass any argument into the say() function, therefore message parameter took the default value 'Hi'.
* In the second function call, we passed the undefined into the say() function, hence the message parameter also took the default value 'Hi'.
* In the third function call, we passed the 'Hello' string into the say() function, therefore message parameter took the string 'Hello' as the default value.

## **More JavaScript default parameter examples**

Let’s look at some more examples to learn some available options for setting default values of the function parameters.

### **1) Passing undefined arguments**

The following createDiv() function creates a new <div> element in the document with a specific height, width, and border style:

function createDiv(height = '100px', width = '100px', border = 'solid 1px red') {

let div = document.createElement('div');

div.style.height = height;

div.style.width = width;

div.style.border = border;

document.body.appendChild(div);

return div;

}

Code language: JavaScript (javascript)

The following doesn’t pass any arguments to the function so the createDiv() function uses the default values for the parameters.

createDiv();

Suppose you want to use the default values for the height and width parameters and specific border style. In this case, you need to pass undefined values to the first two parameters as follows:

createDiv(undefined,undefined,'solid 5px blue');

Code language: JavaScript (javascript)

### **2) Evaluating default parameters**

JavaScript engine evaluates the default arguments at the time you call the function. See the following example:

function put(toy, toyBox = []) {

toyBox.push(toy);

return toyBox;

}

console.log(put('Toy Car'));

*// -> ['Toy Car']*

console.log(put('Teddy Bear'));

*// -> ['Teddy Bear'], not ['Toy Car','Teddy Bear']*

Code language: JavaScript (javascript)

The parameter can take a default value which is a result of a function.

Consider the following example:

function date(d = today()) {

console.log(d);

}

function today() {

return (new Date()).toLocaleDateString("en-US");

}

date();

Code language: JavaScript (javascript)

The date() function takes one parameter whose default value is the returned value of the today() function. The today() function returns today’s date in a specified string format.

When we declared the date() function, the today() function has not yet evaluated until we called the date() function.

We can use this feature to make arguments are mandatory. If the caller doesn’t pass any argument, we throw an error as follows:

function requiredArg() {

throw new Error('The argument is required');

}

function add(x = requiredArg(), y = requiredArg()){

return x + y;

}

add(10); *// error*

add(10,20); *// OK*

Code language: JavaScript (javascript)

### **3) Using other parameters in default values**

You can assign a parameter a default value that references to other default parameters as shown in the following example:

function add(x = 1, y = x, z = x + y) {

return x + y + z;

}

console.log(add()); *// 4*

Code language: JavaScript (javascript)

In the add() function:

* The default value of the y is set to x parameter.
* The default value of the z is the sum of x and y
* The add() function returns the sum of x, y, and z.

The parameter list seems to have its own [scope](https://www.javascripttutorial.net/javascript-variable-scope/). If you reference the parameter that has not been initialized yet, you will get an error. For example:

function subtract( x = y, y = 1 ) {

return x - y;

}

subtract(10);

Code language: JavaScript (javascript)

Error message:

Uncaught ReferenceError: Cannot access 'y' before initialization

Code language: JavaScript (javascript)

### **Using functions**

You can use a return value of a function as a default value for a parameter. For example:

let taxRate = () => 0.1;

let getPrice = function( price, tax = price \* taxRate() ) {

return price + tax;

}

let fullPrice = getPrice(100);

console.log(fullPrice); *// 110*

Code language: JavaScript (javascript)

In the getPrice() function, we called the taxRate() function to get the tax rate and use this tax rate to calculate the tax amount from the price.

### **The arguments object**

The value of the arguments object inside the function is the number of actual arguments that you pass to the function. For example:

function add(x, y = 1, z = 2) {

console.log( arguments.length );

return x + y + z;

}

add(10); *// 1*

add(10, 20); *// 2*

add(10, 20, 30); *// 3*

Code language: JavaScript (javascript)

Now, you should understand the JavaScript default function parameters and how to use them effectively.

# JavaScript Rest Parameters

**Summary**: in this tutorial, you will learn how to use the JavaScript rest parameters to gather parameters and put them all in an array.

## **Introduction to JavaScript rest parameters**

ES6 provides a new kind of parameter so-called rest parameter that has a prefix of three dots (...). A rest parameter allows you to represent an indefinite number of arguments as an [array](https://www.javascripttutorial.net/javascript-array/). See the following syntax:

function fn(a,b,...args) {

*//...*

}

Code language: JavaScript (javascript)

The last parameter (args) is prefixed with the three-dots ( ...). It’s called a rest parameter ( ...args).

All the arguments you pass to the [function](https://www.javascripttutorial.net/javascript-function/) will map to the parameter list. In the syntax above, the first argument maps to a, the second one maps to b, and the third, the fourth, etc., will be stored in the rest parameter args as an array. For example:

fn(1, 2, 3, "A", "B", "C");

Code language: JavaScript (javascript)

The args array stores the following values:

[3,'A','B','C']

Code language: JSON / JSON with Comments (json)

If you pass only the first two parameters, the rest parameter will be an empty array:

fn(1,2);

The args will be:

[]

Code language: JSON / JSON with Comments (json)

Notice that the rest parameters must appear at the end of the argument list. The following code will result in an error:

function fn(a,...rest, b) {

*// error*

}

Code language: JavaScript (javascript)

Error:

SyntaxError: Rest parameter must be last formal parameter

Code language: JavaScript (javascript)

## **More JavaScript rest parameters examples**

See the following example:

function sum(...args) {

let total = 0;

for (const a of args) {

total += a;

}

return total;

}

sum(1, 2, 3);

Code language: JavaScript (javascript)

The output of the script is:

6

In this example, args in an array. Therefore, you could use the [for..of](https://www.javascripttutorial.net/es6/javascript-for-of/) loop to iterate over its elements and sum them up.

Assuming that the caller of the sum() function may pass arguments with various kinds of data types such as [number](https://www.javascripttutorial.net/javascript-data-types/#number), [string](https://www.javascripttutorial.net/javascript-data-types/#string), and [boolean](https://www.javascripttutorial.net/javascript-data-types/#boolean), and you want to calculate the total of numbers only:

function sum(...args) {

return args

.filter(function (e) {

return typeof e === 'number';

})

.reduce(function (prev, curr) {

return prev + curr;

});

}

Code language: JavaScript (javascript)

The following script uses the new sum() function to sum only numeric arguments:

let result = sum(10,'Hi',null,undefined,20);

console.log(result);

Code language: JavaScript (javascript)

Output:

30

Note that without the rest parameters, you have to use the arguments object of the function.

However, the arguments object itself is not an instance of the Array type. Therefore, you cannot use the filter() method directly. In ES5, you have to use Array.prototype.filter.call() as follows:

function sum() {

return Array.prototype.filter

.call(arguments, function (e) {

return typeof e === 'number';

})

.reduce(function (prev, curr) {

return prev + curr;

});

}

Code language: JavaScript (javascript)

As you see, the rest parameter makes the code more elegant. Suppose you need to filter the arguments based on a specific type such as numbers, strings, boolean, and null. The following function helps you to do it:

function filterBy(type, ...args) {

return args.filter(function (e) {

return typeof e === type;

});

}

Code language: JavaScript (javascript)

## **JavaScript rest parameters and arrow function**

An [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) does not have the arguments object. Therefore, if you want to pass some arguments to the arrow function, you must use the rest parameters. See the following example:

const combine = (...args) => {

return args.reduce(function (prev, curr) {

return prev + ' ' + curr;

});

};

let message = combine('JavaScript', 'Rest', 'Parameters'); *// =>*

console.log(message); *// JavaScript Rest Parameters*

Code language: JavaScript (javascript)

Output:

JavaScript Rest Parameters

The combine() function is an arrow that takes an indefinite number of arguments and concatenates these arguments.

## **JavaScript rest parameter in a dynamic function**

JavaScript allows you to create dynamic functions through the [Function](https://www.javascripttutorial.net/javascript-function-type/) constructor. And it is possible to use the rest parameter in a dynamic function. Here is an example:

var showNumbers = new Function('...numbers', 'console.log(numbers)');

showNumbers(1, 2, 3);

Code language: PHP (php)

Output:

[ 1, 2, 3 ]

Code language: JSON / JSON with Comments (json)

In this tutorial, you have learned how to use the JavaScript rest parameter to represent an indefinite number of arguments as an array.

# JavaScript Spread Operator

**Summary**: in this tutorial, you will learn about the JavaScript spread operator that spreads out elements of an iterable object.

## **Introduction to the JavaScript spread operator**

ES6 provides a new operator called spread operator that consists of three dots (...). The spread operator allows you to spread out elements of an iterable object such as an [array](https://www.javascripttutorial.net/javascript-array/), [map](https://www.javascripttutorial.net/es6/javascript-map/), or [set](https://www.javascripttutorial.net/es6/javascript-set/). For example:

const odd = [1,3,5];

const combined = [2,4,6, ...odd];

console.log(combined);

Code language: JavaScript (javascript)

Output:

[ 2, 4, 6, 1, 3, 5 ]

Code language: JSON / JSON with Comments (json)

In this example, the three dots ( ...) located in front of the odd array is the spread operator. The spread operator (...) unpacks the elements of the odd array.

Note that ES6 also has the three dots ( ...) which is a [rest parameter](https://www.javascripttutorial.net/es6/javascript-rest-parameters/) that collects all remaining arguments of a function into an array.

function f(a, b, ...args) {

console.log(args);

}

f(1, 2, 3, 4, 5);

Code language: JavaScript (javascript)

Output:

[ 3, 4, 5 ]

Code language: JSON / JSON with Comments (json)

In this example, the rest parameter (...) collects the arguments 3, 4, and 5 into an array args. So the three dots ( ...) represent both the spread operator and the rest parameter.

Here are the main differences:

* The spread operator (...) unpacks the elements of an iterable object.
* The rest parameter (...) packs the elements into an array.

The rest parameters must be the last arguments of a [function](https://www.javascripttutorial.net/javascript-function/). However, the spread operator can be anywhere:

const odd = [1,3,5];

const combined = [...odd, 2,4,6];

console.log(combined);

Code language: JavaScript (javascript)

Output:

[ 1, 3, 5, 2, 4, 6 ]

Code language: JSON / JSON with Comments (json)

Or

const odd = [1,3,5];

const combined = [2,...odd, 4,6];

console.log(combined);

Code language: JavaScript (javascript)

Output:

[ 2, 1, 3, 5, 4, 6 ]

Code language: JSON / JSON with Comments (json)

Note that ES2018 expands the spread operator to objects, which is known as [object spread](https://www.javascripttutorial.net/es-next/javascript-object-spread/).

Let’s look at some scenarios where you can use the spread operators.

## **JavaScript spread operator and apply() method**

See the following compare() function compares two numbers:

function compare(a, b) {

return a - b;

}

Code language: JavaScript (javascript)

In ES5, to pass an array of two numbers to the compare() function, you often use the [apply()](https://www.javascripttutorial.net/javascript-apply-method/) method as follows:

let result = compare.apply(null, [1, 2]);

console.log(result); *// -1*

Code language: JavaScript (javascript)

However, by using the spread operator, you can pass an array of two numbers to the compare() function:

let result = compare(...[1, 2]);

console.log(result); *// -1*

Code language: JavaScript (javascript)

The spread operator spreads out the elements of the array so a is 1 and b is 2 in this case.

## **A better way to use the Array’s push() method example**

Sometimes, a function may accept an indefinite number of arguments. Filling arguments from an array is not convenient.

For example, the push() method of an array object allows you to add one or more elements to an array. If you want to pass an array to the push() method, you need to use apply() method as follows:

let rivers = ['Nile', 'Ganges', 'Yangte'];

let moreRivers = ['Danube', 'Amazon'];

[].push.apply(rivers, moreRivers);

console.log(rivers);

Code language: JavaScript (javascript)

This solution looks verbose.

The following example uses the spread operator to improve the readability of the code:

rivers.push(...moreRivers);

Code language: CSS (css)

As you can see, using the spread operator is much cleaner.

## **JavaScript spread operator and array manipulation**

### **1) Constructing array literal**

The spread operator allows you to insert another array into the initialized array when you construct an array using the literal form. See the following example:

let initialChars = ['A', 'B'];

let chars = [...initialChars, 'C', 'D'];

console.log(chars); *// ["A", "B", "C", "D"]*

Code language: JavaScript (javascript)

### **2) Concatenating arrays**

Also, you can use the spread operator to concatenate two or more arrays:

let numbers = [1, 2];

let moreNumbers = [3, 4];

let allNumbers = [...numbers, ...moreNumbers];

console.log(allNumbers); *// [1, 2, 3, 4]*

Code language: JavaScript (javascript)

### **3) Copying an array**

In addition, you can copy an array instance by using the spread operator:

let scores = [80, 70, 90];

let copiedScores = [...scores];

console.log(copiedScores); *// [80, 70, 90]*

Code language: JavaScript (javascript)

Note that the spread operator only copies the array itself to the new one, not the elements. This means that the copy is shallow, not deep.

### **JavaScript spread operator and strings**

Consider the following example:

let chars = ['A', ...'BC', 'D'];

console.log(chars); *// ["A", "B", "C", "D"]*

Code language: JavaScript (javascript)

In this example, we constructed the chars array from individual strings. When we applied the spread operator to the 'BC'string, it spreads out each individual character of the string 'BC' into individual characters.

## **Summary**

* The spread operator is denoted by three dots (…).
* The spread operator unpacks elements of iterable objects such as arrays, sets, and maps into a list.
* The rest paramter is also denoted by three dots (…). However, it packs the remaining arguments of a function into an array.
* The spread operator can be used to clone an iterable object or merge iterable objects into one.

# Object Literal Syntax Extensions in ES6

**Summary**: in this tutorial, you will learn about the syntax extensions of the object literal in ES6 that make your code cleaner and more flexible.

The [object](https://www.javascripttutorial.net/javascript-objects/) literal is one of the most popular [patterns for creating objects in JavaScript](https://www.javascripttutorial.net/create-objects-in-javascript/) because of its simplicity. ES6 makes the object literal more succinct and powerful by extending the syntax in some ways.

## **Object property initializer shorthand**

Prior to ES6, an object literal is a collection of name-value pairs. For example:

function createMachine(name, status) {

return {

name: name,

status: status

};

}

Code language: JavaScript (javascript)

The createMachine() [function](https://www.javascripttutorial.net/javascript-function/) takes two arguments name and status and returns a new object literal with two properties: name and status.

The name and status properties take the values of the name and status parameters. This syntax looks redundant because name and status mentioned twice in both the name and value of properties.

ES6 allows you to eliminate the duplication when a property of an object is the same as the local variable name by including the name without a colon and value.

For example, you can rewrite the createMachine() function in ES6 as follows:

function createMachine(name, status) {

return {

name,

status

};

}

Code language: JavaScript (javascript)

Internally, when a property of an object literal only has a name, the JavaScript engine searches for a variable with the same name in the surrounding scope. If the JavaScript engine can find one, it assigns the property the value of the variable.

In this example, the JavaScript engine assigns the name and status property values of the name and status arguments.

Similarly, you can construct an object literal from local variables as shown in the following example:

let name = 'Computer',

status = 'On';

let machine = {

name,

status

};

Code language: JavaScript (javascript)

## **Computed property name**

Prior to ES6, you could use the square brackets( [])  to enable the **computed property names** for the properties on objects.

The square brackets allow you to use the string literals and variables as the property names.

See the following example:

let name = 'machine name';

let machine = {

[name]: 'server',

'machine hours': 10000

};

console.log(machine[name]); *// server*

console.log(machine['machine hours']); *// 10000*

Code language: JavaScript (javascript)

The name variable was initialized to a value of 'machine name'. Since both properties of the machine object contains a space, you can only reference them using the square brackets.

In ES6, the computed property name is a part of the object literal syntax, and it uses the square bracket notation.

When a property name is placed inside the square brackets, the JavaScript engine evaluates it as a string. It means that you can use an expression as a property name. For example:

let prefix = 'machine';

let machine = {

[prefix + ' name']: 'server',

[prefix + ' hours']: 10000

};

console.log(machine['machine name']); *// server*

console.log(machine['machine hours']); *// 10000*

Code language: JavaScript (javascript)

The machine object’s properties evaluate to 'machine name' and 'machine hours', therefore you can reference them as the properties of the machine object.

## **Concise method syntax**

Prior to ES6, when defining a method for an object literal, you need to specify the name and full function definition as shown in the following example:

let server = {

name: "Server",

restart: function () {

console.log("The" + this.name + " is restarting...");

}

};

Code language: JavaScript (javascript)

ES6 makes the syntax for making a method of the object literal more succinct by removing the colon (:) and the function keyword.

The following example rewrites the server object above using the ES6 syntax.

let server = {

name: 'Server',

restart() {

console.log("The" + this.name + " is restarting...");

}

};

Code language: JavaScript (javascript)

This shorthand syntax is also known as the **concise method syntax**. It’s valid to have spaces in the property name. For example:

let server = {

name: 'Server',

restart() {

console.log("The " + this.name + " is restarting...");

},

'starting up'() {

console.log("The " + this.name + " is starting up!");

}

};

server['starting up']();

Code language: JavaScript (javascript)

In this example, the method 'starting up' has spaces in its name. To call the method, you use the following syntax:

object\_name['property name']();

Code language: CSS (css)

In this tutorial, you have learned how to use some new object literal syntax extensions in ES6 including property initializer shorthand, computed properties, and concise method syntax.

# JavaScript for…of Loop

**Summary**: in this tutorial, you’ll how to use JavaScript for...of statement to iterate over iterable objects.

## **Introduction to the JavaScript for…of loop**

ES6 introduced a new statement for...of that iterates over an iterable object such as:

* Built-in [Array](https://www.javascripttutorial.net/javascript-array/), [String](https://www.javascripttutorial.net/javascript-string/), [Map](https://www.javascripttutorial.net/es6/javascript-map/), [Set](https://www.javascripttutorial.net/es6/javascript-set/), …
* Array-like objects such as arguments or NodeList
* User-defined objects that implement the [iterator protocol](https://www.javascripttutorial.net/es6/javascript-iterator/).

The following illustrates the syntax of the for...of:

for (variable of iterable) {

*// ...*

}

Code language: JavaScript (javascript)

### **variable**

In each iteration, a property of the iterable object is assigned to the variable. You can use [var](https://www.javascripttutorial.net/javascript-variables/), [let](https://www.javascripttutorial.net/es6/javascript-let/), or [const](https://www.javascripttutorial.net/es6/javascript-const/) to declare the variable.

### **iterable**

The iterable is an object whose iterable properties are iterated.

## **JavaScript for of loop examples**

Let’s take a look at some examples of using the for...of loop.

### **1) Iterating over arrays**

The following example shows how to use the for...of to iterate over elements of an array:

let scores = [80, 90, 70];

for (let score of scores) {

score = score + 5;

console.log(score);

}

Code language: JavaScript (javascript)

Output:

85

95

75

In this example, the for...of iterates over every element of the scores array. It assigns the element of the scores array to the variable score in each iteration.

If you don’t change the variable inside the loop, you should use the [const](https://www.javascripttutorial.net/es6/javascript-const/) keyword instead of the [let](https://www.javascripttutorial.net/es6/javascript-let/) keyword as follows:

let scores = [80, 90, 70];

for (const score of scores) {

console.log(score);

}

Code language: JavaScript (javascript)

Output:

80

90

70

To access the index of the array elements inside the loop, you can use the for...of statement with the entries() method of the array.

The array.entries() method returns a pair of [index, element] in each iteration. For example:

let colors = ['Red', 'Green', 'Blue'];

for (const [index, color] of colors.entries()) {

console.log(`${color} is at index ${index}`);

}

Code language: JavaScript (javascript)

Output:

Red is at index 0

Green is at index 1

Blue is at index 2

In this example, we used the [array destructuring](https://www.javascripttutorial.net/es6/destructuring/)to assign the result of the entries() method to the index and color variables in each iteration:

const [index, color] of colors.entries()

Code language: CSS (css)

### **2) In-place object destructuring with for…of**

Consider the following example:

const ratings = [

{user: 'John',score: 3},

{user: 'Jane',score: 4},

{user: 'David',score: 5},

{user: 'Peter',score: 2},

];

let sum = 0;

for (const {score} of ratings) {

sum += score;

}

console.log(`Total scores: ${sum}`); *// 14*

Code language: JavaScript (javascript)

Output:

Total scores: 14

How it works:

* The ratings is an array of objects. Each object has two properties user and score.
* The for...of iterate over the ratings array and calculate the total scores of all objects.
* The expression const {score} of ratings uses [object destructing](https://www.javascripttutorial.net/es6/javascript-object-destructuring/) to assign the score property of the current iterated element to the score variable.

### **3) Iterating over strings**

The following example uses the for...of loop to iterate over characters of a string.

let str = 'abc';

for (let c of str) {

console.log(c);

}

Code language: JavaScript (javascript)

Output:

a

b

c

### **3) Iterating over Map objects**

The following example illustrates how to use the for...of statement to iterate over a Map object.

let colors = new Map();

colors.set('red', '#ff0000');

colors.set('green', '#00ff00');

colors.set('blue', '#0000ff');

for (let color of colors) {

console.log(color);

}

Code language: JavaScript (javascript)

Output:

[ 'red', '#ff0000' ]

[ 'green', '#00ff00' ]

[ 'blue', '#0000ff' ]

Code language: JSON / JSON with Comments (json)

### **4) Iterating over set objects**

The following example shows how to iterate over a set object using the for...of loop:

let nums = new Set([1, 2, 3]);

for (let num of nums) {

console.log(num);

}

Code language: JavaScript (javascript)

## **for...of vs. for...in**

The for...in iterates over all [enumerable properties](https://www.javascripttutorial.net/javascript-enumerable-properties/) of an object. It doesn’t iterate over a collection such as Array, Map or Set.

Unlike the for...in loop, the for...of iterates a collection, rather than an object. In fact, the for...of iterates over elements of any collection that has the [[Symbol.iterator]](https://www.javascripttutorial.net/es6/symbol/#iterator) property.

The following example illustrates the differences between for...of and for...in

let scores = [10,20,30];

scores.message = 'Hi';

console.log("for...in:");

for (let score in scores) {

console.log(score);

}

console.log('for...of:');

for (let score of scores) {

console.log(score);

}

Code language: JavaScript (javascript)

Output:

for...in:

0

1

2

message

for...of:

10

20

30

Code language: CSS (css)

In this example, the for…in statement iterates over the properties of the scores array:

for...in:

0

1

2

message

Code language: CSS (css)

while the for…of iterates over the element of an array:

for...of:

10

20

30

Code language: CSS (css)

## **Summary**

* Use the for...of loop to iterate over elements of an iterable object.

# A Quick Look at Octal and Binary Literals in ES6

**Summary**: in this tutorial, you will learn how to represent the octal and binary literals in ES6.

ES5 provided numeric literals in octal (prefix 0), decimal (no prefix), and hexadecimal (0x). ES6 added support for binary literals and changed how it represents octal literals.

## **Octal literals**

To represent an octal literal in ES5, you use the zero prefix (0) followed by a sequence of octal digits (from 0 to 7). For example:

let a = 051;

console.log(a); *// 41*

Code language: JavaScript (javascript)

If the octal literal contains a number that is out of range, JavaScript ignores the leading 0 and treats the octal literal as a decimal, as shown in the following example:

let b = 058; *// invalid octal*

console.log(b); *// 58*

Code language: JavaScript (javascript)

In this example, since 8 is an invalid digit for representing the octal number, JavaScript ignores the 0 and treats the whole number as a decimal with a value of 58.

Note you can use the octal literals in non-strict mode. If you use them in strict mode, JavaScript will throw an error.

"use strict"

let b = 058; *// invalid octal*

console.log(b);

Code language: JavaScript (javascript)

Here is the error message:

SyntaxError: Decimals with leading zeros are not allowed in strict mode.

