3/22/2022

Nagendra Mirajkar

Zensar AES Practices

JavaScript

Detailed

Table of Contents

[What is JavaScript 18](#_Toc98942369)

[Client-side vs. Server-side JavaScript 18](#_Toc98942370)

[JavaScript History 18](#_Toc98942371)

[JavaScript overview 19](#_Toc98942372)

[JavaScript Code Editors 21](#_Toc98942373)

[Popular JavaScript Code Editors 21](#_Toc98942374)

[Visual Studio Code 21](#_Toc98942375)

[**Download Visual Studio Code** 22](#_Toc98942376)

[**Installing Visual Studio Code** 22](#_Toc98942377)

[**A) Windows** 22](#_Toc98942378)

[**B) macOS** 23](#_Toc98942379)

[**Installing the Live Server extension** 23](#_Toc98942380)

[Web Development Tools 23](#_Toc98942381)

[Google Chrome 24](#_Toc98942382)

[Firefox and Edge 25](#_Toc98942383)

[Safari 25](#_Toc98942384)

[JavaScript Hello World Example 26](#_Toc98942385)

[Embed JavaScript code in an HTML page 26](#_Toc98942386)

[Include an external JavaScript file 27](#_Toc98942387)

[The async and defer attributes 28](#_Toc98942388)

[Summary 29](#_Toc98942389)

[JavaScript Syntax 29](#_Toc98942390)

[Whitespace 29](#_Toc98942391)

[Statements 30](#_Toc98942392)

[**Blocks** 30](#_Toc98942393)

[Identifiers 30](#_Toc98942394)

[Comments 30](#_Toc98942395)

[**Single-line comments** 30](#_Toc98942396)

[**Block comments** 31](#_Toc98942397)

[Expressions 31](#_Toc98942398)

[Keywords & Reserved words 31](#_Toc98942399)

[Summary 32](#_Toc98942400)

[JavaScript Variables 32](#_Toc98942401)

[Declare a variable 32](#_Toc98942402)

[Initialize a variable 33](#_Toc98942403)

[Change a variable 34](#_Toc98942404)

[Undefined vs. undeclared variables 34](#_Toc98942405)

[Constants 34](#_Toc98942406)

[Summary 35](#_Toc98942407)

[JavaScript Data Types 35](#_Toc98942408)

[The undefined type 36](#_Toc98942409)

[The null type 36](#_Toc98942410)

[The number type 36](#_Toc98942411)

[**NaN** 37](#_Toc98942412)

[The string type 37](#_Toc98942413)

[The boolean type 38](#_Toc98942414)

[The symbol type 39](#_Toc98942415)

[The bigint type 39](#_Toc98942416)

[The object type 39](#_Toc98942417)

[Summary 41](#_Toc98942418)

[JavaScript Numbers 41](#_Toc98942419)

[Introduction to the JavaScript Number 41](#_Toc98942420)

[Integer numbers 41](#_Toc98942421)

[**Octal numbers** 41](#_Toc98942422)

[**Hexadecimal numbers** 42](#_Toc98942423)

[Floating-point numbers 42](#_Toc98942424)

[Big Integers 43](#_Toc98942425)

[Summary 44](#_Toc98942426)

[JavaScript Numeric Separator 44](#_Toc98942427)

[Introduction to the JavaScript numeric separator 44](#_Toc98942428)

[Summary 45](#_Toc98942429)

[A Quick Look at Octal and Binary Literals in ES6 45](#_Toc98942430)

[Octal literals 45](#_Toc98942431)

[Binary literals 46](#_Toc98942432)

[Summary 46](#_Toc98942433)

[JavaScript Boolean 46](#_Toc98942434)

[JavaScript boolean primitive type 46](#_Toc98942435)

[JavaScript Boolean object 47](#_Toc98942436)

[JavaScript boolean vs. Boolean 47](#_Toc98942437)

[JavaScript string 48](#_Toc98942438)

[Introduction to the JavaScript strings 49](#_Toc98942439)

[Escaping special characters 49](#_Toc98942440)

[Getting the length of the string 50](#_Toc98942441)

[Accessing characters 50](#_Toc98942442)

[Concatenating strings via + operator 50](#_Toc98942443)

[Converting values to string 50](#_Toc98942444)

[Comparing strings 51](#_Toc98942445)

[Summary 51](#_Toc98942446)

[JavaScript Objects 51](#_Toc98942447)

[Introduction to the JavaScript objects 52](#_Toc98942448)

[Accessing properties 52](#_Toc98942449)

[**1) The dot notation (.)** 52](#_Toc98942450)

[**2) Array-like notation ( [])** 53](#_Toc98942451)

[Modifying the value of a property 54](#_Toc98942452)

[Adding a new property to an object 54](#_Toc98942453)

[Deleting a property of an object 54](#_Toc98942454)

[Checking if a property exists 54](#_Toc98942455)

[Summary 55](#_Toc98942456)

[JavaScript Primitive vs. Reference Values 55](#_Toc98942457)

[Stack and heap memory 55](#_Toc98942458)

[Dynamic properties 56](#_Toc98942459)

[Copying values 57](#_Toc98942460)

[Summary 59](#_Toc98942461)

[JavaScript Arrays 59](#_Toc98942462)

[Introduction to JavaScript arrays 59](#_Toc98942463)

[Creating JavaScript arrays 60](#_Toc98942464)

[Accessing JavaScript array elements 61](#_Toc98942465)

[Getting the array size 61](#_Toc98942466)

[Basic operations on arrays 61](#_Toc98942467)

[**1) Adding an element to the end of an array** 62](#_Toc98942468)

[**2) Adding an element to the beginning of an array** 62](#_Toc98942469)

[**3) Removing an element from the end of an array** 62](#_Toc98942470)

[**4) Removing an element from the beginning of an array** 62](#_Toc98942471)

[**5) Finding an index of an element in the array** 63](#_Toc98942472)

[**6) Check if an value is an array** 63](#_Toc98942473)

[Summary 63](#_Toc98942474)

[JavaScript Arithmetic Operators 63](#_Toc98942475)

[Introduction to the JavaScript Arithmetic Operators 63](#_Toc98942476)

[Addition operator (+) 64](#_Toc98942477)

[Subtraction operator (-) 65](#_Toc98942478)

[Multiplication operator (\*) 66](#_Toc98942479)

[Divide operator (/) 67](#_Toc98942480)

[Using JavaScript arithmetic operators with objects 67](#_Toc98942481)

[Summary 68](#_Toc98942482)

[JavaScript Remainder Operator 68](#_Toc98942483)

[Introduction to the JavaScript remainder operator 69](#_Toc98942484)

[JavaScript remainder operator examples 69](#_Toc98942485)

[**1) Using the remainder operator with positive dividend example** 69](#_Toc98942486)

[**2) Using the remainder operator with negative dividend example** 69](#_Toc98942487)

[**3) Using the remainder operator special values** 69](#_Toc98942488)

[Using the remainder operator to check if a number is an odd number 70](#_Toc98942489)

[Remainder vs Modulo operator 71](#_Toc98942490)

[Summary 71](#_Toc98942491)

[JavaScript Assignment Operators 71](#_Toc98942492)

[Introduction to JavaScript assignment operators 72](#_Toc98942493)

[Chaining JavaScript assignment operators 73](#_Toc98942494)

[Summary 73](#_Toc98942495)

[JavaScript Unary Operators Explained 73](#_Toc98942496)

[Introduction to the JavaScript unary operators 73](#_Toc98942497)

[Unary plus (+) 74](#_Toc98942498)

[Unary minus (-) 76](#_Toc98942499)

[Increment / Decrement operators 76](#_Toc98942500)

[Summary 78](#_Toc98942501)

[JavaScript Comparison Operators 78](#_Toc98942502)

[Introduction to JavaScript comparison operators 78](#_Toc98942503)

[**Compare numbers** 79](#_Toc98942504)

[**Compare strings** 79](#_Toc98942505)

[**Compare a number with a value of another type** 80](#_Toc98942506)

[**Compare an object with a non-object** 80](#_Toc98942507)

[**Compare a Boolean with another value** 81](#_Toc98942508)

[**Compare null and undefined** 81](#_Toc98942509)

[**Compare NaN with other values** 81](#_Toc98942510)

[Strict equal (===) and not strict equal (!==) 82](#_Toc98942511)

[An Introduction to JavaScript Logical Operators 82](#_Toc98942512)

[1) The Logical NOT operator (!) 83](#_Toc98942513)

[**Double-negation (!!)** 84](#_Toc98942514)

[2) The Logical AND operator (&&) 84](#_Toc98942515)

[**Short-circuit evaluation** 85](#_Toc98942516)

[**A Chain of && operators** 86](#_Toc98942517)

[3) The Logical OR operator (||) 86](#_Toc98942518)

[**The || operator is also short-circuited** 87](#_Toc98942519)

[**A chain of || operators** 87](#_Toc98942520)

[Logical operator precedence 88](#_Toc98942521)

[Summary 88](#_Toc98942522)

[JavaScript Logical Assignment Operators 88](#_Toc98942523)

[The Logical OR assignment operator 88](#_Toc98942524)

[The Logical AND assignment operator 89](#_Toc98942525)

[The nullish coalescing assignment operator 90](#_Toc98942526)

[Summary 90](#_Toc98942527)

[JavaScript Nullish Coalescing Operator 91](#_Toc98942528)

[Introduction to the JavaScript nullish coalescing operator 91](#_Toc98942529)

[Why nullish coalescing operator 91](#_Toc98942530)

[The nullish coalescing operator is short-circuited 92](#_Toc98942531)

[Chaining with the AND or OR operator 92](#_Toc98942532)

[Summary 92](#_Toc98942533)

[JavaScript Exponentiation Operator 93](#_Toc98942534)

[Introduction to the JavaScript exponentiation operator 93](#_Toc98942535)

[Summary 94](#_Toc98942536)

[JavaScript if 94](#_Toc98942537)

[Introduction to the JavaScript if statement 94](#_Toc98942538)

[JavaScript if statement examples 95](#_Toc98942539)

[Nested if statement 96](#_Toc98942540)

[Summary 96](#_Toc98942541)

[JavaScript if else 97](#_Toc98942542)

[Introduction to the JavaScript if…else statement 97](#_Toc98942543)

[JavaScript if…else statement 97](#_Toc98942544)

[Summary 98](#_Toc98942545)

[JavaScript if else if 98](#_Toc98942546)

[Introduction to the JavaScript if else if statement 98](#_Toc98942547)

[JavaScript if else if examples 99](#_Toc98942548)

[**1) A simple JavaScript if…else…if statement example** 99](#_Toc98942549)

[**2) Using JavaScript if…else…if statement to calculate the body mass index** 100](#_Toc98942550)

[Summary 100](#_Toc98942551)

[JavaScript Ternary Operator 101](#_Toc98942552)

[Introduction to JavaScript ternary operator 101](#_Toc98942553)

[JavaScript ternary operator examples 102](#_Toc98942554)

[**1) Using the JavaScript ternary operator to perform multiple statements** 102](#_Toc98942555)

[**2) Simplifying ternary operator example** 102](#_Toc98942556)

[**3) Using multiple JavaScript ternary operators example** 102](#_Toc98942557)

[Summary 103](#_Toc98942558)

[JavaScript switch case 103](#_Toc98942559)

[Introduction to the JavaScript switch case statement 103](#_Toc98942560)

[JavaScript switch case examples 104](#_Toc98942561)

[**1) Using JavaScript switch statement to get the day of the week** 104](#_Toc98942562)

[**2) Using the JavaScript switch statement to get the day count based of a month** 105](#_Toc98942563)

[Summary 106](#_Toc98942564)

[JavaScript while Loop 106](#_Toc98942565)

[Introduction to the JavaScript while loop statement 106](#_Toc98942566)

[JavaScript while loop example 107](#_Toc98942567)

[Summary 107](#_Toc98942568)

[JavaScript do…while Loop 107](#_Toc98942569)

[Introduction to the JavaScript do…while statement 107](#_Toc98942570)

[JavaScript do while statement examples 108](#_Toc98942571)

[**1) Simple JavaScript do while statement example** 108](#_Toc98942572)

[**2) Using the JavaScript do while statement to make a simple number guessing game** 109](#_Toc98942573)

[Summary 110](#_Toc98942574)

[JavaScript for Loop 110](#_Toc98942575)

[Introduction to the JavaScript for loop statement 110](#_Toc98942576)

[**1) iterator** 110](#_Toc98942577)

[**2) condition** 110](#_Toc98942578)

[**3) iterator** 110](#_Toc98942579)

[JavaScript for loop examples 110](#_Toc98942580)

[**1) A simple JavaScript for loop example** 111](#_Toc98942581)

[**2) Using the JavaScript for loop without the initializer example** 111](#_Toc98942582)

[**3) Using the JavaScript for loop without the condition example** 111](#_Toc98942583)

[**3) Using the JavaScript for loop statement without any expression example** 112](#_Toc98942584)

[**4) Using the JavaScript for loop without the loop body example** 112](#_Toc98942585)

[Summary 112](#_Toc98942586)

[JavaScript break 112](#_Toc98942587)

[The label statement 113](#_Toc98942588)

[Introduction to JavaScript break statement 113](#_Toc98942589)

[Using JavaScript break statement in a for loop 113](#_Toc98942590)

[**Using the break statement to terminate a nested loop** 114](#_Toc98942591)

[Using JavaScript break statement in a while loop 115](#_Toc98942592)

[Using JavaScript break statement in a do…while loop 116](#_Toc98942593)

[Summary 117](#_Toc98942594)

[JavaScript continue 117](#_Toc98942595)

[Introduction to the JavaScript continue statement 117](#_Toc98942596)

[Using the continue statement in a for loop 117](#_Toc98942597)

[Using the continue statement in a while loop 118](#_Toc98942598)

[Using the continue statement with a label example 119](#_Toc98942599)

[Summary 119](#_Toc98942600)

[JavaScript Comma Operator 120](#_Toc98942601)

[Introduction to the JavaScript comma operator 120](#_Toc98942602)

[Summary 121](#_Toc98942603)

[JavaScript Functions 121](#_Toc98942604)

[Introduction to JavaScript functions 121](#_Toc98942605)

[Declare a function 122](#_Toc98942606)

[Calling a function 122](#_Toc98942607)

[**Parameters vs. Arguments** 123](#_Toc98942608)

[Returning a value 123](#_Toc98942609)

[The arguments object 124](#_Toc98942610)

[Function hoisting 125](#_Toc98942611)

[Summary 125](#_Toc98942612)

[JavaScript Functions are First-Class Citizens 126](#_Toc98942613)

[Storing functions in variables 126](#_Toc98942614)

[Passing a function to another function 126](#_Toc98942615)

[Returning functions from functions 127](#_Toc98942616)

[More JavaScript Functions are First-Class Citizens example 129](#_Toc98942617)

[Summary 130](#_Toc98942618)

[JavaScript Anonymous Functions 131](#_Toc98942619)

[Introduction to JavaScript anonymous functions 131](#_Toc98942620)

[Using anonymous functions as arguments 131](#_Toc98942621)

[Immediately invoked function execution 132](#_Toc98942622)

[Arrow functions 132](#_Toc98942623)

[Summary 133](#_Toc98942624)

[Understanding JavaScript Pass-By-Value 133](#_Toc98942625)

[JavaScript pass-by-value or pass-by-reference 133](#_Toc98942626)

[Pass-by-value of primitives values 133](#_Toc98942627)

[Pass-by-value of reference values 134](#_Toc98942628)

[Summary 136](#_Toc98942629)

[JavaScript Recursive Function 136](#_Toc98942630)

[Introduction to the JavaScript recursive functions 136](#_Toc98942631)

[JavaScript recursive function examples 137](#_Toc98942632)

[**1) A simple JavaScript recursive function example** 137](#_Toc98942633)

[**2) Calculate the sum of n natural numbers example** 139](#_Toc98942634)

[Summary 139](#_Toc98942635)

[JavaScript Default Parameters 139](#_Toc98942636)

[Arguments vs. Parameters 140](#_Toc98942637)

[Setting JavaScript default parameters for a function 140](#_Toc98942638)

[More JavaScript default parameter examples 141](#_Toc98942639)

[**1) Passing undefined arguments** 141](#_Toc98942640)

[**2) Evaluating default parameters** 142](#_Toc98942641)

[**3) Using other parameters in default values** 142](#_Toc98942642)

[**Using functions** 143](#_Toc98942643)

[**The arguments object** 143](#_Toc98942644)

[Javascript Object Methods 144](#_Toc98942645)

[Introduction to the JavaScript object methods 144](#_Toc98942646)

[Object method shorthand 145](#_Toc98942647)

[**The this value** 145](#_Toc98942648)

[Summary 146](#_Toc98942649)

[JavaScript Constructor Function 146](#_Toc98942650)

[Introduction to JavaScript constructor functions 146](#_Toc98942651)

[Adding methods to JavaScript constructor functions 148](#_Toc98942652)

[Returning from constructor functions 148](#_Toc98942653)

[Calling a constructor function without the new keyword 148](#_Toc98942654)

[Summary 150](#_Toc98942655)

[JavaScript Prototype 150](#_Toc98942656)

[Introduction to JavaScript prototype 150](#_Toc98942657)

[Defining methods in the JavaScript prototype object 152](#_Toc98942658)

[Defining methods in an individual object 154](#_Toc98942659)

[Getting prototype linkage 155](#_Toc98942660)

[Shadowing 155](#_Toc98942661)

[Summary 156](#_Toc98942662)

[JavaScript Constructor/Prototype Pattern 156](#_Toc98942663)

[Introduction to the JavaScript Constructor / Prototype pattern 156](#_Toc98942664)

[JavaScript Constructor / Prototype example 156](#_Toc98942665)

[Classes in ES6 158](#_Toc98942666)

[Summary 159](#_Toc98942667)

[JavaScript Prototypal Inheritance 159](#_Toc98942668)

[Introduction to JavaScript prototypal inheritance 159](#_Toc98942669)

[JavaScript prototypal inheritance and \_\_proto\_\_ 159](#_Toc98942670)

[A standard way to implement prototypal inheritance in ES5 161](#_Toc98942671)

[Summary 162](#_Toc98942672)

[Demystifying the JavaScript this Keyword 162](#_Toc98942673)

[What is the this keyword 163](#_Toc98942674)

[Global context 163](#_Toc98942675)

[Function context 164](#_Toc98942676)

[**1) Simple function invocation** 164](#_Toc98942677)

[**2) Method invocation** 165](#_Toc98942678)

[**3) Constructor invocation** 166](#_Toc98942679)

[**4) Indirect Invocation** 167](#_Toc98942680)

[Arrow functions 168](#_Toc98942681)

[JavaScript globalThis 168](#_Toc98942682)

[Introduction to the JavaScript globalThis object 169](#_Toc98942683)

[Summary 170](#_Toc98942684)

[JavaScript Object Properties 170](#_Toc98942685)

[Object Property types 170](#_Toc98942686)

[**1) Data properties** 170](#_Toc98942687)

[**2) Accessor properties** 173](#_Toc98942688)

[Define multiple properties: Object.defineProperties() 174](#_Toc98942689)

[JavaScript object property descriptor 174](#_Toc98942690)

[Summary 175](#_Toc98942691)

[JavaScript for…in Loop 175](#_Toc98942692)

[Introduction to JavaScript for...in loop 175](#_Toc98942693)

[The for...in loop & inheritance 176](#_Toc98942694)

[The for...in loop and Array 177](#_Toc98942695)

[Summary 178](#_Toc98942696)

[A Basic Guide to Enumerable Properties of an Object in JavaScript 178](#_Toc98942697)

[Introduction to JavaScript enumerable properties 178](#_Toc98942698)

[Summary 180](#_Toc98942699)

[Understanding Own Properties of an Object in JavaScript 180](#_Toc98942700)

[Summary 181](#_Toc98942701)

[JavaScript Object.values() 181](#_Toc98942702)

[Object.values() vs. for...in 182](#_Toc98942703)

[JavaScript Object.entries() 183](#_Toc98942704)

[Introduction to JavaScript Object.entries() method 183](#_Toc98942705)

[Object.entries() vs. for...in 183](#_Toc98942706)

[JavaScript Object.is() 184](#_Toc98942707)

[Negative zero 184](#_Toc98942708)

[NaN 184](#_Toc98942709)

[JavaScript Factory Functions 186](#_Toc98942710)

[Introduction to the factory functions in JavaScript 187](#_Toc98942711)

[The Object.create() method 189](#_Toc98942712)

[Summary 190](#_Toc98942713)

[JavaScript Object Destructuring 190](#_Toc98942714)

[Introduction to the JavaScript object destructuring assignment 190](#_Toc98942715)

[Setting default values 191](#_Toc98942716)

[Destructuring a null object 192](#_Toc98942717)

[Nested object destructuring 192](#_Toc98942718)

[Destructuring function arguments 193](#_Toc98942719)

[Summary 194](#_Toc98942720)

[JavaScript Optional Chaining Operator 194](#_Toc98942721)

[Introduction to the JavaScript optional chaining operator 194](#_Toc98942722)

[**Stacking the optional chaining operator** 196](#_Toc98942723)

[**Combining with the nullish coalescing operator** 196](#_Toc98942724)

[Using optional chaining operator with function calls 196](#_Toc98942725)

[Summary 198](#_Toc98942726)

[Object Literal Syntax Extensions in ES6 198](#_Toc98942727)

[Object property initializer shorthand 198](#_Toc98942728)

[Computed property name 199](#_Toc98942729)

[Concise method syntax 200](#_Toc98942730)

[JavaScript Class 201](#_Toc98942731)

[Classes prior to ES6 revisited 201](#_Toc98942732)

[ES6 class declaration 202](#_Toc98942733)

[Class vs. Custom type 203](#_Toc98942734)

[Summary 203](#_Toc98942735)

[JavaScript Getters and Setters 203](#_Toc98942736)

[Introduction to the JavaScript getters and setters 204](#_Toc98942737)

[Using getter in an object literal 206](#_Toc98942738)

[Summary 207](#_Toc98942739)

[JavaScript Class Expressions 207](#_Toc98942740)

[Introduction to JavaScript class expressions 207](#_Toc98942741)

[First-class citizen 208](#_Toc98942742)

[Singleton 209](#_Toc98942743)

[Summary 210](#_Toc98942744)

[JavaScript Computed Property 210](#_Toc98942745)

[Introduction to JavaScript Computed Property 210](#_Toc98942746)

[JavaScript Inheritance Using extends & super 211](#_Toc98942747)

[Implementing JavaScript inheritance using extends and super 211](#_Toc98942748)

[Shadowing methods 214](#_Toc98942749)

[Inheriting static members 215](#_Toc98942750)

[Inheriting from built-in types 215](#_Toc98942751)

[Summary 216](#_Toc98942752)

[Introduction to JavaScript new.target Metaproperty 216](#_Toc98942753)

[Introduction to JavaScript new.target 216](#_Toc98942754)

[JavaScript new.target in functions 217](#_Toc98942755)

[JavaScript new.target in constructors 217](#_Toc98942756)

[JavaScript Static Methods 218](#_Toc98942757)

[Introduction to the JavaScript static methods 218](#_Toc98942758)

[JavaScript static methods in ES6 219](#_Toc98942759)

[Calling a static method from the class constructor or an instance method 219](#_Toc98942760)

[Summary 220](#_Toc98942761)

[JavaScript Static Properties 220](#_Toc98942762)

[Introduction to the JavaScript static properties 220](#_Toc98942763)

[Summary 222](#_Toc98942764)

[JavaScript Private Fields 222](#_Toc98942765)

[Introduction to the JavaScript private fields 222](#_Toc98942766)

[Using getter and setter to access private fields 223](#_Toc98942767)

[Private fields and subclasses 223](#_Toc98942768)

[The in operator: check private fields exist 224](#_Toc98942769)

[Static private fields 224](#_Toc98942770)

[Summary 225](#_Toc98942771)

[JavaScript Private Methods 226](#_Toc98942772)

[Introduction to JavaScript private methods 226](#_Toc98942773)

[JavaScript private method examples 227](#_Toc98942774)

[**1) Private instance method example** 227](#_Toc98942775)

[**2) Private static method example** 228](#_Toc98942776)

[**Summary** 229](#_Toc98942777)

[JavaScript instanceof 229](#_Toc98942778)

[Introduction to the JavaScript instanceof operator 229](#_Toc98942779)

[JavaScript instanceof operator example 229](#_Toc98942780)

[ES6 class and instanceof operator 230](#_Toc98942781)

[The instanceof operator and inheritance 231](#_Toc98942782)

[Symbol.hasInstance 231](#_Toc98942783)

[Summary 232](#_Toc98942784)

[JavaScript Function Type 232](#_Toc98942785)

[Introduction to the JavaScript Function type 232](#_Toc98942786)

[**Functions properties** 232](#_Toc98942787)

[**new.target** 232](#_Toc98942788)

[Function methods: apply, call, and bind 233](#_Toc98942789)

[**The apply() and call() methods** 233](#_Toc98942790)

[**The bind() method** 235](#_Toc98942791)

[Summary 236](#_Toc98942792)

[JavaScript call() Method 236](#_Toc98942793)

[Introduction to the JavaScript call() method 236](#_Toc98942794)

[Using the JavaScript call() method to chain constructors for an object 238](#_Toc98942795)

[Using the JavaScript call() method for function borrowing 238](#_Toc98942796)

[JavaScript apply() method 241](#_Toc98942797)

[Introduction to the JavaScript apply() method 241](#_Toc98942798)

[JavaScript apply() method examples 241](#_Toc98942799)

[**1) Simple JavaScript apply() method example** 241](#_Toc98942800)

[**2) Function borrowing** 242](#_Toc98942801)

[**3) Using the apply() method to append an array to another** 243](#_Toc98942802)

[Summary 243](#_Toc98942803)

[JavaScript bind() Method 243](#_Toc98942804)

[Introduction to JavaScript bind() method 244](#_Toc98942805)

[Using JavaScript bind() for function binding 244](#_Toc98942806)

[Using bind() to borrow methods from a different object 245](#_Toc98942807)

[Summary 246](#_Toc98942808)

[JavaScript Closures 246](#_Toc98942809)

[Introduction to JavaScript closures 246](#_Toc98942810)

[**Lexical scoping** 246](#_Toc98942811)

[**JavaScript closures** 247](#_Toc98942812)

[**More JavaScript Closure example** 248](#_Toc98942813)

[JavaScript closures in a loop 249](#_Toc98942814)

[**1) Using the IIFE solution** 249](#_Toc98942815)

[**2) Using let keyword in ES6** 250](#_Toc98942816)

[Summary 250](#_Toc98942817)

[JavaScript Immediately Invoked Function Expression 250](#_Toc98942818)

[Why IIFEs 250](#_Toc98942819)

[Named IIFE 252](#_Toc98942820)

[IIFE starting with a semicolon (;) 252](#_Toc98942821)

[IIFE in actions 253](#_Toc98942822)

[jQuery & IIFE 255](#_Toc98942823)

[Returning Multiple Values from a Function 255](#_Toc98942824)

[Returning multiple values from a function using an array 256](#_Toc98942825)

[Returning multiple values from an function using an object 256](#_Toc98942826)

[Summary 257](#_Toc98942827)

[An Introduction to JavaScript Arrow Functions 257](#_Toc98942828)

[Introduction to JavaScript arrow functions 257](#_Toc98942829)

[**JavaScript arrow functions with multiple parameters** 258](#_Toc98942830)

[**JavaScript arrow functions with a single parameter** 259](#_Toc98942831)

[**JavaScript arrow functions with no parameter** 259](#_Toc98942832)

[Line break between parameter definition and arrow 259](#_Toc98942833)

[Statements & expressions in the arrow function body 260](#_Toc98942834)

[JavaScript arrow functions and object literal 260](#_Toc98942835)

[Arrow function vs. regular function 261](#_Toc98942836)

[**JavaScript arrow functions and this value** 261](#_Toc98942837)

[**JavaScript arrow functions and the arguments object** 262](#_Toc98942838)

[**JavaScript arrow functions and the prototype property** 262](#_Toc98942839)

[Summary 263](#_Toc98942840)

[When You Should Not Use Arrow Functions 263](#_Toc98942841)

[1) Event handlers 263](#_Toc98942842)

[2) Object methods 264](#_Toc98942843)

[3) Prototype methods 265](#_Toc98942844)

[4) Functions that use the arguments object 265](#_Toc98942845)

[Summary 266](#_Toc98942846)

[JavaScript Rest Parameters 266](#_Toc98942847)

[Introduction to JavaScript rest parameters 266](#_Toc98942848)

[More JavaScript rest parameters examples 267](#_Toc98942849)