Code language: JavaScript (javascript)

ES6 allows you to specify the octal literal by using the prefix 0o followed by a sequence of octal digits from 0 through 7. Here is an example:

let c = 0o51;

console.log(c); *// 41*

Code language: JavaScript (javascript)

If you use an invalid number in the octal literal, JavaScript will throw a SyntaxError as shown in the following example:

let d = 0o58;

console.log(d); *// SyntaxError*

Code language: JavaScript (javascript)

## **Binary literals**

In ES5, JavaScript didn’t provide any literal form for binary numbers. To parse a binary string, you use the parseInt() function as follows:

let e = parseInt('111',2);

console.log(e); *// 7*

Code language: JavaScript (javascript)

ES6 added support for binary literals by using the 0b prefix followed by a sequence of binary numbers (0 and 1). Here is an example:

let f = 0b111;

console.log(f); *// 7*

Code language: JavaScript (javascript)

## **Summary**

* Octal literals start with 0o followed by a sequence of numbers between 0 and 7.
* Binary literals start with 0b followed by a sequence of number 0 and 1.

# JavaScript Template Literals In Depth

**Summary**: in this tutorial, you will learn about JavaScript template literal, which allows you to work with a string template more easily.

Before ES6, you use single quotes (') or double quotes (") to wrap a string literal. And the strings have very limited functionality.

To enable you to solve more complex problems, ES6 template literals provide the syntax that allows you to work with strings more safely and cleanly.

In ES6, you create a template literal by wrapping your text in backticks (`) as follows:

let simple = `This is a template literal`;

Code language: JavaScript (javascript)

and you get the following features:

* A multiline string: a string that can span multiple lines.
* String formatting: the ability to substitute part of the string for the values of variables or expressions. This feature is also called string interpolation.
* HTML escaping: the ability to transform a string so that it is safe to include in HTML.

## **The basic syntax of JavaScript template literals**

As mentioned earlier, instead of using the single quotes or double quotes, a template literal uses backticks, as shown in the following example:

let str = `Template literal in ES6`;

console.log(str);*// Template literal in ES6*

console.log(str.length); *// 23*

console.log(typeof str);*// string*

Code language: JavaScript (javascript)

Using the backticks, you can freely use the single or double quotes in the template literal without escaping.

let anotherStr = `Here's a template literal`;

Code language: JavaScript (javascript)

If a string contains a backtick, you must escape it using a backslash (\) :

let strWithBacktick = `Template literals use backticks \` insead of quotes`;

Code language: JavaScript (javascript)

## **Multiline strings**

Before ES6, you use the following technique to create a multi-line string by manually including the newline character ( \n) in the string as follows:

let msg = 'Multiline \n\

string';

console.log(msg);

*//Multiline*

*//string*

Code language: JavaScript (javascript)

Note that the backslash ( \) placed after the newline character ( \n) indicates the continuation of the string rather than a new line.

This technique, however, is not consistent across JavaScript engines. Therefore, it was pretty common to create a multiline string that relies on an array and [string concatenation](https://www.javascripttutorial.net/javascript-string-concat/) as follows:

let msg = ['This text',

'can',

'span multiple lines'].join('\n');

Code language: JavaScript (javascript)

The template literals allow you to define multiline strings more easily because you need to add a new line in the string wherever you want:

let p =

`This text

can

span multiple lines`;

Code language: JavaScript (javascript)

Note that the whitespace is a part of the string. Therefore, you need to ensure that the text lines up with proper indentation. Suppose you have a post object:

let post = {

title: 'JavaScript Template Literals',

excerpt: 'Introduction to JavaScript template literals in ES6',

body: 'Content of the post will be here...',

tags: ['es6', 'template literals', 'javascript']

};

Code language: JavaScript (javascript)

The following code returns the HTML code of the post object. Note that we use [the object destructuring technique](https://www.javascripttutorial.net/es6/javascript-object-destructuring/) to assign the properties of the post object to individual variables : title, excerpt, body, and tags.

let {title, excerpt, body, tags} = post;

let postHtml = `**<article>**

**<header>**

**<h1>**${title}**</h1>**

**</header>**

**<section>**

**<div>**${excerpt}**</div>**

**<div>**${body}**</div>**

**</section>**

**<footer>**

**<ul>**

${tags.map(tag => `**<li>**${tag}**</li>**`).join('\n ')}

**</ul>**

**</footer>**`;

Code language: HTML, XML (xml)

The following is the output of the variable postHtml. Notice how we used the spacing to indent the <li> tags correctly.

**<article>**

**<header>**

**<h1>**JavaScript Template Literals**</h1>**

**</header>**

**<section>**

**<div>**Introduction to JavaScript template literals in ES6**</div>**

**<div>**Content of the post will be here...**</div>**

**</section>**

**<footer>**

**<ul>**

**<li>**es6**</li>**

**<li>**template literals**</li>**

**<li>**javascript**</li>**

**</ul>**

**</footer>**

Code language: HTML, XML (xml)

## **Variable and expression substitutions**

At this point, a template literal is just like a better version of a regular JavaScript string. The big difference between a template literal and a regular string is substitutions.

The substitutions allow you to embed variables and expressions in a string. The JavaScript engine will automatically replace these variables and expressions with their values. This feature is known as string interpolation.

To instruct JavaScript to substitute a variable and expression, you place the variable and expression in a special block as follows:

${variable\_name}

See the following example:

let firstName = 'John',

lastName = 'Doe';

let greeting = `Hi ${firstName}, ${lastName}`;

console.log(greeting); *// Hi John, Doe*

Code language: JavaScript (javascript)

The substitution ${firstName} and ${lastName} access the variables  firstName and lastName to insert their values into the greeting string.

The greeting variable then holds the result of the substitutions. The following example substitutes an expression instead:

let price = 8.99,

tax = 0.1;

let netPrice = `Net Price:$${(price \* (1 + tax)).toFixed(2)}`;

console.log(netPrice); *// netPrice:$9.89*

Code language: JavaScript (javascript)

## **Tagged templates**

A template tag carries the transformation on the template literal and returns the result string.

You place the tag at the beginning of the template before the backtick (`) character as follows:

let greeting = tag`Hi`;

Code language: JavaScript (javascript)

In this example, tag is the template tag that applies to the Hi template literal. The tag can be any function with the following signature:

function tag(literals, ...substitutions) {

*// return a string*

}

Code language: JavaScript (javascript)

In this function:

* The literals parameter is an array that contains the literal strings.
* The substitutions parameter contains the subsequent arguments interpreted for each substitution.

See the following example:

function format(literals, ...substitutions) {

let result = '';

for (let i = 0; i < substitutions.length; i++) {

result += literals[i];

result += substitutions[i];

}

*// add the last literal*

result += literals[literals.length - 1];

return result;

}

let quantity = 9,

priceEach = 8.99,

result = format`${quantity} items cost $${(quantity \* priceEach).toFixed(2)}.`;

console.log(result); *// 9 items cost $80.91.*

Code language: JavaScript (javascript)

In this example, the format() function accepts three arguments: the literals array and two other arguments stored in the substitutions array.

The first argument is the literals array that contains three elements:

* An empty string before the first substitution (”). Note that the first argument of the literals array is an empty string.
* A string 'items cost' that locates between the first and the second substitutions.
* A string that follows the second substitution ('.')

The second argument is 9, which is the interpreted value of the quantity variable. It becomes the first element of the substitutions array. The third argument is 80.91, which is the interpreted value of the expression (quantity \* priceEach).toFixed(2). It becomes the second element of the substitutions array.

## **Summary**

* Use the backtick to create a string literal for string interpolation.

# ES6 Destructuring Assignment

**Summary**: in this tutorial, you will learn how to use the ES6 destructuring assignment that allows you to destructure an array into individual variables.

ES6 provides a new feature called destructing assignment that allows you to [destructure properties of an object](https://www.javascripttutorial.net/es6/javascript-object-destructuring/) or elements of an [array](https://www.javascripttutorial.net/javascript-array/) into individual variables.

Let’s start with the array destructuring.

## **Introduction to JavaScript Array destructuring**

Assuming that you have a function that returns an array of numbers as follows:

function getScores() {

return [70, 80, 90];

}

Code language: JavaScript (javascript)

The following invokes the getScores() function and assigns the returned value to a variable:

let scores = getScores();

Code language: JavaScript (javascript)

To get the individual score, you need to do like this:

let x = scores[0],

y = scores[1],

z = scores[2];

Code language: JavaScript (javascript)

Prior to ES6, there was no direct way to assign the elements of the returned array to multiple variables such as x, y and z.

Fortunately, starting from ES6, you can use the destructing assignment as follows:

let [x, y, z] = getScores();

console.log(x); *// 70*

console.log(y); *// 80*

console.log(z); *// 90*

Code language: JavaScript (javascript)

The variables x, y and z will take the values of the first, second, and third elements of the returned array.

Note that the square brackets [] look like the array syntax but they are not.

If the getScores() function returns an array of two elements, the third variable will be undefined, like this:

function getScores() {

return [70, 80];

}

let [x, y, z] = getScores();

console.log(x); *// 70*

console.log(y); *// 80*

console.log(z); *// undefined*

Code language: JavaScript (javascript)

In case the getScores() function returns an array that has more than three elements, the remaining elements are discarded. For example:

function getScores() {

return [70, 80, 90, 100];

}

let [x, y, z] = getScores();

console.log(x); *// 70*

console.log(y); *// 80*

console.log(z); *// 90*

Code language: JavaScript (javascript)

## **Array Destructuring Assignment and Rest syntax**

It’s possible to take all remaining elements of an array and put them in a new array by using the [rest syntax](https://www.javascripttutorial.net/es6/javascript-rest-parameters/) (...):

let [x, y ,...args] = getScores();

console.log(x); *// 70*

console.log(y); *// 80*

console.log(args); *// [90, 100]*

Code language: JavaScript (javascript)

The variables x and y receive values of the first two elements of the returned array. And the args variable receives all the remaining arguments, which are the last two elements of the returned array.

Note that it’s possible to destructure an array in the assignment that separates from the variable’s declaration. For example:

let a, b;

[a, b] = [10, 20];

console.log(a); *// 10*

console.log(b); *// 20*

Code language: JavaScript (javascript)

## **Setting default values**

See the following example:

function getItems() {

return [10, 20];

}

let items = getItems();

let thirdItem = items[2] != undefined ? items[2] : 0;

console.log(thirdItem); *// 0*

Code language: JavaScript (javascript)

How it works:

* First, declare the getItems() function that returns an array of two numbers.
* Then, assign the items variable to the returned array of the getItems() function.
* Finally, check if the third element exists in the array. If not, assign the value 0 to the thirdItem variable.

It’ll be simpler with the destructuring assignment with a default value:

let [, , thirdItem = 0] = getItems();

console.log(thirdItem); *// 0*

Code language: JavaScript (javascript)

If the value taken from the array is undefined, you can assign the variable a default value, like this:

let a, b;

[a = 1, b = 2] = [10];

console.log(a); *// 10*

console.log(b); *// 2*

Code language: JavaScript (javascript)

If the getItems() function doesn’t return an array and you expect an array, the destructing assignment will result in an error. For example:

function getItems() {

return null;

}

let [x = 1, y = 2] = getItems();

Code language: JavaScript (javascript)

Error:

Uncaught TypeError: getItems is not a function or its return value is not iterable

Code language: JavaScript (javascript)

A typical way to solve this is to fallback the returned value of the getItems() function to an empty array like this:

function getItems() {

return null;

}

let [a = 10, b = 20] = getItems() || [];

console.log(a); *// 10*

console.log(b); *// 20*

Code language: JavaScript (javascript)

## **Nested array destructuring**

The following function returns an array that contains an element which is another array, or nested array:

function getProfile() {

return [

'John',

'Doe',

['Red', 'Green', 'Blue']

];

}

Code language: JavaScript (javascript)

Since the third element of the returned array is another array, you need to use the nested array destructuring syntax to destructure it, like this:

let [

firstName,

lastName,

[

color1,

color2,

color3

]

] = getProfile();

console.log(color1, color2, color3); *// Red Green Blue*

Code language: JavaScript (javascript)

## **Array Destructuring Assignment Applications**

Let’s see some practical examples of using the array destructuring assignment syntax.

### **1) Swapping variables**

The array destructuring makes it easy to swap values of variables without using a temporary variable:

let a = 10,

b = 20;

[a, b] = [b, a];

console.log(a); *// 20*

console.log(b); *// 10*

Code language: JavaScript (javascript)

### **2) Functions that return multiple values**

In JavaScript, a function can return a value. However, you can return an array that contains multiple values, for example:

function stat(a, b) {

return [

a + b,

(a + b) / 2,

a - b

]

}

Code language: JavaScript (javascript)

And then you use the array destructuring assignment syntax to destructure the elements of the return array into variables:

let [sum, average, difference] = stat(20, 10);

console.log(sum, average, difference); *// 30, 15, 10*

Code language: JavaScript (javascript)

In this tutorial, you have learned how to use the ES6 destructuring assignment to destructure elements in an array into individual variables.

# JavaScript Object Destructuring

**Summary**: in this tutorial, you’ll learn about JavaScript object destructuring that assigns properties of an object to individual variables.

If you want to learn how to destructure an [array](https://www.javascripttutorial.net/javascript-array/), you can check out the [array destructuring tutorial](https://www.javascripttutorial.net/es6/destructuring/).

## **Introduction to the JavaScript object destructuring assignment**

Suppose you have a person object with two properties: firstName and lastName.

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

Prior to ES6, when you want to assign properties of the person object to variables, you typically do it like this:

let firstName = person.firstName;

let lastName = person.lastName;

Code language: JavaScript (javascript)

ES6 introduces the object destructuring syntax that provides an alternative way to assign [properties](https://www.javascripttutorial.net/javascript-object-properties/) of an [object](https://www.javascripttutorial.net/javascript-objects/) to variables:

let { firstName: fname, lastName: lname } = person;

Code language: JavaScript (javascript)

In this example, the firstName and lastName properties are assigned to the fName and lName variables respectively.

In this syntax:

let { property1: variable1, property2: variable2 } = object;

Code language: JavaScript (javascript)

The identifier before the colon (:) is the property of the object and the identifier after the colon is the variable.

Notice that the property name is always on the left whether it’s an object literal or object destructuring syntax.

If the variables have the same names as the properties of the object, you can make the code more concise as follows:

let { firstName, lastName } = person;

console.log(firstName); *// 'John'*

console.log(lastName); *// 'Doe'*

Code language: JavaScript (javascript)

In this example, we declared two variables firstName and lastName, and assigned the properties of the person object to the variables in the same statement.

It’s possible to separate the declaration and assignment. However, you must surround the variables in parentheses:

({firstName, lastName} = person);

If you don’t use the parentheses, the JavaScript engine will interpret the left-hand side as a block and throw a syntax error.

When you assign a property that does not exist to a variable using the object destructuring, the variable is set to undefined. For example:

let { firstName, lastName, middleName } = person;

console.log(middleName); *// undefined*

Code language: JavaScript (javascript)

In this example, the middleName property doesn’t exist in the person object, therefore, the middleName variable is undefined.

## **Setting default values**

You can assign a default value to the variable when the property of an object doesn’t exist. For example:

let person = {

firstName: 'John',

lastName: 'Doe',

currentAge: 28

};

let { firstName, lastName, middleName = '', currentAge: age = 18 } = person;

console.log(middleName); *// ''*

console.log(age); *// 28*

Code language: JavaScript (javascript)

In this example, we assign an empty string to the middleName variable when the person object doesn’t have the middleName property.

Also, we assign the currentAge property to the age variable with the default value of 18.

However, when the person object does have the middleName property, the assignment works as usual:

let person = {

firstName: 'John',

lastName: 'Doe',

middleName: 'C.',

currentAge: 28

};

let { firstName, lastName, middleName = '', currentAge: age = 18 } = person;

console.log(middleName); *// 'C.'*

console.log(age); *// 28*

Code language: JavaScript (javascript)

## **Destructuring a null object**

A function may return an object or null in some situations. For example:

function getPerson() {

return null;

}

Code language: JavaScript (javascript)

And you use the object destructuring assignment:

let { firstName, lastName } = getPerson();

console.log(firstName, lastName);

Code language: JavaScript (javascript)

The code will throw a TypeError:

TypeError: Cannot destructure property 'firstName' of 'getPerson(...)' as it is null.

Code language: JavaScript (javascript)

To avoid this, you can use the OR operator (||) to fallback the null object to an empty object:

let { firstName, lastName } = getPerson() || {};

Code language: JavaScript (javascript)

Now, no error will occur. And the firstName and lastName will be undefined.

## **Nested object destructuring**

Assuming that you have an employee object which has a name object as the property:

let employee = {

id: 1001,

name: {

firstName: 'John',

lastName: 'Doe'

}

};

Code language: JavaScript (javascript)

The following statement destructures the properties of the nested name object into individual variables:

let {

name: {

firstName,

lastName

}

} = employee;

console.log(firstName); *// John*

console.log(lastName); *// Doe*

Code language: JavaScript (javascript)

It’s possible to do multiple assignement of a property to multiple variables:

let employee = {

id: 1001,

name: {

firstName: 'John',

lastName: 'Doe'

}

};

let {

name: {

firstName,

lastName

},

name

} = employee;

console.log(firstName); *// John*

console.log(lastName); *// Doe*

console.log(name); *// { firstName: 'John', lastName: 'Doe' }*

Code language: JavaScript (javascript)

## **Destructuring function arguments**

Suppose you have a function that displays the person object:

let display = (person) => console.log(`${person.firstName} ${person.lastName}`);

let person = {

firstName: 'John',

lastName: 'Doe'

};

display(person);

Code language: JavaScript (javascript)

It’s possible to destructure the object argument passed into the function like this:

let display = ({firstName, lastName}) => console.log(`${firstName} ${lastName}`);

let person = {

firstName: 'John',

lastName: 'Doe'

};

display(person);

Code language: JavaScript (javascript)

It looks less verbose especially when you use many properties of the argument object. This technique is often used in React.

## **Summary**

* Object destructuring assigns the properties of an object to variables with the same names by default.

# A Comprehensive Look at ES6 Modules

**Summary**: in this tutorial, you will learn about **ES6 modules** and how to export variables, functions, classes from a module, and reuse them in other modules.

An ES6 module is a JavaScript file that executes in strict mode only. It means that any [variables](https://www.javascripttutorial.net/es6/javascript-let/) or [functions](https://www.javascripttutorial.net/javascript-function/) declared in the module won’t be added automatically to the global scope.

## **Executing modules on web browsers**

First, create a new file called message.js and add the following code:

export let message = 'ES6 Modules';

Code language: JavaScript (javascript)

The message.js is a module in ES6 that contains the message variable. The export statement exposes the message variable to other modules.

Second, create another new file named app.js that uses the message.js module. The app.js module creates a new heading 1 (h1) element and attaches it to an HTML page. The import statement imports the message variable from the message.js module.

import { message } from './message.js'

const h1 = document.createElement('h1');

h1.textContent = message

document.body.appendChild(h1)

Code language: JavaScript (javascript)

Third, create a new HTML page that uses the app.js module:

<!DOCTYPE html>

**<html>**

**<head>**

**<meta charset="utf-8">**

**<title>**ES6 Modules**</title>**

**</head>**

**<body>**

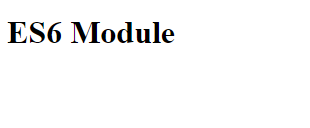
**<script type="module" src="./app.js"></script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

Note that we used the type="module" in the script tag to load the app.js module. If you view the page on a web browser, you will see the following page:



Let’s examine the export and import statements in more detail.

## **Exporting**

To export a [variable](https://www.javascripttutorial.net/javascript-variables/), a [function](https://www.javascripttutorial.net/javascript-function/), or a [class](https://www.javascripttutorial.net/es6/javascript-class/), you place the export keyword in front of it as follows:

*// log.js*

export let message = 'Hi';

export function getMessage() {

return message;

}

export function setMessage(msg) {

message = msg;

}

export class Logger {

}

Code language: JavaScript (javascript)

In this example, we have the log.js module with a variable, two functions, and one class. We used the export keyword to exports all identifiers in the module.

Note that the export keyword requires the function or class to have a name to be exported. You can’t export an anonymous function or class using this syntax.

JavaScript allows you to define a variable, a function, or a class first then export it later as follows:

*// foo.js*

function foo() {

console.log('foo');

}

function bar() {

console.log('bar');

}

export foo;

Code language: JavaScript (javascript)

In this example, we defined the foo() function first and then exported it. Since we didn’t export the bar() function, we couldn’t access it in other modules. The bar() function is inaccessible outside the module or we say it is private.

## **Importing**

Once you define a module with exports, you can access the exported variables, functions, and classes in another module by using the import keyword. The following illustrates the syntax:

import { what, ever } from './other\_module.js';

Code language: JavaScript (javascript)

In this syntax:

* First, specify what to import inside the curly braces, which are called bindings.
* Then, specify the module from which you import the given bindings.

Note that when you import a binding from a module, the binding behaves like it was defined using [const](https://www.javascripttutorial.net/es6/javascript-const/). It means you can’t have another identifier with the same name or change the value of the binding.

See the following example:

*// greeting.js*

export let message = 'Hi';

export function setMessage(msg) {

message = msg;

}

Code language: JavaScript (javascript)

When you import the message variable and setMessage() function, you can use the setMessage() function to change the value of the message variable as shown below:

*// app.js*

import {message, setMessage } from './greeting.js';

console.log(message); *// 'Hi'*

setMessage('Hello');

console.log(message); *// 'Hello'*

Code language: JavaScript (javascript)

However, you can’t change the value of the message variable directly. The following expression causes an error:

message = 'Hallo'; *// error*

Code language: JavaScript (javascript)

Behind the scenes, when you called the setMessage() function. JavaScript went back to the greeting.js module and executed code in there and changed the message variable. The change was then automatically reflected on the imported message binding.

The message binding in the app.js is the local name for exported message identifier. So basically the message variables in the app.js and greeting.js modules aren’t the same.

### **Import a single binding**

Suppose you have a module with the foo variable as follows:

*// foo.js*

export foo = 10;

Code language: JavaScript (javascript)

Then, in another module, you can reuse the foo variable:

*// app.js*

import { foo } from './foo.js';

console.log(foo); *// 10;*

Code language: JavaScript (javascript)

However, you can’t change the value of foo. If you attempt to do so, you will get an error:

foo = 20; *// throws an error*

Code language: JavaScript (javascript)

### **Import multiple bindings**

Suppose you have the cal.js module as follows:

*// cal.js*

export let a = 10,

b = 20,

result = 0;

export function sum() {

result = a + b;

return result;

}

export function multiply() {

result = a \* b;

return result;

}

Code language: JavaScript (javascript)

And you want to import these bindings from the cal.js, you can explicitly list them as follows:

import {a, b, result, sum, multiply } from './cal.js';

sum();

console.log(result); *// 30*

multiply();

console.log(result); *// 200*

Code language: JavaScript (javascript)

### **Import an entire module as an object**

To import everything from a module as a single object, you use the asterisk (\*) pattern as follows:

import \* as cal from './cal.js';

Code language: JavaScript (javascript)

In this example, we imported all bindings from the cal.js module as the cal object. In this case, all the bindings become properties of the cal object, so you can access them as shown below:

cal.a;

cal.b;

cal.sum();

Code language: CSS (css)

This import is called namespace import.

It’s important to keep in mind that the imported module executes only once even import it multiple times. Consider this example:

import { a } from './cal.js';

import { b } from './cal.js';

import {result} from './cal.js';

Code language: JavaScript (javascript)

After the first import statement, the cal.js module is executed and loaded into the memory, and it is reused whenever it is referenced by the subsequent import statement.

## **Limitation of import and export statements**

Note that you must use the import or export statement outside other statements and functions. The following example causes a SyntaxError:

if( requiredSum ) {

export sum;

}

Code language: JavaScript (javascript)

Because we used the export statement inside the [if](https://www.javascripttutorial.net/javascript-if-else/) statement. Similarly, the following import statement also causes a SyntaxError:

function importSum() {

import {sum} from './cal.js';

}

Code language: JavaScript (javascript)

Because we used the import statement inside a function.

The reason for the error is that JavaScript must statically determine what will be exported and imported.

Note that ES2020 introduced the function-like object [import()](https://www.javascripttutorial.net/es-next/javascript-import/) that allows you to dynamically import a module.

## **Aliasing**

JavaScript allows you to create aliases for variables, functions, or classes when you export and import. See the following math.js module:

*// math.js*

function add( a, b ) {

return a + b;

}

export { add as sum };

Code language: JavaScript (javascript)

In this example, instead of exporting the add() function, we used the as keyword to assign the sum() function an alias.

So when you import the add() function from the math.js module, you must use sum instead:

import { sum } from './math.js';

Code language: JavaScript (javascript)

If you want to use a different name when you import, you can use the as keyword as follows:

import {sum as total} from './math.js';

Code language: JavaScript (javascript)

## **Re-exporting a binding**

It’s possible to export bindings that you have imported. This is called re-exporting. For example:

import { sum } from './math.js';

export { sum };

Code language: JavaScript (javascript)

In this example, we imported sum from the math.js module and re-export it. The following statement is equivalent to the statements above:

export {sum} from './math.js';

Code language: JavaScript (javascript)

In case you want to rename the bindings before re-exporting, you use the as keyword. The following example imports sum from the math.js module and re-export it as add.

export { sum as add } from './math.js';

Code language: JavaScript (javascript)

If you want to export all the bindings from another module, you can use the asterisk (\*):

export \* from './cal.js';

Code language: JavaScript (javascript)

## **Importing without bindings**

Sometimes, you want to develop a module that doesn’t export anything, for example, you may want to add a new method to a built-in object such as the [Array](https://www.javascripttutorial.net/javascript-array/).

*// array.js*

if (!Array.prototype.contain) {

Array.prototype.contain = function(e) {

*// contain implementation*

*// ...*

}

}

Code language: JavaScript (javascript)

Now, you can import the module without any binding and use the contain() method defined in the array.js module as follows:

import './array.js';

[1,2,3].contain(2); *// true*

Code language: JavaScript (javascript)

## **Default exports**

A module can have one and only one default export. The default export is easier to import. The default for a module can be a variable, a function, or a class.

The following is the sort.js module with a default export.

*// sort.js*

export default function(arr) {

*// sorting here*

}

Code language: JavaScript (javascript)

Note that you don’t need to specify the name for the function because the module represents the function name.

import sort from sort.js;

sort([2,1,3]);

Code language: JavaScript (javascript)

As you see, the  sort identifier represents the default function of the sort.js module. Notice that we didn’t use the curly brace {} surrounding the  sort identifier.

Let’s change the sort.js module to include the default export as well as the non-default one:

*// sort.js*

export default function(arr) {

*// sorting here*

}

export function heapSort(arr) {

*// heapsort*

}

Code language: JavaScript (javascript)

To import both default and non-default bindings, you use the specify a list of bindings after the import keyword with the following rules:

* The default binding must come first.
* The non-default binding must be surrounded by curly braces.

See the following example:

import sort, {heapSort} from './sort.js';

sort([2,1,3]);

heapSort([3,1,2]);

Code language: JavaScript (javascript)

To rename the default export, you also use the as keyword as follows:

import { default as quicksort, heapSort} from './sort.js';

Code language: JavaScript (javascript)

In this tutorial, you have learned about ES6 modules and how to export bindings from a module and import them into another module.

# JavaScript Class

**Summary**: in this tutorial, you’ll learn about the JavaScript class and how to use it effectively.

A JavaScript class is a blueprint for creating [objects](https://www.javascripttutorial.net/javascript-objects/). A class encapsulates data and functions that manipulate data.

Unlike other programming languages such as Java and [C#](https://www.csharptutorial.net/), JavaScript classes are syntactic sugar over the [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/). In other words, ES6 classes are just special [functions](https://www.javascripttutorial.net/javascript-function/).

## **Classes prior to ES6 revisited**

Prior to ES6, JavaScript had no concepts of classes. To mimic a class, you often use the [constructor/prototype pattern](https://www.javascripttutorial.net/javascript-constructor-prototype/) as shown in the following example:

function Person(name) {

this.name = name;

}

Person.prototype.getName = function () {

return this.name;

};

var john = new Person("John Doe");

console.log(john.getName());

Code language: JavaScript (javascript)

Output:

John Doe

How it works.

First, create the Person as a constructor function that has a property name called name. The getName() function is assigned to the prototype so that it can be shared by all instances of the Person type.

Then, create a new instance of the Person type using the new operator. The john object, hence, is an instance of the Person and Object through [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/).

The following statements use the instanceof operator to check if john is an instance of the Person and Object type:

console.log(john instanceof Person); *// true*

console.log(john instanceof Object); *// true*

Code language: JavaScript (javascript)

## **ES6 class declaration**

ES6 introduced a new syntax for declaring a class as shown in this example:

class Person {

constructor(name) {

this.name = name;

}

getName() {

return this.name;

}

}

Code language: JavaScript (javascript)

This Person class behaves like the Person type in the previous example. However, instead of using a constructor/prototype pattern, it uses the class keyword.

In the Person class, the constructor() is where you can initialize the properties of an instance. JavaScript automatically calls the constructor() method when you instantiate an object of the class.

The following creates a new Person object, which will automatically call the constructor() of the Person class:

let john = new Person("John Doe");

Code language: JavaScript (javascript)

The getName() is called a method of the Person class. Like a constructor function, you can call the methods of a class using the following syntax:

objectName.methodName(args)

Code language: CSS (css)

For example:

let name = john.getName();

console.log(name); *// "John Doe"*

Code language: JavaScript (javascript)

To verify the fact that classes are special functions, you can use the typeof operator of to check the type of the Person class.

console.log(typeof Person); *// function*

Code language: JavaScript (javascript)

It returns function as expected.

The john object is also an instance of the Person and Object types:

console.log(john instanceof Person); *// true*

console.log(john instanceof Object); *// true*

Code language: JavaScript (javascript)

## **Class vs. Custom type**

Despite the similarities between a class and a custom type defined via a constructor function, there are some important differences.

First, class declarations are not [hoisted](https://www.javascripttutorial.net/javascript-hoisting/) like function declarations.

For example, if you place the following code above the Person class declaration section, you will get a ReferenceError.

let john = new Person("John Doe");

Code language: JavaScript (javascript)

Error:

Uncaught ReferenceError: Person is not defined

Code language: JavaScript (javascript)

Second, all the code inside a class automatically executes in the strict mode. And you cannot change this behavior.

Third, class methods are [non-enumerable](https://www.javascripttutorial.net/javascript-enumerable-properties/). If you use a constructor/prototype pattern, you have to use the Object.defineProperty() method to make a property non-enumerable.

Finally, calling the class constructor without the new operator will result in an error as shown in the following example.

let john = Person("John Doe");

Code language: JavaScript (javascript)

Error:

Uncaught TypeError: Class constructor Person cannot be invoked without 'new'

Code language: JavaScript (javascript)

Note that it’s possible to call the constructor function without the new operator. In this case, the constructor function behaves like a regular function.

## **Summary**

* Use JavaScript class keyword declares a new class.
* A class declaration is syntactic sugar over [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/) with additional enhancements.

# JavaScript Getters and Setters

**Summary**: in this tutorial, you will learn about JavaScript getters and setters and how to use them effectively.

## **Introduction to the JavaScript getters and setters**

The following example [defines a class](https://www.javascripttutorial.net/es6/javascript-class/) called Person:

class Person {

constructor(name) {

this.name = name;

}

}

let person = new Person("John");

console.log(person.name); *// John*

Code language: JavaScript (javascript)

The Person class has a property name and a constructor. The constructor initializes the name property to a string.

Sometimes, you don’t want the name property to be accessed directly like this:

person.name

Code language: CSS (css)

To do that, you may come up with a pair of methods that manipulate the name property. For example:

class Person {

constructor(name) {

this.setName(name);

}

getName() {

return this.name;

}

setName(newName) {

newName = newName.trim();

if (newName === '') {

throw 'The name cannot be empty';

}

this.name = newName;

}

}

let person = new Person('Jane Doe');

console.log(person); *// Jane Doe*

person.setName('Jane Smith');

console.log(person.getName()); *// Jane Smith*

Code language: JavaScript (javascript)

In this example, the Person class has the name property. Also, it has two additional methods getName() and setName().

The getName() method returns the value of the name property.

The setName() method assigns an argument to the name property. The setName() removes the whitespaces from both ends of the newName argument and throws an exception if the newName is empty.

The constructor() calls the setName() method to initialize the name property:

constructor(name) {

this.setName(name);

}

Code language: JavaScript (javascript)

The getName() and setName() methods are known as getter and setter in other programming languages such as Java and C++.

ES6 provides specific syntax for defining the getter and setter using the get and set keywords. For example:

class Person {

constructor(name) {

this.name = name;

}

get name() {

return this.\_name;

}

set name(newName) {

newName = newName.trim();

if (newName === '') {

throw 'The name cannot be empty';

}

this.\_name = newName;

}

}

Code language: JavaScript (javascript)

How it works.

First, the name property is changed to \_name to avoid the name collision with the getter and setter.

Second, the getter uses the get keyword followed by the method name:

get name() {

return this.\_name;

}

Code language: JavaScript (javascript)

To call the getter, you use the following syntax:

let name = person.name;

Code language: JavaScript (javascript)

When JavaScript sees the access to name property of the Person class, it checks if the Person class has any name property.

If not, JavaScript checks if the Person class has any method that binds to the name property. In this example, the name() method binds to the name property via the get keyword. Once JavaScript finds the getter method, it executes the getter method and returns a value.

Third, the setter uses the set keyword followed by the method name:

set name(newName) {

newName = newName.trim();

if (newName === '') {

throw 'The name cannot be empty';

}

this.\_name = newName;

}

Code language: JavaScript (javascript)

JavaScript will call the name() setter when you assign a value to the name property like this:

person.name = 'Jane Smith';

Code language: JavaScript (javascript)

If a class has only a getter but not a setter and you attempt to use the setter, the change won’t take any effect. See the following example:

class Person {

constructor(name) {

this.\_name = name;

}

get name() {

return this.\_name;

}

}

let person = new Person("Jane Doe");

console.log(person.name);

*// attempt to change the name, but cannot*

person.name = 'Jane Smith';

console.log(person.name); *// Jane Doe*

Code language: JavaScript (javascript)

In this example, the Person class has the name getter but not the name setter. It attempts to call the setter. However, the change doesn’t take effect since the Person class doesn’t have the name setter.

## **Using getter in an object literal**

The following example defines a getter called latest to return the latest attendee of the meeting object:

let meeting = {

attendees: [],

add(attendee) {

console.log(`${attendee} joined the meeting.`);

this.attendees.push(attendee);

return this;

},

get latest() {

let count = this.attendees.length;

return count == 0 ? undefined : this.attendees[count - 1];

}

};

meeting.add('John').add('Jane').add('Peter');

console.log(`The latest attendee is ${meeting.latest}.`);

Code language: JavaScript (javascript)

Output:

John joined a meeting.

Jane joined a meeting.

Peter joined a meeting.

The latest attendee is Peter.

## **Summary**

* Use the get and set keywords to define the JavaScript getters and setters for a class or an object.
* The get keyword binds an object property to a method that will be invoked when that property is looked up.
* The set keyword binds an object property to a method that will be invoked when that property is assigned.

# JavaScript Class Expressions

**Summary**: in this tutorial, you’ll learn how to use JavaScript class expressions to declare new classes.

## **Introduction to JavaScript class expressions**

Similar to [functions](https://www.javascripttutorial.net/javascript-function/), [classes](https://www.javascripttutorial.net/es6/javascript-class/) have expression forms. A class expression provides you with an alternative way to define a new class.

A class expression doesn’t require an identifier after the class keyword. And you can use a class expression in a [variable declaration](https://www.javascripttutorial.net/javascript-variables/) and pass it into a function as an argument.

For example, the following defines a class expression:

let Person = class {

constructor(name) {

this.name = name;

}

getName() {

return this.name;

}

}

Code language: Python (python)

How it works.

On the left side of the expression is the Person variable. It’s assigned to a class expression.

The class expression starts with the keyword class followed by the class definition.

A class expression may have a name or not. In this example, we have an unnamed class expression.

If a class expression has a name, its name can be local to the class body.

The following creates an instance of the Person class expression. Its syntax is the same as if it were a class declaration.

let person = new Person('John Doe');

Code language: Python (python)

Like a [class declaration](https://www.javascripttutorial.net/es6/javascript-class/), the type of a class expression is also a [function](https://www.javascripttutorial.net/javascript-function/):

console.log(typeof Person); // function

Code language: Python (python)

Similar to function expressions, class expressions are not [hoisted](https://www.javascripttutorial.net/javascript-hoisting/). It means that you cannot create an instance of the class before defining the class expression.

## **First-class citizen**

[JavaScript classes are first-class citizens](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/). It means that you can pass a class into a function, return it from a function, and assign it to a variable.

See the following example:

function factory(aClass) {

return new aClass();

}

let greeting = factory(class {

sayHi() { console.log('Hi'); }

});

greeting.sayHi(); // 'Hi'

Code language: Python (python)

How it works.

First, define a factory() function that takes a class expression as an argument and return the instance of the class:

function factory(aClass) {

return new aClass();

}

Code language: Python (python)

Second, pass an unnamed class expression to the factory() function and assign its result to the greeting variable:

let greeting = factory(class {

sayHi() { console.log('Hi'); }

});

Code language: Python (python)

The class expression has a method called sayHi(). And the greeting variable is an instance of the class expression.

Third, call the sayHi() method on the greeting object:

greeting.sayHi(); // 'Hi'

Code language: Python (python)

## **Singleton**

Singleton is a design pattern that limits the instantiation of a class to a single instance. It ensures that only one instance of a class can be created throughout the system.

Class expressions can be used to create a singleton by calling the class constructor immediately.

To do that, you use the new operator with a class expression and include the parentheses at the end of class declaration as shown in the following example:

let app = new class {

constructor(name) {

this.name = name;

}

start() {

console.log(`Starting the ${this.name}...`);

}

}('Awesome App');

app.start(); // Starting the Awesome App...

Code language: Python (python)

How it works.

The following is an unnamed class expresion:

new class {

constructor(name) {

this.name = name;

}

start() {

console.log(`Starting the ${this.name}...`);

}

}

Code language: Python (python)

The class has a constructor() that accepts an argument. It aslo has a method called start().

The class expression evaluates to a class. Therefore, you can call its constructor immediately by placing parentheses after the expression:

new class {

constructor(name) {

this.name = name;

}

start() {

console.log(`Starting the ${this.name}...`);

}

}('Awesome App')

Code language: Python (python)

This expression returns an instance of the class expression which is assigned to the app variable.

The following calls the start() method on the app object:

app.start(); // Starting the Awesome App...

Code language: Python (python)

## **Summary**

* ES6 provides you with an alternative way to defining a new class using a class expression.
* Class expressions can be named or unnamed.
* The class expression can be used to create a singleton object.

# JavaScript Static Methods

**Summary**: in this tutorial, you’ll learn about the JavaScript static methods and how to use them effectively.

## **Introduction to the JavaScript static methods**

By definition, static methods are bound to a [class](https://www.javascripttutorial.net/es6/javascript-class/), not the instances of that class. Therefore, static methods are useful for defining helper or utility methods.

To define a static method before ES6, you add it directly to the constructor of the class. For example, suppose you have a Person type as follows:

function Person(name) {

this.name = name;

}

Person.prototype.getName = function () {

return this.name;

};

Code language: JavaScript (javascript)

The following adds a static method called createAnonymous() to the Person type:

Person.createAnonymous = function (gender) {

let name = gender == "male" ? "John Doe" : "Jane Doe";

return new Person(name);

};

Code language: JavaScript (javascript)

The createAnonymous() method is considered a static method because it doesn’t depend on any instance of the Person type for its property values.

To call the createAnonymous() method, you use the Person type instead of its instances:

var anonymous = Person.createAnonymous();

Code language: JavaScript (javascript)

## **JavaScript static methods in ES6**

In ES6, you define static methods using the static keyword. The following example defines a static method called createAnonymous() for the Person class:

class Person {

constructor(name) {

this.name = name;

}

getName() {

return this.name;

}

static createAnonymous(gender) {

let name = gender == "male" ? "John Doe" : "Jane Doe";

return new Person(name);

}

}

Code language: JavaScript (javascript)

To invoke the static method, you use the following syntax:

let anonymous = Person.createAnonymous("male");

Code language: JavaScript (javascript)

If you attempt to call the static method from an instance of the class, you’ll get an error. For example:

let person = new Person('James Doe');

let anonymous = person.createAnonymous("male");

Code language: JavaScript (javascript)

Error:

TypeError: person.createAnonymous is not a function

Code language: JavaScript (javascript)

## **Calling a static method from the class constructor or an instance method**

To call a static method from a class constructor or an instance method, you use the class name, followed by the . and the static method:

className.staticMethodName();

Code language: CSS (css)

Alternatively, you can use the following syntax:

this.constructor.staticMethodName();

Code language: CSS (css)

## **Summary**

* JavaScript static methods are shared among instances of a class. Therefore, they are bound to the class.
* Call the static methods via the class name, not the instances of that class.
* Use the className.staticMethodName() or this.constructor.staticMethodName() to call a static method in a class constructor or an instance method.

# JavaScript Static Properties

**Summary**: in this tutorial, you’ll learn about the JavaScript static properties of a class and how to access the static properties in a static method, class constructor, and other instance methods.

## **Introduction to the JavaScript static properties**

Like a [static method](https://www.javascripttutorial.net/es6/javascript-static-method/), a static property is shared by all instances of a [class](https://www.javascripttutorial.net/es6/javascript-class/). To define static property, you use the static keyword followed by the property name like this:

class Item {

static count = 0;

}

Code language: JavaScript (javascript)

To access a static property, you use the class name followed by the . operator and the static property name. For example:

console.log(Item.count); *// 0*

Code language: JavaScript (javascript)

To access the static property in a static method, you use the class name followed by the . operator and the static property name. For example:

class Item {

static count = 0;

static getCount() {

return Item.count;

}

}

console.log(Item.getCount()); *// 0*

Code language: JavaScript (javascript)

To access a static property in a class constructor or instance methods, you use the following syntax:

className.staticPropertyName;

Code language: CSS (css)

Or

this.constructor.staticPropertyName;

Code language: CSS (css)

The following example increases the count static property in the class constructor:

class Item {

constructor(name, quantity) {

this.name = name;

this.quantity = quantity;

this.constructor.count++;

}

static count = 0;

static getCount() {

return Item.count++;

}

}

Code language: JavaScript (javascript)

When you create a new instance of the Item class, the following statement increases the count static property by one:

this.constructor.count++;

Code language: CSS (css)

For example:

*// Item class ...*

let pen = new Item("Pen", 5);

let notebook = new Item("notebook", 10);

console.log(Item.getCount()); *// 2*

Code language: JavaScript (javascript)

This example creates two instances of the Item class, which calls the class constructor. Since the class constructor increases the count property by one each time it’s called, the value of the count is two.

Put it all together.

class Item {

constructor(name, quantity) {

this.name = name;

this.quantity = quantity;

this.constructor.count++;

}

static count = 0;

static getCount() {

return Item.count++;

}

}

let pen = new Item("Pen", 5);

let notebook = new Item("notebook", 10);

console.log(Item.getCount()); *// 2*

Code language: JavaScript (javascript)

## **Summary**

* A static property of a class is shared by all instances of that class.
* Use the static keyword to define a static property.
* Use the className.staticPropertyName to access the static property in a static method.
* Use the this.constructor.staticPropertyName or className.staticPropertyName to access the static property in a constructor.

# JavaScript Computed Property

**Summary**: in this tutorial, you’ll learn about the JavaScript computed properties introduced in ES6.

## **Introduction to JavaScript Computed Property**

ES6 allows you to use an expression in brackets []. It’ll then use the result of the expression as the property name of an object. For example:

let propName = 'c';

const rank = {

a: 1,

b: 2,

[propName]: 3,

};

console.log(rank.c); *// 3*

Code language: JavaScript (javascript)

In this example, the [propName] is a computed property of the rank object. The property name is derived from the value of the propName variable.

When you access c property of the rank object, JavaScript evaluates propName and returns the property’s value.

Like an [object literal](https://www.javascripttutorial.net/javascript-objects/), you can use computed properties for getters and setters of a [class](https://www.javascripttutorial.net/es6/javascript-class/). For example:

let name = 'fullName';

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

get [name]() {

return `${this.firstName} ${this.lastName}`;

}

}

let person = new Person('John', 'Doe');

console.log(person.fullName);

Code language: JavaScript (javascript)

Output:

John Doe

How it works:

The get[name] is a computed property name of a getter of the Person class. At runtime, when you access the fullName property, the person object calls the getter and returns the full name.

# JavaScript Inheritance Using extends & super

**Summary**: in this tutorial, you will learn how to implement JavaScript inheritance by using extends and super in ES6.

## **Implementing JavaScript inheritance using extends and super**

Prior to ES6, implementing a proper inheritance required multiple steps. One of the most commonly used strategies is [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/).

The following illustrates how the Bird inherits properties from the Animal using the prototypal inheritance technique:

function Animal(legs) {

this.legs = legs;

}

Animal.prototype.walk = function() {

console.log('walking on ' + this.legs + ' legs');

}

function Bird(legs) {

Animal.call(this, legs);

}

Bird.prototype = Object.create(Animal.prototype);

Bird.prototype.constructor = Animal;

Bird.prototype.fly = function() {

console.log('flying');

}

var pigeon = new Bird(2);

pigeon.walk(); *// walking on 2 legs*

pigeon.fly(); *// flying*

Code language: JavaScript (javascript)

ES6 simplified these steps by using the extends and super keywords.

The following example defines the Animal and Bird classes and establishes the inheritance through the extends and super keywords.

class Animal {

constructor(legs) {

this.legs = legs;

}

walk() {

console.log('walking on ' + this.legs + ' legs');

}

}

class Bird extends Animal {

constructor(legs) {

super(legs);

}

fly() {

console.log('flying');

}

}

let bird = new Bird(2);

bird.walk();

bird.fly();

Code language: JavaScript (javascript)

How it works.

First, use the extends keyword to make the Bird class inheriting from the Animal class:

class Bird extends Animal {

*// ...*

}

Code language: JavaScript (javascript)

The Animal class is called a **base class** or **parent class** while the Bird class is known as a **derived class** or **child class**. By doing this, the Bird class inherits all methods and properties of the Animal class.

Second, in the Bird‘s constructor, call super() to invoke the Animal‘s constructor with the legs argument.

JavaScript requires the child class to call super() if it has a constructor. As you can see in the Bird class, the super(legs) is equivalent to the following statement in ES5:

Animal.call(this, legs);

Code language: JavaScript (javascript)

If the Bird class doesn’t have a constructor, you can don’t need to do anything else:

class Bird extends Animal {

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

It is equivalent to the following class:

class Bird extends Animal {

constructor(...args) {

super(...args);

}

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

However, the child class has a constructor, it needs to call super(). For example, the following code results in an error:

class Bird extends Animal {

constructor(legs) {

}

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

Error:

ReferenceError: Must call super constructor in derived class before accessing 'this' or returning from derived constructor

Code language: JavaScript (javascript)

Because the super() initializes the this object, you need to call the super() before accessing the this object. Trying to access this before calling super() also results in an error.