[JavaScript rest parameters and arrow function 268](#_Toc98942850)

[JavaScript rest parameter in a dynamic function 269](#_Toc98942851)

[JavaScript Callbacks 269](#_Toc98942852)

[What are callbacks 269](#_Toc98942853)

[Synchronous callbacks 272](#_Toc98942854)

[Asynchronous callbacks 272](#_Toc98942855)

[**Handling errors** 274](#_Toc98942856)

[**Nesting callbacks and the Pyramid of Doom** 274](#_Toc98942857)

[Summary 275](#_Toc98942858)

[JavaScript Promises 275](#_Toc98942859)

[Why JavaScript promises 275](#_Toc98942860)

[**Using callbacks to deal with an asynchronous operation** 278](#_Toc98942861)

[Understanding JavaScript Promises 278](#_Toc98942862)

[Creating a promise 279](#_Toc98942863)

[Consuming a Promise: then, catch, finally 280](#_Toc98942864)

[**1) The then() method** 280](#_Toc98942865)

[**2) The catch() method** 282](#_Toc98942866)

[**3) The finally() method** 283](#_Toc98942867)

[A practical JavaScript Promise example 284](#_Toc98942868)

[Summary 286](#_Toc98942869)

[Promise Chaining 286](#_Toc98942870)

[Introduction to the JavaScript promise chaining 286](#_Toc98942871)

[Multiple handlers for a promise 288](#_Toc98942872)

[Returning a Promise 289](#_Toc98942873)

[**Promise chaining syntax** 290](#_Toc98942874)

[JavaScript Promise.all() 292](#_Toc98942875)

[Introduction to the JavaScript Promise.all() method 292](#_Toc98942876)

[JavaScript Promise.all() method examples 293](#_Toc98942877)

[**1) Resolved promises example** 293](#_Toc98942878)

[**2) Rejected promises example** 293](#_Toc98942879)

[Summary 294](#_Toc98942880)

[JavaScript Promise.race() 294](#_Toc98942881)

[Introduction to JavaScript Promise.race() static method 295](#_Toc98942882)

[JavaScript Promise.race() examples 295](#_Toc98942883)

[**1) Simple JavaScript Promise.race() examples** 295](#_Toc98942884)

[**2) Practical JavaScript Promise.race() example** 297](#_Toc98942885)

[Summary 300](#_Toc98942886)

[JavaScript Promise.any() 300](#_Toc98942887)

[Introduction to JavaScript Promise.any() method 300](#_Toc98942888)

[JavaScript Promise.any() examples 301](#_Toc98942889)

[**1) All promises fulfilled example** 301](#_Toc98942890)

[**2) One promise rejected example** 302](#_Toc98942891)

[**3) All promises rejected example** 302](#_Toc98942892)

[When to use the JavaScript Promise.any() method 303](#_Toc98942893)

[**The index.html file** 303](#_Toc98942894)

[**The app.js file** 304](#_Toc98942895)

[Summary 304](#_Toc98942896)

[JavaScript Promise.allSettled() 304](#_Toc98942897)

[Introduction to the Promise.allSettled() method 305](#_Toc98942898)

[JavaScript Promise.allSettled() example 305](#_Toc98942899)

[Summary 306](#_Toc98942900)

[JavaScript Promise finally() 306](#_Toc98942901)

[Introduction to the JavaScript Promise finally() method 306](#_Toc98942902)

[The JavaScript Promise finally() method example 307](#_Toc98942903)

[Summary 308](#_Toc98942904)

[Promise Error Handling 309](#_Toc98942905)

[Normal error 309](#_Toc98942906)

[Errors inside the Promises 310](#_Toc98942907)

[Calling reject() function 311](#_Toc98942908)

[Missing the catch() method 311](#_Toc98942909)

[Summary 312](#_Toc98942910)

[JavaScript async/await 312](#_Toc98942911)

[Introduction to JavaScript async / await keywords 312](#_Toc98942912)

[The async keyword 314](#_Toc98942913)

[The await keyword 315](#_Toc98942914)

[Error handling 316](#_Toc98942915)

[The Essential Guide to JavaScript Iterators 317](#_Toc98942916)

[The for loop issues 317](#_Toc98942917)

[Iteration protocols 317](#_Toc98942918)

[**Iterator protocol** 317](#_Toc98942919)

[**Iterable protocol** 318](#_Toc98942920)

[Iterators 318](#_Toc98942921)

[Cleaning up 319](#_Toc98942922)

[JavaScript Generators 320](#_Toc98942923)

[Introduction to JavaScript Generators 321](#_Toc98942924)

[More generator examples 323](#_Toc98942925)

[Using generators to implement iterators 323](#_Toc98942926)

[Using a generator to implement the Bag data structure 324](#_Toc98942927)

[Summary 325](#_Toc98942928)

[JavaScript yield 325](#_Toc98942929)

[Introduction to the JavaScript yield keyword 325](#_Toc98942930)

[JavaScript yield examples 326](#_Toc98942931)

[**A) Returning a value** 326](#_Toc98942932)

[**B) Returning undefined** 326](#_Toc98942933)

[**C) Passing a value to the next() method** 327](#_Toc98942934)

[**D) Using yield in an array** 327](#_Toc98942935)

[**E) Using yield to return an array** 328](#_Toc98942936)

[**F) Using the yield to return individual elements of an array** 328](#_Toc98942937)

[JavaScript for…of Loop 329](#_Toc98942938)

[Introduction to the JavaScript for…of loop 329](#_Toc98942939)

[**variable** 329](#_Toc98942940)

[**iterable** 329](#_Toc98942941)

[JavaScript for of loop examples 329](#_Toc98942942)

[**1) Iterating over arrays** 329](#_Toc98942943)

[**2) In-place object destructuring with for…of** 331](#_Toc98942944)

[**3) Iterating over strings** 331](#_Toc98942945)

[**3) Iterating over Map objects** 331](#_Toc98942946)

[**4) Iterating over set objects** 332](#_Toc98942947)

[for...of vs. for...in 332](#_Toc98942948)

[Summary 333](#_Toc98942949)

[JavaScript Asynchronous Iterators 333](#_Toc98942950)

[Introduction to JavaScript Asynchronous Iterators 333](#_Toc98942951)

[The for await...of statement 335](#_Toc98942952)

[JavaScript Async Generators 336](#_Toc98942953)

[What is an async generator 336](#_Toc98942954)

[Introduction to the JavaScript async generators 336](#_Toc98942955)

[A Comprehensive Look at ES6 Modules 338](#_Toc98942956)

[Executing modules on web browsers 338](#_Toc98942957)

[Exporting 339](#_Toc98942958)

[Importing 340](#_Toc98942959)

[**Import a single binding** 341](#_Toc98942960)

[**Import multiple bindings** 341](#_Toc98942961)

[**Import an entire module as an object** 341](#_Toc98942962)

[Limitation of import and export statements 342](#_Toc98942963)

[Aliasing 342](#_Toc98942964)

[Re-exporting a binding 343](#_Toc98942965)

[Importing without bindings 343](#_Toc98942966)

[Default exports 344](#_Toc98942967)

[JavaScript import 345](#_Toc98942968)

[Introduction to the JavaScript import() 345](#_Toc98942969)

[Some practical use cases of JavaScript import() 347](#_Toc98942970)

[**1) Loading module on demand** 347](#_Toc98942971)

[**2) Loading modules based on conditions** 347](#_Toc98942972)

[**3) Computed module specifiers** 347](#_Toc98942973)

[More on the JavaScript import() 348](#_Toc98942974)

[**Using object destructuring** 348](#_Toc98942975)

[**Dynamically loading multiple modules** 348](#_Toc98942976)

[**Accessing the default export** 348](#_Toc98942977)

[Summary 349](#_Toc98942978)

[JavaScript Top-level await 349](#_Toc98942979)

[Introduction to the JavaScript top-level await 349](#_Toc98942980)

[**Workaround** 352](#_Toc98942981)

[**Using the top-level await** 353](#_Toc98942982)

[JavaScript top-level await use cases 354](#_Toc98942983)

[**Dynamic dependency pathing** 354](#_Toc98942984)

[**Dependency fallback** 354](#_Toc98942985)

[Summary 354](#_Toc98942986)

[JavaScript try…catch 354](#_Toc98942987)

[Introduction to JavaScript try…catch statement 354](#_Toc98942988)

[JavaScript try…catch statement examples 355](#_Toc98942989)

[**Ingoring the catch block** 356](#_Toc98942990)

[The exception identifier 357](#_Toc98942991)

[Summary 358](#_Toc98942992)

[JavaScript Throw Exception 358](#_Toc98942993)

[Introduction to the JavaScript throw statement 358](#_Toc98942994)

[JavaScript throw exception examples 358](#_Toc98942995)

[**1) Using the JavaScript throw statement to throw an exception** 358](#_Toc98942996)

[**2) Using JavaScript throw statement to throw an instance of the Error class** 359](#_Toc98942997)

[**3) Using JavaScript throw statement to throw a user-defined exception** 360](#_Toc98942998)

[Summary 361](#_Toc98942999)

[Optional catch Binding 361](#_Toc98943000)

[Introduction to the optional catch binding 361](#_Toc98943001)

[Summary 362](#_Toc98943002)

[JavaScript Execution Context 362](#_Toc98943003)

[Introduction to the JavaScript execution context 362](#_Toc98943004)

[The creation phase 363](#_Toc98943005)

[The execution phase 363](#_Toc98943006)

[JavaScript Call Stack 366](#_Toc98943007)

[Introduction to JavaScript Call Stack 366](#_Toc98943008)

[JavaScript call stack example 367](#_Toc98943009)

[Stack overflow 370](#_Toc98943010)

[Asynchronous JavaScript 370](#_Toc98943011)

[Summary 371](#_Toc98943012)

[JavaScript Event Loop 371](#_Toc98943013)

[JavaScript single-threaded model 371](#_Toc98943014)

[Callbacks to the rescue 373](#_Toc98943015)

[JavaScript Hoisting 377](#_Toc98943016)

[Introduction to the JavaScript hoisting 377](#_Toc98943017)

[Variable hoisting 377](#_Toc98943018)

[**The let keyword** 378](#_Toc98943019)

[Function hoisting 378](#_Toc98943020)

[**Function expressions** 379](#_Toc98943021)

[**Arrow functions** 380](#_Toc98943022)

[Summary 380](#_Toc98943023)

[JavaScript Variable Scopes 380](#_Toc98943024)

[What is variable scope 380](#_Toc98943025)

[The global scope 380](#_Toc98943026)

[Local scope 381](#_Toc98943027)

[Scope chain 382](#_Toc98943028)

[**More scope chain example** 383](#_Toc98943029)

[Global variable leaks: the weird part of JavaScript 384](#_Toc98943030)

[Block scope 385](#_Toc98943031)

What is JavaScript

JavaScript is a programming language initially designed to interact with elements of web pages. In web browsers, JavaScript consists of three main parts:

* ECMAScript provides the core functionality.
* The [Document Object Model (DOM)](https://www.javascripttutorial.net/javascript-dom/) provides interfaces for interacting with elements on web pages
* The [Browser Object Model (BOM)](https://www.javascripttutorial.net/javascript-bom/) provides the browser API for interacting with the web browser.

JavaScript allows you to add interactivity to a web page. Typically, you use JavaScript with HTML and CSS to enhance a web page’s functionality, such as [validating forms](https://www.javascripttutorial.net/javascript-dom/javascript-form-validation/), creating interactive maps, and displaying animated charts.

When a web page is loaded, i.e., after HTML and CSS have been downloaded, the JavaScript engine in the web browser executes the JavaScript code. The JavaScript code then modifies the HTML and CSS to update the user interface dynamically.

The JavaScript engine is a program that executes JavaScript code. In the beginning, JavaScript engines were implemented as interpreters.

However, modern JavaScript engines are typically implemented as just-in-time compilers that compile JavaScript code to bytecode for improved performance.

## **Client-side vs. Server-side JavaScript**

When JavaScript is used on a web page, it is executed in web browsers. In this case, JavaScript works as a client-side language.

JavaScript can run on both web browsers and servers. A popular JavaScript server-side environment is [Node.js](https://www.javascripttutorial.net/nodejs-tutorial/). Unlike client-side JavaScript, server-side JavaScript executes on the server that allows you to access databases, file systems, etc.

## **JavaScript History**

In 1995, JavaScript was created by a Netscape developer named [Brendan Eich](https://en.wikipedia.org/wiki/Brendan_Eich). First, its name was Mocha. And then, its name was changed to LiveScript.

Netscape decided to change LiveScript to JavaScript to leverage Java’s fame, which was popular. The decision was made just before Netscape released its web browser product Netscape Navigator 2. As a result, JavaScript entered version 1.0.

Netscape released JavaScript 1.1 in Netscape Navigator 3. In the meantime, Microsoft introduced a web browser product called the [Internet Explorer](https://en.wikipedia.org/wiki/Internet_Explorer) 3 (IE 3), which competed with Netscape. However, IE came with its own JavaScript implementation called [JScript](https://en.wikipedia.org/wiki/JScript). Microsoft used the name JScript to avoid possible license issues with Netscape.

Hence, two different JavaScript versions were in the market:

* JavaScript in Netscape Navigator
* JScript in Internet Explorer.

JavaScript had no standards that governed its syntax and features. And the community decided that it was time to standardize the language.

In 1997, JavaScript 1.1 was submitted to the [European Computer Manufacturers Association](https://www.ecma-international.org/) (ECMA) as a proposal. [Technical Committee #39](https://www.ecma-international.org/memento/tc39-m.htm) (TC39) was assigned to standardize the language to make it a general-purpose, cross-platform, and vendor-neutral scripting language.

TC39 came up with ECMA-262, a standard for defining a new scripting language named ECMAScript (often pronounced Ek-ma-script).

After that, the International Organization for Standardization and International Electrotechnical Commissions (ISO/IEC) adopted ECMAScript (ISO/IEC-16262).

## **JavaScript overview**

To [define a variable](https://www.javascripttutorial.net/javascript-variables/) in JavaScript, you use var keyword. For example:

var x = 10;

var y = 20;

Code language: JavaScript (javascript)

ES6 added a new way to declare a variable with the [let](https://www.javascripttutorial.net/es6/javascript-let/) keyword:

let x = 10;

let y = 20;

Code language: JavaScript (javascript)

There are differences between [var and let](https://www.javascripttutorial.net/es6/difference-between-var-and-let/). And it’s a good practice to use the let keyword to declare variables.

To declare a [function](https://www.javascripttutorial.net/javascript-function/), you use the function keyword. The following example defines a function that calculates the sum of two arguments:

function add( a, b ) {

return a + b;

}

Code language: JavaScript (javascript)

To call the add() function, you use the following syntax:

let result = add(x, y);

Code language: JavaScript (javascript)

To log the result into the console window of the web browser, you use the console.log() :

console.log(result);

Code language: JavaScript (javascript)

Now, you should see 30 in the console window.

JavaScript provides you with condition statements such as [if-else](https://www.javascripttutorial.net/javascript-if-else/) and [switch](https://www.javascripttutorial.net/javascript-switch-case/) statements. For example:

let a = 20,

b = 30;

function divide(a, b) {

if(b == 0) {

throw 'Division by zero';

}

return a / b;

}

Code language: JavaScript (javascript)

In the divide() function, we check whether the de-numerator (b) is zero. If yes, we [throw an exception](https://www.javascripttutorial.net/javascript-try-catch/). Otherwise, we return the result of a / b.

To declare an [array](https://www.javascripttutorial.net/javascript-array/), you use the following syntax:

let items = [];

Code language: JavaScript (javascript)

To declare an array with some initial elements, you specify the elements in the square brackets:

let items = [1, 2, 3];

Code language: JavaScript (javascript)

You can access the number of elements in the items array through its length property:

console.log(items.length); *// 3*

Code language: JavaScript (javascript)

To iterate over the elements of the items array, you use the [for](https://www.javascripttutorial.net/javascript-for-loop/) loop statement as follows:

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

console.log(items[i]);

}

Code language: JavaScript (javascript)

Or use the [for...of](https://www.javascripttutorial.net/es6/javascript-for-of/) loop in [ES6](https://www.javascripttutorial.net/es6/):

for(let item of items) {

console.log(item);

}

Code language: JavaScript (javascript)

JavaScript is an evolving language. It has many other features that you’ll learn in the following tutorials.

In this tutorial, you learned what JavaScript is and the overview of the JavaScript language.

# JavaScript Code Editors

**Summary**: in this tutorial, you will learn about JavaScript code editors and how to install the Visual Studio Code for coding JavaScript.

## **Popular JavaScript Code Editors**

To edit JavaScript source code, you need a plain text editor such as Notepad on Windows. However, to simplify and speed up typing of JavaScript code, you need a JavaScript code editor.

Besides basic editing features, a JavaScript code editor provides you with syntax highlighting, indentation, autocomplete, and brace matching functionality. Some editors also allow you to debug JavaScript.

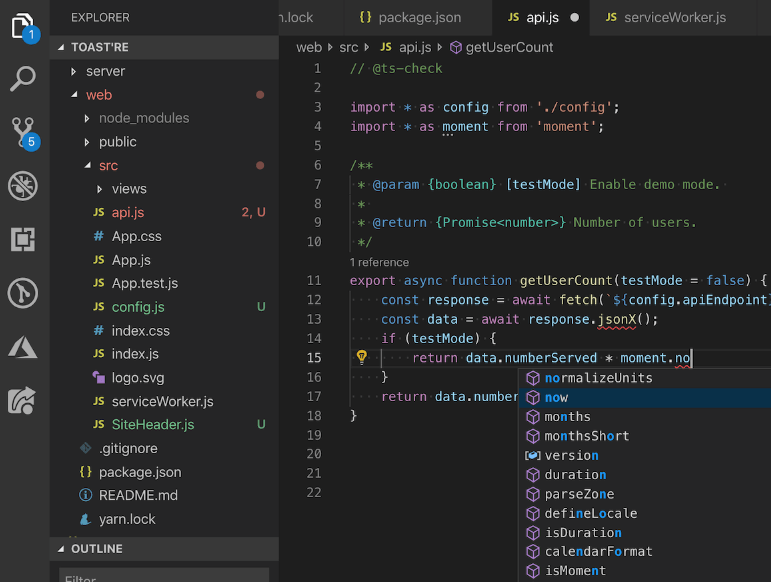
The following are some popular JavaScript code editors:

* [Visual Studio Code](https://code.visualstudio.com/)
* [Atom](https://atom.io/)
* [Notepad++](https://notepad-plus-plus.org/)
* [Vim](https://www.vim.org/)
* [GNU Emacs](https://www.gnu.org/software/emacs/)

Note that all these JavaScript editors are free. As a matter of choice, we will use the Visual Studio Code.

## **Visual Studio Code**

Visual Studio Code is a free and open-source code editor developed by Microsoft. Visual Studio Code is often called VS Code.



VS Code works across platforms including Windows, Linux, and macOS.

VS Code is highly customizable. It allows you to change the theme, keyboard shortcuts, preferences. It has lots of useful extensions that add extra functionality to the editor.

VS Code includes built-in support for JavaScript, which includes IntelliSense, debugging, formatting, code navigation, refactoring, and many other advanced language features.

To learn all the features supported by VS code, you check it out the [JavaScript in Visual Studio Code](https://code.visualstudio.com/docs/languages/javascript).

### **Download Visual Studio Code**

To download the Visual Studio Code, you go to the following download link:

[Download Visual Studio Code](https://code.visualstudio.com/download)

### **Installing Visual Studio Code**

Setting up the Visual Studio Code is easy and quick. It is a small download so that you can install it in a few minutes.

### **A) Windows**

To install the VS Code on Windows, you follow these steps:

* First, execute the installer from the downloaded file.
* Then, open the Visual Studio code.

Note that the installer will add the Visual Studio Code to your %PATH%. It will allow you to type the command code . to launch the VS Code on that folder.

### **B) macOS**

You follow these steps to install the VS Code on macOS:

* First, double-click on the downloaded archive to expands the contents.
* Then, drag Visual Studio Code.app to the Applications to make it available in the launchpad.

### **Installing the Live Server extension**

The live server extension allows you to launch a development local server with the hot reload feature for static pages. Once you change the JavaScript code, you don’t need to refresh the page to see the changes.

To install the Live Server extension, you follow these steps:

* First, click the Extensions.
* Second, search for the Live Server and select the Live Server extension on the list.
* Finally, click the Install button.

A screenshot of a computer

Description automatically generated with medium confidence

In this tutorial, you have learned about the JavaScript code editor and how to install the Visual Studio Code for editing JavaScript source code.

# Web Development Tools

**Summary**: in this tutorial, you will learn how to open the Console tab of web development tools to view the messages.

Web development tools allow you to test and debug the JavaScript code. Web development tools are often called devtools.

Modern web browsers such as Google Chrome, Firefox, Edge, Safari, and Opera provide the devtools as built-in features.

Generally, devtools allow you to work with a variety of web technologies such as HTML, CSS, DOM, and JavaScript.

In this tutorial, you will learn how to open the Console tab of the devtools to view messages output by JavaScript.

## **Google Chrome**

First, open the [devtools.html](https://www.javascripttutorial.net/sample/devtools.html) file.

The devtools.html file has the following JavaScript code:

**<script>**

console.log('Hello, devtools!');

*// the following code causes an error*

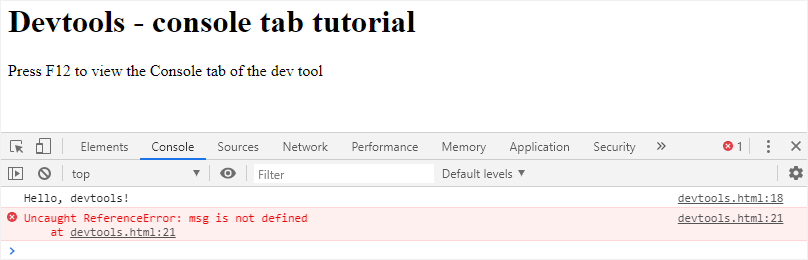
let greeting = msg;

**</script>**

Code language: HTML, XML (xml)

Second, press F12 on Windows or Cmd+Opt+J if you are on Mac.

The devtools will open the Console tab by default. It will look like this:



The first message is 'Hello, DevTools!' which is the output of the following command:

console.log('Hello, DevTools!');

Code language: JavaScript (javascript)

To output the value of the [variable](https://www.javascripttutorial.net/javascript-variables/), you use the following console.log() method. For example:

let message = 'Good Morning!';

console.log(message);

Code language: JavaScript (javascript)

The second message that appears on the Console tab is an error.

Uncaught ReferenceError: msg is not defined

Code language: JavaScript (javascript)

This is because the variable msg has not been defined in the code but was referenced in the assignment.

Now, you can see both normal messages issued by the console.log() and the error messages. It’s enough to start. We’ll dive into the devtools in the later tutorial.

## **Firefox and Edge**

Typically, you open the Console tab of the devtools in Firefox and Edge using F12. They have similar user interfaces.

## **Safari**

If you are using Safari browser on Mac, you need to enable the Developer Menu first:

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application, email

Description automatically generated

And then press Cmd+Opt+C to toggle the Console window:

Graphical user interface, text, application

Description automatically generated

In this tutorial, you have learned how to open the Console tab of the devtools for checking messages issued by JavaScript code.

# JavaScript Hello World Example

**Summary**: This tutorial helps you get started with JavaScript by showing you how to embed JavaScript code into an HTML page.

To insert JavaScript into an HTML page, you use the <script> element. There are two ways to use the <script> element in an HTML page:

* Embed JavaScript code directly into the HTML page.
* Reference an external JavaScript code file.

## **Embed JavaScript code in an HTML page**

Placing JavaScript code inside the <script> element directly is not recommended and should be used only for proof of concept or testing purposes.

The JavaScript code in the <script> element is interpreted from top to bottom. For example:

<!DOCTYPE html>

**<html lang="en">**

**<head>**

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

**<title>**JavaScript Hello World Example**</title>**

**<script>**

alert('Hello, World!');

**</script>**

**</head>**

**<body>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

In the <script> element, we use the [alert()](https://www.javascripttutorial.net/javascript-bom/javascript-alert/) function to display the Hello, World! message.

## **Include an external JavaScript file**

To include a JavaScript from an external file:

* First, create a file whose extension is .js e.g., app.js and place it in the js subfolder. Note that placing the JavaScript file in the js folder is not required however it is a good practice.
* Then, use the URL to the JavasScript source code file in the src attribute of the <script> element.

The following shows the contents of the app.js file:

alert('Hello, World!');

Code language: JavaScript (javascript)

And the following shows the helloworld.html file:

<!DOCTYPE html>

**<html lang="en">**

**<head>**

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

**<title>**JavaScript Hello World Example**</title>**

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

**</head>**

**<body>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

If you launch the helloworld.html file in the web browser, you will see an alert that displays the Hello, World! message.

Note that you can load a JavaScript file from a remote server. This allows you to serve up JavaScript from various domains e.g., content delivery network (CDN) to speed up the page.

When you have multiple JavaScript files on a page, the JavaScript engine interprets the files in the order that they appear. For example:

**<script src="js/service.js"></script>**

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

Code language: HTML, XML (xml)

In this example, JavaScript engine will interpret the service.js and the app.js files in sequence. It completes interpreting the service.js file first before interpreting the app.js file.

For the page that includes many external JavaScript files, the blank page is shown during the page rendering phase.

To avoid this, you include the JavaScript file just before the </body> tag as shown in this example:

<!DOCTYPE html>

**<html lang="en">**

**<head>**

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

**<title>**JavaScript Hello World Example**</title>**

**</head>**

**<body>**

*<!-- end of page content here-->*

**<script src="js/service.js"></script>**

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

**</body>**

**</html>**

Code language: HTML, XML (xml)

## **The async and defer attributes**

To change how the browser load and execute JavaScript files, you use one of two attributes of the <script> element async and defer.

These attributes take effect only on the external script files. The async attribute instructs the web browser to execute the JavaScript file asynchronously. The async attribute does not guarantee the script files to execute in the order that they appear. For example:

**<script async src="service.js"></script>**

**<script async src="app.js"></script>**

Code language: HTML, XML (xml)

The app.js file might execute before the service.js file. Therefore, you must ensure that there is no dependency between them.

The defer attribute requests the web browser to execute the script file after the HTML document has been parsed.

<!DOCTYPE html>

**<html lang="en">**

**<head>**

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

**<title>**JavaScript defer demonstration**</title>**

**<script defer src="defer-script.js"></script>**

**</head>**

**<body>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

Even though we place the <script> element in the <head> section, the script will wait for the browser to receive the closing tag <html> to start executing.

## **Summary**

* Use <script> element to include a JavaScript file in a HTML page.
* The async attribute of the <script> element instructs the web browser to fetch the JavaScript file in parallel and then parse and execute as soon as the JavaScript file is available.
* The defer attribute of the <script> element allows the web browser to execute the JavaScript file after the document has been parsed.

# JavaScript Syntax

**Summary**: in this tutorial, you will learn about JavaScript syntax, including whitespace, statements, identifiers, comments, expressions, and keywords.

## **Whitespace**

Whitespace refers to characters that provide the space between other characters. JavaScript has the following whitespace:

* Carriage return
* Space
* New Line
* tab

JavaScript engine ignores whitespace. However, you can use whitespace to format the code to make it easy to read and maintain.

The following JavaScript code doesn’t use whitespace:

let formatted = true; if (formatted) {console.log('The code is easy to read');}

Code language: JavaScript (javascript)

It’s is equivalent to the following code that uses whitespace. Hence, this code is much easy to read:

let formatted = true;

if (formatted) {

console.log('The code is easy to read');

}

Code language: JavaScript (javascript)

Note that JavaScript bundlers remove all whitespace from JavaScript files and put them into a single file for deployment. By doing this, JavaScript bundlers make the JavaScript code lighter and faster to load in the web browsers.

## **Statements**

A statement is a code that declares a variable or instructs the JavaScript engine to do a task. A simple statement is terminated by a semicolon (;).

Although the semicolon (;) is optional; you should always use it to terminate a statement. For example, the following [declares a variable](https://www.javascripttutorial.net/javascript-variables/) and shows it to the console:

let message = "Welcome to JavaScript";

console.log(message);

Code language: JavaScript (javascript)

### **Blocks**

A block is a sequence of zero or more simple statements. A block is delimited by a pair of curly brackets {}. For example:

if (window.localStorage) {

console.log('The local storage is supported');

}

Code language: JavaScript (javascript)

## **Identifiers**

An identifier is a name you choose for variables, parameters, [functions](https://www.javascripttutorial.net/javascript-function/), classes, etc. An identifier name starts with a letter (a-z, or A-Z), an underscore(\_), or a dollar sign ($) and is followed by a sequence of characters including (a-z, A-Z), numbers (0-9), underscores (\_), and dollar signs ($).