For example, if you want to initialize the color property of the Bird class, you can do it as follows:

class Bird extends Animal {

constructor(legs, color) {

super(legs);

this.color = color;

}

fly() {

console.log("flying");

}

getColor() {

return this.color;

}

}

let pegion = new Bird(2, "white");

console.log(pegion.getColor());

Code language: JavaScript (javascript)

## **Shadowing methods**

ES6 allows the child class and parent class to have methods with the same name. In this case, when you call the method of an object of the child class, the method in the child class will shadow the method in the parent class.

The following Dog class extends the Animal class and redefines the walk() method:

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

console.log(`go walking`);

}

}

let bingo = new Dog();

bingo.walk(); *// go walking*

Code language: JavaScript (javascript)

To call the method of the parent class in the child class, you use super.method(arguments) like this:

class Dog extends Animal {

constructor() {

super(4);

}

walk() {

super.walk();

console.log(`go walking`);

}

}

let bingo = new Dog();

bingo.walk();

*// walking on 4 legs*

*// go walking*

Code language: JavaScript (javascript)

## **Inheriting static members**

Besides the properties and methods, the child class also inherits all static properties and methods of the parent class. For example:

class Animal {

constructor(legs) {

this.legs = legs;

}

walk() {

console.log('walking on ' + this.legs + ' legs');

}

static helloWorld() {

console.log('Hello World');

}

}

class Bird extends Animal {

fly() {

console.log('flying');

}

}

Code language: JavaScript (javascript)

In this example, the Animal class has the helloWorld() static method and this method is available as Bird.helloWorld() and behaves the same as the Animal.helloWorld() method:

Bird.helloWorld(); *// Hello World*

Code language: JavaScript (javascript)

## **Inheriting from built-in types**

JavaScript allows you to extend a built-in type such as [Array](https://www.javascripttutorial.net/javascript-array/), String, [Map](https://www.javascripttutorial.net/es6/javascript-map/), and [Set](https://www.javascripttutorial.net/es6/javascript-set/)through inheritance.

The following Queue class extends the Array reference type. The syntax is much cleaner than the [Queue](https://www.javascripttutorial.net/javascript-queue/) implemented using the [constructor/prototype pattern](https://www.javascripttutorial.net/create-objects-in-javascript/#constructor_prototype_pattern).

class Queue extends Array {

enqueue(e) {

super.push(e);

}

dequeue() {

return super.shift();

}

peek() {

return !this.empty() ? this[0] : undefined;

}

empty() {

return this.length === 0;

}

}

var customers = new Queue();

customers.enqueue('A');

customers.enqueue('B');

customers.enqueue('C');

while (!customers.empty()) {

console.log(customers.dequeue());

}

Code language: JavaScript (javascript)

## **Summary**

* Use the extends keyword to implement the inheritance in ES6. The class to be extended is called a base class or parent class. The class that extends the base class or parent class is called the derived class or child class.
* Call the super(arguments) in the child class’s constructor to invoke the parent class’s constructor.
* Use super keyword to call methods of the parent class in the methods of the child class.

# Introduction to JavaScript new.target Metaproperty

**Summary**: in this tutorial, you learn about the JavaScript new.target metaproperty that detects whether a function or constructor was called using the new operator.

## **Introduction to JavaScript new.target**



ES6 provides a metaproperty named new.target that allows you to detect whether a [function](https://www.javascripttutorial.net/javascript-function/)or constructor was called using the new operator.

The new.target consists of the new keyword, a dot, and target property. The new.target is available in all functions.

However, in [arrow functions](https://www.javascripttutorial.net/es6/javascript-arrow-function/), the new.target is the one that belongs to the surrounding function.

The new.target is very useful to inspect at runtime whether a function is being executed as a function or as a constructor. It is also handy to determine a specific derived class that was called by using the new operator from within a parent class.

## **JavaScript new.target in functions**

Let’s see the following Person constructor function:

function Person(name) {

this.name = name;

}

Code language: JavaScript (javascript)

You can create a new object from the Person function by using the new operator as follows:

let john = new Person('John');

console.log(john.name); *// john*

Code language: JavaScript (javascript)

Or you can call the Person as a function:

Person('Lily');

Code language: JavaScript (javascript)

Because the this is set to the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) i.e., the window object when you run JavaScript in the web browser, the name property is added to the window object as follows:

console.log(window.name); *//Lily*

Code language: JavaScript (javascript)

To help you detect whether a function was called using the new operator, you use the new.target metaproperty.

In a regular function call, the new.target returns undefined. If the function was called with the new operator, the new.target returns a reference to the function.

Suppose you don’t want the Person to be called as a function, you can use the new.target as follows:

function Person(name) {

if (!new.target) {

throw "must use new operator with Person";

}

this.name = name;

}

Code language: JavaScript (javascript)

Now, the only way to use Person is to instantiate an object from it by using the new operator. If you try to call it as a regular function, you will get an error.

## **JavaScript new.target in constructors**

In a [class](https://www.javascripttutorial.net/es6/javascript-class/)constructor, the new.target refers to the constructor that was invoked directly by the new operator. It is true if the constructor is in the parent class and was delegated from the constructor of the child class:

class Person {

constructor(name) {

this.name = name;

console.log(new.target.name);

}

}

class Employee extends Person {

constructor(name, title) {

super(name);

this.title = title;

}

}

let john = new Person('John Doe'); *// Person*

let lily = new Employee('Lily Bush', 'Programmer'); *// Employee*

Code language: JavaScript (javascript)

In this example, new.target.name is the human-friendly name of the constructor reference of new.target

In this tutorial, you have learned how to use the JavaScript new.target metaproperty to detect whether a function or constructor was called using the new operator.

# An Introduction to JavaScript Arrow Functions

**Summary**: in this tutorial, you will learn how to use the JavaScript arrow function to write more concise code for function expressions.

## **Introduction to JavaScript arrow functions**

ES6 arrow functions provide you with an alternative way to write a shorter syntax compared to the function expression.

The following example defines a function expression that returns the sum of two numbers:

let add = function (x, y) {

return x + y;

};

console.log(add(10, 20)); *// 30*

Code language: JavaScript (javascript)

The following example is equivalent to the above add() function expression but use an arrow function instead:

let add = (x, y) => x + y;

console.log(add(10, 20)); *// 30;*

Code language: JavaScript (javascript)

In this example, the arrow function has one expression x + y so it returns the result of the expression.

However, if you use the block syntax, you need to specify the return keyword:

let add = (x, y) => { return x + y; };

Code language: JavaScript (javascript)

The  typeof operator returns function indicating the type of arrow function.

console.log(typeof add); *// function*

Code language: JavaScript (javascript)

The arrow function is also an instance of the [Function type](https://www.javascripttutorial.net/javascript-function-type/) as shown in the following example:

console.log(add instanceof Function); *// true*

Code language: JavaScript (javascript)

### **JavaScript arrow functions with multiple parameters**

If an arrow function has two or more parameters, you use the following syntax:

(p1, p2, ..., pn) => expression;

Code language: PHP (php)

The following expression:

=> expression

Code language: PHP (php)

is equivalent to the following expression:

=> { return expression; }

Code language: PHP (php)

For example, to [sort an array](https://www.javascripttutorial.net/javascript-array-sort/) of numbers in the descending order, you use the sort() method of the array object as follows:

let numbers = [4,2,6];

numbers.sort(function(a,b){

return b - a;

});

console.log(numbers); *// [6,4,2]*

Code language: JavaScript (javascript)

The code is more concise with the arrow function syntax:

let numbers = [4,2,6];

numbers.sort((a,b) => b - a);

console.log(numbers); *// [6,4,2]*

Code language: JavaScript (javascript)

### **JavaScript arrow functions with a single parameter**

If an arrow function takes a single parameter, you use the following syntax:

(p1) => { statements }

Code language: PHP (php)

Note that you can omit the parentheses as follows:

p => { statements }

Code language: PHP (php)

The following example uses an arrow function as an argument of the [map()](https://www.javascripttutorial.net/javascript-array-map/) method that transforms an array of strings into an array of the string’s lengths.

let names = ['John', 'Mac', 'Peter'];

let lengths = names.map(name => name.length);

console.log(lengths);

Code language: JavaScript (javascript)

Output:

[ 4, 3, 5 ]

Code language: JSON / JSON with Comments (json)

### **JavaScript arrow functions with no parameter**

If the arrow function has no parameter, you need to use parentheses, like this:

() => { statements }

Code language: PHP (php)

For example:

let logDoc = () => console.log(window.document);

logDoc();

Code language: JavaScript (javascript)

## **Line break between parameter definition and arrow**

JavaScript doesn’t allow you to have a line break between the parameter definition and the arrow (=>) in an arrow function.

For example, the following code causes a SyntaxError:

let multiply = (x,y)

=> x \* y;

Code language: JavaScript (javascript)

However, the following code works perfectly fine:

let multiply = (x,y) =>

x \* y;

Code language: JavaScript (javascript)

JavaScript allows you to have line breaks between parameters as shown in the following example:

let multiply = (

x,

y

) =>

x \* y;

Code language: JavaScript (javascript)

## **Statements & expressions in the arrow function body**

In JavaScript, an expression evaluates to a value as shown in the following example.

10 + 20;

A statement does a specific task such as:

if (x === y) {

console.log('x equals y');

}

Code language: JavaScript (javascript)

If you use an expression in the body of an arrow function, you don’t need to use the curly braces.

let square = x => x \* x;

Code language: JavaScript (javascript)

However, if you use a statement, you must wrap it inside a pair of curly braces as in the following example:

let except = msg => {

throw msg;

};

Code language: JavaScript (javascript)

## **JavaScript arrow functions and object literal**

Consider the following example:

let setColor = function (color) {

return {value: color}

};

let backgroundColor = setColor('Red');

console.log(backgroundColor.value); *// "Red"*

Code language: JavaScript (javascript)

The setColor() function expression returns an object that has the value property set to the color argument.

If you use the following syntax to return an object literal from an arrow function, you will get an error.

p => {object:literal}

Code language: PHP (php)

For example, the following code causes an error.

let setColor = color => {value: color };

Code language: JavaScript (javascript)

Since both block and object literal use curly brackets, the JavasScript engine cannot distinguish between a block and an object.

To fix this, you need to wrap the object literal in parentheses as follows:

let setColor = color => ({value: color });

Code language: JavaScript (javascript)

## **Arrow function vs. regular function**

There are two main differences between an arrow function and a regular function.

1. First, in the arrow function, the [this](https://www.javascripttutorial.net/javascript-this/), arguments, super, [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) are lexical. It means that the arrow function uses these variables (or constructs) from the enclosing lexical scope.
2. Second, an arrow function cannot be used as a function constructor. If you use the new keyword to create a new object from an arrow function, you will get an error.

### **JavaScript arrow functions and this value**

In JavaScript, a new function defines its own this value. However, it is not the case for the arrow function. See the following example:

function Car() {

this.speed = 0;

this.speedUp = function (speed) {

this.speed = speed;

setTimeout(function () {

console.log(this.speed); *// undefined*

}, 1000);

};

}

let car = new Car();

car.speedUp(50);

Code language: JavaScript (javascript)

Inside the anonymous function of the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function, the this.speed is undefined. The reason is that the this of the [anonymous function](https://www.javascripttutorial.net/javascript-anonymous-functions/) shadows the this of the speedUp() method.

To fix this, you assign the this value to a variable that doesn’t shadow inside the anonymous function as follows:

function Car() {

this.speed = 0;

this.speedUp = function (speed) {

this.speed = speed;

let self = this;

setTimeout(function () {

console.log(self.speed);

}, 1000);

};

}

let car = new Car();

car.speedUp(50); *// 50;*

Code language: JavaScript (javascript)

Unlike an anonymous function, an arrow function captures the this value of the enclosing context instead of creating its own this context. The following code should work as expected:

function Car() {

this.speed = 0;

this.speedUp = function (speed) {

this.speed = speed;

setTimeout(

() => console.log(this.speed),

1000);

};

}

let car = new Car();

car.speedUp(50); *// 50;*

Code language: JavaScript (javascript)

### **JavaScript arrow functions and the arguments object**

An arrow function doesn’t have the arguments object. For example:

function show() {

return x => x + arguments[0];

}

let display = show(10, 20);

let result = display(5);

console.log(result); *// 15*

Code language: JavaScript (javascript)

The arrow function inside the showMe() function references the arguments object. However, this arguments object belongs to the show() function, not the arrow function.

Also, an arrow function doesn’t have the [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) keyword.

### **JavaScript arrow functions and the prototype property**

When you define a [function](https://www.javascripttutorial.net/javascript-function/) using a function keyword, the function has a property called [prototype](https://www.javascripttutorial.net/javascript-prototype/):

function dump( message ) {

console.log(message);

}

console.log(dump.hasOwnProperty('prototype')); *// true*

Code language: JavaScript (javascript)

However, arrow functions don’t have the prototype property:

let dump = message => console.log(message);

console.log(dump.hasOwnProperty('prototype')); *// false*

Code language: JavaScript (javascript)

It is a good practice to use arrow functions for [callbacks](https://www.javascripttutorial.net/javascript-callback/) and [closures](https://www.javascripttutorial.net/javascript-closure/) because the syntax of arrow functions is cleaner.

## **Summary**

* Use the (...args) => expression; to define an arrow function.
* Use the (...args) => { statements } to define an arrow function that has multiple statements.
* An arrow function doesn’t have its binding to this or super.
* An arrow function doesn’t have arguments object, new.target keyword, and prototype property.

# When You Should Not Use Arrow Functions

**Summary**: in this tutorial, you will learn when you **should not use** the arrow functions in ES6.

An [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) doesn’t have its own [this](https://www.javascripttutorial.net/javascript-this/) value and the arguments object. Therefore, you should not use it as an event handler, a method of an object literal, a prototype method, or when you have a function that uses the arguments object.

## **1) Event handlers**

Suppose that you have the following input text field:

**<input type="text" name="username" id="username" placeholder="Enter a username">**

Code language: HTML, XML (xml)

And you want to show a greeting message when users type their usernames. The following shows the <div> element that will display the greeting message:

**<div id="greeting"></div>**

Code language: HTML, XML (xml)

Once users type their usernames, you capture the current value of the input and update it to the <div> element:

const greeting = document.querySelector('#greeting');

const username = document.querySelector('#username');

username.addEventListener('keyup', () => {

greeting.textContent = 'Hello ' + this.value;

});

Code language: JavaScript (javascript)

However, when you execute the code, you will get the following message regardless of whatever you type:

Hello undefined

Code language: JavaScript (javascript)

It means that the this.value in the event handler always returns undefined.

As mentioned earlier, the arrow function doesn’t have its own this value. It uses the this value of the enclosing lexical scope. In the above example, the this in arrow function references the global object.

In the web browser, the global object is window. The window object doesn’t have the value property. Therefore, the JavaScript engine adds the value property to the window object and sets its values to undefined.

To fix this issue, you need to use a regular function instead. The this value will be bound to the <input> element that triggers the event.

username.addEventListener('keyup', function () {

input.textContent = 'Hello ' + this.value;

});

Code language: JavaScript (javascript)

## **2) Object methods**

See the following counter object:

const counter = {

count: 0,

next: () => ++this.count,

current: () => this.count

};

Code language: JavaScript (javascript)

The counter object has two methods: current() and next(). The current() method returns the current counter value and the next() method returns the next counter value.

The following shows the next counter value which should be 1:

console.log(counter.next());

Code language: CSS (css)

However, it returns NaN.

The reason is that when you use the arrow function inside the object, it inherits the this value from the enclosing lexical scope which is the global scope in this example.

The this.count inside the next() method is equivalent to the window.count (in the web browser).

The window.count is undefined by default because the window object doesn’t have the count property. The next() method adds one to undefined that results in NaN.

To fix this, you use regular functions as the method of an object literal as follows:

const counter = {

count: 0,

next() {

return ++this.count;

},

current() {

return this.count;

}

};

Code language: JavaScript (javascript)

Now, calling the next() method will return one as expected:

console.log(counter.next()); *// 1*

Code language: JavaScript (javascript)

## **3) Prototype methods**

See the following Counter object that uses the [prototype](https://www.javascripttutorial.net/javascript-prototype/) pattern:

function Counter() {

this.count = 0;

}

Counter.prototype.next = () => {

return this.count;

};

Counter.prototype.current = () => {

return ++this.next;

}

Code language: JavaScript (javascript)

The this value in these next() and current() methods reference the global object. Since you want the this value inside the methods to reference the Counter object, you need to use the regular functions instead:

function Counter() {

this.count = 0;

}

Counter.prototype.next = function () {

return this.count;

};

Counter.prototype.current = function () {

return ++this.next;

}

Code language: JavaScript (javascript)

## **4) Functions that use the arguments object**

Arrow functions don’t have the arguments object. Therefore, if you have a function that uses arguments object, you cannot use the arrow function.

For example, the following concat() function won’t work:

const concat = (separator) => {

let args = Array.prototype.slice.call(arguments, 1);

return args.join(separator);

}

Code language: JavaScript (javascript)

Instead, you use a regular function like this:

function concat(separator) {

let args = Array.prototype.slice.call(arguments, 1);

return args.join(separator);

}

Code language: JavaScript (javascript)

## **Summary**

* An arrow function doesn’t have its own this value. Instead, it uses the this value of the enclosing lexical scope. An arrow function also doesn’t have the arguments object.
* Avoid using the arrow function for event handlers, object methods, prototype methods, and functions that use the arguments object.

# The Ultimate Guide to JavaScript Symbol

**Summary**: in this tutorial, you will learn about the JavaScript symbol primitive type and how to use the symbol effectively.

## **Creating symbols**

ES6 added Symbol as a new primitive type. Unlike other primitive types such as [number](https://www.javascripttutorial.net/javascript-data-types/#number), [boolean](https://www.javascripttutorial.net/javascript-data-types/#boolean), [null](https://www.javascripttutorial.net/javascript-data-types/#null), [undefined](https://www.javascripttutorial.net/javascript-data-types/#undefined), and [string](https://www.javascripttutorial.net/javascript-data-types/#string), the symbol type doesn’t have a literal form.

To create a new symbol, you use the global Symbol() function as shown in this example:

let s = Symbol('foo');

Code language: JavaScript (javascript)

The Symbol() function creates a new unique value each time you call it:

console.log(Symbol() === Symbol()); *// false*

Code language: JavaScript (javascript)

The Symbol() function accepts a description as an optional argument. The description argument will make your symbol more descriptive.

The following example creates two symbols: firstName and lastName.

let firstName = Symbol('first name'),

lastName = Symbol('last name');

Code language: JavaScript (javascript)

You can access the symbol’s description property using the toString() method. The console.log() method calls the toString() method of the symbol implicitly as shown in the following example:

console.log(firstName); *// Symbol(first name)*

console.log(lastName); *// Symbol(last name)*

Code language: JavaScript (javascript)

Since symbols are primitive values, you can use the  typeof operator to check whether a variable is a symbol. ES6 extended  typeof to return the symbol string when you pass in a symbol variable:

console.log(typeof firstName); *// symbol*

Code language: JavaScript (javascript)

Since a symbol is a primitive value, if you attempt to create a symbol using the new operator, you will get an error:

let s = new Symbol(); *// error*

Code language: JavaScript (javascript)

## **Sharing symbols**

ES6 provides you with the global symbol registry that allows you to share symbols globally. If you want to create a symbol that will be shared, you use the Symbol.for() method instead of calling the Symbol() function.

The Symbol.for() method accepts a single parameter that can be used for symbol’s description, as shown in the following example:

let ssn = Symbol.for('ssn');

Code language: JavaScript (javascript)

The Symbol.for() method first searches for the symbol with the  ssn key in the global symbol registry. It returns the existing symbol if there is one. Otherwise, the Symbol.for() method creates a new symbol, registers it to the global symbol registry with the specified key, and returns the symbol.

Later, if you call the Symbol.for() method using the same key, the Symbol.for() method will return the existing symbol.

let citizenID = Symbol.for('ssn');

console.log(ssn === citizenID); *// true*

Code language: JavaScript (javascript)

In this example, we used the Symbol.for() method to look up the symbol with the  ssn key. Since the global symbol registry already contained it, the Symbol.for() method returned the existing symbol.

To get the key associated with a symbol, you use the Symbol.keyFor() method as shown in the following example:

console.log(Symbol.keyFor(citizenID)); *// 'ssn'*

Code language: JavaScript (javascript)

If a symbol that does not exist in the global symbol registry, the System.keyFor() method returns undefined.

let systemID = Symbol('sys');

console.log(Symbol.keyFor(systemID)); *// undefined*

Code language: JavaScript (javascript)

## **Symbol usages**

### **A) Using symbols as unique values**

Whenever you use a string or a number in your code, you should use symbols instead. For example, you have to manage the status in the task management application. Before ES6, you would use strings such as open, in progress, completed, canceled, and on hold to represent different statuses of a task. In ES6, you can use symbols as follows:

let statuses = {

OPEN: Symbol('Open'),

IN\_PROGRESS: Symbol('In progress'),

COMPLETED: Symbol('Completed'),

HOLD: Symbol('On hold'),

CANCELED: Symbol('Canceled')

};

*// complete a task*

task.setStatus(statuses.COMPLETED);

Code language: JavaScript (javascript)

### **B) Using symbol as the computed property name of an object**

You can use symbols as [computed property](https://www.javascripttutorial.net/es6/javascript-computed-property/) names. See the following example:

let status = Symbol('status');

let task = {

[status]: statuses.OPEN,

description: 'Learn ES6 Symbol'

};

console.log(task);

Code language: JavaScript (javascript)

To get all the enumerable properties of an object, you use the Object.keys() method.

console.log(Object.keys(task)); *// ["description"]*

Code language: JavaScript (javascript)

To get all properties of an object whether the properties are enumerable or not, you use the Object.getOwnPropertyNames() method.

console.log(Object.getOwnPropertyNames(task)); *// ["description"]*

Code language: JavaScript (javascript)

To get all property symbols of an object, you use the Object.getOwnPropertySymbols() method, which has been added in ES6.

console.log(Object.getOwnPropertySymbols(task)); *//[Symbol(status)]*

Code language: JavaScript (javascript)

The Object.getOwnPropertySymbols() method returns an array of own property symbols from an object.

## **Well-known symbols**

ES6 provides predefined symbols which are called well-known symbols. The well-known symbols represent the common behaviors in JavaScript. Each well-known symbol is a static property of the Symbol object.

### **Symbol.hasInstance**

The Symbol.hasInstance is a symbol that changes the behavior of the instanceof operator. Typically, when you use the instanceof operator:

obj instanceof type;

Code language: JavaScript (javascript)

JavaScript will call the Symbol.hasIntance method as follows:

type[Symbol.hasInstance](obj);

Code language: JavaScript (javascript)

It then depends on the method to determine if  obj is an instance of the type object. See the following example.

class Stack {

}

console.log([] instanceof Stack); *// false*

Code language: JavaScript (javascript)

The [] array is not an instance of the Stack class, therefore, the instanceof operator returns false in this example.

Assuming that you want the [] array is an instance of the Stack class, you can add the Symbol.hasInstance method as follows:

class Stack {

static [Symbol.hasInstance](obj) {

return Array.isArray(obj);

}

}

console.log([] instanceof Stack); *// true*

Code language: JavaScript (javascript)

### **Symbol.iterator**

The Symbol.iterator specifies whether a function will return an iterator for an object.

The objects that have Symbol.iterator property are called iterable objects.

In ES6, all collection objects (A[rray](https://www.javascripttutorial.net/javascript-array/), [Set](https://www.javascripttutorial.net/es6/javascript-set/) and [Map](https://www.javascripttutorial.net/es6/javascript-map/)) and strings are iterable objects.