Note that the letter is not limited to the ASCII character and may include extended ASCII or Unicode though not recommended.

Identifiers are case-sensitive. For example, the message is different from the Message.

## **Comments**

Comments allow you to add notes or hints to JavaScript code. When executing the code, the JavaScript engine ignores the comments.

JavaScript supports single-line and block comments.

### **Single-line comments**

A single-line comment starts with two forward-slashes characters (//). A single-line comment makes all the text following the // on the same line into a comment. For example:

*// this is a single-line comment*

Code language: JSON / JSON with Comments (json)

### **Block comments**

A delimited comment begins with a forward slash and asterisk /\* and ends with the opposite \*/ as in the following example:

*/\* This is a block comment*

*that can span multiple lines \*/*

Code language: JSON / JSON with Comments (json)

## **Expressions**

An expression is a piece of code that evaluates to a value. For example:

2 + 1

The above expression returns three.

## **Keywords & Reserved words**

JavaScript defines a list of reserved keywords that have specific uses. Therefore, you cannot use the reserved keywords as identifiers or property names by rules.

The following table shows the JavaScript reserved words defined in ECMA-262:

|  |  |  |
| --- | --- | --- |
| [break](https://www.javascripttutorial.net/javascript-break/) | case | catch |
| [continue](https://www.javascripttutorial.net/javascript-continue/) | debugger | default |
| [else](https://www.javascripttutorial.net/javascript-if-else/) | export | [extends](https://www.javascripttutorial.net/es6/javascript-inheritance/) |
| [function](https://www.javascripttutorial.net/javascript-function/) | [if](https://www.javascripttutorial.net/javascript-if/) | [import](https://www.javascripttutorial.net/nodejs-tutorial/nodejs-es-module/) |
| new | return | super |
| throw | try | null |
| void | [while](https://www.javascripttutorial.net/javascript-while-loop/) | with |
| [class](https://www.javascripttutorial.net/es6/javascript-class/) | delete | finally |
| [in](https://www.javascripttutorial.net/javascript-for-loop/) | [switch](https://www.javascripttutorial.net/javascript-switch-case/) | typeof |
| yield | [const](https://www.javascripttutorial.net/es6/javascript-const/) | [do](https://www.javascripttutorial.net/javascript-do-while/) |
| [for](https://www.javascripttutorial.net/javascript-for-loop/) | instanceof | [this](https://www.javascripttutorial.net/javascript-this/) |
| var |  |  |

In addition to the reserved keywords, ECMA-252 also define a list of future reserved words that cannot be used as identifiers or property names:

|  |  |  |
| --- | --- | --- |
| enum | implements | let |
| protected | private | public |
| await | interface | package |
| implements | public |  |

## **Summary**

* Use whitespace including cariage return, space, newline, and tab to format the code. The JavaScript engine ignores the whiespace.
* Use a semicolon (;) to terminate a simple statement.
* Use the curly braces ({}) to form a block that groups one or more simple statments.
* A single-line comment start with // followed by a text. A block comment begins with /\* and ends with \*/. JavaScript engine also ignores the comments.
* Identifers are names that you choose for variables, functions, classes, etc.
* Do not use the reserved keywords and reserved words for identifers.

# JavaScript Variables

**Summary**: in this tutorial, you’ll learn about JavaScript variables and how to use variables to store values in the application.

A variable is a label that references a value like a number or string. Before using a variable, you need to declare it.

## **Declare a variable**

To declare a variable, you use the var keyword followed by the variable name as follows:

var message;

Code language: JavaScript (javascript)

A variable name can be any valid identifier. By default, the message variable has a special value [undefined](https://www.javascripttutorial.net/javascript-data-types/#undefined) if you have not assigned a value to it.

Variable names follow these rules:

* Variable names are case-sensitive. This means that the message and Message are different variables.
* Variable names can only contain letters, numbers, underscores, or dollar signs and cannot contain spaces. Also, variable names must begin with a letter, an underscore (\_) or a dollar sign ($).
* Variable names cannot use the reserved words.

By convention, variable names use camelcase like message, yourAge, and myName.

JavaScript is a dynamically typed language. This means that you don’t need to specify the variable’s [type](https://www.javascripttutorial.net/javascript-data-types/) in the declaration like other static typed languages such as Java or [C#](https://www.csharptutorial.net/csharp-tutorial/csharp-variables/).

Starting in ES6, you can use the let keyword to declare a variable like this:

let message;

Code language: JavaScript (javascript)

It’s a good practice to use the let keyword to declare a variable. Later, you’ll learn the differences [between var and let keywords](https://www.javascripttutorial.net/es6/difference-between-var-and-let/). And you should not worry about it for now.

## **Initialize a variable**

Once you have declared a variable, you can initialize it with a value. To initialize a variable, you specify the variable name, followed by an equals sign (=) and a value.

For example, The following declares the message variable and initializes it with a literal string "Hello":

let message;

message = "Hello";

Code language: JavaScript (javascript)

To declare and initialize a variable at the same time, you use the following syntax:

let variableName = value;

Code language: JavaScript (javascript)

For example, the following statement declares the message variable and initializes it with the literal string "Hello":

let message = "Hello";

Code language: JavaScript (javascript)

JavaScript allows you to declare two or more variables using a single statement. To separate two variable declarations, you use a comma (,) like this:

let message = "Hello",

counter = 100;

Code language: JavaScript (javascript)

Since JavaScript is a dynamically typed language, you can assign a value of a different type to a variable. Although, it is not recommended. For example:

let message = 'Hello';

message = 100;

Code language: JavaScript (javascript)

## **Change a variable**

Once you initialize a variable, you can change its value by assigning a different value. For example:

let message = "Hello";

message = 'Bye';

Code language: JavaScript (javascript)

## **Undefined vs. undeclared variables**

It’s important to distinguish between undefined and undeclared variables.

An undefined variable is a variable that has been declared but has not been initialized with a value. For example:

let message;

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

Code language: JavaScript (javascript)

In this example, the message variable is declared but not initialized. Therefore, the message variable is undefined.

In contrast, an undeclared variable is a variable that has not been declared. For example:

console.log(counter);

Code language: JavaScript (javascript)

Output:

console.log(counter);

^

ReferenceError: counter is not defined

Code language: JavaScript (javascript)

In this example, the counter variable has not been declared. Hence, accessing it causes a ReferenceError.

## **Constants**

A constant holds a value that doesn’t change. To declare a constant, you use the const keyword. When defining a constant, you need to initialize it with a value. For example:

const workday = 5;

Code language: JavaScript (javascript)

Once defining a constant, you cannot change its value.

The following example attempts to change the value of the workday constant to 4 and causes an error:

workday = 2;

Error:

Uncaught TypeError: Assignment to constant variable.

Code language: JavaScript (javascript)

Later, you’ll learn that the const keyword actually defines a read-only reference to a value in the [constants](https://www.javascripttutorial.net/es6/javascript-const/) tutorial.

## **Summary**

* A variable is a label that references a value.
* Use the let keyword to declare a variable.
* An undefined variable is a variable that has been declared but not initialized while an undeclared variable is variable that has not been declared.
* Use the const keyword to define a readonly reference to a value.

# JavaScript Data Types

**Summary**: in this tutorial, you will learn about the JavaScript data types and their unique characteristics.

JavaScript has the primitive data types:

1. [null](https://www.javascripttutorial.net/javascript-data-types/#null)
2. [undefined](https://www.javascripttutorial.net/javascript-data-types/#undefined)
3. [boolean](https://www.javascripttutorial.net/javascript-data-types/#boolean)
4. [number](https://www.javascripttutorial.net/javascript-data-types/#number)
5. [string](https://www.javascripttutorial.net/javascript-data-types/#string)
6. [symbol](https://www.javascripttutorial.net/javascript-data-types/#symbol) – available from ES2015
7. [bigint](https://www.javascripttutorial.net/es-next/javascript-bigint/) – available from ES2020

and a complex data type [object](https://www.javascripttutorial.net/javascript-data-types/#object).

JavaScript is a dynamically typed language. It means that a [variable](https://www.javascripttutorial.net/javascript-variables/) doesn’t associate with a type. In other words, a variable can hold a value of different types. For example:

let counter = 120; *// counter is a number*

counter = false; *// counter is now a boolean*

counter = "foo"; *// counter is now a string*

Code language: JavaScript (javascript)

To get the current type of the value that the variable stores, you use the [typeof](https://www.javascripttutorial.net/javascript-typeof/) operator:

let counter = 120;

console.log(typeof(counter)); *// "number"*

counter = false;

console.log(typeof(counter)); *// "boolean"*

counter = "Hi";

console.log(typeof(counter)); *// "string"*

Code language: JavaScript (javascript)

Output:

"number"

"boolean"

"string"

Code language: JSON / JSON with Comments (json)

## **The undefined type**

The undefined type is a primitive type that has only one value undefined. By default, when a variable is declared but not initialized, it is assigned the value of undefined.

Consider the following example:

let counter;

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

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

Code language: JavaScript (javascript)

In this example, the counter is a variable. Since counter hasn’t been initialized, it is assigned the value undefined. The type of counter is also undefined.

It’s important to note that the typeof operator also returns undefined when you call it on a variable that hasn’t been declared:

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

Code language: JavaScript (javascript)

## **The null type**

The null type is the second primitive data type that also has only one value null. For example:

let obj = null;

console.log(typeof obj); *// object*

Code language: JavaScript (javascript)

JavaScript defines that null is equal to undefined as follows:

console.log(null == undefined); *// true*

Code language: JavaScript (javascript)

## **The number type**

JavaScript uses the number type to represent both integer and floating-point numbers.

The following statement declares a variable and initializes its value with an integer:

let num = 100;

Code language: JavaScript (javascript)

To represent a floating-point number, you include a decimal point followed by at least one number. For example:

let price= 12.5;

let discount = 0.05;

Code language: JavaScript (javascript)

Note that JavaScript automatically converts a floating-point number into an integer number if the number appears to be a whole number.

The reason is that Javascript always wants to use less memory since a floating-point value uses twice as much memory as an integer value. For example:

let price = 200.00; *// interpreted as an integer 200*

Code language: JavaScript (javascript)

To get the range of the number type, you use Number.MIN\_VALUE and Number.MAX\_VALUE. For example:

console.log(Number.MAX\_VALUE); *// 1.7976931348623157e+308*

console.log(Number.MIN\_VALUE); *// 5e-324*

Code language: JavaScript (javascript)

Also, you can use Infinity and -Infinity to represent the infinite number. For example:

console.log(Number.MAX\_VALUE + Number.MAX\_VALUE); *// Infinity*

console.log(-Number.MAX\_VALUE - Number.MAX\_VALUE); *// -Infinity*

Code language: JavaScript (javascript)

### **NaN**

NaN stands for Not a Number. It is a special numeric value that indicates an invalid number. For example, the division of a string by a number returns NaN:.

console.log('a'/2); *// NaN;*

Code language: JavaScript (javascript)

The NaN has two special characteristics:

* Any operation with NaN returns NaN.
* The NaN does not equal any value, including itself.

Here are some examples:

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

console.log(NaN == NaN); *// false*

Code language: JavaScript (javascript)

## **The string type**

In JavaScript, a string is a sequence of zero or more characters. A string literal begins and ends with either a single quote(') or a double quote (").

A string that begins with a double quote must end with a double quote. Likewise, a string that begins with a single quote must also end with a single quote:

let greeting = 'Hi';

let message = "Bye";

Code language: JavaScript (javascript)

If you want to single quote or double quotes in a literal string, you need to use the backslash to escape it. For example:

let message = 'I\'m also a valid string'; *// use \ to escape the single quote (')*

Code language: JavaScript (javascript)

JavaScript strings are immutable. This means that it cannot be modified once created. However, you can create a new string from an existing string. For example:

let str = 'JavaScript';

str = str + ' String';

Code language: JavaScript (javascript)

In this example:

* First, declare the str variable and initialize it to a string of 'JavaScript'.
* Second, use the + operator to combine 'JavaScript' with ' String' to make its value as 'Javascript String'.

Behind the scene, the JavaScript engine creates a new string that holds the new string 'JavaScript String' and destroys the original strings 'JavaScript' and ' String'.

## **The boolean type**

The boolean type has two literal values: true and false in lowercase. The following example declares two variables that hold the boolean values.

let inProgress = true;

let completed = false;

console.log(typeof completed); *// boolean*

Code language: JavaScript (javascript)

JavaScript allows values of other types to be converted into boolean values of true or false.

To convert a value of another data type into a boolean value, you use the [Boolean()](https://www.javascripttutorial.net/javascript-boolean/) function. The following table shows the conversion rules:

| **Type** | **true** | **false** |
| --- | --- | --- |
| string | non-empty string | empty string |
| number | non-zero number and Infinity | 0, NaN |
| object | non-null object | null |
| undefined |  | undefined |

For example:

console.log(Boolean('Hi'));*// true*

console.log(Boolean('')); *// false*

console.log(Boolean(20)); *// true*

console.log(Boolean(Infinity)); *// true*

console.log(Boolean(0)); *// false*

console.log(Boolean({foo: 100})); *// true on non-empty object*

console.log(Boolean(null));*// false*

Code language: JavaScript (javascript)

## **The symbol type**

JavaScript added a primitive type in ES6: the [symbol](https://www.javascripttutorial.net/es6/symbol/). Different from other primitive types, the symbol type does not have a literal form.

To create a symbol, you call the Symbol function as follows:

let s1 = Symbol();

Code language: JavaScript (javascript)

The Symbol function creates a new unique value every time you call it.

console.log(Symbol() == Symbol()); *// false*

Code language: JavaScript (javascript)

Note that you’ll learn more about symbols in the [symbol tutorial](https://www.javascripttutorial.net/es6/symbol/).

## **The bigint type**

The bigint type represents the whole numbers that are larger than 253 – 1. To form a bigint literal number, you append the letter n at the end of the number:

let pageView = 9007199254740991n;

console.log(typeof(pageView)); *// 'bigint'*

Code language: JavaScript (javascript)

And you’ll learn more about the [bigint type here](https://www.javascripttutorial.net/es-next/javascript-bigint/" \t "_blank).

## **The object type**

In JavaScript, an [object](https://www.javascripttutorial.net/home/javascript-objects/) is a collection of [properties](https://www.javascripttutorial.net/home/javascript-object-properties/), where each property is defined as a key-value pair.

The following example defines an empty object using the object literal syntax:

let emptyObject = {};

Code language: JavaScript (javascript)

The following example defines the person object with two properties: firstName and lastName.

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

A property name of an object can be any string. You can use quotes around the property name if it is not a valid identifier.

For example, if the person object has a property first-name, you must place it in the quotes such as "first-name".

A property of an object can hold an object. For example:

let contact = {

firstName: 'John',

lastName: 'Doe',

email: 'john.doe@example.com',

phone: '(408)-555-9999',

address: {

building: '4000',

street: 'North 1st street',

city: 'San Jose',

state: 'CA',

country: 'USA'

}

}

Code language: JavaScript (javascript)

The contact object has the firstName, lastName, email, phone, and address properties.

The address property itself holds an object that has building,  street, city, state, and country properties.

To access a object’s property, you can use

* The dot notation (.)
* The array-like notation ([]).

The following example uses the dot notation (.) to access the firstName and lastName properties of the contact object.

console.log(contact.firstName);

console.log(contact.lastName);

Code language: CSS (css)

If you reference a property that does not exist, you’ll get an undefined value. For example:

console.log(contact.age); *// undefined*

Code language: JavaScript (javascript)

The following example uses the array-like notation to access the email and phone properties of the contact object.

console.log(contact['phone']); *// '(408)-555-9999'*

console.log(contact['email']); *// 'john.doe@example.com'*

Code language: JavaScript (javascript)

## **Summary**

* JavaScript has the primitive types: number, string, boolean, null, undefined, symbol and bigint and a complex type: object.

# JavaScript Numbers

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

## **Introduction to the JavaScript Number**

JavaScript uses the number type to represent both integers and floating-point values. Technically, the JavaScript number type uses the IEEE-754 format.

ES2020 introduced a new primitive type [bigint](https://www.javascripttutorial.net/es-next/javascript-bigint/" \t "_blank) that represent big integer numbers whose values are larger than 253 – 1.

To support various types of numbers, JavaScript uses different number literal formats.

## **Integer numbers**

The following shows how to declare a variable that holds a decimal integer:

let counter = 100;

Code language: JavaScript (javascript)

Integers can be represented in the following formats:

* Octal (base 8)
* Hexadecimal (based 16)

When you use the octal and hexadecimal numbers in arithmetic operations, JavaScript treats them as decimal numbers.

### **Octal numbers**

An octal literal number starts with the digit zero (0) followed by a sequence of octal digits (numbers from 0 through 7). For example:

let num = 071;

console.log(num);

Code language: JavaScript (javascript)

Output:

57

If an octal number contains a number that is not in the range from 0 to 7, the JavaScript engine ignores the 0 and treats the number as a decimal. For example:

let num = 080;

console.log(num);

Code language: JavaScript (javascript)

Output:

80

This implicit behavior might cause issues. Therefore, ES6 introduced a new [octal literal](https://www.javascripttutorial.net/es6/octal-and-binary-literals/) that starts with the 0o followed by a sequence of octal digits (from 0 to 7). For example:

let num = 0o71;

console.log(num);

Code language: JavaScript (javascript)

Output:

57

If you an invalid number after 0o, JavaScript will issue a syntax error like this:

let num = 0o80;

console.log(num);

Code language: JavaScript (javascript)

Output:

let num = 0o80;

^^

SyntaxError: Invalid or unexpected token

Code language: JavaScript (javascript)

### **Hexadecimal numbers**

Hexadecimal numbers start with 0x or 0X followed by any number of hexadecimal digits (0 through 9, and a through f). For example:

let num = 0x1a;

console.log(num);

Code language: JavaScript (javascript)

Output:

26

## **Floating-point numbers**

To define a floating-point literal number, you include a decimal point and at least one number after that. For example:

let price = 9.99;

let tax = 0.08;

let discount = .05; *// valid but not recommeded*

Code language: JavaScript (javascript)

When you have a very big number, you can use e-notation. E-notation indicates a number should be multiplied by 10 raised to a given power. For example:

let amount = 3.14e7;

console.log(amount);

Code language: JavaScript (javascript)

Output:

31400000

The notation 3.14e7 means that take 3.14 and multiply it by 107.

Likewise, you can use the E-notation to represent a very small number. For example:

let amount = 5e-7;

console.log(amount);

Code language: JavaScript (javascript)

Output:

0.0000005

Code language: CSS (css)

The 5e-7 notation means that take 5 and divide it by 10,000,000.

Also, JavaScript automatically converts any floating-point number with at least six zeros after the decimal point into e-notation. For example:

let amount = 0.0000005;

console.log(amount);

Code language: JavaScript (javascript)

Output:

5e-7

Floating-point numbers are accurate up to 17 decimal places. When you perform arithmetic operations on floating-point numbers, you often get the approximate result. For example:

let amount = 0.2 + 0.1;

console.log(amount);

Code language: JavaScript (javascript)

Output:

0.30000000000000004

Code language: CSS (css)

## **Big Integers**

JavaScript introduced the bigint type starting in ES2022. The bigint type stores whole numbers whose values are greater than 253 – 1.

A big integer literal has the n character at the end of an integer literal like this:

let pageView = 9007199254740991n;

Code language: JavaScript (javascript)

## **Summary**

* JavaScript Number type reprensents both integer and floating-point numbers.

# JavaScript Numeric Separator

**Summary**: in this tutorial, you’ll how to use the JavaScript numeric separator to make the numeric literals more readable.

## **Introduction to the JavaScript numeric separator**

The numeric separator allows you to create a visual separation between groups of digits by using underscores (\_) as separators.

For example, the following number is very difficult to read especially when it contains long digit repetitions:

const budget = 1000000000;

Code language: JavaScript (javascript)

Is this a billion or a hundred million?

The numeric separator fixes this readability issue as follows:

const budget = 1\_000\_000\_000;

Code language: JavaScript (javascript)

As you can see, the number is now very easy to interpret.

JavaScript allows you to use numeric separators for both integer and floating-point numbers. For example:

let amount = 120\_201\_123.05; *// 120,201,123.05*

let expense = 123\_450; *// 12345*

let fee = 12345\_00; *// 1234500*

Code language: JavaScript (javascript)

It’s important to note that all numbers in JavaScript are floating-point numbers.

Also, you can use the numeric separators for factional and exponent parts. For example:

let amount = 0.000\_001; *// 1 millionth*

Code language: JavaScript (javascript)

It’s important to notice that you can use the numeric separator for [bigint](https://www.javascripttutorial.net/es-next/javascript-bigint/) literal, binary literal, octal literal, and hex literal. For example:

*// BigInt*

const max = 9\_223\_372\_036\_854\_775\_807n;

*// binary*

let nibbles = 0b1011\_0101\_0101;

*// octal*

let val = 0o1234\_5670;

*// hex*

let message = 0xD0\_E0\_F0;

Code language: JavaScript (javascript)

## **Summary**

* Use underscores (\_) as the numeric separators to create a visual separation between group of digits.

# 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 Boolean

**Summary**: in this tutorial, you will learn about the JavaScript Boolean object and the differences between the Boolean object and the boolean primitive type.

## **JavaScript boolean primitive type**

JavaScript provides a [boolean](https://www.javascripttutorial.net/javascript-data-types/" \l "boolean) primitive type that has two values of true and false. The following example declares two [variables](https://www.javascripttutorial.net/javascript-variables/) that hold boolean values of false and true:

let isPending = false;

let isDone = true;

Code language: JavaScript (javascript)

When you apply the  typeof operator to a variable that holds a primitive boolean value, you get the boolean as the following example:

console.log(typeof(isPending)); *// boolean*

console.log(typeof(isDone)); *// boolean*

Code language: JavaScript (javascript)

## **JavaScript Boolean object**

In addition to the boolean primitive type, JavaScript also provides you with the global Boolean() function, with the letter B in uppercase, to cast a value of another type to boolean.

The following example shows how to use the Boolean() function to convert a string into a boolean value. Because the string is not empty, it returns true.

let a = Boolean('Hi');

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

console.log(typeof(a)); *// boolean*

Code language: JavaScript (javascript)

The Boolean is also a wrapper object of the boolean primitive type. It means that when you pass either true or false to the Boolean constructor, it’ll create a Boolean object. For example:

let b = new Boolean(false);

Code language: JavaScript (javascript)

To get the primitive value back, you call the valueOf() method of the Boolean object as follows:

console.log(b.valueOf()); *// false*

Code language: JavaScript (javascript)

However, if you call the toString() method of a Boolean object, you get a string value "true" or "false". For example:

console.log(b.toString()); *// "false"*

Code language: JavaScript (javascript)

## **JavaScript boolean vs. Boolean**

Consider this example:

let completed = true;

let active = new Boolean(false);

Code language: JavaScript (javascript)

First, active is an object so you can add a property to it:

active.primitiveValue = active.valueOf();

console.log(active.primitiveValue); *// false*

Code language: JavaScript (javascript)

However, you cannot do it with the primitive boolean variable like the completed variable:

completed.name = 'primitive';

console.log(completed.name); *// undefined*

Code language: JavaScript (javascript)

Second, the typeof of Boolean object returns object, whereas the typeof of a primitive boolean value returns boolean.

console.log(typeof completed); *// boolean*

console.log(typeof active); *// object*

Code language: JavaScript (javascript)

Third, when applying the  instanceof operator to a Boolean object, it returns true. However, it returns false if you apply the  instanceof operator to a boolean value.

console.log(completed instanceof Boolean); *// false*

console.log(active instanceof Boolean); *// true*

Code language: JavaScript (javascript)

It is a good practice **never** to use the Boolean object because it will create many confusions, especially when using in an expression. For example:

let falseObj = new Boolean(false);

if (falseObj) {

console.log('weird part of the Boolean object');

}

Code language: JavaScript (javascript)

How the script works.

* First, create falseObj as a Boolean object wrapper for the false value.
* Second, use falseObj in the  [if](https://www.javascripttutorial.net/javascript-if-else/) statement. Because falseObj is an object, and JavaScript engine coerces it to a boolean value of true. As a result, the statement inside the if block is executed.

The following table summarizes the differences between the JavaScript Boolean and boolean:

| **Operator** | **boolean** | **Boolean** |
| --- | --- | --- |
| typeof | boolean | object |
| instanceof Boolean | false | true |

It is recommended that you use the Boolean() function to convert a value of a different type to a Boolean type, but you should never use the Boolean as a wrapper object of a primitive boolean value.

In this tutorial, you have learned about the JavaScript Boolean object and the differences between the Boolean object and boolean primitive type.

# JavaScript string

**Summary**: in this tutorial, you’ll learn about the JavaScript string primitive type and how to use it to define strings.

## **Introduction to the JavaScript strings**

JavaScript strings are primitive values. Also, strings are immutable. It means that if you modify a string, you will always get a new string. The original string doesn’t change.

To create literal strings, you use either single quotes (') or double quotes (") like this:

let str = 'Hi';

let greeting = "Hello";

Code language: JavaScript (javascript)

ES6 introduced [template literals](https://www.javascripttutorial.net/es6/javascript-template-literals/) that allow you to define a string backtick (`) characters:

let name = `John`';

Code language: JavaScript (javascript)

The template literals allow you to use the single quotes and double quotes inside a string without the need of escaping them. For example:

let mesage = `"I'm good". She said";

Code language: JavaScript (javascript)

Also, you can place the [variables](https://www.javascripttutorial.net/javascript-variables/) and expressions inside a template literal. JavaScript will replace the variables with their value in the string. This is called string interpolation. For example:

let name = 'John';

let message = `Hi, I'm ${name}.`;

console.log(message);

Code language: JavaScript (javascript)

Output:

Hi, I'm John.

In this example, JavaScript replaces the name variable with its value inside the template literal.

## **Escaping special characters**

To escape special characters, you use the backslash \ character. For example:

* Windows line break: '\r\n'
* Unix line break: '\n'
* Tab: '\t'
* Backslash '\'

The following example uses the backslash character to escape the single quote character in a string:

let str = 'I\'m a string!';

Code language: JavaScript (javascript)

## **Getting the length of the string**

The length property returns the length of a string:

let str = "Good Morning!";

console.log(str.length); *// 13*

Code language: JavaScript (javascript)

Note that JavaScript has the [String](https://www.javascripttutorial.net/javascript-string-type/) type (with the letter S in uppercase), which is the [primitive wrapper type](https://www.javascripttutorial.net/javascript-primitive-wrapper-types/) of the primitive string type. Therefore, you can access all properties and methods of the String type from a primitive string.

## **Accessing characters**

To access the characters in a string, you use the array-like [] notation with the zero-based index. The following example returns the first character of a string with the index zero:

let str = "Hello";

console.log(str[0]); *// "H"*

Code language: JavaScript (javascript)

To access the last character of the string, you use the length - 1 index:

let str = "Hello";

console.log(str[str.length -1]); *// "o"*

Code language: JavaScript (javascript)

## **Concatenating strings via + operator**

To [concatenate two or more strings](https://www.javascripttutorial.net/string/javascript-string-concatenation/), you use the + operator:

let name = 'John';

let str = 'Hello ' + name;

console.log(str); *// "Hello John"*

Code language: JavaScript (javascript)

If you want to assemble a string piece by piece, you can use the += operator:

let className = 'btn';

className += ' btn-primary'

className += ' none';

console.log(className);

Code language: JavaScript (javascript)

Output:

btn btn-primary none

Code language: JavaScript (javascript)

## **Converting values to string**

To convert a non-string value to a string, you use one of the following:

* String(n);
* ” + n
* n.toString()

Note that the toString() method doesn’t work for undefined and null.

When you convert a string to a boolean, you cannot convert it back. For example:

let status = false;

let str = status.toString(); *// "false"*

let back = Boolean(str); *// true*

Code language: JavaScript (javascript)

In this example:

* First, declare the status variable and initialize its with the value of false.
* Second, convert the status variable to a string using the toString() method.
* Third, convert the string back to a boolean value using the Boolean() function. The Boolean() function converts the string "false" to a boolean value. The result is true because "false" is a non-empty string.

Note that only string for which the Boolean() returns false, is the empty string ('');

## **Comparing strings**

To compare two strings, you use [comparison operators](https://www.javascripttutorial.net/javascript-comparison-operators/) such as >, >=, <, <=, and == operators.

The comparison operators compare strings based on the numeric values of the characters. And it may return the string order that is different from the one used in dictionaries. For example:

let result = 'a' < 'b';

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

Code language: JavaScript (javascript)

However:

let result = 'a' < 'B';

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

Code language: JavaScript (javascript)

## **Summary**

* JavaScript strings are primitive values and immutable.
* Literal strings are delimited by single quotes ('), double quotes ("), or backticks (`).
* The length property returns the length of the string.
* Use the comparison operators `>, >=, <, <=, == to compare strings.

# JavaScript Objects

**Summary**: in this tutorial, you will learn about JavaScript objects and how to manipulate object properties effectively.

## **Introduction to the JavaScript objects**

In JavaScript, an object is an unordered collection of key-value pairs. Each key-value pair is called a property.

The key of a property can be a string. And the value of a property can be any value, e.g., a [string](https://www.javascripttutorial.net/javascript-string-type/), a [number](https://www.javascripttutorial.net/javascript-number/), an [array](https://www.javascripttutorial.net/javascript-array/), and even a [function](https://www.javascripttutorial.net/javascript-function/).

JavaScript provides you with many ways to create an object. The most commonly used one is to use the object literal notation.

The following example creates an empty object using the object literal notation:

let empty = {};

Code language: JavaScript (javascript)

To create an object with properties, you use the key:value within the curly braces. For example, the following creates a new person object:

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

The person object has two properties firstName and lastName with the corresponding values 'John' and 'Doe'.

When an object has multiple properties, you use a comma (,) to separate them like the above example.

## **Accessing properties**

To access a property of an object, you use one of two notations: the dot notation and array-like notation.

### **1) The dot notation (.)**

The following illustrates how to use the dot notation to access a property of an object:

objectName.propertyName

Code language: CSS (css)

For example, to access the firstName property of the person object, you use the following expression:

person.firstName

Code language: CSS (css)

This example creates a person object and shows the first name and last name to the console:

let person = {

firstName: 'John',

lastName: 'Doe'

};

console.log(person.firstName);

console.log(person.lastName);

Code language: JavaScript (javascript)

### **2) Array-like notation ( [])**

The following illustrates how to access the value of an object’s property via the array-like notation:

objectName['propertyName']

Code language: CSS (css)

For example:

let person = {

firstName: 'John',

lastName: 'Doe'

};

console.log(person['firstName']);

console.log(person['lastName']);

Code language: JavaScript (javascript)

When a property name contains spaces, you need to place it inside quotes. For example, the following address object has the 'building no' as a property:

let address = {

'building no': 3960,

street: 'North 1st street',

state: 'CA',

country: 'USA'

};

Code language: JavaScript (javascript)

To access the 'building no' property, you need to use the array-like notation:

address['building no'];

Code language: CSS (css)

If you use the dot notation, you’ll get an error:

address.'building no';

Code language: JavaScript (javascript)

Error:

SyntaxError: Unexpected string

Code language: JavaScript (javascript)

Note that it is not a good practice to use spaces in the property names of an object.

Reading from a property that does not exist will result in an [undefined](https://www.javascripttutorial.net/javascript-undefined/). For example:

console.log(address.district);

Code language: CSS (css)

Output:

undefined

Code language: JavaScript (javascript)

## **Modifying the value of a property**

To change the value of a property, you use the [assignment operator](https://www.javascripttutorial.net/javascript-assignment-operators/) (=). For example:

let person = {

firstName: 'John',

lastName: 'Doe'

};

person.firstName = 'Jane';

console.log(person);

Code language: JavaScript (javascript)

Output:

{ firstName: 'Jane', lastName: 'Doe' }

Code language: CSS (css)

In this example, we changed the value of the firstName property of the person object from 'John' to 'Jane'.

## **Adding a new property to an object**

Unlike objects in other programming languages such as Java and [C#](https://www.csharptutorial.net/csharp-tutorial/csharp-class/), you can add a property to an object after object creation.

The following statement adds the age property to the person object and assigns 25 to it:

person.age = 25;

## **Deleting a property of an object**

To delete a property of an object, you use the delete operator:

delete objectName.propertyName;

Code language: JavaScript (javascript)

The following example removes the age property from the person object:

delete person.age;

Code language: JavaScript (javascript)

If you attempt to reaccess the age property, you’ll get an undefined value.

## **Checking if a property exists**

To check if a property exists in an object, you use the in operator:

propertyName in objectName

The in operator returns true if the propertyName exists in the objectName.

The following example creates an employee object and uses the in operator to check if the ssn and employeeId properties exist in the object:

let employee = {

firstName: 'Peter',

lastName: 'Doe',

employeeId: 1

};

console.log('ssn' in employee);

console.log('employeeId' in employee);

Code language: JavaScript (javascript)

Output:

false

true

Code language: JavaScript (javascript)

## **Summary**

* An object is a collection of key-value pairs.
* Use the dot notation ( .) or array-like notation ([]) to access a property of an object.
* The delete operator removes a property from an object.
* The in operator check if a property exists in an object.

# JavaScript Primitive vs. Reference Values

**Summary**: in this tutorial, you’ll learn about two different types of values in JavaScript including primitive and reference values.

JavaScript has two different types of values:

* Primitive values
* Reference values

Primitive values are atomic pieces of data while reference values are objects that might consist of multiple values.

## **Stack and heap memory**

When you declare [variables](https://www.javascripttutorial.net/javascript-variables/), the JavaScript engine allocates the memory for them on two memory locations: stack and heap.

Static data is the data whose size is fixed at compile time. Static data includes:

* Primitive values ([null](https://www.javascripttutorial.net/object/javascript-null/), [undefined](https://www.javascripttutorial.net/javascript-undefined/), [boolean](https://www.javascripttutorial.net/javascript-boolean-type/), [number](https://www.javascripttutorial.net/javascript-number/), [string](https://www.javascripttutorial.net/string/), [symbol](https://www.javascripttutorial.net/es6/symbol/), and [BigInt](https://www.javascripttutorial.net/es-next/javascript-bigint/))
* Reference values that refer to objects.

Because static data has a size that does not change, the JavaScript engine allocates a fixed amount of memory space to the static data and store it on the stack.

For example, the following declares two variables and initializes their values to a literal string and a number:

let name = 'John';

let age = 25;

Code language: JavaScript (javascript)

Because name and age are primitive values, the JavaScript engine stores these variables on the stack as shown in the following picture:

Note that strings are objects in many programming languages, including Java and C#. However, strings are primitive values in JavaScript.

Unlike the stack, JavaScript stores objects (and functions) on the heap. The JavaScript engine doesn’t allocate a fixed amount of memory for these objects. Instead, it’ll allocate more space as needed.

The following example defines the name, age, and person variables:

let name = 'John';

let age = 25;

let person = {

name: 'John',

age: 25,

};

Code language: JavaScript (javascript)

Internally, the JavaScript engine allocates the memory as shown in the following picture:

In this picture, JavaScript allocates memory on the stack for the three variables name, age, and person.

The JavaScript engine creates a new object on the heap memory. Also, it links the person variable on the stack memory to the object on the heap memory.

Because of this, we say that the person variable is a reference that refers to an object.

## **Dynamic properties**

A reference value allows you to add, change, or delete properties at any time. For example:

let person = {

name: 'John',

age: 25,

};

*// add the ssn property*

person.ssn = '123-45-6789';

*// change the name*

person.name = 'John Doe';

*// delete the age property*

delete person.age;

console.log(person);

Code language: JavaScript (javascript)

Output:

{ name: 'John Doe', ssn: '123-45-6789' }

Code language: CSS (css)

Unlike a reference value, a primitive value cannot have properties. This means that you cannot add a property to a primitive value.

JavaScript allows you to add a property to a primitive value. However, it won’t take any effect. For example:

let name = 'John';

name.alias = 'Knight';

console.log(name.alias); *// undefined*

Code language: JavaScript (javascript)

Output:

undefined

Code language: JavaScript (javascript)

In this example, we add the alias property to the name primitive value. But when we access the alias property via the name primitive value, it returns undefined.

## **Copying values**

When you assign a primitive value from one variable to another, the JavaScript engine creates a copy of that value and assigns it to the variable. For example:

let age = 25;

let newAge = age;

Code language: JavaScript (javascript)

In this example:

* First, declare a new variable age and initialize its value with 25.
* Second, declare another variable newAge and assign the age to the newAge variable.

Behind the scene, the JavaScript engine creates a copy of the primitive value 25 and assign it to the newAge variable.

The following picture illustrates the stack memory after the assignment:

On the stack memory, the newAge and age are separate variables. If you change the value of one variable, it won’t affect the other.

For example:

let age = 25;

let newAge = age;

newAge = newAge + 1;

console.log(age, newAge);

Code language: JavaScript (javascript)

When you assign a reference value from one variable to another, the JavaScript engine creates a reference so that both variables refer to the same object on the heap memory. This means that if you change one variable, it’ll affect the other.

For example:

let person = {

name: 'John',

age: 25,

};

let member = person;

member.age = 26;

console.log(person);

console.log(member);

Code language: JavaScript (javascript)

How it works.

First, declare a person variable and initialize its value with an object with two properties name and age.

Second, assign the person variable to the member variable. In the memory, both variables reference the same object, as shown in the following picture:

Third, change the age property of the object via the member variable:

Since both person and member variables reference the same object, changing the object via the member variable is also reflected in the person variable.

## **Summary**

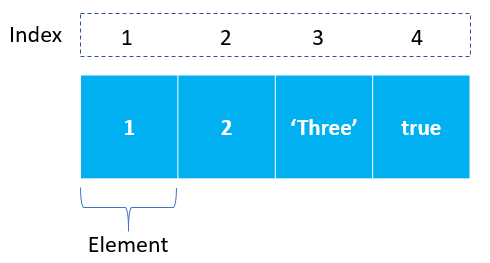
* Javascript has two types of values: primitive values and reference values.
* You can add, change, or delete properties to a reference value, whereas you cannot do it with a primitive value.
* Copying a primitive value from one variable to another creates a separate value copy. It means that changing the value in one variable does not affect the other.
* Copying a reference from one variable to another creates a reference so that two variables refer to the same object. This means that changing the object via one variable reflects in another variable.

# JavaScript Arrays

**Summary**: in this tutorial, you’ll learn about JavaScript arrays and their basic operations.

## **Introduction to JavaScript arrays**

In JavaScript, an array is an ordered list of values. Each value is called an element specified by an index.



An JavaScript array has the following characteristics:

1. First, an array can hold values of mixed types. For example, you can have an array that stores elements with the types number, string, and boolean.
2. Second, the size of an array is dynamic and auto-growing. In other words, you don’t need to specify the array size upfront.

## **Creating JavaScript arrays**

JavaScript provides you with two ways to create an array. The first one is to use the Array constructor as follows:

let scores = new Array();

Code language: JavaScript (javascript)

The scores array is empty, which does hold any elements.

If you know the number of elements that the array will hold, you can create an array with an initial size as shown in the following example:

let scores = Array(10);

Code language: JavaScript (javascript)

To create an array and initialize it with some elements, you pass the elements as a comma-separated list into the Array() constructor.

For example, the following creates the scores array that has five elements (or numbers):

let scores = new Array(9,10,8,7,6);

Code language: JavaScript (javascript)

Note that if you use the Array() constructor to create an array and pass a number into it, you are creating an array with an initial size.

However, when you pass a value of another type like string into the Array() constructor, you create an array with an element of that value. For example:

let athletes = new Array(3); *// creates an array with initial size 3*

let scores = new Array(1, 2, 3); *// create an array with three numbers 1,2 3*

let signs = new Array('Red'); *// creates an array with one element 'Red'*

Code language: JavaScript (javascript)

JavaScript allows you to omit the new operator when you use the Array() constructor. For example, the following statement creates the artists array.

let artists = Array();

Code language: JavaScript (javascript)

In practice, you’ll rarely use the Array() constructor to create an array.

The more preferred way to create an array is to use the array literal notation:

let arrayName = [element1, element2, element3, ...];

Code language: JavaScript (javascript)

The array literal form uses the square brackets [] to wrap a comma-separated list of elements.

The following example creates the colors array that holds string elements:

let colors = ['red', 'green', 'blue'];

Code language: JavaScript (javascript)

To create an empty array, you use square brackets without specifying any element like this:

let emptyArray = [];

Code language: JavaScript (javascript)

## **Accessing JavaScript array elements**

JavaScript arrays are zero-based indexed. In other words, the first element of an array starts at index 0, the second element starts at index 1, and so on.

To access an element in an array, you specify an index in the square brackets []:

arrayName[index]

Code language: CSS (css)

The following shows how to access the elements of the mountains array:

let mountains = ['Everest', 'Fuji', 'Nanga Parbat'];

console.log(mountains[0]); *// 'Everest'*

console.log(mountains[1]); *// 'Fuji'*

console.log(mountains[2]); *// 'Nanga Parbat'*

Code language: JavaScript (javascript)

To change the value of an element, you assign that value to the element like this:

let mountains = ['Everest', 'Fuji', 'Nanga Parbat'];

mountains[2] = 'K2';

console.log(mountains);

Code language: JavaScript (javascript)

Output:

[ 'Everest', 'Fuji', 'K2' ]

Code language: JSON / JSON with Comments (json)

## **Getting the array size**

Typically, the [length](https://www.javascripttutorial.net/javascript-array-length/) property of an array returns the number of elements. The following example shows how to use the length property:

let mountains = ['Everest', 'Fuji', 'Nanga Parbat'];

console.log(mountains.length); *// 3*

Code language: JavaScript (javascript)

## **Basic operations on arrays**

The following explains some basic operations on arrays. And you’ll learn advanced operations such as [map()](https://www.javascripttutorial.net/javascript-array-map/), [filter()](https://www.javascripttutorial.net/javascript-array-filter/), and [reduce()](https://www.javascripttutorial.net/javascript-array-reduce/) in the next tutorials.

### **1) Adding an element to the end of an array**

To add an element to the end of an array, you use the push() method:

let seas = ['Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea'];

seas.push('Red Sea');

console.log(seas);

Code language: JavaScript (javascript)

Output:

[ 'Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea', 'Red Sea' ]

Code language: JSON / JSON with Comments (json)

### **2) Adding an element to the beginning of an array**

To add an element to the beginning of an array, you use the unshift() method:

let seas = ['Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea'];

seas.unshift('Red Sea');

console.log(seas);

Code language: JavaScript (javascript)

Output:

[ 'Red Sea', 'Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea' ]

Code language: JSON / JSON with Comments (json)

### **3) Removing an element from the end of an array**

To remove an element from the end of an array, you use the pop() method:

let seas = ['Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea'];

const lastElement = seas.pop();

console.log(lastElement);

Code language: JavaScript (javascript)

Output:

Baltic Sea

### **4) Removing an element from the beginning of an array**

To remove an element from the beginning of an array, you use the shift() method:

let seas = ['Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea'];

const firstElement = seas.shift();

console.log(firstElement);

Code language: JavaScript (javascript)

Output:

Black Sea

### **5) Finding an index of an element in the array**

To find the index of an element, you use the [indexOf()](https://www.javascripttutorial.net/javascript-string-indexof/) method:

let seas = ['Black Sea', 'Caribbean Sea', 'North Sea', 'Baltic Sea'];

let index = seas.indexOf('North Sea');

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

Code language: JavaScript (javascript)

### **6) Check if an value is an array**

To check if a value is an array, you use [Array.isArray()](https://www.javascripttutorial.net/array/how-to-check-if-a-variable-is-an-array-in-javascript/) method:

console.log(Array.isArray(seas)); *// true*

Code language: JavaScript (javascript)

## **Summary**

* In JavaScript, an array is an order list of values. Each value is called an element specified by an index.
* An array can hold values of mixed types.
* JavaScript arrays are dynamic, which mean that they grow or shrink as needed.

# JavaScript Arithmetic Operators

**Summary**: in this tutorial, you will learn how to use JavaScript arithmetic operators to perform arithmetic calculations.

## **Introduction to the JavaScript Arithmetic Operators**

JavaScript supports the following standard arithmetic operators:

| **Operator** | **Sign** |
| --- | --- |
| Addition | + |
| Subtraction | - |
| Multiplication | \* |
| Division | / |

An arithmetic operator accepts numerical values as operands and returns a single numerical value. The numerical values can be literals or variables.

## **Addition operator (+)**

The addition operator returns the sum of two values. For example, the following uses the addition operator to calculate the sum of two numbers:

let sum = 10 + 20;

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

Code language: JavaScript (javascript)

Also, you can use the addition operator with two [variables](https://www.javascripttutorial.net/javascript-variables/). For example:

let netPrice = 9.99,

shippingFee = 1.99;

let grossPrice = netPrice + shippingFee;

console.log(grossPrice);

Code language: JavaScript (javascript)

Output:

11.98

Code language: CSS (css)

If either value is a [string](https://www.javascripttutorial.net/string/), the addition operator uses the following rules:

* If both values are strings, it concatenates the second string to the first one.
* If one value is a string, it implicitly converts the numeric value into a string and conatenate two strings.

For example, the following uses the addition operator to add concatenate two strings:

let x = '10',

y = '20';

let result = x + y;

console.log(result);

Code language: JavaScript (javascript)

Output:

1020

The following example shows how to use the addition operator to calculate the sum of a number and a string:

let result = 10 + '20';

console.log(result);

Code language: JavaScript (javascript)

Output:

1020

In this example, JavaScript converts the number 10 into a string '10' and concatenates the second string '20' to it.

The following table shows the result when using the addition operator with special numbers:

| **First Value** | **Second Value** | **Result** | **Explanation** |
| --- | --- | --- | --- |
| NaN |  | NaN | If either value is NaN, the result is NaN |
| Infinity | Infinity | Infinity | Infinity + Infinity = Infinity |
| -Infinity | -Infinity | -Infinity | -Infinity + ( -Infinity) = – Infinity |
| Infinity | -Infinity | NaN | Infinity + -Infinity = NaN |
| +0 | +0 | +0 | +0 + (+0) = +0 |
| -0 | +0 | +0 | -0 + (+0) = +0 |
| -0 | -0 | -0 | -0 + (-0) = -0 |

## **Subtraction operator (-)**

The subtraction operator (-) subtracts one number from another. For example:

let result = 30 - 10;

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

Code language: JavaScript (javascript)

If a value is a string, a boolean, null, or undefined, the JavaScript engine will:

* First, convert the into a number using the Number() function.
* Second, perform the subtraction.

The following table shows how to use the subtraction operator with special values:

| **First Value** | **Second Value** | **Result** | **Explanation** |
| --- | --- | --- | --- |
| NaN |  | NaN | If either value is NaN, the result is NaN |
| Infinity | Infinity | NaN | Infinity – Infinity = NaN |
| -Infinity | -Infinity | NaN | -Infinity – ( -Infinity) = NaN |
| Infinity | -Infinity | -Infinity | -Infinity |
| +0 | +0 | +0 | +0 – (+0) = +0 |
| +0 | -0 | -0 | +0 – (-0) = -0 |
| -0 | -0 | +0 | -0 – (-0) = +0 |

## **Multiplication operator (\*)**

JavaScript uses the asterisk (\*) to represent the multiplication operator. The multiplication operator multiplies two numbers and returns a single value. For example:

let result = 2 \* 3;

console.log(result);

Code language: JavaScript (javascript)

Output:

6

If either value is not a number, the JavaScript engine implicitly converts it into a number using the Number() function and performs the multiplication. For example:

let result = '5' \* 2;

console.log(result);

Code language: JavaScript (javascript)

Output:

10

The following table shows how the multiply operator behaves with special values:

| **First Value** | **Second Value** | **Result** | **Explanation** |
| --- | --- | --- | --- |
| NaN |  | NaN | If either value is NaN, the result is NaN |
| Infinity | 0 | NaN | Infinity \* 0 = NaN |
| Infinity | Positive number | Infinity | -Infinity \* 100 = Infinity |
| Infinity | Negative number | -Infinity | Infinity \* (-100) = -Infinity |
| Infinity | Infinity | Infinity | Infinity \* Infinity = Infinity |

## **Divide operator (/)**

Javascript uses the slash (/) character to represent the divide operator. The divide operator divides the first value by the second one. For example:

let result = 20 / 10;

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

Code language: JavaScript (javascript)

If either value is not a number, the JavaScript engine converts it into a number for division. For example:

let result = '20' / 2;

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

Code language: JavaScript (javascript)

The following table shows the divide operators’ behavior when applying to special values:

| **First Value** | **Second Value** | **Result** | **Explanation** |
| --- | --- | --- | --- |
| NaN |  | NaN | If either value is NaN, the result is NaN |
| A number | 0 | Infinity | 1/0 = Infinity |
| Infinity | Infinity | Infinity | Infinity / Infinity = NaN |
| 0 | 0 | NaN | 0/0 = NaN |
| Infinity | A positive number | Infinity | Infinity / 2 = Infinity |
| Infinity | A negative number | -Infinity | Infinity / -2 = -Infinity |

## **Using JavaScript arithmetic operators with objects**

If a value is an [object](https://www.javascripttutorial.net/javascript-objects/), the JavaScript engine will call the valueOf() method of the object to get the value for calculation. For example:

let energy = {

valueOf() {

return 100;

},

};

let currentEnergy = energy - 10;

console.log(currentEnergy);

currentEnergy = energy + 100;

console.log(currentEnergy);

currentEnergy = energy / 2;

console.log(currentEnergy);

currentEnergy = energy \* 1.5;

console.log(currentEnergy);

Code language: JavaScript (javascript)

Output:

90

200

50

150

If the object doesn’t have the valueOf() method but has the toString() method, the JavaScript engine will call the toString() method to get the value for calculation. For example:

let energy = {

toString() {

return 50;

},

};

let currentEnergy = energy - 10;

console.log(currentEnergy);

currentEnergy = energy + 100;

console.log(currentEnergy);

currentEnergy = energy / 2;

console.log(currentEnergy);

currentEnergy = energy \* 1.5;

console.log(currentEnergy);

Code language: JavaScript (javascript)

Output:

40

150

25

75

## **Summary**

* Use the JavaScript arithmetic operators including addition (+), subtraction (-), multiply (\*) and divide (/) to perform arithmetic opeations.

# JavaScript Remainder Operator

**Summary**: in this tutorial, you’ll learn about the JavaScript remainder operator (%) to get the remainder of a number divided by another number.

## **Introduction to the JavaScript remainder operator**

JavaScript uses the % to represent the remainder operator. The remainder operator returns the remainder left over when one value is divided by another value.

Here’s the syntax of the remainder operator:

dividend % divisor

The following shows the equation for the remainder:

dividend = divisor \* quotient + remainder

where |remainder| < |divisor|

In this equation, the dividend, divisor, quotient, and remainder are all integers. The sign of the remainder is the same as the sign of the dividend.

The sign of the remainder is the same as the sign of the dividend.

## **JavaScript remainder operator examples**

Let’s take some examples of using the JavaScript remainder operator.

### **1) Using the remainder operator with positive dividend example**

The following example shows how to use the remainder operator with a positive dividend:

let remainder = 5 % -2;

console.log(remainder); *// 1*

remainder = 5 % 2;

console.log(remainder); *// 1*

Code language: JavaScript (javascript)

### **2) Using the remainder operator with negative dividend example**

The following example uses the remainder operator with a negative dividend:

let remainder = -5 % 3;

console.log(remainder); *// -2*

remainder = -5 % -3;

console.log(remainder); *// -2*

Code language: JavaScript (javascript)

### **3) Using the remainder operator special values**

If a dividend is an Infinity and a divisor is a finite number, the remainder is NaN. For example:

let remainder = Infinity % 2;

console.log(remainder); *// NaN*

Code language: JavaScript (javascript)

If a dividend is a finite number and a divisor is zero, the remainder is NaN:

let remainder = 10 % 0;

console.log(remainder); *// NaN*

Code language: JavaScript (javascript)

If both dividend and divisor are Infinity, the remainder is NaN:

let remainder = Infinity % Infinity;

console.log(remainder); *// NaN*

Code language: JavaScript (javascript)

If a dividend is a finite number and the divisor is an Infinity, the remainder is the dividend. For example:

let remainder = 10 % Infinity;

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

Code language: JavaScript (javascript)

If the dividend is zero and the divisor is non-zero, the remainder is zero:

let remainder = 0 % 10;

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

Code language: JavaScript (javascript)

If either dividend or divisor is not a number, it’s converted to a number using the Number() function and applied the above rules. For example:

let remainder = '10' % 3;

console.log(remainder); *// 1*

Code language: JavaScript (javascript)

## **Using the remainder operator to check if a number is an odd number**

To check if a number is an odd number, you use the remainder operator (%) like the following example:

let num = 13;

let isOdd = num % 2;

Code language: JavaScript (javascript)

In this example, if the num is an odd number, the remainder is one. But if the num is an even number, the remainder is zero.

Later, you’ll learn how to [define a function](https://www.javascripttutorial.net/javascript-function/) that returns true if a number is odd or false otherwise like this:

function isOdd(num) {

return num % 2;

}

Code language: JavaScript (javascript)

Or using an [arrow function in ES6](https://www.javascripttutorial.net/es6/javascript-arrow-function/):

const isOdd = (num) => num % 2;

Code language: JavaScript (javascript)

## **Remainder vs Modulo operator**

In JavaScript, the remainder operator (%) is not the modulo operator.

If you have been working with Python, you may find the % represents the modulo operator in this language. However, it is not the case in JavaScript.

To get a modulo in JavaScript, you use the following expression:

((dividend % divisor) + divisor) % divisor

Or wrap it in a function:

const mod = (dividend, divisor) => ((dividend % divisor) + divisor) % divisor;

Code language: JavaScript (javascript)

If the division and divisor have the same sign, the remainder and modulo operators return the same result. Otherwise, they return different results.

For example:

const mod = (dividend, divisor) => ((dividend % divisor) + divisor) % divisor;

*// dividen and divisor have the same sign*

console.log('remainder:', 5 % 3); *// 2*

console.log('modulo:', mod(5, 3)); *// 2*

*// dividen and divisor have the different signs*

console.log('remainder:', -5 % 3); *// -2*

console.log('modulo:', mod(-5, 3)); *// 1*

Code language: JavaScript (javascript)

Output:

remainder: 2

modulo: 2

remainder: -2

modulo: 1

Code language: HTTP (http)

## **Summary**

* Use the JavaScript remainder operator (%) get the the remainder of a value divided by another value.

# JavaScript Assignment Operators

**Summary**: in this tutorial, you will learn how to use JavaScript assignment operators to assign a value to a variable.

## **Introduction to JavaScript assignment operators**

An assignment operator (=) assigns a value to a variable. The syntax of the assignment operator is as follows:

let a = b;

Code language: JavaScript (javascript)

In this syntax, JavaScript evaluates the expression b first and assigns the result to the variable a.

The following example declares the counter variable and initializes its value to zero:

let counter = 0;

Code language: JavaScript (javascript)

The following example increases the counter variable by one and assigns the result to the counter variable:

let counter = 0;

counter = counter + 1;

Code language: JavaScript (javascript)

When evaluating the second statement, JavaScript evaluates the expression on the right hand first (counter + 1) and assigns the result to the counter variable. After the second assignment, the counter variable is 1.

To make the code more concise, you can use the += operator like this:

let counter = 0;

counter += 1;

Code language: JavaScript (javascript)

In this syntax, you don’t have to repeat the counter variable twice in the assignment.

The following table illustrates assignment operators that are shorthand for another operator and the assignment:

| **Operator** | **Meaning** | **Description** |
| --- | --- | --- |
| a = b | a = b | Assigns the value of b to a. |
| a += b | a = a + b | Assigns the result of a plus b to a. |
| a -= b | a = a - b | Assigns the result of a minus b to a. |
| a \*= b | a = a \* b | Assigns the result of a times b to a. |
| a /= b | a = a / b | Assigns the result of a divided by b to a. |
| a %= b | a = a % b | Assigns the result of a modulo b to a. |
| a &=b | a = a & b | Assigns the result of a AND b to a. |
| a |=b | a = a | b | Assigns the result of a OR b to a. |
| a ^=b | a = a ^ b | Assigns the result of a XOR b to a. |
| a <<= b | a = a << b | Assigns the result of a shifted left by b to a. |
| a >>= b | a = a >> b | Assigns the result of a shifted right (sign preserved) by b to a. |
| a >>>= b | a = a >>> b | Assigns the result of a shifted right by b to a. |

## **Chaining JavaScript assignment operators**

If you want to assign a single value to multiple variables, you can chain the assignment operators. For example:

let a = 10, b = 20, c = 30;

a = b = c; *// all variables are 30*

Code language: JavaScript (javascript)

In this example, JavaScript evaluates from right to left. Therefore, it does the following:

let a = 10, b = 20, c = 30;

b = c; *// b is 30*

a = b; *// a is also 30*

Code language: JavaScript (javascript)

## **Summary**

* Use the assignment operator (=) to assign a value to a variable.
* Chain the assignment operators if you want to assign a single value to multiple variables.