ES6 provides the [for…of](https://www.javascripttutorial.net/es6/javascript-for-of/) loop that works with the iterable object as in the following example.

var numbers = [1, 2, 3];

for (let num of numbers) {

console.log(num);

}

*// 1*

*// 2*

*// 3*

Code language: JavaScript (javascript)

Internally, JavaScript engine first calls the Symbol.iterator method of the numbers array to get the iterator object. Then, it calls the iterator.next() method and copies the value property fo the iterator object into the num variable. After three iterations, the done property of the result object is true, the loop exits.

You can access the default iterator object via System.iterator symbol as follows:

var iterator = numbers[Symbol.iterator]();

console.log(iterator.next()); *// Object {value: 1, done: false}*

console.log(iterator.next()); *// Object {value: 2, done: false}*

console.log(iterator.next()); *// Object {value: 3, done: false}*

console.log(iterator.next()); *// Object {value: undefined, done: true}*

Code language: JavaScript (javascript)

By default, a collection is not iterable. However, you can make it iterable by using the Symbol.iterator as shown in the following example:

class List {

constructor() {

this.elements = [];

}

add(element) {

this.elements.push(element);

return this;

}

\*[Symbol.iterator]() {

for (let element of this.elements) {

yield element;

}

}

}

let chars = new List();

chars.add('A')

.add('B')

.add('C');

*// because of the Symbol.iterator*

for (let c of chars) {

console.log(c);

}

*// A*

*// B*

*// C*

Code language: JavaScript (javascript)

### **Symbol.isConcatSpreadable**

To concatenate two arrays, you use the concat() method as shown in the following example:

let odd = [1, 3],

even = [2, 4];

let all = odd.concat(even);

console.log(all); *// [1, 3, 2, 4]*

Code language: JavaScript (javascript)

In this example, the resulting array contains the single elements of both arrays. In addition, the concat() method also accepts a non-array argument as illustrated below.

let extras = all.concat(5);

console.log(extras); *// [1, 3, 2, 4, 5]*

Code language: JavaScript (javascript)

The number 5 becomes the fifth element of the array.

As you see in the above example, when we pass an array to the concat() method, the concat() method spreads the array into individual elements. However, it treats a single primitive argument differently. Prior to ES6, you could not change this behavior.

This is why the Symbol.isConcatSpreadable symbol comes into play.

The Symbol.isConcatSpreadable property is a Boolean value that determines whether an object is added individually to the result of the concat() function.

Consider the following example:

let list = {

0: 'JavaScript',

1: 'Symbol',

length: 2

};

let message = ['Learning'].concat(list);

console.log(message); *// ["Learning", Object]*

Code language: JavaScript (javascript)

The list object is concatenated to the ['Learning'] array. However, its individual elements are not spreaded.

To enable the elements of the list object added to the array individually when passing to the concat() method, you need to add the Symbol.isConcatSpreadable property to the list object as follows:

let list = {

0: 'JavaScript',

1: 'Symbol',

length: 2,

[Symbol.isConcatSpreadable]: true

};

let message = ['Learning'].concat(list);

console.log(message); *// ["Learning", "JavaScript", "Symbol"]*

Code language: JavaScript (javascript)

Note that if you set the value of the Symbol.isConcatSpreadable to false and pass the list object to the concat() method, it will be concatenated to the array as the whole object.

### **Symbol.toPrimitive**

The Symbol.toPrimitive method determines what should happen when an object is converted into a primitive value.

The JavaScript engine defines the Symbol.toPrimitive method on the prototype of each standard type.

The Symbol.toPrimitive method takes a hint argument which has one of three values: “number”, “string”, and “default”. The hint argument specifies the type of the return value. The hint parameter is filled by the JavaScript engine based on the context in which the object is used.

Here is an example of using the Symbol.toPrimitive method.

function Money(amount, currency) {

this.amount = amount;

this.currency = currency;

}

Money.prototype[Symbol.toPrimitive] = function(hint) {

var result;

switch (hint) {

case 'string':

result = this.amount + this.currency;

break;

case 'number':

result = this.amount;

break;

case 'default':

result = this.amount + this.currency;

break;

}

return result;

}

var price = new Money(799, 'USD');

console.log('Price is ' + price); *// Price is 799USD*

console.log(+price + 1); *// 800*

console.log(String(price)); *// 799USD*

Code language: JavaScript (javascript)

In this tutorial, you have learned about JavaScript symbols and how to use symbols for unique values and object properties. Also, you learned how to use well-known symbols to modify object behaviors.

# The Essential Guide to JavaScript Iterators

**Summary**: in this tutorial, you will learn about JavaScript iterator and how to use iterators to process a sequence of data more efficiently.

## **The for loop issues**

When you have an [array](https://www.javascripttutorial.net/javascript-array/) of data, you typically use a [for](https://www.javascripttutorial.net/javascript-for-loop/) loop to iterate over its elements. For example:

let ranks = ['A', 'B', 'C'];

for (let i = 0; i < ranks.length; i++) {

console.log(ranks[i]);

}

Code language: JavaScript (javascript)

The for loop uses the variable  i to track the index of the ranks array. The value of  i increments each time the loop executes as long as the value of i is less than the number of elements in the ranks array.

This code is straightforward. However, its complexity grows when you nest a loop inside another loop. In addition, keeping track of multiple variables inside the loops is error-prone.

ES6 introduced a new loop construct called [for...of](https://www.javascripttutorial.net/es6/javascript-for-of/) to eliminate the standard loop’s complexity and avoid the errors caused by keeping track of loop indexes.

To iterate over the elements of the ranks array, you use the following for...of construct:

for(let rank of ranks) {

console.log(rank);

}

Code language: JavaScript (javascript)

The for...of is far more elegant than the for loop because it shows the true intent of the code – iterate over an array to access each element in the sequence.

On top of this, the for...of loop has the ability to create a loop over any **iterable** object, not just an array.

To understand the iterable object, you need to understand the iteration protocols first.

## **Iteration protocols**

There are two iteration protocols: **iterable protocol** and **iterator protocol**.

### **Iterator protocol**

An object is an iterator when it implements an interface (or API) that answers two questions:

* Is there any element left?
* If there is, what is the element?

Technically speaking, an object is qualified as an iterator when it has a next() method that returns an object with two properties:

* done: a boolean value indicating whether or not  there are any more elements that could be iterated upon.
* value: the current element.

Each time you call the next(), it returns the next value in the collection:

{ value: 'next value', done: false }

Code language: CSS (css)

If you call the next() method after the last value has been returned, the next() returns the result object as follows:

{done: true: value: undefined}

Code language: CSS (css)

The value of the done property indicates that there is no more value to return and the value of the property is set to undefined.

### **Iterable protocol**

An object is iterable when it contains a method called [Symbol.iterator] that takes no argument and returns an object which conforms to the iterator protocol.

The [Symbol.iterator] is one of the built-in well-known [symbols](https://www.javascripttutorial.net/es6/symbol/) in ES6.

## **Iterators**

Since ES6 provides built-in iterators for the collection types  [Array](https://www.javascripttutorial.net/javascript-array/), [Set](https://www.javascripttutorial.net/es6/javascript-set/), and [Map](https://www.javascripttutorial.net/es6/javascript-map/), you don’t have to create iterators for these objects.

If you have a custom type and want to make it iterable so that you can use the for...of loop construct, you need to implement the iteration protocols.

The following code creates a Sequence object that returns a list of numbers in the range of ( start, end) with an interval between subsequent numbers.

class Sequence {

constructor( start = 0, end = Infinity, interval = 1 ) {

this.start = start;

this.end = end;

this.interval = interval;

}

[Symbol.iterator]() {

let counter = 0;

let nextIndex = this.start;

return {

next: () => {

if ( nextIndex <= this.end ) {

let result = { value: nextIndex, done: false }

nextIndex += this.interval;

counter++;

return result;

}

return { value: counter, done: true };

}

}

}

};

Code language: JavaScript (javascript)

The following code uses the Sequence iterator in a for...of loop:

let evenNumbers = new Sequence(2, 10, 2);

for (const num of evenNumbers) {

console.log(num);

}

Code language: JavaScript (javascript)

Output:

2

4

6

8

10

You can explicitly access the [Symbol.iterator]() method as shown in the following script:

let evenNumbers = new Sequence(2, 10, 2);

let iterator = evenNumbers[Symbol.iterator]();

let result = iterator.next();

while( !result.done ) {

console.log(result.value);

result = iterator.next();

}

Code language: JavaScript (javascript)

## **Cleaning up**

In addition to the next() method, the [Symbol.iterator]() may optionally return a method called return().

The return() method is invoked automatically when the iteration is stopped prematurely. It is where you can place the code to clean up the resources.

The following example implements the return() method for the Sequence object:

class Sequence {

constructor( start = 0, end = Infinity, interval = 1 ) {

this.start = start;

this.end = end;

this.interval = interval;

}

[Symbol.iterator]() {

let counter = 0;

let nextIndex = this.start;

return {

next: () => {

if ( nextIndex <= this.end ) {

let result = { value: nextIndex, done: false }

nextIndex += this.interval;

counter++;

return result;

}

return { value: counter, done: true };

},

return: () => {

console.log('cleaning up...');

return { value: undefined, done: true };

}

}

}

}

Code language: JavaScript (javascript)

The following snippet uses the Sequence object to generate a sequence of odd numbers from 1 to 10. However, it prematurely stops the iteration. As a result, the return() method is automatically invoked.

let oddNumbers = new Sequence(1, 10, 2);

for (const num of oddNumbers) {

if( num > 7 ) {

break;

}

console.log(num);

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

cleaning up...

In this tutorial, you have learned about the JavaScript iterator and how to use the iteration protocols to implement customize iteration logic.

# JavaScript Generators

**Summary**: in this tutorial, you will learn about JavaScript Generators and how to use them effectively.

## **Introduction to JavaScript Generators**

In JavaScript, a regular [function](https://www.javascripttutorial.net/javascript-function/) is executed based on the run-to-completion model. It cannot pause midway and then continues from where it paused. For example:

function foo() {

console.log('I');

console.log('cannot');

console.log('pause');

}

Code language: JavaScript (javascript)

The foo() function executes from top to bottom. The only way to exit the foo() is by returning from it or throwing an error. If you invoke the foo() function again, it will start the execution from the top to bottom.

foo();

Output:

I

cannot

pause

ES6 introduces a new kind of function that is different from a regular function: function generator or generator.

A generator can pause midway and then continues from where it paused. For example:

function\* generate() {

console.log('invoked 1st time');

yield 1;

console.log('invoked 2nd time');

yield 2;

}

Code language: JavaScript (javascript)

Let’s examine the generate() function in detail.

* First, you see the asterisk (\*) after the function keyword. The asterisk denotes that the generate() is a generator, not a normal function.
* Second, the [yield](https://www.javascripttutorial.net/es6/javascript-yield/) statement returns a value and pauses the execution of the function.

The following code invokes the generate() generator:

let gen = generate();

Code language: JavaScript (javascript)

When you invoke the generate() generator:

* First, you see nothing in the console. If the generate() were a regular function, you would expect to see some messages.
* Second, you get something back from generate() as a returned value.

Let’s show the returned value on the console:

console.log(gen);

Code language: JavaScript (javascript)

Output:

Object [Generator] {}

Code language: CSS (css)

So, a generator returns a Generator object without executing its body when it is invoked.

The Generator object returns another object with two properties: done and value. In other words, a Generator object is [iterable](https://www.javascripttutorial.net/es6/javascript-iterator/).

The following calls the next() method on the Generator object:

let result = gen.next();

console.log(result);

Code language: JavaScript (javascript)

Output:

invoked 1st time

{ value: 1, done: false }

Code language: CSS (css)

As you can see, the Generator object executes its body which outputs message 'invoked 1st time' at line 1 and returns the value 1 at line 2.

The yield statement returns 1 and pauses the generator at line 2.

Similarly, the following code invokes the next() method of the Generator second time:

result = gen.next();

console.log(result);

Code language: JavaScript (javascript)

Output:

invoked 2nd time

{ value: 2, done: false }

Code language: CSS (css)

This time the Generator resumes its execution from line 3 that outputs the message 'invoked 2nd time' and returns (or yield) 2.

The following invokes the next() method of the generator object third time:

result = gen.next();

console.log(result);

Code language: JavaScript (javascript)

Output:

{ value: undefined, done: true }

Code language: CSS (css)

Since a generator is iterable, you can use the [for...of](https://www.javascripttutorial.net/es6/javascript-for-of/) loop:

for (const g of gen) {

console.log(g);

}

Code language: JavaScript (javascript)

Here is the output:

invoked 1st time

1

invoked 2nd time

2

## **More generator examples**

The following example illustrates how to use a generator to generate a never-ending sequence:

function\* forever() {

let index = 0;

while (true) {

yield index++;

}

}

let f = forever();

console.log(f.next()); *// 0*

console.log(f.next()); *// 1*

console.log(f.next()); *// 2*

Code language: JavaScript (javascript)

Each time you call the next() method of the forever generator, it returns the next number in the sequence starting from 0.

## **Using generators to implement iterators**

When you implement an iterator, you have to manually define the next() method. In the next() method, you also have to manually save the state of the current element.

Since generators are iterables, they can help you simplify the code for implementing iterator.

The following is a Sequence iterator created in the [iterator tutorial](https://www.javascripttutorial.net/es6/javascript-iterator/):

class Sequence {

constructor( start = 0, end = Infinity, interval = 1 ) {

this.start = start;

this.end = end;

this.interval = interval;

}

[Symbol.iterator]() {

let counter = 0;

let nextIndex = this.start;

return {

next: () => {

if ( nextIndex < this.end ) {

let result = { value: nextIndex, done: false }

nextIndex += this.interval;

counter++;

return result;

}

return { value: counter, done: true };

}

}

}

}

Code language: JavaScript (javascript)

And here is the new Sequence iterator that uses a generator:

class Sequence {

constructor( start = 0, end = Infinity, interval = 1 ) {

this.start = start;

this.end = end;

this.interval = interval;

}

\* [Symbol.iterator]() {

for( let index = this.start; index <= this.end; index += this.interval ) {

yield index;

}

}

}

Code language: JavaScript (javascript)

As you an see, the method Symbol.iterator is much simpler by using the generator.

The following script uses the Sequence iterator to generate a sequence of odd numbers from 1 to 10:

let oddNumbers = new Sequence(1, 10, 2);

for (const num of oddNumbers) {

console.log(num);

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

9

## **Using a generator to implement the Bag data structure**

A Bag is a data structure that has the ability to collect elements and iterate through elements. It doesn’t support removing items.

The following script implements the Bag data structure:

class Bag {

constructor() {

this.elements = [];

}

isEmpty() {

return this.elements.length === 0;

}

add(element) {

this.elements.push(element);

}

\* [Symbol.iterator]() {

for (let element of this.elements) {

yield element;

}

}

}

let bag = new Bag();

bag.add(1);

bag.add(2);

bag.add(3);

for (let e of bag) {

console.log(e);

}

Code language: JavaScript (javascript)

Output:

1

2

3

## **Summary**

* Generators are created by the generator function function\* f(){}.
* Generators do not execute its body immediately when they are invoked.
* Generators can pause midway and resumes their executions where they were paused. The yield statement pauses the execution of a generator and returns a value.
* Generators are iterable so you can use them with the for...of loop.

# JavaScript yield

**Summary**: in this tutorial, you will learn about the JavaScript yield keyword and how to use it in generator functions.

## **Introduction to the JavaScript yield keyword**

The yield keyword allows you to pause and resume a [generator](https://www.javascripttutorial.net/es6/javascript-generators/) function (function\*).

The following shows the syntax of the yield keyword:

[variable\_name] = yield [expression];

Code language: JavaScript (javascript)

In this syntax:

* The expression specifies the value to return from a generator function via the iteration protocol. If you omit the expression, the yield returns undefined.
* The variable\_name stores the optional value passed to the next() method of the iterator object.

## **JavaScript yield examples**

Let’s take some examples of using the yield keyword.

### **A) Returning a value**

The following trivial example illustrates how to use the yield keyword to return a value from a generator function:

function\* foo() {

yield 1;

yield 2;

yield 3;

}

let f = foo();

console.log(f.next());

Code language: JavaScript (javascript)

Output:

{ value: 1, done: false }

Code language: CSS (css)

As you can see the value that follows the yield is added to the value property of the return object when the next() is called:

yield 1;

Code language: JavaScript (javascript)

### **B) Returning undefined**

This example illustrates how to use the yield keyword to return undefined:

function\* bar() {

yield;

}

let b = bar();

console.log(b.next());

Code language: JavaScript (javascript)

Output:

{ value: undefined, done: false }

Code language: CSS (css)

### **C) Passing a value to the next() method**

In the following example, the yield keyword is an expression that evaluates to the argument passed to the next() method:

function\* generate() {

let result = yield;

console.log(`result is ${result}`);

}

let g = generate();

console.log(g.next());

console.log(g.next(1000));

Code language: JavaScript (javascript)

The first call g.next() returns the following object:

{ value: undefined, done: false }

Code language: CSS (css)

The second call g.next() carries the following tasks:

* Evaluate yield to 1000.
* Assign result the value of yield, which is 1000.
* Output the message and return the object

Output:

result is 1000

{ value: undefined, done: true }

Code language: CSS (css)

### **D) Using yield in an array**

The following example uses the yield keyword as elements of an [array](https://www.javascripttutorial.net/javascript-array/):

function\* baz() {

let arr = [yield, yield];

console.log(arr);

}

var z = baz();

console.log(z.next());

console.log(z.next(1));

console.log(z.next(2));

Code language: JavaScript (javascript)

The first call z.next() sets the first element of the arr array to 1 and returns the following object:

{ value: undefined, done: false }

Code language: CSS (css)

The second call z.next() sets the second of the arr array to 2 and returns the following object:

{ value: undefined, done: false }

Code language: CSS (css)

The third call z.next() shows the contents of the arr array and returns the following object:

[ 1, 2 ]

{ value: undefined, done: true }

Code language: CSS (css)

### **E) Using yield to return an array**

The following generator function uses the yield keyword to return an array:

function\* yieldArray() {

yield 1;

yield [ 20, 30, 40 ];

}

let y = yieldArray();

console.log(y.next());

console.log(y.next());

console.log(y.next());

Code language: JavaScript (javascript)

The first call y.next() returns the following object:

{ value: 1, done: false }

Code language: CSS (css)

The second call y.next() returns the following object:

{ value: [ 20, 30, 40 ], done: false }

Code language: CSS (css)

In this case, yield sets the array [ 20, 30, 40 ] as the value of the value property of the return object.

The third call y.next() returns the following object:

{ value: undefined, done: true }

Code language: CSS (css)

### **F) Using the yield to return individual elements of an array**

See the following generator function:

function\* yieldArrayElements() {

yield 1;

yield\* [ 20, 30, 40 ];

}

let a = yieldArrayElements();

console.log(a.next()); *// { value: 1, done: false }*

console.log(a.next()); *// { value: 20, done: false }*

console.log(a.next()); *// { value: 30, done: false }*

console.log(a.next()); *// { value: 40, done: false }*

Code language: JavaScript (javascript)

In this example, yield\* is the new syntax. The yield\* expression is used to delegate to another iterable object or generator.

As a result, the following expression returns the individual elements of the array [20, 30, 40]:

yield\* [20, 30, 40];

Code language: JavaScript (javascript)

In this tutorial, you have learned about the JavaScript yield keyword and how to use it in function generators.

# JavaScript Promises

**Summary**: in this tutorial, you will learn about JavaScript promises and how to use them effectively.

## **Why JavaScript promises**

The following example [defines a function](https://www.javascripttutorial.net/javascript-function/) getUsers() that returns a list of user [objects](https://www.javascripttutorial.net/javascript-objects/):

function getUsers() {

return [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}

Code language: JavaScript (javascript)

Each user object has two properties username and email.

To find a user by username from the user list returned by the getUsers() function, you can use the findUser() function as follows:

function findUser(username) {

const users = getUsers();

const user = users.find((user) => user.username === username);

return user;

}

Code language: JavaScript (javascript)

In the findUser() function:

* First, get a user array by calling the getUsers() function
* Second, find the user with a specific username by using the [find()](https://www.javascripttutorial.net/es6/javascript-array-find/) method of the Array object.
* Third, return the matched user.

The following shows the complete code for finding a user with the username 'john':

function getUsers() {

return [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}

function findUser(username) {

const users = getUsers();

const user = users.find((user) => user.username === username);

return user;

}

console.log(findUser('john'));

Code language: JavaScript (javascript)

Output:

{ username: 'john', email: 'john@test.com' }

Code language: CSS (css)

The code in the findUser() function is synchronous and blocking. The findUser() function executes the getUsers() function to get a user array, calls the find() method on the users array to search for a user with a specific username, and returns the matched user.

In practice, the getUsers() function may access a database or call an API to get the user list. Therefore, the getUsers() function will have a delay.

To simulate the delay, you can use the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function. For example:

function getUsers() {

let users = [];

*// delay 1 second (1000ms)*

setTimeout(() => {

users = [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}, 1000);

return users;

}

Code language: JavaScript (javascript)

How it works.

* First, define an array users and initialize its value with an empty array.
* Second, assign an array of the users to the users variable inside the callback of the setTimeout() function.
* Third, return the users array

The getUsers() won’t work properly and always returns an empty array. Therefore, the findUser() function won’t work as expected:

function getUsers() {

let users = [];

setTimeout(() => {

users = [

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

];

}, 1000);

return users;

}

function findUser(username) {

const users = getUsers(); *// A*

const user = users.find((user) => user.username === username); *// B*

return user;

}

console.log(findUser('john'));

Code language: JavaScript (javascript)

Output:

undefined

Code language: JavaScript (javascript)

Because the getUsers() returns an empty array, the users array is empty (line A). When calling the find() method on the users array, the method returns undefined (line B)

The challenge is how to access the users returned from the getUsers() function after one second. One classical approach is to use the [callback](https://www.javascripttutorial.net/javascript-callback/).

### **Using callbacks to deal with an asynchronous operation**

The following example adds a callback argument to the getUsers() and findUser() functions:

function getUsers(callback) {

setTimeout(() => {

callback([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

}, 1000);

}

function findUser(username, callback) {

getUsers((users) => {

const user = users.find((user) => user.username === username);

callback(user);

});

}

findUser('john', console.log);

Code language: JavaScript (javascript)

Output:

{ username: 'john', email: 'john@test.com' }

Code language: CSS (css)

In this example, the getUsers() function accepts a callback function as an argument and invokes it with the users array inside the setTimeout() function. Also, the findUser() function accepts a callback function that processes the matched user.

The callback approach works very well. However, it makes the code more difficult to follow. Also, it adds complexity to the functions with callback arguments.

If the number of functions grows, you may end up with the callback hell problem. To resolve this, JavaScript comes up with the concept of promises.

## **Understanding JavaScript Promises**

By definition, a promise is an **object** that encapsulates the result of an **asynchronous operation**.

A promise object has a state that can be one of the following:

* Pending
* Fulfilled with a **value**
* Rejected for a **reason**

In the beginning, the state of a promise is pending, indicating that the asynchronous operation is in progress. Depending on the result of the asynchronous operation, the state changes to either fulfilled or rejected.

The fulfilled state indicates that the asynchronous operation was completed successfully:

The rejected state indicates that the asynchronous operation failed.

## **Creating a promise**

To create a promise object, you use the Promise() constructor:

const promise = new Promise((resolve, reject) => {

*// contain an operation*

*// ...*

*// return the state*

if (success) {

resolve(value);

} else {

reject(error);

}

});

Code language: JavaScript (javascript)

The promise constructor accepts a callback function that typically performs an asynchronous operation. This function is often referred to as an executor.

In turn, the executor accepts two callback functions with the name resolve and reject.

Note that the callback functions passed into the executor are resolve and reject by convention only.

If the asynchronous operation completes successfully, the executor will call the resolve() function to change the state of the promise from pending to fulfilled with a value.

In case of an error, the executor will call the reject() function to change the state of the promise from pending to rejected with the error reason.