# JavaScript Unary Operators Explained

**Summary**: in this tutorial, you will learn how to use JavaScript unary operators that take a single operand and performs an operation.

## **Introduction to the JavaScript unary operators**

Unary operators work on one value. The following table shows the unary operators and their meanings:

| **Unary Operators** | **Name** | **Meaning** |
| --- | --- | --- |
| +x | Unary Plus | Convert a value into a number |
| -x | Unary Minus | Convert a value into a number and negate it |
| ++x | Increment Operator (Prefix) | Add one to the value |
| –x | Decrement Operator (Prefix) | Subtract one from the value |
| x++ | Increment Operator (Postfix) | Add one to the value |
| x– | Decrement Operator (Postfix) | Subtract one from the value |

## **Unary plus (+)**

The unary plus operator is a simple plus sign (+). If you place the unary plus before a numeric value, it does nothing. For example

let x = 10;

let y = +x;

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

Code language: JavaScript (javascript)

When you apply the unary plus operator to a non-numeric value, it performs a number conversion using the Number() function wit the rules in the following table:

| **Value** | **Result** |
| --- | --- |
| boolean | false to 0, true to 1 |
| string | Convert the string value based on a set of specific rules |
| object | Call the valueOf() and/or toString() method to get the value to convert into a number |

For example, the following uses the unary plus operator to convert the string '10' to the number 10:

let s = '10';

console.log(+s); *// 10*

Code language: JavaScript (javascript)

The following example uses the unary plus operator (+) converts a boolean value into a number, false to 0 and true to 1.

let f = false,

t = true;

console.log(+f); *// 0*

console.log(+t); *// 1*

Code language: JavaScript (javascript)

Output:

0

1

Suppose you have a product object with the toString() method as follows:

let person = {

name: 'John',

toString: function () {

return '25';

},

};

console.log(+person);

Code language: JavaScript (javascript)

Output:

25

In this example, we apply the unary plus operator (+) on the person object that has the toString() method, JavaScript engine calls toString() method to get the value ('25') and convert it into a number.

The following adds the valueOf() method to the person object:

let person = {

name: 'John',

toString: function () {

return '25';

},

valueOf: function () {

return '30';

},

};

console.log(+person);

Code language: JavaScript (javascript)

Output:

30

In this example, the person object has the valueOf() method, the JavaScript engine calls it instead of the toString() method to get the value to convert.

## **Unary minus (-)**

The unary minus operator is a single minus sign (-). If you apply the unary minus operator to a number, it negates the number. For example:

let x = 10;

let y = -x;

console.log(y); *// -10*

Code language: JavaScript (javascript)

If you apply the unary minus operator to a non-numeric value, it converts the value into a number using the same rules as the unary plus operator and then negates the value.

## **Increment / Decrement operators**

The increment operator has two plus signs (++) while the decrement operator has two minus signs (--).

Both increment and decrement operators have two versions: prefix and postfix. And you place the prefix and postfix versions of the increment or decrement operators before and after the variable to which they apply.

The following example uses the prefix increment operator to add one to a variable:

let age = 25;

++age;

console.log(age); *// 26*

Code language: JavaScript (javascript)

It’s equivalent to the following:

let age = 25;

age = age + 1;

console.log(age); *// 26*

Code language: JavaScript (javascript)

The following example uses the prefix decrement operator to subtract one from a variable:

let weight = 90;

--weight;

console.log(weight); *// 89*

Code language: JavaScript (javascript)

It is equivalent to the following:

let weight = 90;

weight = weight - 1;

console.log(weight); *// 89*

Code language: JavaScript (javascript)

When you apply the prefix increment or decrement, JavaScript changes the variable before evaluating the statement. For example:

let weight = 90;

weight = ++weight + 5;

console.log(weight); *// 96*

Code language: JavaScript (javascript)

In this example:

* First, increase the weight on the right-hand side so ++weight is 91
* Second, add five to the ++weight that returns 96
* Third, assign the result to the weight on the left-hand side.

Likewise, the following example uses a prefix decrement operator:

let weight = 90;

weight = --weight + 5;

console.log(weight); *// 94*

Code language: JavaScript (javascript)

In this example:

* First, subtract one from the weight, –weight returns 89
* Second, add five to the –weight that returns 94
* Third, assign the result to the weight on the left-hand side.

The postfix increment or decrement operator changes the value after the statement is evaluated. For example:

let weight = 90;

let newWeight = weight++ + 5;

console.log(newWeight); *// 95*

console.log(weight); *// 91*

Code language: JavaScript (javascript)

How it works.

* First, add five to weight (90) and assign the result to the new weight (95)
* Second, add one to the weight variable after the second statement completes, the weight becomes 91.
* Third, output both newWeight and weight to the console.

When applying the increment/decrement operator to a non-numeric value, it performs the following steps :

* First, convert the value into a number using the same rules as the unary plus (+) operator.
* Then, add one to or subtract one from the value.

## **Summary**

* Unary operators work on one value.
* Unary plus (+) or minus (-) converts a non-numeric value into a number. The unary minus negates the value after the conversion.
* The prefix increment operator adds one to a value. The value is changed before the statement is evaluted.
* The postfix increment operator adds one to a value. The value is changed after the statement is evaluted.
* The prefix decrement operator subtracts one from a value. The value is changed before the statement is evaluted.
* The postfix decrement operator subtracts one from a value. The value is changed after the statement is evaluted.

# JavaScript Comparison Operators

**Summary**: in this tutorial, you will learn how to use JavaScript comparison operators to compare two values.

## **Introduction to JavaScript comparison operators**

To compare two values, you use a comparison operator. The following table shows the comparison operators in JavaScript:

| **Operator** | **Meaning** |
| --- | --- |
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |
| == | equal to |
| != | not equal to |

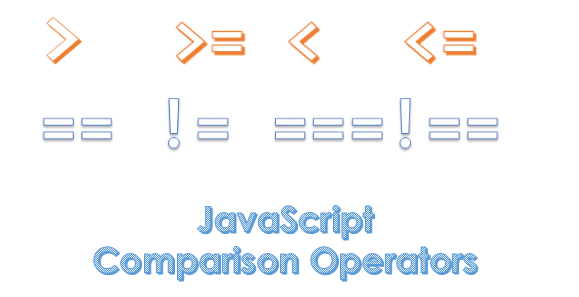
A comparison operator returns a [Boolean](https://www.javascripttutorial.net/javascript-boolean/) value indicating that the comparison is true or not. See the following example:

let r1 = 20 > 10; *// true*

let r2 = 20 < 10; *// false*

let r3 = 10 == 10; *// true*

Code language: JavaScript (javascript)



A comparison operator takes two values. If the types of the values are not comparable, the comparison operator converts them into values of comparable types according to specific rules.

### **Compare numbers**

If values are numbers, the comparison operators perform a numeric comparison. For example:

let a = 10,

b = 20;

console.log(a >= b); *// false*

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

Code language: JavaScript (javascript)

This example is straightforward. The variable a is 10, b is 20. The expression a >= b expression returns false and the expression a == 10 expression returns true.

### **Compare strings**

If the operands are strings, JavaScript compares the character codes numerically one by one in the string.

let name1 = 'alice',

name2 = 'bob';

let result = name1 < name2;

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

console.log(name1 == 'alice'); *// true*

Code language: JavaScript (javascript)

Because JavaScript compares the character codes in the strings numerically, you may receive an unexpected result, for example:

let f1 = 'apple',

f2 = 'Banana';

let result = f2 < f1;

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

Code language: JavaScript (javascript)

In this example, f2 is less than f1 because the letter B has the character code 66 while the letter a has the character code 97.

To fix this, you need to:

* First, convert the strings into a common format, either lowercase or uppercase
* Second, compare the converted values

For example:

let f1 = 'apple',

f2 = 'Banana';

let result = f2.toLowerCase() < f1.toLowerCase();

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

Code language: JavaScript (javascript)

Note that the toLowerCase() is a method of the String object that converts the string to lowercase.

### **Compare a number with a value of another type**

If a value is a number and the other is not, the comparison operator will convert the non-numeric value into a number and compare them numerically. For example:

console.log(10 < '20'); *// true*

Code language: JavaScript (javascript)

In this example, the comparison operator converts the string '20' into the number 20 and compares with the number 10. Here is an example:

console.log(10 == '10'); *// true*

Code language: JavaScript (javascript)

In this example, the comparison operator converts the string '10' into the number 10 and compares them numerically.

### **Compare an object with a non-object**

If a value is an object, the valueOf() method of that object is called to return the value for comparison. If the object doesn’t have the valueOf() method, the toString() method is called instead. For example:

let apple = {

valueOf: function () {

return 10;

},

};

let orange = {

toString: function () {

return '20';

},

};

console.log(apple > 10); *// false*

console.log(orange == 20); *// true*

Code language: JavaScript (javascript)

In this first comparison, the apple object has the valueOf() method that returns 10. Therefore, the comparison operator uses the number 10 for comparison.

In the second comparison, JavaScript first calls the valueOf() method. However, the orange object doesn’t have the valueOf() method. So JavaScript calls the toString() method to get the returned value of 20 for comparison.

### **Compare a Boolean with another value**

If a value is a Boolean value, JavaScript converts it to a number and compares the converted value with the other value; true is converted to 1 and false is converted to 0. For example:

console.log(true > 0); *// true*

console.log(false < 1); *// true*

console.log(true > false); *// true*

console.log(false > true); *// false*

console.log(true >= true); *// true*

console.log(true <= true); *// true*

console.log(false <= false); *// true*

console.log(false >= false); *// true*

Code language: JavaScript (javascript)

In addition to the above rules, the equal (==) and not-equal(!=) operators also have the following rules.

### **Compare null and undefined**

In JavaScript, null equals undefined. It means that the following expression returns true.

console.log(null == undefined); *// true*

Code language: JavaScript (javascript)

### **Compare NaN with other values**

If either value is NaN, then the equal operator(==) returns false.

console.log(NaN == 1); *// false*

Code language: JavaScript (javascript)

Even

console.log(NaN == NaN); *// false*

Code language: JavaScript (javascript)

The not-equal (!=) operator returns true when comparing the NaN with another value:

console.log(NaN != 1); *// true*

Code language: JavaScript (javascript)

And also

console.log(NaN != NaN); *// true*

Code language: JavaScript (javascript)

## **Strict equal (===) and not strict equal (!==)**

Besides the comparison operators above, JavaScript provides the strict equal ( ===) and not strict equal  ( !==) operators.

| **Operator** | **Meaning** |
| --- | --- |
| === | strict equal |
| !== | not strict equal |

The strict equal and not strict equal operators behave like the equal and not equal operators except that they don’t convert the operand before comparison. See the following example:

console.log("10" == 10); *// true*

console.log("10" === 10); *// false*

Code language: JavaScript (javascript)

In the first comparison, since we use the equality operator, JavaScript converts the string into the number and performs the comparison.

However, in the second comparison, we use the strict equal operator ( ===), JavaScript doesn’t convert the string before comparison, therefore the result is false.

In this tutorial, you have learned how to use the JavaScript comparison operators to compare values.

# An Introduction to JavaScript Logical Operators

**Summary**: in this tutorial, you will learn how to use the JavaScript logical operators including the logical NOT operator( !), the logical AND operator ( &&) and the logical OR operator ( ||).

The logical operators are important in JavaScript because they allow you to compare variables and do something based on the result of that comparison.

For example, if the result of the comparison is true, you perform a block of code; if it’s false, you perform another block of code.



JavaScript provides three logical operators:

* ! (Logical NOT)
* || (Logical OR)
* && (Logical AND)

## **1) The Logical NOT operator (!)**

JavaScript uses an exclamation point ! to represent the logical NOT operator. The ! operator can be applied to a single value of any type, not just a Boolean value.

When you apply the ! operator to a boolean value, the ! returns true if the value is false and returns false if the value if true. For example:

let eligible = false,

required = true;

console.log(!eligible);

console.log(!required);

Code language: JavaScript (javascript)

Output:

true

false

Code language: JavaScript (javascript)

In this example, the eligible is true so !eligible returns false. And because the required is false, the !required returns false.

When you apply the ! operator to a non-Boolean value. The ! operator first converts the value to a [boolean](https://www.javascripttutorial.net/javascript-data-types/" \l "boolean) value and then negates it. For example:

The following example shows how to use the ! operator:

!a

The logical ! operator works based on the following rules:

* If a is undefined, the result is true.
* If a is null, the result is true.
* If a is a number other than 0, the result is false.
* If a is NaN, the result is true.
* If a is null, the result is true.
* If a is an object, the result is false.
* If a is an empty string, the result is true. In case a is a non-empty string, the result is false

The following demonstrates the results of the logical ! operator when we apply it to a non-boolean value:

console.log(!undefined); *// true*

console.log(!null); *// true*

console.log(!20); *//false*

console.log(!0); *//true*

console.log(!NaN); *//true*

console.log(!{}); *// false*

console.log(!''); *//true*

console.log(!'OK'); *//false*

console.log(!false); *//true*

console.log(!true); *//false*

Code language: JavaScript (javascript)

### **Double-negation (!!)**

Sometimes, you may see the double negation (!!) in the code. The !! uses the logical NOT operator (!) twice to convert a value to its real boolean value. The result is the same as using the [Boolean()](https://www.javascripttutorial.net/javascript-boolean/) function. For example:

let counter = 10;

console.log(!!counter); *// true*

Code language: JavaScript (javascript)

The first ! operator returns a Boolean value of the counter variable. And the second one ! negates that result and returns the real boolean value of the counter variable.

## **2) The Logical AND operator (&&)**

JavaScript uses the double ampersand (&&) to represent the logical AND operator.

The following expression uses the && operator:

let result = a && b;

Code language: JavaScript (javascript)

If a can be converted to true, the && operator returns the b; otherwise, it returns the a. In fact, this rule is applied to boolean values.

The following truth table illustrates the result of the && operator when it is applied to two Boolean values:

| **a** | **b** | **a & b** |
| --- | --- | --- |
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

The result of the && operator is true only if both values are true, otherwise, it is false. For example:

let eligible = false,

required = true;

console.log(eligible && required); *// false*

Code language: JavaScript (javascript)

In this example, the eligible is false, therefore, the value of the expression eligible && required is false.

See the following example:

let eligible = true,

required = true;

console.log(eligible && required); *// true*

Code language: JavaScript (javascript)

In this example, both eligible and required are true, therefore, the value of the expression eligible && required is false.

### **Short-circuit evaluation**

The && operator is short-circuited. It means that the && operator evaluates the second value only if the first one doesn’t suffice to determine the value of an expression. For example:

let b = true;

let result = b && (1 / 0);

console.log(result);

Code language: JavaScript (javascript)

Output:

Infinity

Code language: JavaScript (javascript)

In this example, b is true therefore the && operator could not determine the result without further evaluating the second expression (1/0).

The result is Infinity which is the result of the expression (1/0). However:

let b = false;

let result = b && (1 / 0);

console.log(result);

Code language: JavaScript (javascript)

Output:

false

Code language: JavaScript (javascript)

In this case, b is false, the && operator doesn’t need to evaluate the second expression because it can determine the final result as false based value of the first value.

### **A Chain of && operators**

The following expression uses multiple && operators:

let result = value1 && value2 && value3;

Code language: JavaScript (javascript)

The && operator carries the following:

* Evaluates values from left to right.
* For each value, converts it to a boolean. If the result is false, stops and returns the original value.
* If all values are truthy values, returns the last value.

In other words, The && operator returns the first falsy value or the last value if none were found.

If a value can be converted to true, it is so-called a truthy value. If a value can be converted to false, it is a so-called falsy value.

## **3) The Logical OR operator (||)**

JavaScript uses the double pipe || to represent the logical OR operator. You can apply the || operator to two values of any type:

let result = a || b;

Code language: JavaScript (javascript)

If a can be converted to true, returns a; else, returns b. This rule is also applied to boolean values.

The following truth table illustrates the result of the || operator based on the value of the operands:

| **a** | **b** | **a || b** |
| --- | --- | --- |
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

The || operator returns false if both values evaluate to false. In case either value is , the || operator returns true. For example:

let eligible = true,

required = false;

console.log(eligible || required); *// true*

Code language: JavaScript (javascript)

The following expression applies the || operator to the two non-boolean values:

In this example, the eligible || required returns true because one of the values (required) is true. See another example:

let eligible = false,

required = false;

console.log(eligible || required); *// false*

Code language: JavaScript (javascript)

In this example, the expression eligible || required returns false because both values are false.

### **The || operator is also short-circuited**

Similar to the && operator, the || operator is short-circuited. It means that if the first value evaluates to true, the && operator doesn’t evaluate the second one.

### **A chain of || operators**

The following example shows how to use multiple || operators in an expression:

let result = value1 || value2 || value3;

Code language: JavaScript (javascript)

The || operator does the following:

* Evaluates values from left to right.
* For each value, converts it to a boolean value. If the result of the conversion is true, stops and returns the value.
* If all values have been evaluated to false, returns the last value.

In other words, the chain of the || operators return the first truthy value or the last one if no truthy value was found.

## **Logical operator precedence**

When you use the mixed logical operators in an expression, the JavaScript engine evaluates the operators based on a specified order, and this order is called the operator precedence. In other words, the operator precedence is the order that an operator is executed.

The precedence of the logical operator is in the following order from the highest to lowest:

1. Logical NOT (!)
2. Logical AND (&&)
3. Logical OR (||)

## **Summary**

* The NOT operator (!) negates a boolean value. The (!!) converts a value into its real boolean value.
* The AND operator (&&) is applied to two Boolean values and returns true if both values are true.
* The OR operator (||) is applied to two Boolean values and returns true if one of the operands is true.
* Both && and || operator are short-circuited. They cab be also applied to non-Boolean values.
* The logical operator precedence from the highest to the lowest is !, && and ||.

# JavaScript Logical Assignment Operators

**Summary**: in this tutorial, you’ll learn about JavaScript logical assignment operators, including the logical OR assignment operator (||=), the logical AND assignment operator (&&=), and the nullish assignment operator (??=).

ES2021 introduces three logical assignment operators including:

* Logical OR assignment operator (||=)
* Logical AND assignment operator (&&=)
* Nullish coalesing assignment operator (??=)

## **The Logical OR assignment operator**

The logical OR assignment operator (||=) accepts two operands and assigns the right operand to the left operand if the left operand is falsy:

x ||= y

In this syntax, the ||= operator only assigns y to x if x is falsy. For example:

let title;

title ||= 'untitled';

console.log(title);

Code language: JavaScript (javascript)

Output:

untitled

In this example, the title variable is undefined, therefore, it’s falsy. Since the title is falsy, the operator ||= assigns the 'untitled' to the title. The output shows the untitled as expected.

See another example:

let title = 'JavaScript Awesome';

title ||= 'untitled';

console.log(title);

Code language: JavaScript (javascript)

Output:

'JavaScript Awesome'

Code language: JavaScript (javascript)

In this example, the title is 'JavaScript Awesome' so it is truthy. Therefore, the logical OR assignment operator (||=) doesn’t assign the string 'untitled' to the title variable.

The logical OR assignment operator:

x ||= y

is equivalent to the following statement that uses the [logical OR operator](https://www.javascripttutorial.net/javascript-logical-operators/):

x || (x = y)

Like the logical OR operator, the logical OR assignment also short-circuits. It means that the logical OR assignment operator only performs an assignment when the x is falsy.

The following example uses the logical assignment operator to display a default message if the search result element is empty:

document.querySelector('.search-result').textContent ||= 'Sorry! No result found';

Code language: JavaScript (javascript)

## **The Logical AND assignment operator**

The logical AND assignment operator only assigns y to x if x is truthy:

x &&= y;

The logical AND assignment operator also short-circuits. It means that

x &&= y;

is equivalent to:

x && (x = y);

The following example uses the logical AND assignment operator to change the last name of a person object if the last name is truthy:

let person = {

firstName: 'Jane',

lastName: 'Doe',

};

person.lastName &&= 'Smith';

console.log(person);

Code language: JavaScript (javascript)

Output:

{firstName: 'Jane', lastName: 'Smith'}

Code language: CSS (css)

## **The nullish coalescing assignment operator**

The nullish coalescing assignment operator only assigns y to x if x is null or undefined:

x ??= y;

It’s equivalent to the following statement that uses the nullish coalescing operator:

x ?? (x = y);

The following example uses the nullish coalescing assignment operator to add a missing property to an object:

let user = {

username: 'Satoshi'

};

user.nickname ??= 'anonymous';

console.log(user);

Code language: JavaScript (javascript)

Output:

{username: 'Satoshi', nickname:'anonymous'}

Code language: CSS (css)

In this example, the user.nickname is undefined, therefore, it’s nullish. The nullish coalescing assignment operator assigns the string 'anonymous' to the user.nickname property.

The following table illustrates how the logical assignment operators work:

## **Summary**

* The logical OR assignment (x ||= y) operator only assigns y to x if x is falsy.
* The logical AND assignment (x &&= y) operator only assigns y to x if x is truthy.
* The nullish coalesing assignment (x ??= y) operator only assigns y to x if x is nullish.

# JavaScript Nullish Coalescing Operator

**Summary**: in this tutorial, you’ll learn about the JavaScript nullish coalescing operator (??) that accepts two values and returns the second value if the first one is null or undefined.

## **Introduction to the JavaScript nullish coalescing operator**

ES2020 introduced the nullish coalescing operator denoted by the double question marks (??). The nullish coalescing operator is a [logical operator](https://www.javascripttutorial.net/javascript-logical-operators/) that accepts two values:

value1 ?? value2

The nullish coalescing operator returns the second value (value2) if the first value (value2) is null or undefined. Technically, the bullish coalescing operator is equivalent to the following block:

const result = value1;

if(result === null || result === undefined) {

result = value2;

}

Code language: JavaScript (javascript)

A nullish value is a value that is either null or undefined.

The following example uses the nullish coalescing operator (??) to return the string 'John' because the first value is null:

const name = null ?? 'John';

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

Code language: JavaScript (javascript)

And this example returns 28 because the first value is undefined:

const age = undefined ?? 28;

console.log(age);

Code language: JavaScript (javascript)

## **Why nullish coalescing operator**

When assigning a default value to a [variable](https://www.javascripttutorial.net/javascript-variables/), you often use the [logical OR operator](https://www.javascripttutorial.net/javascript-logical-operators/) (||). For example:

let count;

let result = count || 1 ;

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

Code language: JavaScript (javascript)

In this example, the count variable is undefined, it is coerced to false. Therefore, the result is 1.

However, the logical OR operator (||) sometimes confuses if you consider 0 or empty strings '' as a valid value like this:

let count = 0;

let result = count || 1;

Code language: JavaScript (javascript)

The result is 1, not 0, which you may not expect.

The nullish coalescing operator helps you to avoid this pitfall. It only returns the second value when the first one is either null or undefined.

## **The nullish coalescing operator is short-circuited**

Similar to the logical OR and AND operators, the nullish coalescing operator does not evaluate the second value if the first operand is neither undefined nor null.

For example:

let result = 1 ?? console.log('Hi');

Code language: JavaScript (javascript)

In this example, the operator ?? does not evaluate the expression 1/0 because the first value is 1, which is not null and undefined.

The following example evaluates the second expression because the first one is undefined:

let result = undefined ?? console.log('Hi');

Code language: JavaScript (javascript)

Output:

'Hi'

Code language: JavaScript (javascript)

## **Chaining with the AND or OR operator**

A SyntaxError will occur if you combine the logical AND or OR operator directly with the nullish coalescing operator like this:

const result = null || undefined ?? 'OK'; *// SyntaxError*

Code language: JavaScript (javascript)

However, you can avoid this error by wrapping the expression on the left of the ?? operator in parentheses to explicitly specify the operator precedences:

const result = (null || undefined) ?? 'OK';

console.log(result); *// 'OK'*

Code language: JavaScript (javascript)

## **Summary**

* The nullish coalescing operator (??) is a logical operator that accepts two values and returns the second value if the first one is null or undefined.
* The nullish coalescing operator is short-circuited and cannot directly combine with the logical AND or OR operator.

# JavaScript Exponentiation Operator

**Summary**: in this tutorial, you will learn how to use the JavaScript exponentiation operator (\*\*) to raise a number to the power of an exponent.

## **Introduction to the JavaScript exponentiation operator**

To raise a number to the power of an exponent, you often use the static method Math.power() with the following syntax:

Math.pow(base, exponent)

Code language: JavaScript (javascript)

For example:

let result = Math.pow(2,2);

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

result = Math.pow(2,3);

console.log(result); *// 8*

Code language: JavaScript (javascript)

ECMAScript 2016 provided an alternative way to get a base to the exponent power by using the exponentiation operator ( \*\*) with the following syntax:

x\*\*n

The operator \*\* raises the x to the power of an exponent n.

Note that some languages use the caret symbol ^ for exponentiation. However, JavaScript already uses that symbol for the bitwise logical XOR operator.

The following example illustrates how to use the exponentiation operator (\*\*):

let result = 2 \*\* 2;

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

result = 2 \*\* 3;

console.log(result); *// 8*

Code language: JavaScript (javascript)

Both Math.pow() and operator \*\* accept values of the number type. However, the operator \*\* also accepts the numbers of the [bigint](https://www.javascripttutorial.net/es-next/javascript-bigint/) type. For example:

let result = 2n \*\* 3n;

console.log(result); *// 8n*

Code language: JavaScript (javascript)

In addition, you can use the exponentiation operator  ( \*\*) in the infix notation. For example:

let x = 2;

x \*\*= 4;

console.log(x); *// 16*

Code language: JavaScript (javascript)

JavaScript does not allow you to put a unary operator immediately before the base number. If you attempt to do so, you’ll get a SyntaxError.

The following causes a syntax error:

let result = -2\*\*3;

Code language: JavaScript (javascript)

Error:

Uncaught SyntaxError: Unary operator used immediately before exponentiation expression. Parenthesis must be used to disambiguate operator precedence

Code language: JavaScript (javascript)

To fix this, you use parentheses like this:

let result = (-2)\*\*3;

console.log(result); *// -8*

Code language: JavaScript (javascript)

## **Summary**

* The exponentiation operator \*\* raises a number to the power of an exponent.
* The exponentiation operator \*\* accepts values of the type number or bigint.

# JavaScript if

**Summary**: in this tutorial, you will learn how to use the JavaScript if statement to execute a block when a condition is true.

## **Introduction to the JavaScript if statement**

The if statement executes block if a condition is true. The following shows the syntax of the if statement:

if( condition )

statement;

Code language: JavaScript (javascript)

The condition can be a value or an expression. Typically, the condition evaluates to a [Boolean](https://www.javascripttutorial.net/javascript-data-types/#boolean) value, which is true or false.

If the condition evaluates to true, the if statement executes the statement. Otherwise, the if statement passes the control to the next statement after it.

The following flowchart illustrates how the if statement works:

If the condition evaluates to a non-Boolean value, JavaScript implicitly converts its result to a Boolean value by calling the [Boolean()](https://www.javascripttutorial.net/javascript-boolean/) function.

If you have more than one statement to execute, you need to use wrap them in a block using a pair of curly braces as follows:

if (condition) {

*// statements to execute*

}

Code language: JavaScript (javascript)

However, it’s a good practice to always use curly braces with the if statement. By doing this, you make your code easier to maintain and avoid possible mistakes.

## **JavaScript if statement examples**

The following example uses the if statement to check if the age is equal to or greater than 18:

let age = 18;

if (age >= 18) {

console.log('You can sign up');

}

Code language: JavaScript (javascript)

Output:

You can sign up

Code language: JavaScript (javascript)

How it works.

First, declare and initialize the [variable](https://www.javascripttutorial.net/javascript-variables/) age to 18:

let age = 18;

Code language: JavaScript (javascript)

Second, check if the age is greater or equal to 18 using the if statement. Because the expression age >= 18 is true, the code inside the if statement executes that outputs a message to the console:

if (age >= 18) {

console.log('You can sign up');

}

Code language: JavaScript (javascript)

The following example also uses the if statement. However, the age is 16 which causes the condition to evaluate to false. Therefore, you won’t see any message in the output:

let age = 16;

if (age >= 18) {

console.log('You can sign up');

}

Code language: JavaScript (javascript)

## **Nested if statement**

It’s possible to use an if statement inside another if statement. For example:

let age = 16;

let state = 'CA';

if (state == 'CA') {

if (age >= 16) {

console.log('You can drive.');

}

}

Code language: JavaScript (javascript)

Output:

You can drive.

Code language: JavaScript (javascript)

How it works.

First, declare and initialize the age and state variables:

let age = 16;

let state = 'CA';

Code language: JavaScript (javascript)

Second, check if the state is 'CA' using an if statement. If yes, check if the age is greater than 16 using a nested if statement and output a message to the console:

if (state == 'CA') {

if (age == 16) {

console.log('You can drive.');

}

}

Code language: JavaScript (javascript)

In practice, you should avoid using nested if statements as much as possible. For example, you can use the && operator to combine the conditions and use an if statements as follows:

let age = 16;

let state = 'CA';

if (state == 'CA' && age == 16) {

console.log('You can drive.');

}

Code language: JavaScript (javascript)

## **Summary**

* Use the JavaScript if statement to execute a statement if a condition is true.
* Avoid using nested if statement as much as possible.

# JavaScript if else

**Summary**: in this tutorial, you will learn how to use the JavaScript if...else statement to execute a block based on a condition.

## **Introduction to the JavaScript if…else statement**

The [if](https://www.javascripttutorial.net/javascript-if/) statement executes a block if a condition is true. When the condition is false, it does nothing. But if you want to execute a statement if the condition is false, you can use an if...else statement.

The following shows the syntax of the if...else statement:

if( condition ) {

*// ...*

} else {

*// ...*

}

Code language: JavaScript (javascript)

In this syntax, the condition is a value or an expression that evaluates to true or false. If the condition is true, the if...else statement executes the block that follows the if branch.

If the condition is false, the if...else statement executes the block that follows the else branch.

Typically, the condition evaluates to a boolean value, which is true or false. However, if it evaluates to a non-boolean value, the if...else statement will convert it to the boolean value.

The following flowchart illustrates how the if...else statement works:

## **JavaScript if…else statement**

The following example uses the if...else statement to check if the age is greater than or equal to 18:

let age = 18;

if (age >= 18) {

console.log('You can sign up.');

} else {

console.log('You must be at least 18 to sign up.');

}

Code language: JavaScript (javascript)

In this example, the age is 18. Therefore, the expression age >= 18 is true. Hence, you’ll see the following message in the console:

You can sign up.

The following example is the same as above except that the age is 18 instead of 16:

let age = 16;

if (age >= 18) {

console.log('You can sign up.');

} else {

console.log('You must be at least 18 to sign up.');

}

Code language: JavaScript (javascript)

Output:

You must be at least 18 to sign up.

In this example, the age is 16. Therefore, the expression age >= 18 evaluates to false. Hence, the statement in the else branch executes that output the message to the console.

## **Summary**

* Use the JavaScript if...else statement to execute a block if a condition is true and another block otherwise.

# JavaScript if else if

**Summary**: In this tutorial, you will learn how to use the JavaScript if...else...if statement to check multiple conditions and execute the corresponding block if a condition is true.

## **Introduction to the JavaScript if else if statement**

The [if](https://www.javascripttutorial.net/javascript-if/) an [if…else](https://www.javascripttutorial.net/javascript-if-else/) statements accept a single condition and execute the if or else block accordingly based on the condition.

To check multiple conditions and execute the corresponding block if a condition is true, you use the if...else...if statement like this:

if (condition1) {

*// ...*

} else if (condition2) {

*// ...*

} else if (condition3) {

*//...*

} else {

*//...*

}

Code language: JavaScript (javascript)

In this syntax, the if...else...if statement has three conditions. In theory, you can have as many conditions as you want to, where each else...if branch has a condition.

The if...else...if statement checks the conditions from top to bottom and executes the corresponding block if the condition is true.

The if...else...if statement stops evaluating the remaining conditions as soon as a condition is true. For example, if the condition2 is true, the if...else...if statement won’t evaluate the condition3.

If all the conditions are false, the if...else...if statement executes the block in the else branch.

The following flowchart illustrates how the if...else...if statement works:

## **JavaScript if else if examples**

Let’s take some examples of using the if...else...if statement.

### **1) A simple JavaScript if…else…if statement example**

The following example uses the if...else...if statement to get the month name from a month number:

let month = 6;

let monthName;

if (month == 1) {

monthName = 'Jan';

} else if (month == 2) {

monthName = 'Feb';

} else if (month == 3) {

monthName = 'Mar';

} else if (month == 4) {

monthName = 'Apr';

} else if (month == 5) {

monthName = 'May';

} else if (month == 6) {

monthName = 'Jun';

} else if (month == 7) {

monthName = 'Jul';

} else if (month == 8) {

monthName = 'Aug';

} else if (month == 9) {

monthName = 'Sep';

} else if (month == 10) {

monthName = 'Oct';

} else if (month == 11) {

monthName = 'Nov';

} else if (month == 12) {

monthName = 'Dec';

} else {

monthName = 'Invalid month';

}

console.log(monthName);

Code language: JavaScript (javascript)

Output:

Jun

In this example, we compare the month with 12 numbers from 1 to 12 and assign the corresponding month name to the monthName variable.

Since the month is 6, the expression month==6 evaluates to true. Therefore, the if...else...if statement assigns the literal string 'Jun' to the monthName variable. Therefore, you see Jun in the console.

If you change the month to a number that is not between 1 and 12, you’ll see the Invalid Month in the console because the else clause will execute.

### **2) Using JavaScript if…else…if statement to calculate the body mass index**

The following example calculates a body mass index (BMI) of a person. It uses the if...else...if statement to determine the weight status based on the BMI:

let weight = 70; *// kg*

let height = 1.72; *// meter*

*// calculate the body mass index (BMI)*

let bmi = weight / (height \* height);

let weightStatus;

if (bmi < 18.5) {

weightStatus = 'Underweight';

} else if (bmi >= 18.5 && bmi <= 24.9) {

weightStatus = 'Healthy Weight';

} else if (bmi >= 25 && bmi <= 29.9) {

weightStatus = 'Overweight';

} else {

weightStatus = 'Obesity';

}

console.log(weightStatus);

Code language: JavaScript (javascript)

Output:

Healthy Weight

How it works.

* First, declare two variables that hold the weight in kilogram and height in meter. In a realworld application, you’ll get these values from a web form.
* Second, calculate the body mass index by dividing the weight by the square of the height.
* Third, determine the weight status based on the BMI using the if...else..if statement.
* Finally, output the weight status to the console.

## **Summary**

* Use the JavaScript if...else...if statement to check multiple conditions and execute the corresponding block if a condition is true.

# JavaScript Ternary Operator

**Summary**: in this tutorial, you will learn how to use the JavaScript ternary operator to make your code more concise.

## **Introduction to JavaScript ternary operator**

When you want to execute a block if a condition evaluates to true, you often use an [if…else](https://www.javascripttutorial.net/javascript-if-else/) statement. For example:

let age = 18;

let message;

if (age >= 16) {

message = 'You can drive.';

} else {

message = 'You cannot drive.';

}

console.log(message);

Code language: JavaScript (javascript)

In this example, we show a message that a person can drive if the age is greater than or equal to 16. Alternatively, you can use a ternary operator instead of the [if-else](https://www.javascripttutorial.net/javascript-if-else/) statement like this:

let age = 18;

let message;

age >= 16 ? (message = 'You can drive.') : (message = 'You cannot drive.');

console.log(message);

Code language: JavaScript (javascript)

Or you can use the ternary operator in an expression as follows:

let age = 18;

let message;

message = age >= 16 ? 'You can drive.' : 'You cannot drive.';

console.log(message);

Code language: JavaScript (javascript)

Here’s the syntax of the ternary operator:

condition ? expressionIfTrue : expressionIfFalse;

Code language: JavaScript (javascript)

In this syntax, the condition is an expression that evaluates to a Boolean value, either true or false.

If the condition is true, the first expression (expresionIfTrue) executes. If it is false, the second expression (expressionIfFalse) executes.

The following shows the syntax of the ternary operator used in an expression:

let variableName = condition ? expressionIfTrue : expressionIfFalse;

Code language: JavaScript (javascript)

In this syntax, if the condition is true, the variableName will take the result of the first expression (expressionIfTrue) or expressionIfFalse otherwise.

## **JavaScript ternary operator examples**

Let’s take some examples of using the ternary operator.

### **1) Using the JavaScript ternary operator to perform multiple statements**

The following example uses the ternary operator to perform multiple operations, where each operation is separated by a comma. For example:

let authenticated = true;

let nextURL = authenticated

? (alert('You will redirect to admin area'), '/admin')

: (alert('Access denied'), '/403');

*// redirect to nextURL here*

console.log(nextURL); *// '/admin'*

Code language: JavaScript (javascript)

In this example, the returned value of the ternary operator is the last value in the comma-separated list.

### **2) Simplifying ternary operator example**

See the following example:

let locked = 1;

let canChange = locked != 1 ? true : false;

Code language: JavaScript (javascript)

If the locked is 1, then the canChange variable is set to false, otherwise, it is set to true. In this case, you can simplify it by using a Boolean expression as follows:

let locked = 1;

let canChange = locked != 1;

Code language: JavaScript (javascript)

### **3) Using multiple JavaScript ternary operators example**

The following example shows how to use two ternary operators in the same expression:

let speed = 90;

let message = speed >= 120 ? 'Too Fast' : speed >= 80 ? 'Fast' : 'OK';

console.log(message);

Code language: JavaScript (javascript)

Output:

Fast

Code language: JavaScript (javascript)

It’s a good practice to use the ternary operator when it makes the code easier to read. If the logic contains many if...else statements, you should avoid using the ternary operators.

## **Summary**

* Use the JavaScript ternary operator (?:)to make the code more concise.

# JavaScript switch case

**Summary**: in this tutorial, you will learn how to use the JavaScript switch statement to execute a block based on multiple conditions.

## **Introduction to the JavaScript switch case statement**

The switch statement evaluates an expression, compares its result with case values, and executes the statement associated with the matching case value.

The following illustrates the syntax of the switch statement:

switch (expression) {

case value1:

statement1;

break;

case value2:

statement2;

break;

case value3:

statement3;

break;

default:

statement;

}

Code language: JavaScript (javascript)

How it works.

* First, evaluate the expression inside the parentheses after the switch keyword.
* Second, compare the result of the expression with the value1, value2, … in the case branches from top to bottom. The switch statement uses the strict comparison (===).
* Third, execute the statement in the case branch where the result of the expression equals the value that follows the case keyword. The [break](https://www.javascripttutorial.net/javascript-break/) statement exits the switch statement. If you skip the break statement, the code execution falls through the original case branch into the next one. If the result of the expression does not strictly equal to any value, the switch statement will execute the statement in the default branch.

That the switch statement will stop comparing the expression‘s result with the remaining case values as long as it finds a match.

The switch statement is like the [if…else…if](https://www.javascripttutorial.net/javascript-if-else-if/) statement. But it has more readable syntax.

The following flowchart illustrates the switch statement:

In practice, you often use a switch statement to replace a complex if...else...if statement to make the code more readable.

Technically, the switch statement is equivalent to the following  if...else...if statement:

if (expression === value1) {

statement1;

} else if (expression === value2) {

statement2;

} else if (expression === value3) {

statement3;

} else {

statement;

}

Code language: JavaScript (javascript)

## **JavaScript switch case examples**

Let’s take some examples of using the JavaScript switch statement.

### **1) Using JavaScript switch statement to get the day of the week**

The following example uses the switch statement to get the day of the week based on a day number:

let day = 3;

let dayName;

switch (day) {

case 1:

dayName = 'Sunday';

break;

case 2:

dayName = 'Monday';

break;

case 3:

dayName = 'Tuesday';

break;

case 4:

dayName = 'Wednesday';

break;

case 5:

dayName = 'Thursday';

break;

case 6:

dayName = 'Friday';

break;

case 7:

dayName = 'Saturday';

break;

default:

dayName = 'Invalid day';

}

console.log(dayName); *// Tuesday*

Code language: JavaScript (javascript)

Output:

Tuesday

How it works.

First, declare the day variable that holds the day number and the day name variable (dayName).

Second, get the day of the week based on the day number using the switch statement. If the day is 1, the day of the week is Sunday. If the day is 2, the day of the week is Monday, and so on.

Third, output the day of the week to the console.

### **2) Using the JavaScript switch statement to get the day count based of a month**

The following example uses the switch statement to get the day count of a month:

let year = 2016;

let month = 2;

let dayCount;

switch (month) {

case 1:

case 3:

case 5:

case 7:

case 8:

case 10:

case 12:

dayCount = 31;

break;

case 4:

case 6:

case 9:

case 11:

dayCount = 30;

break;

case 2:

*// leap year*

if ((year % 4 == 0 && !(year % 100 == 0)) || year % 400 == 0) {

dayCount = 29;

} else {

dayCount = 28;

}

break;

default:

dayCount = -1; *// invalid month*

}

console.log(dayCount); *// 29*

Code language: JavaScript (javascript)

In this example, we have four cases:

* If the month is 1, 3,5, 7, 8, 10, or 12, the number of days in a month is 31.
* If the month is 4, 6, 9, or 11, the number of days in that month is 30.
* If the month is 2, and the year is not the leap year, the number of days is 28. If the year is the leap year, the number of days is 29.
* If the month is not in the valid range (1-12), the default branch executes and sets the dayCount variable to -1, which indicates the invalid month.

## **Summary**

* The switch statement evaluates an expression, compare its result with case values, and execute the statement associated with the matching case.
* Use the switch statement to rather than a complex if...else...if statement to make the code more redable.
* The switch statement uses the strict comparison (===) to compare the expression with the case values.

# JavaScript while Loop

**Summary**: in this tutorial, you will learn how to use the JavaScript while statement to create a loop that executes a block as long as a condition is true .

## **Introduction to the JavaScript while loop statement**

The JavaScript while statement creates a loop that executes a block as long as a condition evaluates to true.

The following illustrates the syntax of the while statement:

while (expression) {

*// statement*

}

Code language: JavaScript (javascript)

The while statement evaluates the expression before each iteration of the loop.

If the expression evaluates to true, the while statement executes the statement. Otherwise, the while loop exits.

Because the while loop evaluates the expression before each iteration, it is known as a pretest loop.

If the expression evaluates to false before the loop enters, the while loop will never execute.

The following flowchart illustrates the while loop statement:

Note that if you want to execute the statement a least once and check the condition after each iteration, you should use the [do…while](https://www.javascripttutorial.net/javascript-do-while/) statement.

## **JavaScript while loop example**

The following example uses the while statement to output the odd numbers between 1 and 10 to the console:

let count = 1;

while (count < 10) {

console.log(count);

count +=2;

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

9

How the script works

* First, declare and initialize the count variable to 1.
* Second, execute the statement inside the loop if the count variable is less than 10. In each iteration, ouput the count to the console and increase the count by 2.
* Third, after 5 iterations, the count is 11. Therefore, the condition count < 10 is false, the loop exits.

## **Summary**

* Use a while loop statement to create a loop that executes a block as long as a condition evaluates to true.

# JavaScript do…while Loop

**Summary**: in this tutorial, you will learn how to use the JavaScript do...while statement to create a loop that executes a block until a condition is false.

## **Introduction to the JavaScript do…while statement**

The do...while loop statement creates a loop that executes a block until a condition evaluates to false.

The following statement illustrates the syntax of the do...while loop:

do {

statement;

} while(expression);

Code language: JavaScript (javascript)

Unlike the [while](https://www.javascripttutorial.net/javascript-while-loop/) loop, the do-while loop always executes the statement at least once before evaluating the expression.

Because the do...while loop evaluates expression after each iteration, it’s often referred to as a post-test loop.

Inside the loop body, you need to make changes to some [variables](https://www.javascripttutorial.net/javascript-variables/) to ensure that the expression is false after some iterations. Otherwise, you’ll have an indefinite loop.

Note that starting with ES6+, the trailing semicolon (;) after the while(expression) is optional. So you can use the following syntax:

do {

statement;

} while(expression)

Code language: JavaScript (javascript)

The following flowchart illustrates the do-while loop statement:

In practice, you often use the do...while statement when you want to execute the loop body at least once before checking the condition.

## **JavaScript do while statement examples**

Let’s take some examples of using the do…while statement.

### **1) Simple JavaScript do while statement example**

The following example uses the do...while statement to output five numbers from 0 to 4 to the console:

let count = 0;

do {

console.log(count);

count++;

} while (count < 5)

Code language: JavaScript (javascript)

Output:

0

1

2

3

4

In this example:

* First, declare and initialize the count variable to zero.
* Second, show the count and increase its value by one in each iteration until its value is greater or equal 5.

### **2) Using the JavaScript do while statement to make a simple number guessing game**

The following example uses the do...while statement to generate a number guessing game.

The script generates a random integer between 1 and 10. And you have to make a number of guesses until your number matches the random number.

*// generate a secret number between 1 and 10*

const MIN = 1;

const MAX = 10;

let secretNumber = Math.floor(Math.random() \* (MAX - MIN + 1)) + MIN;

let guesses = 0; *// for storing the number of guesses*

let hint = ''; *// for storing hint*

let number = 0;

do {

*// get input from user*

let input = prompt(`Please enter a number between ${MIN} and ${MAX}` + hint);

*// get the integer*

number = parseInt(input);

*// increase the number of guesses*

guesses++;

*// check input number with the secret number*

*// provide hint if needed*

if (number > secretNumber) {

hint = ', and less than ' + number;

} else if (number < secretNumber) {

hint = ', and greater than ' + number;

} else if (number == secretNumber) {

alert(`Bravo! you're correct after ${guesses} guess(es).`);

}

} while (number != secretNumber);

Code language: JavaScript (javascript)

How it works.

* First, generate a random number between 1 and 10.
* Second, [prompt](https://www.javascripttutorial.net/javascript-bom/javascript-prompt/) for input an integer and compare it with the secret number. If they are not equal, show a hint. Otherwise, display the congratulation message.

## **Summary**

* Use the do…while statement to create a loop that executes a block until a condition is false.

# JavaScript for Loop

**Summary**: in this tutorial, you will learn how to use the JavaScript for loop statement to create a loop with various options.

## **Introduction to the JavaScript for loop statement**

The for loop statement creates a loop with three optional expressions. The following illustrates the syntax of the for loop statement:

for (initializer; condition; iterator) {

*// statements*

}

Code language: JavaScript (javascript)

### **1) iterator**

The for statement executes the initializer only once the loop starts. Typically, you declare and initialize a local loop variable in the initializer.

### **2) condition**

The condition is a boolean expression that determines whether the for should execute the next iteration.

The for statement evaluates the condition before each iteration. If the condition is true (or is not present), it executes the next iteration. Otherwise, it’ll end the loop.

### **3) iterator**

The for statement executes the iterator after each iteration.

The following flowchart illustrates the for loop:

In the for loop, the three expressions are optional. The following shows the for loop without any expressions:

for ( ; ; ) {

*// statements*

}

Code language: JavaScript (javascript)

## **JavaScript for loop examples**

Let’s take some examples of using the for loop statement.

### **1) A simple JavaScript for loop example**

The following example uses the for loop statement to show numbers from 1 to 4 to console:

for (let i = 1; i < 5; i++) {

console.log(i);

}

Code language: JavaScript (javascript)

Output:

1

2

3

4

How it works.

* First, declare a variable  counter and initialize it to 1.
* Second, display the value of counter in the console if counter is less than 5.
* Third, increase the value of counter by one in each iteration of the loop.

### **2) Using the JavaScript for loop without the initializer example**

The following example uses a for loop that has no initializer expression:

let j = 1;

for (; j < 10; j += 2) {

console.log(j);

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

9

### **3) Using the JavaScript for loop without the condition example**

Similar to the initializer expression, the condition expression is optional. If you omit the condition expression, you need to use a [break](https://www.javascripttutorial.net/javascript-break/) statement to terminate the loop.

for (let j = 1; ; j += 2) {

console.log(j);

if (j > 10) {

break;

}

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

9

11

### **3) Using the JavaScript for loop statement without any expression example**

All three expressions of the for loop statements are optional. Therefore, you can omit all of them. For example:

let j = 1;

for (;;) {

if (j > 10) {

break;

}

console.log(j);

j += 2;

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

9

### **4) Using the JavaScript for loop without the loop body example**

JavaScript allows the for statement to have an empty statement. In this case, you place a semicolon (;) immediately after the for statement.

For example, the following uses a for loop to calculate the sum of 10 numbers from 1 to 10:

let sum = 0;

for (let i = 0; i <= 9; i++, sum += i);

console.log(sum);

Code language: JavaScript (javascript)

Output:

55

## **Summary**

* Use the JavaScript for statement to create a loop that execute a block based using various options.

# JavaScript break

**Summary**: in this tutorial, you’ll learn how to use the JavaScript break statement to terminate a loop prematurely.

## **The label statement**

In JavaScript, you can label a statement for later use. Here’s the syntax of the label statement:

label: statement;

Code language: HTTP (http)

In this syntax, the label can be any valid identifier. For example, the following shows how to label a for loop using the outer label:

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

console.log(i);

}

Code language: JavaScript (javascript)

Once defining a label, you can reference it in the break or [continue](https://www.javascripttutorial.net/javascript-continue/) statement.

## **Introduction to JavaScript break statement**

The break statement prematurely terminates a loop such as [for](https://www.javascripttutorial.net/javascript-for-loop/), [do...while](https://www.javascripttutorial.net/javascript-do-while/), and [while](https://www.javascripttutorial.net/javascript-while-loop/) loop, a [switch](https://www.javascripttutorial.net/javascript-switch-case/), or a label statement. Here’s the syntax of the break statement:

break [label];

Code language: JavaScript (javascript)

In this syntax, the label is optional if you use the break statement in a loop or switch. However, if you use the break statement with a label statement, you need to specify it.

This tutorial focuses on how to use the break statement to terminate the loop prematurely.

## **Using JavaScript break statement in a for loop**

The following for loop statement outputs five numbers from 0 to 4:

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

console.log(i);

}

Code language: JavaScript (javascript)

Output:

0

1

2

3

4

To terminate the for loop prematurely, you can use a break statement. For example, the following illustrates how to use a break statement inside a for loop:

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

console.log(i);

if (i == 2) {

break;

}

}

Code language: JavaScript (javascript)

Output:

0

1

2

In this example, we use an [if](https://www.javascripttutorial.net/javascript-if/) statement inside the loop. If the current value of i is 2, the if statement executes the break statement that terminates the loop.

This flowchart illustrates how the break statement works in a for loop:

### **Using the break statement to terminate a nested loop**

A nested loop has one loop inside another. For example, the following uses a nested for loop to output a pair of numbers from 1 to 3:

for (let i = 1; i <= 3; i++) {

for (let j = 1; j <= 3; j++) {

console.log(i, j);

}

}

Code language: JavaScript (javascript)

Output:

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

If you use a break statement inside an inner loop, it only terminates the enclosing loop. For example:

for (let i = 1; i <= 3; i++) {

for (let j = 1; j <= 3; j++) {

if (i + j == 4) {

break;

}

console.log(i, j);

}

}

Code language: JavaScript (javascript)

Output:

1 1

1 2

2 1

In this example, if the sum of i and j is 4, the break statement terminates the inner loop. To terminate the nested loop, you use a label statement. For example:

outer: for (let i = 1; i <= 3; i++) {

for (let j = 1; j <= 3; j++) {

if (i + j == 4) {

break outer;

}

console.log(i, j);

}

}

Code language: JavaScript (javascript)

Output:

1 1

1 2

In this example, we label the outer loop with the label outer. Inside the inner loop, we specify the outer label in the break statement. The break statement to terminate the nested loop if the sum of i and j is 4.

## **Using JavaScript break statement in a while loop**

The following output five numbers from 1 to 5 to the console using a while loop:

let i = 0;

while (i < 5) {

i++;

console.log(i);

}

Code language: JavaScript (javascript)

Output:

1

2

3

4

5

Like a for loop, the break statement terminates a while loop prematurely. For example:

let i = 0;

while (i < 5) {

i++;

console.log(i);

if (i == 3) {

break;

}

}

Code language: JavaScript (javascript)

Output:

1

2

3

In this example, when the current value of i is 3, the break statement terminates the loop. Therefore, you see only three numbers in the output.

The following flowchart illustrates how the break statement works in a while loop:

## **Using JavaScript break statement in a do…while loop**

The following example uses a do...while statement to output five numbers from 0 to 5 to the console:

let i = 0;

do {

i++;

console.log(i);

} while (i < 5);

Code language: JavaScript (javascript)

Output:

1

2

3

4

5

Like a while loop, you can use a break statement to terminate a do...while loop. For example:

let i = 0;

do {

i++;

console.log(i);

if (i == 3) {

break;

}

} while (i < 5);

Code language: JavaScript (javascript)

Output:

1

2

3

The following flowchart shows how the break statement works in a do while loop:

## **Summary**

* Use the break statement to terminate a loop including for, while, and do...while prematurely.
* When using in a nested loop, the break statement terminates the enclosing loop. To terminate the nested loop, you use a label statement.

# JavaScript continue

**Summary**: in this tutorial, you will learn how to use the JavaScript continue statement to skip the current iteration of a loop.

## **Introduction to the JavaScript continue statement**

The continue statement terminates the execution of the statement in the current iteration of a loop and immediately continues to the next iteration.

Here’s the syntax of the continue statement:

continue [label];

Code language: JavaScript (javascript)

In this syntax, the label is optional. It is a valid identifier associated with the label of a statement. For more, check out the break statement tutorial for more information on the label statement.

Typically, you use the continue with an [if](https://www.javascripttutorial.net/javascript-if/) statement like this:

*// inside a loop*

if(condition){

continue;

}

Code language: JavaScript (javascript)

In this syntax, the if statement specifies a condition to execute the continue statement inside a loop.

## **Using the continue statement in a for loop**

When using the continue statement in a for loop, it doesn’t terminate the loop entirely. Instead, it jumps to the iterator expression.

The following flowchart illustrates how the continue statement works in a for loop:

JavaScript continue with for loop

The following example uses a continue in a for loop to display the odd number in the console:

for (let i = 0; i < 10; i++) {

if (i % 2 === 0) {

continue;

}

console.log(i);

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

9

In this example, the for loop iterates over the numbers from 0 to 9.

The i%2 returns the remainder of the division of the current value of i by 2.

If the remainder is zero, the if statement executes the continue statement that skips the current iteration of the loop and jumps to the iterator expression i++. Otherwise, it outputs the value of i to the console.

## **Using the continue statement in a while loop**

When using the continue statement in a while loop, it doesn’t terminate the execution of the loop entirely. Instead, it jumps back to the condition.

The following flowchart show hows the continue statement works in a while loop statement:

The following example uses a continue statement in a while loop to display the odd numbers from 1 to 10:

let i = 0;

while (i < 10) {

i++;

if (i % 2 === 0) {

continue;

}

console.log(i);

}

Code language: JavaScript (javascript)

Output:

1

3

5

7

9

## **Using the continue statement with a label example**

The continue statement can include an optional label like this:

continue label;

Code language: JavaScript (javascript)

The following nested loop displays pairs of numbers from 1 to 2:

for (let i = 1; i < 4; i++) {

for (let j = 1; j < 4; j++) {

console.log(i, j);

}

}

Code language: JavaScript (javascript)

Output:

1 1

1 2

1 3

2 1

2 2

2 3

3 1

3 2

3 3

The following shows how to use the continue statement with a label:

outer: for (let i = 1; i < 4; i++) {

for (let j = 1; j < 4; j++) {

if (i + j == 3) continue outer;

console.log(i, j);

}

}

Code language: JavaScript (javascript)

Output:

1 1

3 1

3 2

3 3

## **Summary**

* Use the JavaScript continue statement to skip the current iteration of a loop and continue the next one.

# JavaScript Comma Operator

**Summary**: in this tutorial, you’ll learn about the JavaScript comma operator and its usages.

## **Introduction to the JavaScript comma operator**

JavaScript uses a comma (,) to represent the comma operator. A comma operator takes two expressions, evaluates them from left to right, and returns the value of the right expression.

Here’s the syntax of the comma operator:

leftExpression, rightExpression

For example:

let result = (10, 10 + 20);

console.log(result);

Code language: JavaScript (javascript)

Output:

30

In this example, the 10, 10+20 returns the value of the right expression, which is 10+20. Therefore, the result value is 30.

See the following example:

let x = 10;

let y = (x++, x + 1);

console.log(x, y);

Code language: JavaScript (javascript)

Output:

11 12

In this example, we increase the value of x by one (x++), add one to x (x+1) and assign x to y. Therefore, x is 11, and y is 12 after the statement.

However, to make the code more explicit, you can use two statements rather than one statement with a comma operator like this:

let x = 10;

x++;

let y = x + 1;

console.log(x, y);

Code language: JavaScript (javascript)

This code is more explicit.