Once a promise reaches either fulfilled or rejected state, it stays in that state and can’t go to another state.

In other words, a promise cannot go from the fulfilled state to the rejected state and vice versa. Also, it cannot go back from the fulfilled or rejected state to the pending state.

Once a new Promise object is created, its state is pending. If a promise reaches fulfilled or rejected state, it is resolved.

Note that you will rarely create promise objects in practice. Instead, you will consume promises provided by libraries.

## **Consuming a Promise: then, catch, finally**

### **1) The then() method**

To get the value of a promise when it’s fulfilled, you call the then() method of the promise object. The following shows the syntax of the then() method:

promise.then(onFulfilled,onRejected);

Code language: CSS (css)

The then() method accepts two callback functions: onFulfilled and onRejected.

The then() method calls the onFulfilled() with a value, if the promise is fulfilled or the onRejected() with an error if the promise is rejected.

Note that both onFulfilled and onRejected arguments are optional.

The following example shows how to use then() method of the Promise object returned by the getUsers() function:

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

}, 1000);

});

}

function onFulfilled(users) {

console.log(users);

}

const promise = getUsers();

promise.then(onFulfilled);

Code language: JavaScript (javascript)

Output:

[

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' }

]

Code language: JavaScript (javascript)

In this example:

* First, define the onFulfilled() function to be called when the promise is fulfilled.
* Second, call the getUsers() function to get a promise object.
* Third, call the then() method of the promise object and output the user list to the console.

To make the code more concise, you can use an [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) as the argument of the then() method like this:

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

}, 1000);

});

}

const promise = getUsers();

promise.then((users) => {

console.log(users);

});

Code language: JavaScript (javascript)

Because the getUsers() function returns a promise object, you can chain the function call with the then() method like this:

*// getUsers() function*

*//...*

getUsers().then((users) => {

console.log(users);

});

Code language: JavaScript (javascript)

In this example, the getUsers() function always succeeds. To simulate the error, we can use a success flag like the following:

let success = true;

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (success) {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

} else {

reject('Failed to the user list');

}

}, 1000);

});

}

function onFulfilled(users) {

console.log(users);

}

function onRejected(error) {

console.log(error);

}

const promise = getUsers();

promise.then(onFulfilled, onRejected);

Code language: JavaScript (javascript)

How it works.

First, define the success variable and initialize its value to true.

If the success is true, the promise in the getUsers() function is fulfilled with a user list. Otherwise, it is rejected with an error message.

Second, define the onFulfilled and onRejected functions.

Third, get the promise from the getUsers() function and call the then() method with the onFulfilled and onRejected functions.

The following shows how to use the arrow functions as the arguments of the then() method:

*// getUsers() function*

*// ...*

const promise = getUsers();

promise.then(

(users) => console.log,

(error) => console.log

);

Code language: JavaScript (javascript)

### **2) The catch() method**

If you want to get the error only when the state of the promise is rejected, you can use the catch() method of the Promise object:

promise.catch(onRejected);

Code language: CSS (css)

Internally, the catch() method invokes the then(undefined, onRejected) method.

The following example changes the success flag to false to simulate the error scenario:

let success = false;

function getUsers() {

return new Promise((resolve, reject) => {

setTimeout(() => {

if (success) {

resolve([

{ username: 'john', email: 'john@test.com' },

{ username: 'jane', email: 'jane@test.com' },

]);

} else {

reject('Failed to the user list');

}

}, 1000);

});

}

const promise = getUsers();

promise.catch((error) => {

console.log(error);

});

Code language: JavaScript (javascript)

### **3) The finally() method**

Sometimes, you want to execute the same piece of code whether the promise is fulfilled or rejected. For example:

const render = () => {

*//...*

};

getUsers()

.then((users) => {

console.log(users);

render();

})

.catch((error) => {

console.log(error);

render();

});

Code language: JavaScript (javascript)

As you can see, the render() function call is duplicated in both then() and catch() methods.

To remove this duplicate and execute the render() whether the promise is fulfilled or rejected, you use the finally() method, like this:

const render = () => {

*//...*

};

getUsers()

.then((users) => {

console.log(users);

})

.catch((error) => {

console.log(error);

})

.finally(() => {

render();

});

Code language: JavaScript (javascript)

## **A practical JavaScript Promise example**

The following example shows how to load a JSON file from the server and display its contents on a webpage.

Suppose you have the following JSON file:

https:*//www.javascripttutorial.net/sample/promise/api.json*

Code language: JavaScript (javascript)

with the following contents:

{

"message": "JavaScript Promise Demo"

}

Code language: JSON / JSON with Comments (json)

The following shows the HTML page that contains a button. When you click the button, the page loads data from the JSON file and shows the message:

<!DOCTYPE html>

**<html>**

**<head>**

**<meta charset="utf-8">**

**<title>**JavaScript Promise Demo**</title>**

**<link href="css/style.css" rel="stylesheet">**

**</head>**

**<body>**

**<div id="container">**

**<div id="message"></div>**

**<button id="btnGet">**Get Message**</button>**

**</div>**

**<script src="js/promise-demo.js">**

**</script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

The following shows the promise-demo.js file:

function load(url) {

return new Promise(function (resolve, reject) {

const request = new XMLHttpRequest();

request.onreadystatechange = function () {

if (this.readyState === 4 && this.status == 200) {

resolve(this.response);

} else {

reject(this.status);

}

};

request.open('GET', url, true);

request.send();

});

}

const url = 'https://www.javascripttutorial.net/sample/promise/api.json';

const btn = document.querySelector('#btnGet');

const msg = document.querySelector('#message');

btn.addEventListener('click', () => {

load(URL)

.then((response) => {

const result = JSON.parse(response);

msg.innerHTML = result.message;

})

.catch((error) => {

msg.innerHTML = `Error getting the message, HTTP status: ${error}`;

});

});

Code language: JavaScript (javascript)

How it works.

First, define the load() function that uses the XMLHttpRequest object to load the JSON file from the server:

function load(url) {

return new Promise(function (resolve, reject) {

const request = new XMLHttpRequest();

request.onreadystatechange = function () {

if (this.readyState === 4 && this.status == 200) {

resolve(this.response);

} else {

reject(this.status);

}

};

request.open('GET', url, true);

request.send();

});

}

Code language: JavaScript (javascript)

In the executor, we call resolve() function with the Response if the HTTP status code is 200. Otherwise, we invoke the reject() function with the HTTP status code.

Second, register the button click event listener and call the then() method of the promise object. If the load is successful, then we show the message returned from the server. Otherwise, we show the error message with the HTTP status code.

const url = 'https://www.javascripttutorial.net/sample/promise/api.json';

const btn = document.querySelector('#btnGet');

const msg = document.querySelector('#message');

btn.addEventListener('click', () => {

load(URL)

.then((response) => {

const result = JSON.parse(response);

msg.innerHTML = result.message;

})

.catch((error) => {

msg.innerHTML = `Error getting the message, HTTP status: ${error}`;

});

});

Code language: JavaScript (javascript)

## **Summary**

* A promise is an object that encapsulates the result of an asynchronous operation.
* A promise starts in the pending state and ends in either fulfilled state or rejected state.
* Use then() method to schedule a callback to be executed when the promise is fulfilled, and catch() method to schedule a callback to be invoked when the promise is rejected.
* Place the code that you want to execute in the finally() method whether the promise is fulfilled or rejected.

# Promise Chaining

**Summary**: in this tutorial, you will learn about the JavaScript promise chaining pattern that chains the promises to execute asynchronous operations in sequence.

## **Introduction to the JavaScript promise chaining**

Sometimes, you want to execute two or more related asynchronous operations, where the next operation starts with the result from the previous step. For example:

First, create a new promise that resolves to the number 10 after 3 seconds:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

Code language: JavaScript (javascript)

Note that the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function simulates an asynchronous operation.

Then, invoke the then() method of the promise:

p.then((result) => {

console.log(result);

return result \* 2;

});

Code language: JavaScript (javascript)

The callback passed to the then() method executes once the promise is resolved. In the callback, we show the result of the promise and return a new value multiplied by two (result\*2).

Because the then() method returns a new Promise with a value resolved to a value, you can call the then() method on the return Promise like this:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result);

return result \* 2;

}).then((result) => {

console.log(result);

return result \* 3;

});

Code language: JavaScript (javascript)

Output:

10

20

In this example, the return value in the first then() method is passed to the second then() method. You can keep calling the then() method successively as follows:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result); *// 10*

return result \* 2;

}).then((result) => {

console.log(result); *// 20*

return result \* 3;

}).then((result) => {

console.log(result); *// 60*

return result \* 4;

});

Code language: JavaScript (javascript)

Output:

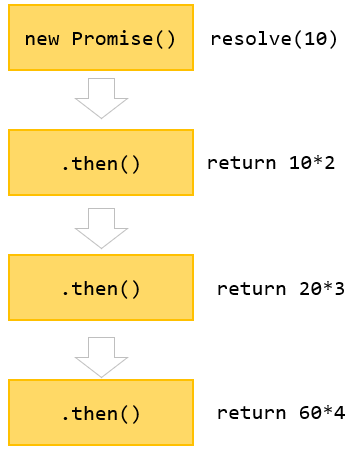
10

20

60

The way we call the then() methods like this is often referred to as a promise chain.

The following picture illustrates the promise chain:



## **Multiple handlers for a promise**

When you call the then() method multiple times on a promise, it is not the promise chaining. For example:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result); *// 10*

return result \* 2;

})

p.then((result) => {

console.log(result); *// 10*

return result \* 3;

})

p.then((result) => {

console.log(result); *// 10*

return result \* 4;

});

Code language: JavaScript (javascript)

Output:

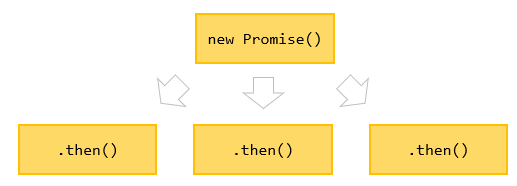
10

10

10

In this example, we have multiple handlers for one promise. These handlers have no relationships. Also, they execute independently and don’t pass the result from one to another like the promise chain above.

The following picture illustrates a promise that has multiple handlers:



In practice, you will rarely use multiple handlers for one promise.

## **Returning a Promise**

When you return a value in the then() method, the then() method returns a new Promise that immediately resolves to the return value.

Also, you can return a new promise in the then() method, like this:

let p = new Promise((resolve, reject) => {

setTimeout(() => {

resolve(10);

}, 3 \* 100);

});

p.then((result) => {

console.log(result);

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve(result \* 2);

}, 3 \* 1000);

});

}).then((result) => {

console.log(result);

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve(result \* 3);

}, 3 \* 1000);

});

}).then(result => console.log(result));

Code language: JavaScript (javascript)

Output:

10

20

60

This example shows 10, 20, and 60 after every 3 seconds. This code pattern allows you to execute some tasks in sequence.

The following modified the above example:

function generateNumber(num) {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve(num);

}, 3 \* 1000);

});

}

generateNumber(10)

.then((result) => {

console.log(result);

return generateNumber(result \* 2);

})

.then((result) => {

console.log(result);

return generateNumber(result \* 3);

})

.then((result) => console.log(result));

Code language: JavaScript (javascript)

### **Promise chaining syntax**

Sometimes, you have multiple asynchronous tasks that you want to execute in sequence. In addition, you need to pass the result of the previous step to the next one. In this case, you can use the following syntax:

step1()

.then(result => step2(result))

.then(result => step3(result))

...

Code language: JavaScript (javascript)

If you need to pass the result from the previous task to the next one without passing the result, you use this syntax:

step1()

.then(step2)

.then(step3)

...

Code language: CSS (css)

Suppose that you want to perform the following asynchronous operations in sequence:

* First, get the user from the database.
* Second, get the services of the selected user.
* Third, calculate the service cost from the user’s services.

The following functions illustrate the three asynchronous operations:

function getUser(userId) {

return new Promise((resolve, reject) => {

console.log('Get the user from the database.');

setTimeout(() => {

resolve({

userId: userId,

username: 'admin'

});

}, 1000);

})

}

function getServices(user) {

return new Promise((resolve, reject) => {

console.log(`Get the services of ${user.username} from the API.`);

setTimeout(() => {

resolve(['Email', 'VPN', 'CDN']);

}, 3 \* 1000);

});

}

function getServiceCost(services) {

return new Promise((resolve, reject) => {

console.log(`Calculate the service cost of ${services}.`);

setTimeout(() => {

resolve(services.length \* 100);

}, 2 \* 1000);

});

}

Code language: JavaScript (javascript)

The following uses the promises to serialize the sequences:

getUser(100)

.then(getServices)

.then(getServiceCost)

.then(console.log);

Code language: CSS (css)

Output

Get the user from the database.

Get the services of admin from the API.

Calculate the service cost of Email,VPN,CDN.

300

Code language: JavaScript (javascript)

Note that ES2017 introduced the [async/await](https://www.javascripttutorial.net/es-next/javascript-async-await/) that helps you write the code that is cleaner than using the promise chaining technique.

In this tutorial, you have learned about the promise chain that executes multiple asynchronous tasks in sequence.

# JavaScript Promise.all()

**Summary**: in this tutorial, you will learn how to use the Promise.all() static method to aggregate results from multiple asynchronous operations.

## **Introduction to the JavaScript Promise.all() method**

The Promise.all() static method takes an [iterable](https://www.javascripttutorial.net/es6/javascript-iterator/) of [promises](https://www.javascripttutorial.net/es6/javascript-promises/):

Promise.all(iterable);

Code language: JavaScript (javascript)

The Promise.all() method returns a single promise that resolves when all the input promises have been resolved. The returned promise resolves to an array of the results of the input promises:

In this diagram, the promise1 resolves to a value v1 at t1 and the promise2 resolves to a value v2 at t2. Hence, the Promise.all(promise1, promise2) returns a promise that resolves to an array containing the results of the promise1 and promise2 [v1, v2] at t2.

In other words, the Promise.all() waits for all the input promises to resolve and returns a new promise that resolves to an array containing the results of the input promises.

If one of the input promise rejects, the Promise.all() method immediately returns a promise that rejects with an error of the first rejected promise:

In this diagram, the promise2 rejects at t1 with an error. Therefore, the Promise.all() returns a new promise that immediately rejects with the same error. Also, the Promise.all() doesn’t care other input promises, whether they will resolve or reject.

In practice, the Promise.all() is useful to aggregate the results from multiple asynchronous operations.

## **JavaScript Promise.all() method examples**

Let’s take some examples to understand how the Promise.all() method works.

### **1) Resolved promises example**

The following promises resolve to 10, 20, and 30 after 1, 2, and 3 seconds. We use the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) to simulate the asynchronous operations:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has resolved');

resolve(20);

}, 2 \* 1000);

});

const p3 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The third promise has resolved');

resolve(30);

}, 3 \* 1000);

});

Promise.all([p1, p2, p3]).then((results) => {

const total = results.reduce((p, c) => p + c);

console.log(`Results: ${results}`);

console.log(`Total: ${total}`);

});

Code language: JavaScript (javascript)

Output

The first promise has resolved

The second promise has resolved

The third promise has resolved

Results: 10,20,30

Total: 60

When all promises have resolved, the values from these promises are passed into the callback of the then() method as an array.

Inside the callback, we use the Array’s [reduce()](https://www.javascripttutorial.net/javascript-array-reduce/) method to calculate the total value and use the console.log to display the array of values as well as the total.

### **2) Rejected promises example**

The Promise.all() returns a Promise that is rejected if any of the input promises are rejected.

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has rejected');

reject('Failed');

}, 2 \* 1000);

});

const p3 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The third promise has resolved');

resolve(30);

}, 3 \* 1000);

});

Promise.all([p1, p2, p3])

.then(console.log) *// never execute*

.catch(console.log);

Code language: JavaScript (javascript)

Output:

The first promise has resolved

The second promise has rejected

Failed

The third promise has resolved

In this example, we have three promises: the first one is resolved after 1 second, the second is rejected after 2 seconds, and the third one is resolved after 3 seconds.

As a result, the returned promise is rejected because the second promise is rejected. The catch() method is executed to display the reason for the rejected promise.

## **Summary**

* The Promise.all() method accepts a list of promises and returns a new promsie that resolve to an array of results of the input promises if all the input promises resolved; or reject with an error of the first rejected promise.
* Use the Promise.all() method to aggregate results from multiple asynchronous operations.

# JavaScript Promise.race()

**Summary**: in this tutorial, you will learn how to use the JavaScript Promise.race() static method.

## **Introduction to JavaScript Promise.race() static method**

The Promise.race() static method accepts a list of [promises](https://www.javascripttutorial.net/es6/javascript-promises/) as an iterable object and returns a new promise that fulfills or rejects as soon as there is one promise that fulfills or rejects, with the value or reason from that promise.

Here’s the syntax of the Promise.race() method:

Promise.race(iterable)

Code language: JavaScript (javascript)

In this syntax, the iterable is an [iterable object](https://www.javascripttutorial.net/es6/javascript-iterator/) that contains a list of promises.

The name of Promise.race() implies that all the promises race against each other with a single winner, either resolved or rejected.

See the following diagram:

In this diagram:

* The promise1 is fulfilled with the value v1 at t1.
* The promise2 is rejected with the error at t2.
* Because the promise1 is resolved earlier than the promise2, the promise1 wins the race. Therefore, the Promise.race([promise1, promise2]) returns a new promise that is fulfilled with the value v1 at t1.

See another diagram:

In this diagram:

* The promise1 is fulfilled with v1 at t2
* The promise2 is rejected with error at t1.
* Because the promise2 is resolved earlier than the promise1, the promise2 wins the race. Therefore, the Promise.race([promise1, promise2]) returns a new promise that is rejected with the error at t1.

## **JavaScript Promise.race() examples**

Let’s take some examples of using the Promise.race() static method.

### **1) Simple JavaScript Promise.race() examples**

The following creates two promises: one resolves in 1 second and the other resolves in 2 seconds. Because the first promise resolves faster than the second one, the Promise.race() resolves with the value from the first promise:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has resolved');

resolve(20);

}, 2 \* 1000);

});

Promise.race([p1, p2])

.then(value => console.log(`Resolved: ${value}`))

.catch(reason => console.log(`Rejected: ${reason}`));

Code language: JavaScript (javascript)

Output:

The first promise has resolved

Resolved: 10

The second promise has resolved

The following example creates two promises. The first promise resolves in 1 second while the second one rejects in 2 seconds. Because the first promise is faster than the second one, the returned promise resolves to the value from the first promise:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The first promise has resolved');

resolve(10);

}, 1 \* 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('The second promise has rejected');

reject(20);

}, 2 \* 1000);

});

Promise.race([p1, p2])

.then(value => console.log(`Resolved: ${value}`))

.catch(reason => console.log(`Rejected: ${reason}`));

Code language: JavaScript (javascript)

Output

The first promise has resolved

Resolved: 10

The second promise has rejected

Note that if the second promise was faster than the first one, the return promise would reject with the reason of the second promise.

### **2) Practical JavaScript Promise.race() example**

Suppose you have to show a spinner if the data loading process from the server is taking longer than a number of seconds.

To do this, you can use the Promise.race() static method. If a timeout occurs, you show the loading indicator, otherwise, you show the message.

The following illustrates the HTML code:

<!DOCTYPE html>

**<html>**

**<head>**

**<meta charset="utf-8">**

**<title>**JavaScript Promise.race() Demo**</title>**

**<link href="css/promise-race.css" rel="stylesheet">**

**</head>**

**<body>**

**<div id="container">**

**<button id="btnGet">**Get Message**</button>**

**<div id="message"></div>**

**<div id="loader"></div>**

**</div>**

**<script src="js/promise-race.js"></script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

To create the loading indicator, we use the CSS animation feature. See the [promise-race.css](https://www.javascripttutorial.net/sample/promise/css/promise-race.css) for more information. Technically speaking, if an element has the .loader class, it shows the loading indicator.

First, define a new function that loads data. It uses the setTimeout() to emulate an asynchronous operation:

const DATA\_LOAD\_TIME = 5000;

function getData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const message = 'Promise.race() Demo';

resolve(message);

}, DATA\_LOAD\_TIME);

});

}

Code language: JavaScript (javascript)

Second, develop a function that shows some contents:

function showContent(message) {

document.querySelector('#message').textContent = message;

}

Code language: JavaScript (javascript)

This function can also be used to set the message to blank.

Third, define the timeout() function returns a promise which rejects when the TIMEOUT is passed.

const TIMEOUT = 500;

function timeout() {

return new Promise((resolve, reject) => {

setTimeout(() => reject(), TIMEOUT);

});

}

Code language: JavaScript (javascript)

Fourth, develop a couple of functions that show and hide the loading indicator:

function showLoadingIndicator() {

document.querySelector('#loader').className = 'loader';

}

function hideLoadingIndicator() {

document.querySelector('#loader').className = '';

}

Code language: JavaScript (javascript)

Fifth, attach a click event listener to the**Get Message** button. Inside the click handler, use the Promise.race() static method:

*// handle button click event*

const btn = document.querySelector('#btnGet');

btn.addEventListener('click', () => {

*// reset UI if users click the 2nd, 3rd, ... time*

reset();

*// show content or loading indicator*

Promise.race([getData()

.then(showContent)

.then(hideLoadingIndicator), timeout()

])

.catch(showLoadingIndicator);

});

Code language: JavaScript (javascript)

We pass two promises to the Promise.race() method:

Promise.race([getData()

.then(showContent)

.then(hideLoadingIndicator), timeout()

])

.catch(showLoadingIndicator);

Code language: JavaScript (javascript)

The first promise gets data from the server, shows the content, and hides the loading indicator. The second promise sets a timeout.

If the first promise takes more than 500 ms to settle, the catch() is called to show the loading indicator. Once the first promise resolves, it hides the loading indicator.

Finally, develop a reset() function that hides the message and loading indicator if the button is clicked from the second time.

*// reset UI*

function reset() {

hideLoadingIndicator();

showContent('');

}

Code language: JavaScript (javascript)

Put it all together.

*// after 0.5 seconds, if the getData() has not resolved, then show*

*// the Loading indicator*

const TIMEOUT = 500;

const DATA\_LOAD\_TIME = 5000;

function getData() {

return new Promise((resolve, reject) => {

setTimeout(() => {

const message = 'Promise.race() Demo';

resolve(message);

}, DATA\_LOAD\_TIME);

});

}

function showContent(message) {

document.querySelector('#message').textContent = message;

}

function timeout() {

return new Promise((resolve, reject) => {

setTimeout(() => reject(), TIMEOUT);

});

}

function showLoadingIndicator() {

document.querySelector('#loader').className = 'loader';

}

function hideLoadingIndicator() {

document.querySelector('#loader').className = '';

}

*// handle button click event*

const btn = document.querySelector('#btnGet');

btn.addEventListener('click', () => {

*// reset UI if users click the second time*

reset();

*// show content or loading indicator*

Promise.race([getData()

.then(showContent)

.then(hideLoadingIndicator), timeout()

])

.catch(showLoadingIndicator);

});

*// reset UI*

function reset() {

hideLoadingIndicator();

showContent('');

}

Code language: JavaScript (javascript)

## **Summary**

* The Promise.race(iterable) method returns a new promise that fulfills or rejects as soon as one of the promises in an iterable fulfills or rejects, with the value or error from that promise.

# Promise Error Handling

**Summary**: in this tutorial, you will learn how to deal with error handling in promises.

Suppose that you have a function called getUserById() that returns a [Promise](https://www.javascripttutorial.net/es6/javascript-promises/):

function getUserById(id) {

return new Promise((resolve, reject) => {

resolve({

id: id,

username: 'admin'

});

});

}

Code language: JavaScript (javascript)

## **Normal error**

First, change the getUserById() function to [throw an error](https://www.javascripttutorial.net/javascript-try-catch/) outside the promise:

function getUserById(id) {

if (typeof id !== 'number' || id <= 0) {

throw new Error('Invalid id argument');

}

return new Promise((resolve, reject) => {

resolve({

id: id,

username: 'admin'

});

});

}

Code language: JavaScript (javascript)

Second, handle the promise by using both then() and catch() methods:

getUserById('a')

.then(user => console.log(user.username))

.catch(err => console.log(err));

Code language: JavaScript (javascript)

The code throws an error:

Uncaught Error: Invalid id argument

Code language: JavaScript (javascript)

When you raise an exception outside the promise, you must catch it with try/catch:

try {

getUserById('a')

.then(user => console.log(user.username))

.catch(err => console.log(`Caught by .catch ${error}`));

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

Output:

Caught by try/catch Error: Invalid id argument

Code language: JavaScript (javascript)

## **Errors inside the Promises**

We change the getUserById() function to throw an error inside the promise:

let authorized = false;

function getUserById(id) {

return new Promise((resolve, reject) => {

if (!authorized) {

throw new Error('Unauthorized access to the user data');

}

resolve({

id: id,

username: 'admin'

});

});

}

Code language: JavaScript (javascript)

And consume the promise:

try {

getUserById(10)

.then(user => console.log(user.username))

.catch(err => console.log(`Caught by .catch ${error}`));

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

Output:

Caught by .catch Error: Unauthorized access to the user data

Code language: JavaScript (javascript)

If you throw an error inside the promise, the catch() method will catch it, not the try/catch.

If you chain promises, the catch() method will catch errors occurred in any promise. For example:

promise1

.then(promise2)

.then(promise3)

.then(promise4)

.catch(err => console.log(err));

Code language: JavaScript (javascript)

In this example, if any error in the promise1, promise2, or promise4, the catch() method will handle it.

## **Calling reject() function**

Throwing an error has the same effect as calling the reject() as illustrated in the following example:

let authorized = false;

function getUserById(id) {

return new Promise((resolve, reject) => {

if (!authorized) {

reject('Unauthorized access to the user data');

}

resolve({

id: id,

username: 'admin'

});

});

}

try {

getUserById(10)

.then(user => console.log(user.username))

.catch(err => console.log(`Caught by .catch ${err}`));

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

In this example, instead of throwing an error inside the promise, we called the reject() explicitly. The catch() method also handles the error in this case.

## **Missing the catch() method**

The following example does not provide the catch() method to handle the error inside the promise. It will cause a runtime error and terminate the program:

function getUserById(id) {

return new Promise((resolve, reject) => {

if (!authorized) {

reject('Unauthorized access to the user data');

}

resolve({

id: id,

username: 'admin'

});

});

}

try {

getUserById(10)

.then(user => console.log(user.username));

*// the following code will not execute*

console.log('next');

} catch (error) {

console.log(`Caught by try/catch ${error}`);

}

Code language: JavaScript (javascript)

Output:

Uncaught (in promise) Unauthorized access to the user data

If the promise is resolved, you can omit the catch() method. In the future, a potential error may cause the program to stop unexpectedly.

## **Summary**

* Inside the promise, the catch() method will catch the error caused by the throw statement and reject().
* If an error occurs and you don’t have the catch() method, the JavaScript engine issues a runtime error and stops the program.

# JavaScript Map Object

**Summary**: in this tutorial, you will learn about the JavaScript Map object that maps a key to a value.

## **Introduction to JavaScript Map object**

Before ES6, we often used an [object](https://www.javascripttutorial.net/javascript-objects/) to emulate a map by mapping a key to a value of any type. But using an object as a map has some side effects:

1. An object always has a default key like the [prototype](https://www.javascripttutorial.net/javascript-prototype/).
2. A key of an object must be a [string](https://www.javascripttutorial.net/javascript-string/) or a [symbol](https://www.javascripttutorial.net/es6/symbol/), you cannot use an object as a key.
3. An object does not have a property that represents the size of the map.

ES6 provides a new collection type called Map that addresses these deficiencies.

By definition, a Map object holds key-value pairs where values of any type can be used as either keys or values. In addition, a Map object remembers the original insertion order of the keys.

To create a new Map, you use the following syntax:

let map = new Map([iterable]);

Code language: JavaScript (javascript)

The Map() accepts an optional [iterable](https://www.javascripttutorial.net/es6/javascript-iterator/) object whose elements are key-value pairs.

## **Useful JavaScript Map methods**

* clear() – removes all elements from the map object.
* delete(key) – removes an element specified by the key. It returns if the element is in the map, or false if it does not.
* entries() – returns a new Iterator object that contains an array of [key, value] for each element in the map object. The order of objects in the map is the same as the insertion order.
* forEach(callback[, thisArg]) – invokes a callback for each key-value pair in the map in the insertion order. The optional thisArg parameter sets the this value for each callback.
* get(key) – returns the value associated with the key. If the key does not exist, it returns undefined.
* has(key) – returns true if a value associated with the key exists, otherwise, return false.
* keys() – returns a new Iterator that contains the keys for elements in insertion order.
* set(key, value) – sets the value for the key in the map object. It returns the map object itself therefore you can chain this method with other methods.
* values() returns a new iterator object that contains values for each element in insertion order.

## **JavaScript Map examples**

### **Create a new Map object**

Suppose you have a list of user objects as follows:

let john = {name: 'John Doe'},

lily = {name: 'Lily Bush'},

peter = {name: 'Peter Drucker'};

Code language: JavaScript (javascript)

Assuming that you have to create a map of users and roles. In this case, you use the following code:

let userRoles = new Map();

Code language: JavaScript (javascript)

The userRoles is an instance of the Map object and its type is an object as illustrated in the following example:

console.log(typeof(userRoles)); *// object*

console.log(userRoles instanceof Map); *// true*

Code language: JavaScript (javascript)

### **Add elements to a Map**

To assign a role to a user, you use the set() method:

userRoles.set(john, 'admin');

Code language: JavaScript (javascript)

The set() method maps user john with the admin role. Since the set() method is chainable, you can save some typings as shown in this example:

userRoles.set(lily, 'editor')

.set(peter, 'subscriber');

Code language: JavaScript (javascript)

### **Initialize a map with an iterable object**

As mentioned earlier, you can pass an iterable object to the Map() constructor:

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber']

]);

Code language: JavaScript (javascript)

### **Get an element from a map by key**

If you want to see the roles of John , you use the get() method:

userRoles.get(john); *// admin*

Code language: JavaScript (javascript)

If you pass a key that does not exist, the get() method will return undefined.

let foo = {name: 'Foo'};

userRoles.get(foo); *//undefined*

Code language: JavaScript (javascript)

### **Check the existence of an element by key**

To check if a key exists in the map, you use the has() method.

userRoles.has(foo); *// false*

userRoles.has(lily); *// true*

Code language: JavaScript (javascript)

### **Get the number of elements in the map**

The size property returns the number of entries of the map.

console.log(userRoles.size); *// 3*

Code language: JavaScript (javascript)

### **Iterate over map keys**

To get the keys of a Map object, you use the keys() method. The keys() returns a new [iterator](https://www.javascripttutorial.net/es6/javascript-iterator/) object that contains the keys of elements in the map.

The following example displays the username of the users in the userRoles map object.

let john = { name: 'John Doe' },

lily = { name: 'Lily Bush' },

peter = { name: 'Peter Drucker' };

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber'],

]);

for (const user of userRoles.keys()) {

console.log(user.name);

}

Code language: JavaScript (javascript)

Output:

John Doe

Lily Bush

Peter Drucker

### **Iterate over map values**

Similarly, you can use the values() method to get an iterator object that contains values for all the elements in the map:

let john = { name: 'John Doe' },

lily = { name: 'Lily Bush' },

peter = { name: 'Peter Drucker' };

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber'],

]);

for (let role of userRoles.values()) {

console.log(role);

}

Code language: JavaScript (javascript)

Output:

admin

editor

subscriber

### **Iterate over map elements**

Also, the entries() method returns an iterator object that contains an array of [key,value] of each element in the Map object:

let john = { name: 'John Doe' },

lily = { name: 'Lily Bush' },

peter = { name: 'Peter Drucker' };

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber'],

]);

for (const role of userRoles.entries()) {

console.log(`${role[0].name}: ${role[1]}`);

}

Code language: JavaScript (javascript)

To make the iteration more natural, you can use [destructuring](https://www.javascripttutorial.net/es6/destructuring/) as follows:

let john = { name: 'John Doe' },

lily = { name: 'Lily Bush' },

peter = { name: 'Peter Drucker' };

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber'],

]);

for (let [user, role] of userRoles.entries()) {

console.log(`${user.name}: ${role}`);

}

Code language: JavaScript (javascript)

In addition to [for...of](https://www.javascripttutorial.net/es6/javascript-for-of/) loop, you can use the forEach() method of the map object:

let john = { name: 'John Doe' },

lily = { name: 'Lily Bush' },

peter = { name: 'Peter Drucker' };

let userRoles = new Map([

[john, 'admin'],

[lily, 'editor'],

[peter, 'subscriber'],

]);

userRoles.forEach((role, user) => console.log(`${user.name}: ${role}`));

Code language: JavaScript (javascript)

### **Convert map keys or values to a array**

Sometimes, you want to work with an array instead of an iterable object, in this case, you can use the [spread operator](https://www.javascripttutorial.net/es6/javascript-spread/).

The following example converts keys for each element into an array of keys:

var keys = [...userRoles.keys()];

console.log(keys);

Code language: JavaScript (javascript)

Output:

[ { name: 'John Doe' },

{ name: 'Lily Bush' },

{ name: 'Peter Drucker' } ]

Code language: JavaScript (javascript)

And the following converts the values of elements to an array:

let roles = [...userRoles.values()];

console.log(roles);

Code language: JavaScript (javascript)

Output

[ 'admin', 'editor', 'subscriber' ]

Code language: JSON / JSON with Comments (json)

### **Delete an element by key**

To delete an entry in the map, you use the delete() method.

userRoles.delete(john);

Code language: CSS (css)

### **Delete all elements in the map**

To delete all entries in the Map object, you use the clear() method:

userRoles.clear();

Code language: CSS (css)

Hence, the size of the map now is zero.

console.log(userRoles.size); *// 0*

Code language: JavaScript (javascript)

## **WeakMap**

A WeakMap is similar to a Map except the keys of a WeakMap must be objects. It means that when a reference to a key (an object) is out of scope, the corresponding value is automatically released from the memory.

A WeakMap only has subset methods of a Map object:

* get(key)
* set(key, value)
* has(key)
* delete(key)

Here are the main difference between a Map and a WeekMap:

* Elements of a WeakMap cannot be iterated.
* Cannot clear all elements at once.
* Cannot check the size of a WeakMap.

In this tutorial, you have learned how to work with the JavaScript Map object and its useful methods to manipulate entries in the map.

# The Beginner’s Guide to JavaScript Set Type in ES6

**Summary**: in this tutorial, you will learn about the JavaScript Set object that allows you to manage a collection of unique values of any type effectively.

## **Introduction to the JavaScript Set object**

ES6 provides a new type named Set that stores a collection of unique values of any type. To create a new empty Set, you use the following syntax:

let setObject = new Set();

Code language: JavaScript (javascript)

The Set constructor also accepts an optional [iterable object](https://www.javascripttutorial.net/es6/javascript-iterator/). If you pass an iterable object to the Set constructor, all the elements of the iterable object will be added to the new set:

let setObject = new Set(iterableObject);

Code language: JavaScript (javascript)

## **Useful Set methods**

The Set object provides the following useful methods:

* add(value) – appends a new element with a specified value to the set. It returns the Set object, therefore, you can chain this method with another Set method.
* clear()  – removes all elements from the Set object.
* delete(value) – deletes an element specified by the value.
* entries()– returns a new Iterator that contains an array of  [value, value] .
* forEach(callback [, thisArg]) – invokes a [callback](https://www.javascripttutorial.net/javascript-callback/) on each element of the Set with the this value sets to thisArg in each call.
* has(value) – returns true if an element with a given value is in the set, or false if it is not.
* keys() – is the same as values() function.
* [@@iterator] – returns a new [Iterator](https://www.javascripttutorial.net/es6/javascript-iterator/) object that contains values of all elements stored in the insertion order.

## **JavaScript Set examples**

### **Create a new Set from an Array**

The following example shows how to create a new Set from an [array](https://www.javascripttutorial.net/javascript-array/).

let chars = new Set(['a', 'a', 'b', 'c', 'c']);

Code language: JavaScript (javascript)

All elements in the set must be unique therefore the  chars only contains 3 distinct elements a, b and c.

console.log(chars);

Code language: JavaScript (javascript)

Output:

Set { 'a', 'b', 'c' }

Code language: JavaScript (javascript)

When you use the  typeof operator to the chars, it returns object.

console.log(typeof(chars));

Code language: JavaScript (javascript)

Output:

object

The chars set is an instance of the Set type so the following statement returns true.

let result = chars instanceof Set;

console.log(result);

Code language: JavaScript (javascript)

### **Get the size of a Set**

To get the number of elements that the set holds, you use the size property of the Set object:

let size = chars.size;

console.log(size);*// 3*

Code language: JavaScript (javascript)

### **Add elements to a Set**

To add an element to the set, you use the add() method:

chars.add('d');

console.log(chars);

Code language: JavaScript (javascript)

Output:

Set { 'a', 'b', 'c', 'd' }

Code language: JavaScript (javascript)

Since the add() method is chainable, you can add multiple items to a set using a chain statement:

chars.add('e')

.add('f');

Code language: JavaScript (javascript)

### **Check if a value is in the Set**

To check if a set has a specific element, you use the has() method. The has() method returns true if the set contains the element, otherwise, it returns false. Since the  chars set contains 'a', the following statement returns true:

let exist = chars.has('a');

console.log(exist);*// true*

Code language: JavaScript (javascript)

The following statement returns false because the  chars set does not contain the 'z' value.

exist = chars.has('z');

console.log(exist); *// false*

Code language: JavaScript (javascript)

### **Remove elements from a set**

To delete a specified element from a set, you use the delete() method. The following statement deletes the 'f' value from the  chars set.

chars.delete('f');

console.log(chars); *// Set {"a", "b", "c", "d", "e"}*

Code language: JavaScript (javascript)

Output:

Set { 'a', 'b', 'c', 'd', 'e' }

Code language: JavaScript (javascript)

The delete() method returns true indicating that the element has been removed successfully. To delete all elements of a set, you use the clear() method:

chars.clear();

console.log(chars); *// Set{}*

Code language: JavaScript (javascript)

### **Looping the elements of a JavaScript Set**

A Set object maintains the insertion order of its elements, therefore, when you iterate over its elements, the order of the elements is the same as the inserted order. Suppose you have a set of user roles as follows.

let roles = new Set();

roles.add('admin')

.add('editor')

.add('subscriber');

Code language: JavaScript (javascript)

The following example uses the [for…of loop](https://www.javascripttutorial.net/es6/javascript-for-of/) to iterate over the chars set.

for (let role of roles) {

console.log(role);

}

Code language: JavaScript (javascript)

Output:

admin

editor

subscriber

The Set also provides the keys(), values(), and entries() methods like the [Map](https://www.javascripttutorial.net/es6/javascript-map/). However, keys and values in the Set are identical. For example:

for (let [key, value] of roles.entries()) {

console.log(key === value);

}

Code language: JavaScript (javascript)

Output

true

true

true

Code language: JavaScript (javascript)

### **Invoke a callback function on each element of a set**

If you want to invoke a [callback](https://www.javascripttutorial.net/javascript-callback/) on every element of a set, you can use the forEach() method.

roles.forEach(role => console.log(role.toUpperCase()));

Code language: JavaScript (javascript)

## **WeakSets**

A WeakSet is similar to a Set except that it contains only objects. Since objects in a WeakSet may be automatically garbage-collected, a WeakSet does not have size property. Like a WeakMap, you cannot iterate elements of a WeakSet, therefore, you will find that WeakSet is rarely used in practice. In fact, you only use a WeakSet to check if a specified value is in the set. Here is an example:

let computer = {type: 'laptop'};

let server = {type: 'server'};

let equipment = new WeakSet([computer, server]);

if (equipment.has(server)) {

console.log('We have a server');

}

Code language: JavaScript (javascript)

Output

We have a server

In this tutorial, you have learned about the JavaScript Set object and how to manipulate its elements.

# JavaScript Array.of()

**Summary**: in this tutorial, you will learn how to improve array construction using the JavaScript Array.of() method in ES6.

## **Introduction to the JavaScript Array.of() method**

In ES5, when you pass a number to the [Array](https://www.javascripttutorial.net/javascript-array/) constructor, JavaScript creates an array whose length equals the number. For example:

let numbers = new Array(2);

console.log(numbers.length); *// 2*

console.log(numbers[0]); *// undefined*

Code language: JavaScript (javascript)

However, when you pass to the Array constructor a value that is not a number, JavaScript creates an array that contains one element with that value. For example:

numbers = new Array("2");

console.log(numbers.length); *// 1*

console.log(numbers[0]); *// "2"*

Code language: JavaScript (javascript)

This behavior is sometimes confusing and error-prone because you may not know the type of data that you pass to the Array constructor.

ES6 introduces the Array.of() method to solve this problem.

The Array.of() method is similar to the Array constructor except the Array.of() method does not treat a single numeric value special.

In other words, the Array.of() method always creates an array that contains the values that you pass to it regardless of the types or the number of arguments.

The following shows the syntax of the Array.of() method:

Array.of(element0[, element1[, ...[, elementN]]])

Code language: CSS (css)

## **JavaScript Array.of() examples**

See the following example:

let numbers = Array.of(3);

console.log(numbers.length); *// 1*

console.log(numbers[0]); *// 3*

Code language: JavaScript (javascript)

In this example, we passed the number 3 to the Array.of() method. The Array.of() method creates an array of one number.

Consider the following example:

let chars = Array.of('A', 'B', 'C');

console.log(chars.length); *// 3*

console.log(chars); *// ['A','B','C']*

Code language: JavaScript (javascript)

In this example, we created an array of three strings by passing 'A', 'B', and 'C' to the Array.of() method. The size of the array is 3.

## **JavaScript Array.of() polyfill**

If you execute the JavaScript in the environment that doesn’t support the  Array.of() method, you can use the following polyfill:

if (!Array.of) {

Array.of = function() {

return Array.prototype.slice.call(arguments);

};

}

Code language: JavaScript (javascript)

In this tutorial, you have learned how to improve array construction using the JavaScript Array.of() method in ES6.

# JavaScript Array.from()

**Summary**: in this tutorial, you will learn about the JavaScript Array.from() method that creates a new array from an array-like or iterable object.

## **Introduction to JavaScript Array Array.from() method**

To create an [array](https://www.javascripttutorial.net/javascript-array/) from an array-like object in ES5, you iterate over all array elements and add each of them to an intermediate array like this:

function arrayFromArgs() {

var results = [];

for (var i = 0; i < arguments.length; i++) {

results.push(arguments[i]);

}

return results;

}

var fruits = arrayFromArgs('Apple', 'Orange', 'Banana');

console.log(fruits);

Code language: JavaScript (javascript)

Output:

[ 'Apple', 'Orange', 'Banana' ]

Code language: JSON / JSON with Comments (json)

To make it more concise, you can use the [slice()](https://www.javascripttutorial.net/javascript-array-slice/) method of the Array.prototype as follows:

function arrayFromArgs() {

return Array.prototype.slice.call(arguments);

}

var fruits = arrayFromArgs('Apple', 'Orange', 'Banana');

console.log(fruits);

Code language: JavaScript (javascript)

ES6 introduces the Array.from() method that creates a new instance of the Array from an array-like or [iterable object](https://www.javascripttutorial.net/es6/javascript-iterator/). The following illustrates the syntax of the Array.from() method:

Array.from(target [, mapFn[, thisArg]])

Code language: CSS (css)

In this syntax:

* target is an array-like or iterable object to convert to an array.
* mapFn is the map function to call on every element of the array
* thisArg is the this value when executing the mapFn function.

The Array.from() returns a new instance of Array that contains all elements of the target object.

## **JavaScript Array.from() method examples**

Let’s take some examples of using the Array.from() method.

### **A) Create an array from an array-like object**

The following example uses the Array.from() method to create a new array from the arguments object of a function:

function arrayFromArgs() {

return Array.from(arguments);

}

console.log(arrayFromArgs(1, 'A'));

Code language: JavaScript (javascript)

Output:

[ 1, 'A' ]

Code language: JSON / JSON with Comments (json)

In this example, we create an array from arguments of the arrayFromArgs() function and return the array.

### **B) JavaScript Array Array.from() with a mapping function**

The Array.from() method accepts a [callback function](https://www.javascripttutorial.net/javascript-callback/) that allows you to execute the mapping function on every element of the array which is being created. See the following example:

function addOne() {

return Array.from(arguments, x => x + 1);

}

console.log(addOne(1, 2, 3));

Code language: JavaScript (javascript)

Output:

[ 2, 3, 4 ]

Code language: JSON / JSON with Comments (json)

In this example, we increased each argument of the addOne() function by one and add the result to the new array.

### **C) JavaScript Array Array.from() with a this value**

If the mapping function belongs to an object, you can optionally pass the third argument to the Array.from() method. The object will represent the this value inside the mapping function. Consider this example:

let doubler = {

factor: 2,

double(x) {

return x \* this.factor;

}

}

let scores = [5, 6, 7];

let newScores = Array.from(scores, doubler.double, doubler);

console.log(newScores);

Code language: JavaScript (javascript)

Output:

[ 10, 12, 14 ]

Code language: JSON / JSON with Comments (json)

### **D) Create an array from an iterable object**

Since the Array.from() method also works on an [iterable object](https://www.javascripttutorial.net/es6/javascript-generators/),  you can use it to create an array from any object that has a [symbol.iterator] property. For example:

let even = {

\*[Symbol.iterator]() {

for (let i = 0; i < 10; i += 2) {

yield i;

}

}

};

let evenNumbers = Array.from(even);

console.log(evenNumbers);

Code language: JavaScript (javascript)

Output:

[0, 2, 4, 6, 8]

Code language: JSON / JSON with Comments (json)

In this example:

* First, define the even object with the [System.iterator] that returns even numbers from 0 to 10.
* Then, use the Array.from() method to create a new array of even numbers from the even object.

In this tutorial, you have learned how to use the JavaScript Array Array.from() method to create an array from an array-like or iterable object.

# JavaScript Array find() Method

**Summary**: in this tutorial, you will learn how to use the JavaScript find() method to search for the first element in an array, which satisfies a test.

## **Introduction to the Array find() method**

In ES5, to find an element in an [array](https://www.javascripttutorial.net/javascript-array/), you use the [indexOf()](https://www.javascripttutorial.net/javascript-array-indexof/) or [lastIndexOf()](https://www.javascripttutorial.net/javascript-array-indexof/) methods. However, these methods are quite limited because they return the index of the first matching element only.

ES6 introduced a new method called find() added to the Array.prototype object.

The find() method returns the first element in an array that satisfies a provided function.

The following shows the syntax of the find() method:

find(callback(element[, index[, array]])[, thisArg])

Code language: CSS (css)

### **Arguments**

The find() accepts two arguments: a callback function and an optional value to use for the this inside the callback function.

### **1) callback**

The callback is a function that executes on each element of the array. It takes three arguments:

* element is the current element.
* index the index of the current element.
* array the array that the find() was called upon.

### **2) thisArg**

The thisArg is the object used as this inside the callback.

### **Return value**

The find() executes the callback function for each element in the array until the callback returns a truthy value.

If the callback returns a truthy value, the find() immediately returns the element and stop searching. Otherwise, it returns undefined.

If you want to find the index of the found element, you can use the [findIndex()](https://www.javascripttutorial.net/es6/javascript-array-findindex/) method.