In practice, you might want to use the comma operator inside a [for](https://www.javascripttutorial.net/javascript-for-loop/) loop to update multiple variables each time through the loop.

The following example uses the comma operator in a for loop to display an array of nine elements as a matrix of 3 rows and three columns:

let board = [1, 2, 3, 4, 5, 6, 7, 8, 9];

let s = '';

for (let i = 0, j = 1; i < board.length; i++, j++) {

s += board[i] + ' ';

if (j % 3 == 0) {

console.log(s);

s = '';

}

}

Code language: JavaScript (javascript)

Output:

1 2 3

4 5 6

7 8 9

## **Summary**

* A comma operator takes two expressions and evaluates them from left to right, and returns the value of the right expression.
* Use the comma operator (,) inside a for loop to update multiple variables once.
* Use two statements rather than the comma operator elsewhere to make the code more explicit and easier to understand.

# JavaScript Functions

**Summary**: in this tutorial, you will learn about JavaScript functions and how to use them to structure the code into smaller and more reusable units.

## **Introduction to JavaScript functions**

When developing an application, you often need to perform the same action in many places. For example, you may want to show a message whenever an error occurs.

To avoid repeating the same code all over places, you can use a function to wrap that code and reuse it.

JavaScript provides many built-in functions such as parseInt() and parseFloat(). In this tutorial, you will learn how to develop custom functions.

## **Declare a function**

To declare a function, you use the function keyword, followed by the function name, a list of parameters, and the function body as follows:

function functionName(parameters) {

*// function body*

*// ...*

}

Code language: JavaScript (javascript)

The function name must be a valid JavaScript identifier. By convention, the function names are in camelCase and start with verbs like getData(), fetchContents(), and isValid().

A function can accept zero, one, or multiple parameters. In the case of multiple parameters, you need to use a comma to separate two parameters.

The following declares a function say() that accepts no parameter:

function say() {

}

Code language: JavaScript (javascript)

The following declares a function named square() that accepts one parameter:

function square(a) {

}

Code language: JavaScript (javascript)

And the following declares a function named add() that accepts two parameters:

function add(a, b) {

}

Code language: JavaScript (javascript)

Inside the function body, you can write the code to implement an action. For example, the following say() function simply shows a message to the console:

function say(message) {

console.log(message);

}

Code language: JavaScript (javascript)

In the body of the say() function, we call the console.log() function to output a message to the console.

## **Calling a function**

To use a function, you need to call it. Calling a function is also known as invoking a function. To call a function, you use its name followed by arguments enclosing in parentheses like this:

functionName(arguments);

Code language: JavaScript (javascript)

When calling a function, JavaScript executes the code inside the function body. For example, the following shows how to call the say() function:

say('Hello');

Code language: JavaScript (javascript)

In this example, we call the say() function and pass a literal string 'Hello' into it.

### **Parameters vs. Arguments**

The terms parameters and arguments are often used interchangeably. However, they are essentially different.

When declaring a function, you specify the parameters. However, when calling a function, you pass the arguments that are corresponding to the parameters.

For example, in the say() function, the message is the parameter and the 'Hello' string is an argument that corresponds to the message parameter.

## **Returning a value**

Every function in JavaScript implicitly returns undefined unless you explicitly specify a return value. For example:

function say(message) {

console.log(message);

}

let result = say('Hello');

console.log('Result:', result);

Code language: JavaScript (javascript)

Output:

Hello

Result: undefined

Code language: JavaScript (javascript)

To specify a return value for a function, you use the return statement followed by an expression or a value, like this:

return expression;

Code language: JavaScript (javascript)

For example, the following add() function returns the sum of the two arguments:

function add(a, b) {

return a + b;

}

Code language: JavaScript (javascript)

The following shows how to call the add() function:

let sum = add(10, 20);

console.log('Sum:', sum);

Code language: JavaScript (javascript)

Output:

Sum: 30

Code language: HTTP (http)

The following example uses multiple return statements in a function to return different values based on conditions:

function compare(a, b) {

if (a > b) {

return -1;

} else if (a < b) {

return 1;

}

return 0;

}

Code language: JavaScript (javascript)

The compare() function compares two values. It returns:

* -1 if the first argument is greater than the second one.
* 1 if the first argument is less than the second one.
* 0 if  the first argument equals the second one.

The function immediately stops executing immediately when it reaches the return statement. Therefore, you can use the return statement without a value to exit the function prematurely, like this:

function say(message) {

*// show nothing if the message is empty*

if (! message ) {

return;

}

console.log(message);

}

Code language: JavaScript (javascript)

In this example, if the message is blank (or undefined), the say() function will show nothing.

The function can return a single value. If you want to [return multiple values from a function](https://www.javascripttutorial.net/javascript-return-multiple-values/), you need to pack these values in an array or an object.

## **The arguments object**

Inside a function, you can access an object called arguments that represents the named arguments of the function.

The arguments object behaves like an [array](https://www.javascripttutorial.net/javascript-array/) though it is not an instance of the [Array](https://www.javascripttutorial.net/javascript-array/) type.

For example, you can use the square bracket [] to access the arguments: arguments[0] returns the first argument, arguments[1] returns the second one, and so on.

Also, you can use the length property of the arguments object to determine the number of arguments.

The following example implements a generic add() function that calculates the sum of any number of arguments.

function add() {

let sum = 0;

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

sum += arguments[i];

}

return sum;

}

Code language: JavaScript (javascript)

Hence, you can pass any number of arguments to the add() function, like this:

console.log(add(1, 2)); *// 3*

console.log(add(1, 2, 3, 4, 5)); *// 15*

Code language: JavaScript (javascript)

## **Function hoisting**

In JavaScript, you can use a function before declaring it. For example:

showMe(); *// a hoisting example*

function showMe(){

console.log('an hoisting example');

}

Code language: JavaScript (javascript)

This feature is called [hoisting](https://www.javascripttutorial.net/javascript-hoisting/).

Function hoisting is a mechanism that the JavaScript engine physically moves function declarations to the top of the code before executing them.

The following shows the version of the code before the JavaScript engine executes it:

function showMe(){

console.log('a hoisting example');

}

showMe(); *// a hoisting example*

Code language: JavaScript (javascript)

## **Summary**

* Use the function keyword to declare a function.
* Use the functionName() to call a function.
* All functions implicitly return undefined if they don’t explicitly return a value.
* Use the return statement to return a value from a function explicitly.
* The arguments variable is an array-like object inside a function, representing function arguments.
* The function hoisting allows you to call a function before declaring it.

# JavaScript Functions are First-Class Citizens

**Summary**: in this tutorial, you’ll learn that JavaScript functions are first-class citizens. This means that you can store functions in variables, pass them to other functions as arguments, and return them from other functions as values.

## **Storing functions in variables**

[Functions](https://www.javascripttutorial.net/javascript-function/) are first-class citizens in JavaScript. In other words, you can treat functions like values of other types.

The following defines the add() function and assigns the function name to the variable sum:

function add(a, b) {

return a + b;

}

let sum = add;

Code language: JavaScript (javascript)

In the assignment statement, we don’t include the opening and closing parentheses at the end of the add identifier. We also don’t execute the function but reference the function.

By doing this, we can have two ways to execute the same function. For example, we can call it normally as follows:

let result = add(10, 20);

Code language: JavaScript (javascript)

Alternatively, we can all the add() function via the sum variable like this:

let result = sum(10,20);

Code language: JavaScript (javascript)

## **Passing a function to another function**

Because functions are values, you can pass a function as an argument into another function.

The following declares the average() function that takes three arguments. The third argument is a function:

function average(a, b, fn) {

return fn(a, b) / 2;

}

Code language: JavaScript (javascript)

Now, you can pass the sum function to the average() function as follows:

let result = average(10, 20, sum);

Code language: JavaScript (javascript)

Put it all together:

function add(a, b) {

return a + b;

}

let sum = add;

function average(a, b, fn) {

return fn(a, b) / 2;

}

let result = average(10, 20, sum);

console.log(result);

Code language: JavaScript (javascript)

Output:

15

## **Returning functions from functions**

Since functions are values, you can return a function from another function.

The following compareBy() function returns a function that compares two [objects](https://www.javascripttutorial.net/javascript-objects/) by a property:

function compareBy(propertyName) {

return function (a, b) {

let x = a[propertyName],

y = b[propertyName];

if (x > y) {

return 1;

} else if (x < y) {

return -1;

} else {

return 0;

}

};

}

Code language: JavaScript (javascript)

Note that a[propertyName] returns the value of the propertyName of the a object. It’s equivalent to a.propertyName. However, if the propertyName contains a space like 'Discount Price', you need to use the square bracket notation to access it.

Suppose that you have an array of product objects where each product object has two properties: name and price.

let products = [

{name: 'iPhone', price: 900},

{name: 'Samsung Galaxy', price: 850},

{name: 'Sony Xperia', price: 700}

];

Code language: JavaScript (javascript)

You can [sort an array](https://www.javascripttutorial.net/javascript-array-sort/) by calling the sort() method. The sort() method accepts a function that compares two elements of the array as an argument.

For example, you can sort the product objects based on the name by passing a function returned from the compareBy() function as follows:

console.log('Products sorted by name:');

products.sort(compareBy('name'));

console.table(products);

Code language: JavaScript (javascript)

Output:

Products sorted by name:

┌─────────┬──────────────────┬───────┐

│ (index) │ name │ price │

├─────────┼──────────────────┼───────┤

│ 0 │ 'Samsung Galaxy' │ 850 │

│ 1 │ 'Sony Xperia' │ 700 │

│ 2 │ 'iPhone' │ 900 │

└─────────┴──────────────────┴───────┘

Code language: plaintext (plaintext)

Similarly, you can sort the product objects by prices:

*// sort products by prices*

console.log('Products sorted by price:');

products.sort(compareBy('price'));

console.table(products);

Code language: JavaScript (javascript)

Output:

Products sorted by price:

┌─────────┬──────────────────┬───────┐

│ (index) │ name │ price │

├─────────┼──────────────────┼───────┤

│ 0 │ 'Sony Xperia' │ 700 │

│ 1 │ 'Samsung Galaxy' │ 850 │

│ 2 │ 'iPhone' │ 900 │

└─────────┴──────────────────┴───────┘

Code language: JavaScript (javascript)

Put it all together.

function compareBy(propertyName) {

return function (a, b) {

let x = a[propertyName],

y = b[propertyName];

if (x > y) {

return 1;

} else if (x < y) {

return -1;

} else {

return 0;

}

};

}

let products = [

{ name: 'iPhone', price: 900 },

{ name: 'Samsung Galaxy', price: 850 },

{ name: 'Sony Xperia', price: 700 },

];

*// sort products by name*

console.log('Products sorted by name:');

products.sort(compareBy('name'));

console.table(products);

*// sort products by price*

console.log('Products sorted by price:');

products.sort(compareBy('price'));

console.table(products);

Code language: JavaScript (javascript)

## **More JavaScript Functions are First-Class Citizens example**

The following example defines two functions that convert a length in centimeters to inches and vice versa:

function cmToIn(length) {

return length / 2.54;

}

function intoCm(length) {

return length \* 2.54;

}

Code language: JavaScript (javascript)

The following convert() function has two parameters. The first parameter is a function and the second one is the length that will be converted based on the first argument:

function convert(fn, length) {

return fn(length);

}

Code language: JavaScript (javascript)

To convert cm to in, you can call the convert() function and pass the cmToIn function into the convert() function as the first argument:

let inches = convert(cmToIn, 10);

console.log(inches);

Code language: JavaScript (javascript)

Output:

3.937007874015748

Code language: CSS (css)

Similarly, to convert a length from inches to centimeters, you can pass the inToCm function into the convert() function, like this:

let cm = convert(intoCm, 10);

console.log(cm);

Code language: JavaScript (javascript)

Output:

25.4

Code language: CSS (css)

Put it all together.

function cmToIn(length) {

return length / 2.54;

}

function intoCm(length) {

return length \* 2.54;

}

function convert(fn, length) {

return fn(length);

}

let inches = convert(cmToIn, 10);

console.log(inches);

let cm = convert(intoCm, 10);

console.log(cm);

Code language: JavaScript (javascript)

Output:

3.937007874015748

25.4

Code language: CSS (css)

## **Summary**

* Functions are first-class citizens in JavaScript.
* You can pass functions to other functions as arguments, return them from other functions as values, and store them in variables.

# JavaScript Anonymous Functions

**Summary**: in this tutorial, you will learn about JavaScript anonymous functions.

## **Introduction to JavaScript anonymous functions**

An anonymous function is a [function](https://www.javascripttutorial.net/javascript-function/) without a name. The following shows how to define an anonymous function:

(function () {

*//...*

});

Code language: JavaScript (javascript)

Note that if you don’t place the anonymous function inside the (), you’ll get a syntax error. The () makes the anonymous function an expression that returns a function object.

An anonymous function is not accessible after its initial creation. Therefore, you often need to assign it to a variable.

For example, the following shows an anonymous function that displays a message:

let show = function() {

console.log('Anonymous function');

};

show();

Code language: JavaScript (javascript)

In this example, the anonymous function has no name between the function keyword and parentheses (). Because we need to call the anonymous function later, we assign the anonymous function to the show variable.

## **Using anonymous functions as arguments**

In practice, you often pass anonymous functions as arguments to other functions. For example:

setTimeout(function() {

console.log('Execute later after 1 second')

}, 1000);

Code language: JavaScript (javascript)

In this example, we pass an anonymous function into the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function. The setTimeout() function executes this anonymous function one second later.

Note that [functions are](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/)first-class citizens in JavaScript. Therefore, you can pass a function to another function as an argument.

## **Immediately invoked function execution**

If you want to create a function and execute it immediately after the declaration, you can declare an anonymous function like this:

(function() {

console.log('IIFE');

})();

Code language: JavaScript (javascript)

How it works.

First, define a function expression:

(function () {

console.log('Immediately invoked function execution');

})

Code language: JavaScript (javascript)

This expression returns a function.

Second, call the function by adding the trailing parentheses ():

(function () {

console.log('Immediately invoked function execution');

})();

Code language: JavaScript (javascript)

and sometimes, you may want to pass arguments into it, like this:

let person = {

firstName: 'John',

lastName: 'Doe'

};

(function () {

console.log(person.firstName} + ' ' + person.lastName);

})(person);

Code language: JavaScript (javascript)

## **Arrow functions**

ES6 introduced [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) expression that provides a shorthand for declaring anonymous functions:

For example, this function:

let show = function () {

console.log('Anonymous function');

};

Code language: JavaScript (javascript)

… can be shortened using the following arrow function:

let show = () => console.log('Anonymous function');

Code language: JavaScript (javascript)

Similarly, the following anonymous function:

let add = function (a, b) {

return a + b;

};

Code language: JavaScript (javascript)

… is functionally equivalent to the following arrow function:

let add = (a, b) => a + b;

Code language: JavaScript (javascript)

## **Summary**

* Anonymous functions are functions without names.
* Anonymous functions can be used as an argument to other functions or as an immediately invoked function execution.

# Understanding JavaScript Pass-By-Value

**Summary**: this tutorial explains how JavaScript pass-by-value works and gives you some examples of passing primitive and reference values to a function.

Before going forward with this tutorial, you should have good knowledge of the[primitive and reference values, and the differences between them](https://www.javascripttutorial.net/javascript-primitive-vs-reference-values/).

## **JavaScript pass-by-value or pass-by-reference**

In JavaScript, all [function](https://www.javascripttutorial.net/javascript-function/) arguments are always passed by value. It means that JavaScript copies the values of the [variables](https://www.javascripttutorial.net/javascript-variables/) into the function arguments.

Any changes that you make to the arguments inside the function do not reflect the passing variables outside of the function. In other words, the changes made to the arguments are not reflected outside of the function.

If function arguments are passed by reference, the changes of variables that you pass into the function will be reflected outside the function. This is not possible in JavaScript.

## **Pass-by-value of primitives values**

Let’s take a look at the following example.

function square(x) {

x = x \* x;

return x;

}

let y = 10;

let result = square(y);

console.log(result); *// 100*

console.log(y); *// 10 -- no change*

Code language: JavaScript (javascript)

How the script works.

First, define a square() function that accepts an argument x. The function assigns the square of x to the x argument.

Next, declare the variable y and initialize its value to 10:

Then, pass the y variable into the square() function. When passing the variable y to the square() function, JavaScript copies y value to the x variable.

After that, the square() function changes the x variable. However, it does not impact the value of the y variable because x and y are separate variables.

Finally, the value of the y variable does not change after the square() function completes.

If JavaScript used the pass-by-reference, the variable y would change to 100 after calling the function.

## **Pass-by-value of reference values**

It’s not obvious to see that reference values are also passed by values. For example:

let person = {

name: 'John',

age: 25,

};

function increaseAge(obj) {

obj.age += 1;

}

increaseAge(person);

console.log(person);

Code language: JavaScript (javascript)

How the script works:

First, define the person variable that references an object with two properties name and age:

Next, define the increaseAge() function that accepts an object obj and increases the age property of the obj argument by one.

Then, pass the person object to the increaseAge() function:

Internally, the JavaScript engine creates the obj reference and make this variable reference the same object that the person variable references.

After that, increase the age property by one inside the increaseAge() function via the obj variable

Finally, accessing the object via the person reference:

It seems that JavaScript passes an object by reference because the change to the object is reflected outside the function. However, this is not the case.

In fact, when passing an object to a function, you are passing the reference of that object, not the actual object. Therefore, the function can modify the properties of the object via its reference.

However, you cannot change the reference passed into the function. For example:

let person = {

name: 'John',

age: 25,

};

function increaseAge(obj) {

obj.age += 1;

*// reference another object*

obj = { name: 'Jane', age: 22 };

}

increaseAge(person);

console.log(person);

Code language: JavaScript (javascript)

Output:

{ name: 'John', age: 26 }

Code language: CSS (css)

In this example, the increaseAage() function changes the age property via the obj argument:

and makes the obj reference another object:

However, the person reference still refers to the original object whose the age property changes to 26. In other words, the increaseAge() function doesn’t change the person reference.

If this concept still confuses you, you can consider function arguments as local variables.

## **Summary**

* JavaScript passes all arguments to a function by values.
* Function arguments are local variables in JavaScript.

# JavaScript Recursive Function

**Summary**: in this tutorial, you will learn how to use the recursion technique to develop a JavaScript recursive function, which is a function that calls itself.

## **Introduction to the JavaScript recursive functions**

A recursive function is a [function](https://www.javascripttutorial.net/javascript-function/) that calls itself until it doesn’t. And this technique is called recursion.

Suppose that you have a function called recurse(). The recurse() is a recursive function if it calls itself inside its body, like this:

function recurse() {

*// ...*

recurse();

*// ...*

}

Code language: JavaScript (javascript)

A recursive function always has a condition to stop calling itself. Otherwise, it will call itself indefinitely. So a recursive function typically looks like the following:

function recurse() {

if(condition) {

*// stop calling itself*

*//...*

} else {

recurse();

}

}

Code language: JavaScript (javascript)

Generally, you use recursive functions to break down a big problem into smaller ones. Typically, you will find the recursive functions in data structures like binary trees and graphs and algorithms such as binary search and quicksort.

## **JavaScript recursive function examples**

Let’s take some examples of using recursive functions.

### **1) A simple JavaScript recursive function example**

Suppose that you need to develop a function that counts down from a specified number to 1. For example, to count down from 3 to 1:

3

2

1

The following shows the countDown() function:

function countDown(fromNumber) {

console.log(fromNumber);

}

countDown(3);

Code language: JavaScript (javascript)

This countDown(3) shows only the number 3.

To count down from the number 3 to 1, you can:

1. show the number 3.
2. and call the countDown(2) that shows the number 2.
3. and call the countDown(1) that shows the number 1.

The following changes the countDown() to a recursive function:

function countDown(fromNumber) {

console.log(fromNumber);

countDown(fromNumber-1);

}

countDown(3);

Code language: JavaScript (javascript)

This countDown(3) will run until the call stack size is exceeded, like this:

Uncaught RangeError: Maximum call stack size exceeded.

Code language: JavaScript (javascript)

… because it doesn’t have the condition to stop calling itself.

The count down will stop when the next number is zero. Therefore, you add an [if condition](https://www.javascripttutorial.net/javascript-if/) as follows:

function countDown(fromNumber) {

console.log(fromNumber);

let nextNumber = fromNumber - 1;

if (nextNumber > 0) {

countDown(nextNumber);

}

}

countDown(3);

Code language: JavaScript (javascript)

Output:

3

2

1

The countDown() seems to work as expected.

However, as mentioned in the [Function type tutorial](https://www.javascripttutorial.net/javascript-function-type/), the function’s name is a reference to the actual function object.

If the function name is set to null somewhere in the code, the recursive function will stop working.

For example, the following code will result in an error:

let newYearCountDown = countDown;

*// somewhere in the code*

countDown = null;

*// the following function call will cause an error*

newYearCountDown(10);

Code language: JavaScript (javascript)

Error:

Uncaught TypeError: countDown is not a function

Code language: JavaScript (javascript)

How the script works:

* First, assign the countDown function name to the variable newYearCountDown.
* Second, set the countDown function reference to null.
* Third, call the newYearCountDown function.

The code causes an error because the body of the countDown() function references the countDown function name, which was set to null at the time of calling the function.

To fix it, you can use a named function expression as follows:

let countDown = function f(fromNumber) {

console.log(fromNumber);

let nextNumber = fromNumber - 1;

if (nextNumber > 0) {

f(nextNumber);

}

}

let newYearCountDown = countDown;

countDown = null;

newYearCountDown(10);

Code language: JavaScript (javascript)

### **2) Calculate the sum of n natural numbers example**

Suppose you need to calculate the sum of natural numbers from 1 to n using the recursion technique. To do that, you need to define the sum() recursively as follows:

sum(n) = n + sum(n-1)

sum(n-1) = n - 1 + sum(n-2)

...

sum(1) = 1

The following illustrates the sum() recursive function:

function sum(n) {

if (n <= 1) {

return n;

}

return n + sum(n - 1);

}

Code language: JavaScript (javascript)

## **Summary**

* A recursive function is a function that calls itself until it doesn’t
* A recursive function always has a condition that stops the function from calling itself.

# 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 Object Methods

**Summary**: in this tutorial, you’ll learn about the JavaScript object methods and how to define methods for an object.

## **Introduction to the JavaScript object methods**

An object is a collection of key/value pairs or [properties](https://www.javascripttutorial.net/javascript-object-properties/). When the value is a function, the property becomes a method. Typically, you use methods to describe the object behaviors.

For example, the following adds the greet method to the person object:

let person = {

firstName: 'John',

lastName: 'Doe'

};

person.greet = function () {

console.log('Hello!');

}

person.greet();

Code language: JavaScript (javascript)

Output:

Hello!

In this example:

* First, use a function expression to define a function and assign it to the greet property of the person object.
* Then, call the method greet() method.

Besides using a function expression, you can define a function and assign it to an object like this:

let person = {

firstName: 'John',

lastName: 'Doe'

};

function greet() {

console.log('Hello, World!');

}

person.greet = greet;

person.greet();

Code language: JavaScript (javascript)

In this example:

* First, define the greet() function as a regular function.
* Second, assign the function name to the the greet property of the person object.
* Third, call the greet() method.

## **Object method shorthand**

JavaScript allows you to define methods of an object using the object literal syntax as shown in the following example:

let person = {

firstName: 'John',

lastName: 'Doe',

greet: function () {

console.log('Hello, World!');

}

};

Code language: JavaScript (javascript)

ES6 provides you with the [concise method syntax](https://www.javascripttutorial.net/es6/object-literal-extensions/) that allows you to define a method for an object:

let person = {

firstName: 'John',

lastName: 'Doe',

greet() {

console.log('Hello, World!');

}

};

person.greet();

Code language: JavaScript (javascript)

This syntax looks much cleaner and less verbose.

### **The this value**

Typically, methods need to access other properties of the object.

For example, you may want to define a method that returns the full name of the person object by concatenating the first name and last name.

Inside a method, the this value references the object that invokes the method. Therefore, you can access a property using the this value as follows:

this.propertyName

Code language: JavaScript (javascript)

The following example uses the this value in the getFullName() method:

let person = {

firstName: 'John',

lastName: 'Doe',

greet: function () {

console.log('Hello, World!');

},

getFullName: function () {

return this.firstName + ' ' + this.lastName;

}

};

console.log(person.getFullName());

Code language: JavaScript (javascript)

Output

'John Doe'

Code language: JavaScript (javascript)

Check out [this tutorial](https://www.javascripttutorial.net/javascript-this/) for more information on [this](https://www.javascripttutorial.net/javascript-this/) value.

## **Summary**

* When a function is a property of an object, it becomes a method.

# JavaScript Constructor Function

**Summary**: in this tutorial, you’ll learn about the JavaScript constructor function and how to use the new keyword to create an object.

## **Introduction to JavaScript constructor functions**

In the [JavaScript objects tutorial](https://www.javascripttutorial.net/javascript-objects/), you learned how to use the object literal syntax to create a new object.

For example, the following creates a new person object with two properties firstName and lastName:

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

In practice, you often need to create many similar objects like the person object.

To do that, you can use a constructor function to define a custom type and the new operator to create multiple objects from this type.

Technically speaking, a constructor function is a regular [function](https://www.javascripttutorial.net/javascript-function/) with the following convention:

* The name of a constructor function starts with a capital letter like Person, Document, etc.
* A constructor function should be called only with the new operator.

Note that ES6 introduces the [class](https://www.javascripttutorial.net/es6/javascript-class/) keyword that allows you to define a custom type. And classes are just syntactic sugar over the constructor functions with some enhancements.

The following example defines a constructor function called Person:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

Code language: JavaScript (javascript)

In this example, the Person is the same as a regular function except that its name starts with the capital letter P.

To create a new instance of the Person, you use the new operator:

let person = new Person('John','Doe');

Code language: JavaScript (javascript)

Basically, the new operator does the following:

* Create a new empty object and assign it to the this variable.
* Assign the arguments 'John' and 'Doe' to the firstName and lastName properties of the object.
* Return the this value.

It’s functionally equivalent to the following:

function Person(firstName, lastName) {

*// this = {};*

*// add properties to this*

this.firstName = firstName;

this.lastName = lastName;

*// return this;*

}

Code language: JavaScript (javascript)

Therefore, the following statement:

let person = new Person('John','Doe');

Code language: JavaScript (javascript)

… returns the same result as the following statement:

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

However, the constructor function Person allows you to create multiple similar objects. For example:

let person1 = new Person('Jane','Doe')

let person2 = new Person('James','Smith')

Code language: JavaScript (javascript)

## **Adding methods to JavaScript constructor functions**

An object may have methods that manipulate its data. To add a method to an object created via the constructor function, you can use the this keyword. For example:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

this.getFullName = function () {

return this.firstName + " " + this.lastName;

};

}

Code language: JavaScript (javascript)

Now, you can create a new Person object and invoke the getFullName() method:

let person = new Person("John", "Doe");

console.log(person.getFullName());

Code language: JavaScript (javascript)

Output:

John Doe

The problem with the constructor function is that when you create multiple instances of the Person, the this.getFullName() is duplicated in every instance, which is not memory efficient.

To resolve this, you can use the [prototype](https://www.javascripttutorial.net/javascript-prototype/) so that all instances of a custom type can share the same methods.

## **Returning from constructor functions**

Typically, a constructor function implicitly returns this that set to the newly created object. But if it has a return statement, then here’s are the rules:

* If return is called with an object, the constructor function returns that object instead of this.
* If return is called with a value other than an object, it is ignored.

## **Calling a constructor function without the new keyword**

Technically, you can call a constructor function like a regular function without using the new keyword like this:

let person = Person('John','Doe');

Code language: JavaScript (javascript)

In this case, the Person just executes like a regular function. Therefore, the this inside the Person function doesn’t bind to the person variable but the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/).

If you attempt to access the firstName or lastName property, you’ll get an error:

console.log(person.firstName);

Code language: CSS (css)

Error:

TypeError: Cannot read property 'firstName' of undefined

Code language: JavaScript (javascript)

Similarly, you cannot access the getFullName() method since it’s bound to the global object.

person.getFullName();

Code language: CSS (css)

Error:

TypeError: Cannot read property 'getFullName' of undefined

Code language: JavaScript (javascript)

To prevent a constructor function to be invoked without the new keyword, ES6 introduced the [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) property.

If a constructor function is called with the new keyword, the new.target returns a reference of the function. Otherwise, it returns undefined.