## **JavaScript find() examples**

The following example uses the find() method to search for the first even number in an array of numbers:

let numbers = [1, 2, 3, 4, 5];

console.log(numbers.find(e => e % 2 == 0));

Code language: JavaScript (javascript)

Output:

2

Suppose that we have a list of customer objects with name and credit properties as follows:

let customers = [{

name: 'ABC Inc',

credit: 100

}, {

name: 'ACME Corp',

credit: 200

}, {

name: 'IoT AG',

credit: 300

}];

Code language: JavaScript (javascript)

The following code uses the find() method to find the first customer whose credit is greater than 100.

console.log(customers.find(c => c.credit > 100));

Code language: JavaScript (javascript)

Output:

{ name: 'ACME Corp', credit: 200 }

Code language: CSS (css)

In this tutorial, you have learned how to use the JavaScript Array’s find() method to search for the first occurrence of an element that satisfies a test.

# JavaScript Array findIndex() Method

**Summary**: in this tutorial, you will learn how to use the [Array](https://www.javascripttutorial.net/javascript-array/) findIndex() method to find the first element that satisfies a given test.

## **Introduction to the JavaScript Array findIndex() Method**

ES6 added a new method called findIndex() to the Array.prototype, which allows you to find the first element in an array that satisfies a provided testing function.

The findIndex() method returns the index of the element that satisfies a testing function or -1 if no element passed the test.

The following illustrates the syntax of the findIndex() method:

findIndex(testFn(element[, index[, array]])[, thisArg])

Code language: CSS (css)

The findIndex() takes two arguments:

### **1) testFn**

The testFn is a function to execute on each element in the array until the function returns true, indicating that the element has been found.

The testFn takes three arguments:

* element is the current element in the array being processed.
* index is the index of the current element being processed.
* array is the array that the findIndex() was called upon.

### **2) thisArg**

The thisArg is an optional object to be used this when executing the callback. If you omit the thisArg argument, the findIndex() function uses undefined.

The findIndex() executes the testFn on every element in the array until it finds the one where testFn returns a truthy value, which is a value that coerces to true.

Once the findIndex() finds such an element, it immediately returns the element’s index.

## **JavaScript Array findIndex() examples**

Let’s take some examples of using the JavaScript Array findIndex() method.

### **1) Using the Array findIndex() method with a simple array example**

The following example returns the index of the first occurrence of the number 7 in the ranks array:

let ranks = [1, 5, 7, 8, 10, 7];

let index = ranks.findIndex(rank => rank === 7);

console.log(index);

Code language: JavaScript (javascript)

Output:

2

### **2) Using the Array findIndex() method with a more complex condition**

This example uses the findIndex() method to get the index of the first occurrence of the number 7 after the index 2 in the ranks array:

let ranks = [1, 5, 7, 8, 10, 7];

let index = ranks.findIndex(

(rank, index) => rank === 7 && index > 2

);

console.log(index);

Code language: JavaScript (javascript)

Output:

5

### **3) Using the Array findIndex() method with an array of objects**

The following example uses the Array findIndex() method to find the index of the first product whose price is greater than 1000:

const products = [

{ name: 'Phone', price: 999 },

{ name: 'Computer', price: 1999 },

{ name: 'Tablet', price: 995 },

];

const index = products.findIndex(product => product.price > 1000);

console.log(index); *// 1*

Code language: JavaScript (javascript)

In this tutorial, you have learned how to use the JavaScript Array findIndex() method to find the first element that satisfies a given test.

# JavaScript Object.assign()

**Summary**: in this tutorial, you will learn how to use the JavaScript Object.assign() method in ES6.

The following shows the syntax of the Object.assign() method:

Object.assign(target, ...sources)

Code language: CSS (css)

The Object.assign() copies all enumerable and own [properties](https://www.javascripttutorial.net/javascript-object-properties/) from the source objects to the target object. It returns the target object.

The Object.assign() invokes the getters on the source objects and setters on the target. It assigns properties only, not copying or defining new properties.

## **Using JavaScript Object.assign() to clone an object**

The following example uses the Object.assign() method to [clone an object](https://www.javascripttutorial.net/object/3-ways-to-copy-objects-in-javascript/).

let widget = {

color: 'red'

};

let clonedWidget = Object.assign({}, widget);

console.log(clonedWidget);

Code language: JavaScript (javascript)

Output

{ color: 'red' }

Code language: CSS (css)

Note that the Object.assign() only carries a shallow clone, not a deep clone.

## **Using JavaScript Object.assign() to merge objects**

The Object.assign() can merge source objects into a target object which has properties consisting of all the properties of the source objects. For example:

let box = {

height: 10,

width: 20

};

let style = {

color: 'Red',

borderStyle: 'solid'

};

let styleBox = Object.assign({}, box, style);

console.log(styleBox);

Code language: JavaScript (javascript)

Output:

{

height: 10,

width: 20,

color: 'Red',

borderStyle: 'solid'

}

Code language: CSS (css)

If the source objects have the same property, the property of the later object overwrites the earlier one:

let box = {

height: 10,

width: 20,

color: 'Red'

};

let style = {

color: 'Blue',

borderStyle: 'solid'

};

let styleBox = Object.assign({}, box, style);

console.log(styleBox);

Code language: JavaScript (javascript)

Output:

{

height: 10,

width: 20,

color: 'Blue',

borderStyle: 'solid'

}

Code language: CSS (css)

## **Summary**

* Object.assign() assigns enumerable and own properties from a source object to a target object.
* Object.assign() can be used to [clone an object](https://www.javascripttutorial.net/object/3-ways-to-copy-objects-in-javascript/) or [merge objects](https://www.javascripttutorial.net/object/javascript-merge-objects/).

# JavaScript Object.is()

**Summary**: in this tutorial, you will learn about the JavaScript Object.is() to check if two values are the same.

The Object.is() behaves like the === operator with two differences:

* -0 and +0
* NaN

## **Negative zero**

The === operator treats -0 and +0 are the same value:

let amount = +0,

volume = -0;

console.log(volume === amount);

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

However, the Object.is() treats +0 and -0 as different values. For example:

let amount = +0,

volume = -0;

console.log(Object.is(amount, volume));

Code language: JavaScript (javascript)

Output

false

Code language: JavaScript (javascript)

## **NaN**

The === operator considers NaN and NaN are different values. The NaN is the only number that does not equal itself. For example:

let quantity = NaN;

console.log(quantity === quantity);

Code language: JavaScript (javascript)

Output:

false

However, Object.is() treats NaN as the same value:

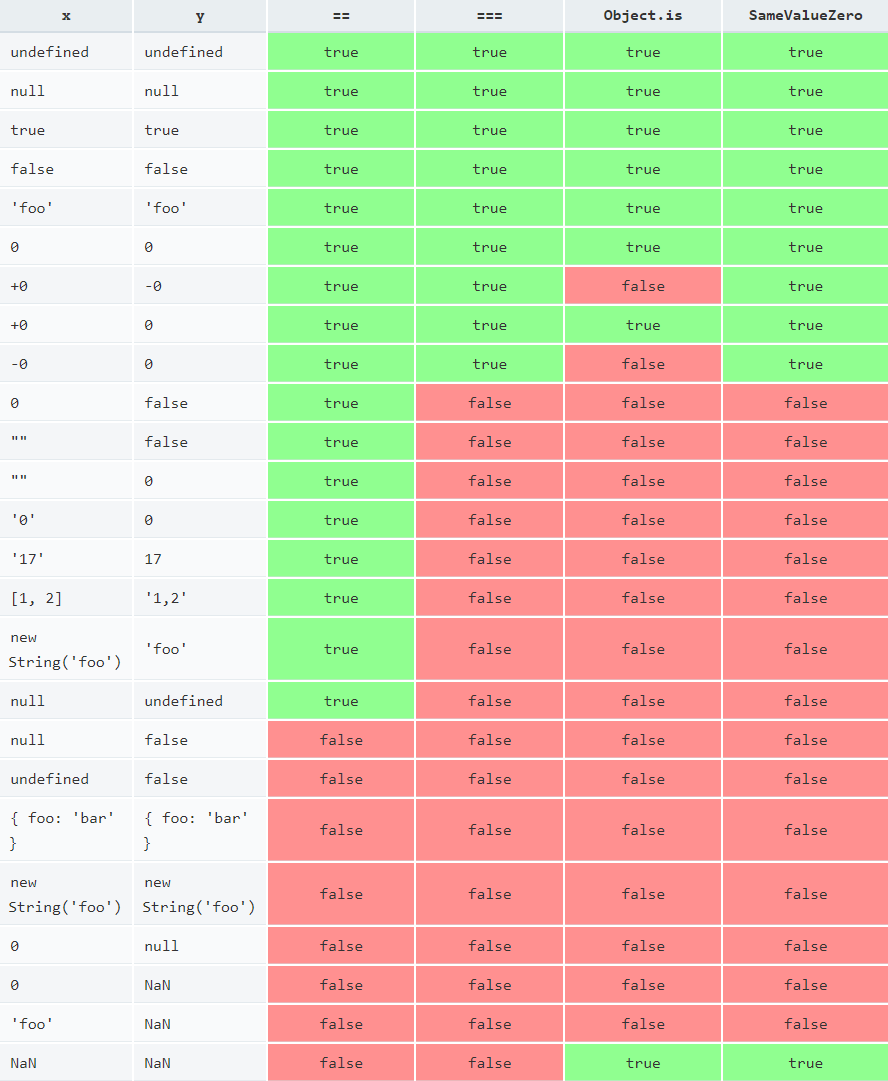
let quantity = NaN;

console.log(Object.is(quantity, quantity));

Output:

true

See the following sameness comparison table for reference:



# JavaScript String startsWith

**Summary**: in this tutorial, you will learn how to use the JavaScript String startsWith() method to check if a string begins with the characters of another string.

The startsWith() returns true if a [string](https://www.javascripttutorial.net/javascript-string/) begins with the characters of a specified string; otherwise false.

The following shows the syntax of the startsWith() method:

String.startsWith(searchString [,position])

Code language: CSS (css)

### **Arguments**

* searchString is the characters to be searched for at the start of this string.
* position is an optional parameter that determines the start position to search for the searchString. It defaults to 0.

## **JavaScript String startsWith() examples**

Suppose that you have a string called title as follows:

const title = 'Jack and Jill Went Up the Hill';

Code language: JavaScript (javascript)

The following example uses the startsWith() method to check if the title starts with the string ‘Jack’:

console.log(title.startsWith('Jack'));

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

The startsWith() method matches characters case-sensitively, so the following statement returns false:

title.startsWith('Jack');

Code language: JavaScript (javascript)

This example uses the startsWith method with the second parameter that determines the begin position to start searching:

console.log(title.startsWith('Jill', 9));

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

Put it all together:

const title = 'Jack and Jill Went Up the Hill';

console.log(title.startsWith('Jack'));

console.log(title.startsWith('jack'));

console.log(title.startsWith('Jill', 9));

Code language: JavaScript (javascript)

Output:

true

false

true

Code language: JavaScript (javascript)

# JavaScript String endsWith

**Summary**: in this tutorial, you will learn how to use the JavaScript String endsWith() method to check if a string ends with the characters of another string.

The endsWith() returns true if a [string](https://www.javascripttutorial.net/javascript-string/) ends with the characters of a specified string; otherwise false.

Here is the syntax of the endsWith() method:

String.endsWith(searchString [,length])

Code language: CSS (css)

### **Arguments**

* searchString is the characters to be searched for at the end of the string.
* length is an optional parameter that determines the length of the string to search. It defaults to the length of the string.

Note that to check if a string begins with the characters of a specified string, you use the [startsWith()](https://www.javascripttutorial.net/es6/javascript-string-startswith/) method.

## **JavaScript String endsWith() examples**

Suppose that you have a string called title:

const title = 'Jack and Jill Went Up the Hill';

Code language: JavaScript (javascript)

The following example uses the endsWith() method to check if the title ends with the string 'Hill':

console.log(title.endsWith('Hill'));

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

The endsWith() method matches characters case-sensitively, therefore, the following example returns false:

title.endsWith('hill');

Code language: JavaScript (javascript)

The following example uses the endsWith() method with the second parameter that determines the length of the string to search:

console.log(title.endsWith('Up', 21));

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

Put it all together:

const title = 'Jack and Jill Went Up the Hill';

console.log(title.endsWith('Hill'));

console.log(title.endsWith('hill'));

console.log(title.endsWith('Up', 21));

Code language: JavaScript (javascript)

Output:

true

false

true

Code language: JavaScript (javascript)

# JavaScript String includes() Method

**Summary**: in this tutorial, you will learn how to use the JavaScript String includes() method to check if a string contains another string.

## **Introduction to JavaScript String includes() method**

The includes() method determines whether a string contains another string:

string.includes(searchString [,position])

Code language: CSS (css)

The includes() method returns true if the searchString found in the string; otherwise false.

The optional position parameter specifies the position within the string at which to begin searching for the searchString. The position defaults to 0.

The includes() matches string case-sensitively.

## **JavaScript String includes() examples**

This example uses the includes() method to check if the string @ is in the string 'admin@example.com':

let email = 'admin@example.com';

console.log(email.includes('@'));

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

The following example checks if the str contains the Script:

let str = 'JavaScript String';

console.log(str.includes('Script'));

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

As mentioned earlier, the includes() matches the string case-sensitively, therefore, the following example returns false:

let str = 'JavaScript String';

console.log(str.includes('script'));

Code language: JavaScript (javascript)

Output:

false

Code language: JavaScript (javascript)

The following example uses the includes() method with the second parameter:

let str = 'JavaScript String';

console.log(str.includes('Script', 5));

Code language: JavaScript (javascript)

Output:

false

Code language: JavaScript (javascript)

In this tutorial, you have learned how to use the JavaScript String includes() method to determine whether a string contains another string.

# JavaScript Proxy

**Summary**: in this tutorial, you will learn about the JavaScript Proxy object in ES6.

## **What is a JavaScript Proxy object**

A JavaScript Proxy is an [object](https://www.javascripttutorial.net/javascript-objects/) that wraps another object (target) and intercepts the fundamental operations of the target object.

The fundamental operations can be the property lookup, assignment, enumeration, and function invocations, etc.

## **Creating a proxy object**

To create a new proxy object, you use the following syntax:

let proxy = new Proxy(target, handler);

Code language: JavaScript (javascript)

In this syntax:

* target – is an object to wrap.
* handler – is an object that contains methods to control the behaviors of the target. The methods inside the handler object are called traps.

## **A simple proxy example**

First, define an object called user:

const user = {

firstName: 'John',

lastName: 'Doe',

email: 'john.doe@example.com',

}

Code language: JavaScript (javascript)

Second, define a handler object:

const handler = {

get(target, property) {

console.log(`Property ${property} has been read.`);

return target[property];

}

}

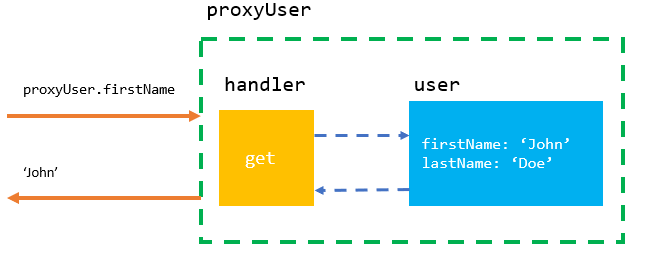
Code language: JavaScript (javascript)

Third, create a proxy object:

const proxyUser = new Proxy(user, handler);

Code language: JavaScript (javascript)

The proxyUser object uses the user object to store data. The proxyUser can access all properties of the user object.



Fourth, access the firstName and lastName properties of the user object via the proxyUser object:

console.log(proxyUser.firstName);

console.log(proxyUser.lastName);

Code language: CSS (css)

Output:

Property firstName has been read.

John

Property lastName has been read.

Doe

When you access a property of the user object via the proxyUser object, the get() method in the handler object is called.

Fifth, if you modify the original object user, the change is reflected in the proxyUser:

user.firstName = 'Jane';

console.log(proxyUser.firstName);

Code language: JavaScript (javascript)

Output:

Property firstName has been read.

Jane

Similarly, a change in the proxyUser object will be reflected in the original object (user):

proxyUser.lastName = 'William';

console.log(user.lastName);

Code language: JavaScript (javascript)

Output:

William

## **Proxy Traps**

### **The get() trap**

The get() trap is fired when a property of the target object is accessed via the proxy object.

In the previous example, a message is printed out when a property of the user object is accessed by the proxyUser object.

Generally, you can develop a custom logic in the get() trap when a property is accessed.

For example, you can use the get() trap to define computed properties for the target object. The computed properties are properties whose values are calculated based on values of existing properties.

The user object does not have a property fullName, you can use the get() trap to create the fullName property based on the firstName and lastName properties:

const user = {

firstName: 'John',

lastName: 'Doe'

}

const handler = {

get(target, property) {

return property === 'fullName' ?

`${target.firstName} ${target.lastName}` :

target[property];

}

};

const proxyUser = new Proxy(user, handler);

console.log(proxyUser.fullName);

Code language: JavaScript (javascript)

Output:

John Doe

### **The set() trap**

The set() trap controls behavior when a property of the target object is set.

Suppose that the age of user must be greater than 18. To enforce this constraint, you develop a set() trap as follows:

const user = {

firstName: 'John',

lastName: 'Doe',

age: 20

}

const handler = {

set(target, property, value) {

if (property === 'age') {

if (typeof value !== 'number') {

throw new Error('Age must be a number.');

}

if (value < 18) {

throw new Error('The user must be 18 or older.')

}

}

target[property] = value;

}

};

const proxyUser = new Proxy(user, handler);

Code language: JavaScript (javascript)

First, set the age of user to a string:

proxyUser.age = 'foo';

Code language: JavaScript (javascript)

Output:

Error: Age must be a number.

Code language: JavaScript (javascript)

Second, set the age of the user to 16:

proxyUser.age = '16';

Code language: JavaScript (javascript)

Output:

The user must be 18 or older.

Third, set the age of the user to 21:

proxyUser.age = 21;

No error occurred.

## **The apply() trap**

The handler.apply() method is a trap for a function call. Here is the syntax:

let proxy = new Proxy(target, {

apply: function(target, thisArg, args) {

*//...*

}

});

Code language: JavaScript (javascript)

See the following example:

const user = {

firstName: 'John',

lastName: 'Doe'

}

const getFullName = function (user) {

return `${user.firstName} ${user.lastName}`;

}

const getFullNameProxy = new Proxy(getFullName, {

apply(target, thisArg, args) {

return target(...args).toUpperCase();

}

});

console.log(getFullNameProxy(user)); *//*

Code language: JavaScript (javascript)

Output

JOHN DOE

## **More traps**

The following are more traps:

* construct – traps usage of the new operator
* getPrototypeOf – traps an internal call to [[GetPrototypeOf]]
* setPrototypeOf – traps a call to Object.setPrototypeOf
* isExtensible – traps a call to Object.isExtensible
* preventExtensions – traps a call to Object.preventExtensions
* getOwnPropertyDescriptor – traps a call to Object.getOwnPropertyDescriptor

In this tutorial, you have learned about the JavaScript Proxy object used to wrap another object to change the fundamental behaviors of that object.

# JavaScript Reflection

**Summary**: in this tutorial, you will learn about the JavaScript reflection and Reflect API in ES6.

## **What is reflection**

In computer programming, reflection is the ability of a program to manipulate [variables](https://www.javascripttutorial.net/javascript-variables/), properties, and methods of [objects](https://www.javascripttutorial.net/javascript-objects/) at runtime.

Prior to ES6, JavaScript already has reflection features even though they were not officially called that by the community or the specification. For example, methods like Object.keys(), Object.getOwnPropertyDescriptor(), and Array.isArray() are the classic reflection features.

ES6 introduces a new global object called Reflect that allows you to call methods, construct objects, get and set properties, manipulate and extend properties.

The Reflect API is important because it allows you to develop programs and frameworks that are able to handle dynamic code.

## **Reflect API**

Unlike the most global objects, the Reflect is not a constructor. It means that you cannot use Reflect with the new operator or invoke the Reflect as a function.  It is similar to the Math and JSON objects. All the methods of the Reflect object are static.

* Reflect.apply() – call a [function](https://www.javascripttutorial.net/javascript-function/) with specified arguments.
* Reflect.construct() – act like the new operator, but as a function. It is equivalent to calling new target(...args).
* Reflect.defineProperty() – is similar to Object.defineProperty(), but return a Boolean value indicating whether or not the property was successfully defined on the object.
* Reflect.deleteProperty() – behave like the delete operator, but as a function. It’s equivalent to calling the  delete objectName[propertyName].
* Reflect.get() – return the value of a property.
* Reflect.getOwnPropertyDescriptor() – is similar to Object.getOwnPropertyDescriptor(). It returns a property descriptor of a property if the property exists on the object, or undefined otherwise.
* Reflect.getPrototypeOf() – is the same as Object.getPrototypeOf().
* Reflect.has() – work like the in operator, but as a function. It returns a boolean indicating whether an property (either owned or inherited) exists.  
  Reflect.isExtensible() – is the same as Object.isExtensible().
* Reflect.ownKeys() – return an array of the owned property keys (not inherited) of an object.
* Reflect.preventExtensions() – is similar to Object.preventExtensions(). It returns a Boolean.
* Reflect.set() – assign a value to a property and return a Boolean value which is true if the property is set successfully.
* Reflect.setPrototypeOf() – set the [prototype](https://www.javascripttutorial.net/javascript-prototype/) of an object.

Let’s take some examples of using the Reflect API:

### **Creating objects: Reflect.construct()**

The Reflect.construct() method behaves like the new operator, but as a function. It is equivalent to calling the new target(...args) with the possibility to specify a different prototype:

Reflect.construct(target, args [, newTarget])

Code language: CSS (css)

The Reflect.construct() returns the new instance of the target, or the newTarget if specified, initialized by the target as a constructor with the given array-like object args. See the following example:

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

get fullName() {

return `${this.firstName} ${this.lastName}`;

}

};

let args = ['John', 'Doe'];

let john = Reflect.construct(

Person,

args

);

console.log(john instanceof Person);

console.log(john.fullName); *// John Doe*

Code language: JavaScript (javascript)

Output

true

John Doe

Code language: JavaScript (javascript)

In this example:

* First, define a class called Person.
* Second, declare an args array that contains two strings.
* Third, create a new instance of the Person class using the Reflect.construct() method. The john object is an instance of the Person class so it has the fullName property.

### **Calling a function: Reflect.apply()**

Prior to ES6, you call a function with a specified this value and arguments by using the Function.prototype.apply() method. For example:

let result = Function.prototype.apply.call(Math.max, Math, [10, 20, 30]);

console.log(result);

Code language: JavaScript (javascript)

Output:

30

This syntax is quite verbose.

The Reflect.apply() provides the same feature as the Function.prototype.apply() but less verbose and easier to understand:

let result = Reflect.apply(Math.max, Math, [10, 20, 30]);

console.log(result);

Code language: JavaScript (javascript)

Here is the syntax of the Reflect.apply() method:

Reflect.apply(target, thisArg, args)

Code language: JavaScript (javascript)

### **Defining a property: Reflect.defineProperty()**

The Reflect.defineProperty() is like the Object.defineProperty(). However, it returns a Boolean indicating whether or not the property was defined successfully instead of throwing an exception:

Reflect.defineProperty(target, propertyName, propertyDescriptor)

Code language: JavaScript (javascript)

See the following example:

let person = {

name: 'John Doe'

};

if (Reflect.defineProperty(person, 'age', {

writable: true,

configurable: true,

enumerable: false,

value: 25,

})) {

console.log(person.age);

} else {

console.log('Cannot define the age property on the person object.');

}

Code language: JavaScript (javascript)

In this tutorial, you have learned about the JavaScript reflection and the Reflect API that contains a number of reflective methods.