The following adds a statement inside the Person function to show the new.target to the console:

function Person(firstName, lastName) {

console.log(new.target);

this.firstName = firstName;

this.lastName = lastName;

this.getFullName = function () {

return this.firstName + " " + this.lastName;

};

}

Code language: JavaScript (javascript)

The following returns undefined because the Person constructor function is called like a regular function:

let person = Person("John", "Doe");

Code language: JavaScript (javascript)

Output:

undefined

Code language: JavaScript (javascript)

However, the following returns a reference to the Person function because it’s called with the new keyword:

let person = new Person("John", "Doe");

Code language: JavaScript (javascript)

Output:

[Function: Person]

Code language: JSON / JSON with Comments (json)

By using the new.target, you can force the callers of the constructor function to use the new keyword. Otherwise, you can throw an error like this:

function Person(firstName, lastName) {

if (!new.target) {

throw Error("Cannot be called without the new keyword");

}

this.firstName = firstName;

this.lastName = lastName;

}

Code language: JavaScript (javascript)

Alternatively, you can make the syntax more flexible by creating a new Person object if the users of the constructor function don’t use the new keyword:

function Person(firstName, lastName) {

if (!new.target) {

return new Person(firstName, lastName);

}

this.firstName = firstName;

this.lastName = lastName;

}

let person = Person("John", "Doe");

console.log(person.firstName);

Code language: JavaScript (javascript)

This pattern is often used in JavaScript libraries and frameworks to make the syntax more flexible.

## **Summary**

* JavaScript constructor function is a regular function used to create multiple similar objects.

# JavaScript Prototype

**Summary**: in this tutorial, you’ll learn about the JavaScript prototype and how it works under the hood.

## **Introduction to JavaScript prototype**

JavaScript has the built-in Object() function. The [typeof](https://www.javascripttutorial.net/javascript-typeof/) operator returns 'function' if you pass the Object function to it. For example:

typeof(Object)

Code language: JavaScript (javascript)

Output:

'function'

Code language: JavaScript (javascript)

Please note that Object() is a function, not an object. It’s confusing if this is the first time you’ve learned about the JavaScript prototype.

Also, JavaScript provides an anonymous [object](https://www.javascripttutorial.net/javascript-objects/) that can be referenced via the prototype property of the Object() function:

console.log(Object.prototype);

Code language: JavaScript (javascript)

[Text, letter

Description automatically generated](https://www.javascripttutorial.net/wp-content/uploads/2021/01/JavaScript-Prototype-Object.prototype.png)

The Object.prototype object has some useful [properties](https://www.javascripttutorial.net/javascript-object-properties/) and [methods](https://www.javascripttutorial.net/javascript-object-methods/) such as toString() and valueOf().

Note when a function is a value of an object property, it’s called a method. Therefore, a method is an object property with value as a function.

The Object.prototype also has an important property called constructor that references the Object() function.

The following statement confirms that the Object.prototype.constructor property references the Object function:

console.log(Object.prototype.constructor === Object); *// true*

Code language: JavaScript (javascript)

Suppose a circle represents a function and a square represents an object. The following picture illustrates the relationship between the Object() function and the Object.prototype object:

First, define a [constructor function](https://www.javascripttutorial.net/javascript-constructor-function/) called Person as follows:

function Person(name) {

this.name = name;

}

Code language: JavaScript (javascript)

In this example, the Person() function accepts a name argument and assigns it to the name property of the this object.

Behind the scenes, JavaScript creates a new function Person() and an anonymous object:

Like the Object() function, the Person() function has a property called prototype that references an anonymous object. And the anonymous object has the constructor property that references the Person() function.

The following shows the Person() function and the anonymous object referenced by the Person.prototype:

console.log(Person);

console.log(Person.prototype);

Code language: CSS (css)

[Graphical user interface, application

Description automatically generated with medium confidence](https://www.javascripttutorial.net/wp-content/uploads/2021/01/JavaScript-Prototype-Person-function.png)

In addition, JavaScript links the Person.prototype object to the Object.prototype object via the [[Prototype]], which is known as a prototype linkage.

The prototype linkage is denoted by [[Prototype]] in the following figure:

## **Defining methods in the JavaScript prototype object**

The following defines a new method called greet() in the Person.prototype object:

Person.prototype.greet = function() {

return "Hi, I'm " + this.name + "!";

}

Code language: JavaScript (javascript)

In this case, the JavaScript engine adds the greet() method to the Person.prototype object:

The following creates a new instance of the Person :

let p1 = new Person('John');

Code language: JavaScript (javascript)

Internally, the JavaScript engine creates a new object named p1 and links the p1 object to the Person.prototype object via the prototype linkage:

The link between p1, Person.prototype, and Object.protoype is called a prototype chain.

The following calls the greet() method on the p1 object:

let greeting = p1.greet();

console.log(greeting);

Code language: JavaScript (javascript)

Because p1 doesn’t have the greet() method, JavaScript follows the prototype linkage and finds it on the Person.prototype object.

Since JavaScript can find the greet() method on the Person.prototype object, it executes the greet() method and returns the result:

The following calls the toString() method on the p1 object:

let s = p1.toString();

console.log(s);

Code language: JavaScript (javascript)

In this case, the JavaScript engine follows the prototype chain to look up for the toString() method in the Person.prototype.

Because the Person.prototype doesn’t have the toString() method, the JavaScript engine goes up to the prototype chain and searches for the toString() method in the Object.prototype object.

Since JavaScript can find the toString() method in the Object.prototype, it executes the toString() method.

If you call a method that doesn’t exist on the Person.prototype and Object.prototype object, the JavaScript engine will follow the prototype chain and throw an error if it cannot find the method. For example:

p1.fly();

Code language: CSS (css)

Because the fly() method doesn’t exist on any object in the prototype chain, the JavaScript engine issues the following error:

TypeError: p1.fly is not a function

Code language: JavaScript (javascript)

The following creates another instance of the Person whose name property is 'Jane':

let p2 = new Person('Jane');

Code language: JavaScript (javascript)

The p2 object has the properties and methods as the p1 object.

In conclusion, when you define a method on the prototype object, this method is shared by all instances.

## **Defining methods in an individual object**

The following defines the draw() method on the p1 object.

p1.draw = function () {

return "I can draw.";

};

Code language: JavaScript (javascript)

The JavaScript engine adds the draw() method to the p1 object, not the Person.prototype object:

It means that you can call the draw() method on the p1 object:

p1.draw();

Code language: CSS (css)

But you cannot call the draw() method on the p2 object:

p2.draw()

Code language: CSS (css)

Error:

TypeError: p2.draw is not a function

Code language: JavaScript (javascript)

When you define a method in an object, the method is only available to that object. It cannot be shared with other objects by default.

## **Getting prototype linkage**

The \_\_proto\_\_ is pronounced as dunder proto. The \_\_proto\_\_ is an [accessor property](https://www.javascripttutorial.net/javascript-object-properties/) of the Object.prototype object. It exposes the internal prototype linkage ( [[Prototype]]) of an object through which it is accessed.

The \_\_proto\_\_ has been standardized in [ES6](https://www.javascripttutorial.net/es6/) to ensure compatibility for web browsers. However, it may be deprecated in favor of Object.getPrototypeOf() in the future. Therefore, you should never use the \_\_proto\_\_ in your production code.

The  p1.\_\_proto\_\_ exposes the [[Prototype]] that references the Person.prototype object.

Similarly, p2.\_\_proto\_\_ also references the same object as p1.\_\_proto\_\_:

console.log(p1.\_\_proto\_\_ === Person.prototype); *// true*

console.log(p1.\_\_proto\_\_ === p2.\_\_proto\_\_); *// true*

Code language: JavaScript (javascript)

As mentioned earlier, you should use the Object.getPrototypeOf() method instead of the \_\_proto\_\_.

The Object.getPrototypeOf() method returns the prototype of a specified object.

console.log(p1.\_\_proto\_\_ === Object.getPrototypeOf(p1)); *// true*

Code language: JavaScript (javascript)

Another popular way to get the prototype linkage is when the Object.getPrototypeOf() method is not available is via the constructor property as follows:

p1.constructor.prototype

Code language: CSS (css)

The p1.constructor returns Person, therefore, p1.constructor.prototype returns the prototype object.

## **Shadowing**

See the following method call:

console.log(p1.greet());

Code language: CSS (css)

The p1 object doesn’t have the greet() method defined, therefore JavaScript goes up to the prototype chain to find it. In this case, it can find the method in the Person.prototype object.

Let’s add a new method to the object p1 with the same name as the method in the Person.prototype object:

p1.greet = function() {

console.log('Hello');

}

Code language: JavaScript (javascript)

And call the greet() method:

console.log(p1.greet());

Code language: CSS (css)

Because the p1 object has the greet() method, JavaScript just executes it immediately without looking it up in the prototype chain.

This is an example of shadowing. The greet() method of the p1 object shadows the greet() method of the prototype object which the p1 object references.

## **Summary**

* The Object() function has a property called prototype that references a Object.prototype object.
* The Object.prototype object has all properties and methods which are available in all objects such as toString() and valueOf().
* The Object.prototype object has the constructor property that references the Object function.
* Every function has a prototype object. This prototype object references the Object.prototype object via [[prototype]] linkage or \_\_proto\_\_ property.
* The prototype chain allows one object to use the methods and properties of its prototype objects via the [[prototype]] linkages.
* The Object.getPrototypeOf() method returns the prototype object of a given object. Do use the Object.getPrototypeOf() method instead of \_\_proto\_\_.

# JavaScript Constructor/Prototype Pattern

**Summary**: in this tutorial, you’ll learn how to use the JavaScript constructor/Prototype pattern to define a custom type in ES5.

## **Introduction to the JavaScript Constructor / Prototype pattern**

The combination of the [constructor](https://www.javascripttutorial.net/javascript-constructor-function/)and [prototype](https://www.javascripttutorial.net/javascript-prototype/) patterns is the most common way to define custom types in ES5. In this pattern:

* The constructor pattern defines the object properties.
* The prototype pattern defines the object methods.

By using this pattern, all objects of the custom type share the methods defined in the prototype. Also, each object has its own properties.

This constructor/prototype pattern takes the best parts of both constructor and prototype patterns.

## **JavaScript Constructor / Prototype example**

Suppose that you want to define a custom type called Person that has:

* Two properties firstName and lastName
* One method getFullName()

First, use the [constructor function](https://www.javascripttutorial.net/javascript-constructor-function/) to initialize the properties:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

Code language: JavaScript (javascript)

Behind the scene, the JavaScript engine defines a Person function denoted by the circle and an anonymous object denoted by the square.

The Person function has the prototype property that references an anonymous object. The anonymous object has a constructor property that references the Person function:

Second, define the getFullName() method in the prototype object of the Person function:

Person.prototype.getFullName = function () {

return this.firstName + ' ' + this.lastName;

};

Code language: JavaScript (javascript)

JavaScript defines the getFullName() method on the Person.prototype object like this:

Third, create multiple instances of the Person type:

let p1 = new Person("John", "Doe");

let p2 = new Person("Jane", "Doe");

console.log(p1.getFullName());

console.log(p2.getFullName());

Code language: JavaScript (javascript)

Output:

'John Doe'

'Jane Doe'

Code language: JavaScript (javascript)

Javascript creates two objects p1 and p2. These objects link to the Person.prototype object via the [[Prototype]] linkage:

Each object has its own properties firstName and lastName. However, they share the same getFullName() method.

When you call the getFullName() method on the p1 or p2 object, the JavaScript engine searches for the method on these objects. Because the JavaScript engine doesn’t find the method there, it follows the prototype linkage and searches for the method in the Person.prototype object.

Because the Person.prototype object has the getFullName() method, JavaScript stops searching and executes the method.

Put it all together:

function Person(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

Person.prototype.getFullName = function () {

return this.firstName + ' ' + this.lastName;

};

let p1 = new Person('John', 'Doe');

let p2 = new Person('Jane', 'Doe');

console.log(p1.getFullName());

console.log(p2.getFullName());

Code language: JavaScript (javascript)

## **Classes in ES6**

ES6 introduces the [class](https://www.javascripttutorial.net/es6/javascript-class/) keyword that makes the constructor/prototype pattern easier to use. For example, the following uses the class keyword to define exactly the same Person type:

class Person {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

getFullName() {

return this.firstName + " " + this.lastName;

}

}

let p1 = new Person('John', 'Doe');

let p2 = new Person('Jane', 'Doe');

console.log(p1.getFullName());

console.log(p2.getFullName());

Code language: JavaScript (javascript)

In this syntax, the class moves the property initialization to the constructor method. It also packs the getFullName() method in the same place as the constructor function.

The class syntax looks cleaner and less verbose. However, it’s syntactic sugar over the constructor/prototype pattern with some enhancements.

For more information on the classes, check out the [JavaScript class tutorial](https://www.javascripttutorial.net/es6/javascript-class/).

## **Summary**

* Use JavaScript constructor/prototype to define a custom type in ES5.
* Initialize the object properties in the constructor function and define methods and properties that can be shared by all instances in the prototype object.

# JavaScript Prototypal Inheritance

**Summary**: in this tutorial, you’ll learn how the JavaScript prototypal inheritance works.

## **Introduction to JavaScript prototypal inheritance**

If you’ve worked with other object-oriented programming languages such as Java or C++, you’ve been familiar with the inheritance concept.

In this programming paradigm, a class is a blueprint for creating objects. If you want a new class to reuse the functionality of an existing class, you can create a new class that extends the existing class. This is called **classical inheritance**.

JavaScript doesn’t use **classical inheritance**. Instead, it uses **prototypal inheritance**.

In prototypal inheritance, an object “inherits” [properties](https://www.javascripttutorial.net/javascript-object-properties/) from another object via the [prototype](https://www.javascripttutorial.net/javascript-prototype/) linkage.

## **JavaScript prototypal inheritance and \_\_proto\_\_**

Let’s take an example to make the concept clear.

The following defines a person object:

let person = {

name: "John Doe",

greet: function () {

return "Hi, I'm " + this.name;

}

};

Code language: JavaScript (javascript)

In this example, the person object has a property and a method:

* name is a property that stores the person’s name.
* greet is a method that returns a greeting as a string.

By default, the JavaScript engine provides you with a built-in Object() function and an anonymous object that can be referenced by the Object.prototype:

Note that the circle represents a function while the square represents an object.

The person object has a link to the anonymous object referenced by the Object() function. The [[Prototype]] represents the linkage:

It means that the person object can call any methods defined in the anonymous object referenced by the Object.prototype likes this. For example, the following shows how to call the toString() method via the person object:

console.log(person.toString());

Code language: JavaScript (javascript)

Output:

[object Object]

Code language: JavaScript (javascript)

The [object Object] is the default string representation of an object.

When you call toString() method via person, the JavaScript engine cannot find it on the person object. Therefore, the JavaScript engine follows the prototype chain and searches for the method in the Object.prototype object.

Since the JavaScript engine can find the toString() method in the Object.prototype object, it executes the toString() method.

To access the prototype of the person object, you can use the \_\_proto\_\_ property as follows

console.log(person.\_\_proto\_\_);

Code language: JavaScript (javascript)

The following shows the person.\_\_proto\_\_ and Object.prototype references the same object:

console.log(person.\_\_proto\_\_ === Object.prototype); *// true*

Code language: JavaScript (javascript)

The following defines the teacher object that has the teach() method:

let teacher = {

teach: function (subject) {

return "I can teach " + subject;

}

};

Code language: JavaScript (javascript)

Like the person object, the teacher.\_\_proto\_\_ references the Object.prototype as illustrated in the following picture:

If you want the teacher object to access all methods and properties of the person object, you can set the prototype of teacher object to the person object like this:

teacher.\_\_proto\_\_ = person;

Code language: JavaScript (javascript)

Note that you should never use the \_\_proto\_\_ property in the production code. Please use it for demonstration purposes only.

Now, the teacher object can access the name property and greet() method from the person object via the prototype chain:

console.log(teacher.name);

console.log(teacher.greet());

Code language: JavaScript (javascript)

Output:

John Doe

Hi, I'm John Doe

Code language: JavaScript (javascript)

When you call the greet() method on the teacher object, the JavaScript engine finds it in the teacher object first.

Since the JavaScript engine cannot find the method in the teacher object, it follows the prototype chain and searches for the method in the person object. Because the JavaScript can engine can find the greet() method in the person object, it executes the method.

In JavaScript, we say that the teacher object inherits the methods and properties of the person object. And this kind of inheritance is called prototypal inheritance.

## **A standard way to implement prototypal inheritance in ES5**

ES5 provided a standard way to work with prototypal inheritance by using the Object.create() method.

Note that now you should use the newer ES6 [class](https://www.javascripttutorial.net/es6/javascript-class/) and [extends](https://www.javascripttutorial.net/es6/javascript-inheritance/) keywords to implement [inheritance](https://www.javascripttutorial.net/es6/javascript-inheritance/). It’s much simpler.

The Object.create() method creates a new object and uses an existing object as a prototype of the new object:

Object.create(proto, [propertiesObject])

Code language: JavaScript (javascript)

The Object.create() method accepts two arguments:

* The first argument (proto) is an object used as the prototype for the new object.
* The second argument (propertiesObject), if provided, is an optional object that defines additional properties for the new object.

Suppose you have a person object:

let person = {

name: "John Doe",

greet: function () {

return "Hi, I'm " + this.name;

}

};

Code language: JavaScript (javascript)

The following creates an empty teacher object with the \_\_proto\_\_ of the person object:

let teacher = Object.create(person);

Code language: JavaScript (javascript)

After that, you can define properties for the teacher object:

teacher.name = 'Jane Doe';

teacher.teach = function (subject) {

return "I can teach " + subject;

}

Code language: JavaScript (javascript)

Or you can do all of these steps in one statement as follows:

let teacher = Object.create(person, {

name: { value: 'John Doe' } ,

teach: { value: function(subject) {

return "I can teach " + subject;

}}

});

Code language: JavaScript (javascript)

ES5 also introduced the Object.getPrototypeOf() method that returns the prototype of an object. For example:

console.log(Object.getPrototypeOf(teacher) === person);

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

## **Summary**

* Inheritance allows an object to use the properties and methods of another object without duplicating the code.
* JavaScript uses the prototypal inheritance.

# Demystifying the JavaScript this Keyword

**Summary**: in this tutorial, you will  learn about the JavaScript this value and understand it clearly in various contexts.

If you have been working with other programming languages such as Java, [C#](https://www.csharptutorial.net/csharp-tutorial/csharp-this/), or [PHP](https://www.phptutorial.net/php-oop/php-this/), you’re already familiar with the this keyword.

In these languages, the this keyword represents the current instance of the class. And it is only relevant within the class.

JavaScript also has this keyword. However, the this keyword in JavaScript behaves differently from other programming languages.

In JavaScript, you can use the this keyword in the [global and function contexts](https://www.javascripttutorial.net/javascript-execution-context/). Moreover, the behavior of the  this keyword changes between strict and non-strict modes.

## **What is the this keyword**

In general, the this references the object of which the function is a property. In other words, the this references the object that is currently calling the function.

Suppose you have an object called counter that has a method next(). When you call the next() method, you can access the this object.

let counter = {

count: 0,

next: function () {

return ++this.count;

},

};

counter.next();

Code language: JavaScript (javascript)

Inside the next() function, the this references the counter object. See the following method call:

counter.next();

Code language: CSS (css)

The next() is a function that is the property of the counter object. Therefore, inside the next() function, the this references the counter object.

## **Global context**

In the global context, the this references the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/), which is the window object on the web browser or global object on Node.js.

This behavior is consistent in both strict and non-strict modes. Here’s the output on the web browser:

console.log(this === window); *// true*

Code language: JavaScript (javascript)

If you assign a property to this object in the global context, JavaScript will add the property to the global object as shown in the following example:

this.color= 'Red';

console.log(window.color); *// 'Red'*

Code language: JavaScript (javascript)

## **Function context**

In JavaScript, you can call a [function](https://www.javascripttutorial.net/javascript-function/) in the following ways:

* Function invocation
* Method invocation
* Constructor invocation
* Indirect invocation

Each function invocation defines its own context. Therefore, the this behaves differently.

### **1) Simple function invocation**

In the non-strict mode, the this references the global object when the function is called as follows:

function show() {

console.log(this === window); *// true*

}

show();

Code language: JavaScript (javascript)

When you call the show() function, the this references the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/), which is the window on the web browser and global on Node.js.

Calling the show() function is the same as:

window.show();

Code language: JavaScript (javascript)

In the strict mode, JavaScript sets the this inside a function to undefined. For example:

"use strict";

function show() {

console.log(this === undefined);

}

show();

Code language: JavaScript (javascript)

To enable the strict mode, you use the directive "use strict" at the beginning of the JavaScript file. If you want to apply the strict mode to a specific function only, you place it at the top of the function body.

Note that the strict mode has been available since ECMAScript 5.1. The strict mode applies to both function and nested functions. For example:

function show() {

"use strict";

console.log(this === undefined); *// true*

function display() {

console.log(this === undefined); *// true*

}

display();

}

show();

Code language: JavaScript (javascript)

Output:

true

true

Code language: JavaScript (javascript)

In the display() inner function, the this also set to undefined as shown in the console.

### **2) Method invocation**

When you call a method of an object, JavaScript sets this to the object that owns the method. See the following car object:

let car = {

brand: 'Honda',

getBrand: function () {

return this.brand;

}

}

console.log(car.getBrand()); *// Honda*

Code language: JavaScript (javascript)

In this example, the this object in the getBrand() method references the car object.

Since a method is a property of an object which is a value, you can store it in a variable.

let brand = car.getBrand;

Code language: JavaScript (javascript)

And then call the method via the variable

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

Code language: JavaScript (javascript)

You get undefined instead of "Honda" because when you call a method without specifying its object, JavaScript sets this to the global object in non-strict mode and undefined in the strict mode.

To fix this issue, you use the [bind()](https://www.javascripttutorial.net/javascript-bind/) method of the Function.prototype object. The bind() method creates a new function whose the this keyword is set to a specified value.

let brand = car.getBrand.bind(car);

console.log(brand()); *// Honda*

Code language: JavaScript (javascript)

In this example, when you call the brand() method, the this keyword is bound to the car object. For example:

let car = {

brand: 'Honda',

getBrand: function () {

return this.brand;

}

}

let bike = {

brand: 'Harley Davidson'

}

let brand = car.getBrand.bind(bike);

console.log(brand());

Code language: JavaScript (javascript)

Output:

Harley Davidson

In this example, the bind() method sets the this to the bike object, therefore, you see the value of the brand property of the bike object on the console.

### **3) Constructor invocation**

When you use the new keyword to create an instance of a function object, you use the function as a constructor.

The following example declares a Car function, then invokes it as a constructor:

function Car(brand) {

this.brand = brand;

}

Car.prototype.getBrand = function () {

return this.brand;

}

let car = new Car('Honda');

console.log(car.getBrand());

Code language: JavaScript (javascript)

The expression new Car('Honda') is a constructor invocation of the Car function.

JavaScript creates a new object and sets this to the newly created object. This pattern works great with only one potential problem.

Now, you can invoke the Car() as a function or as a constructor. If you omit the new keyword as follows:

var bmw = Car('BMW');

console.log(bmw.brand);

*// => TypeError: Cannot read property 'brand' of undefined*

Code language: JavaScript (javascript)

Since the this value in the Car() sets to the global object, the bmw.brand returns undefined.

To make sure that the Car() function is always invoked using constructor invocation, you add a check at the beginning of the Car() function as follows:

function Car(brand) {

if (!(this instanceof Car)) {

throw Error('Must use the new operator to call the function');

}

this.brand = brand;

}

Code language: JavaScript (javascript)

ES6 introduced a meta-property named [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) that allows you to detect whether a function is invoked as a simple invocation or as a constructor.

You can modify the Car() function that uses the new.target metaproperty as follows:

function Car(brand) {

if (!new.target) {

throw Error('Must use the new operator to call the function');

}

this.brand = brand;

}

Code language: JavaScript (javascript)

### **4) Indirect Invocation**

In JavaScript, [functions are first-class citizens](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/). In other words, functions are objects, which are instances of the [Function type](https://www.javascripttutorial.net/javascript-function-type/).

The Function type has two methods: [call()](https://www.javascripttutorial.net/javascript-call/) and [apply()](https://www.javascripttutorial.net/javascript-apply-method/) . These methods allow you to set the this value when calling a function. For example:

function getBrand(prefix) {

console.log(prefix + this.brand);

}

let honda = {

brand: 'Honda'

};

let audi = {

brand: 'Audi'

};

getBrand.call(honda, "It's a ");

getBrand.call(audi, "It's an ");

Code language: JavaScript (javascript)

Output:

It's a Honda

It's an Audi

Code language: PHP (php)

In this example, we called the getBrand() function indirectly using the call() method of the getBrand function. We passed honda and  audi object as the first argument of the call() method, therefore, we got the corresponding brand in each call.

The apply() method is similar to the call() method except that its second argument is an array of arguments.

getBrand.apply(honda, ["It's a "]); *// "It's a Honda"*

getBrand.apply(audi, ["It's an "]); *// "It's a Audi"*

Code language: JavaScript (javascript)

## **Arrow functions**

[ES6](https://www.javascripttutorial.net/es6/)introduced a new concept named [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/). In arrow functions, JavaScript sets the this lexically.

It means the arrow function does not create its own [execution context](https://www.javascripttutorial.net/javascript-execution-context/) but inherits the this from the outer function where the arrow function is defined. See the following example:

let getThis = () => this;

console.log(getThis() === window); *// true*

Code language: JavaScript (javascript)

In this example, the this value is set to the global object i.e., window in the web browser.

Since an arrow function does not create its own execution context, defining a method using an arrow function will cause an issue. For example:

function Car() {

this.speed = 120;

}

Car.prototype.getSpeed = () => {

return this.speed;

}

var car = new Car();

car.getSpeed(); *// TypeError*

Code language: JavaScript (javascript)

Inside the getSpeed() method, the this value reference the global object, not the Car object. Therefore the car.getSpeed() invocation causes an error because the global object does not have the speed property.

# JavaScript globalThis

**Summary**: in this tutorial, you’ll learn how to about the JavaScript globalThis object.

## **Introduction to the JavaScript globalThis object**

ES2020 introduced the globalThis object that provides a standard way to access the global object across environments.

Historically, JavaScript had a global object with different names in different environments.

In web browsers, the global object is [window](https://www.javascripttutorial.net/javascript-bom/javascript-window/) or frames.

However, the Web Workers API doesn’t have the window object because it has no browsing context. Hence, the Web Workers API uses self as a global object.

Node.js, on the other hand, uses the global keyword to reference the global object.

| **Environment** | **Global** |
| --- | --- |
| Web Browsers | this |
| Web Workers | self |
| Node.js | global |

If you write JavaScript code that works across environments and needs to access the global object, you have to use different syntaxes like window, frames, self, or global.

To standardize this, ES2020 introduced the globalThis that is available across environments.

For example, the following code checks if the current environment supports the Fetch API:

const canFetch = typeof globalThis.fetch === 'function';

console.log(canFetch);

Code language: JavaScript (javascript)

The code checks if the fetch() function is a property of the global object. In the web browsers, the globalThis is the window object. Therefore, if you run this code on the modern web browser, the canFetch will be true.

The following code returns true on the web browser:

globalThis === window

Code language: JavaScript (javascript)

Output:

true

Code language: JavaScript (javascript)

## **Summary**

* Use the globalThis object to reference the global object to make the code works across environments.

JavaScript Object Properties

**Summary**: in this tutorial, you will learn about the JavaScript object’s properties and attributes such as configurable, enumerable, writable, get, set, and value.

## **Object Property types**

JavaScript specifies the characteristics of properties of [objects](https://www.javascripttutorial.net/javascript-objects/) via internal attributes surrounded by the two pairs of square brackets, e.g., [[Enumerable]].

Objects have two types of properties: data and accessor properties.

### **1) Data properties**

A data property contains a single location for a data value. A data property has four attributes:

* [[Configurarable]] – determines whether a property can be redefined or removed via delete operator.
* [[Enumerable]] – indicates if a property can be returned in the [for...in](https://www.javascripttutorial.net/javascript-for-in/) loop.
* [[Writable]] – specifies that the value of a property can be changed.
* [[Value]] – contains the actual value of a property.

By default, the [[Configurable]] , [[Enumerable]]And [[Writable]] attributes set to true for all properties defined directly on an object. The default value of the[[Value]] attribute is undefined.

The following example creates a person object with two properties firstName and lastName with the configurable, enumerable, and writable attributes set to true. And their values are set to 'John' and 'Doe' respectively:

let person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

To change any attribute of a property, you use the Object.defineProperty() method.

The Object.defineProperty() method accepts three arguments:

* An object.
* A property name of the object.
* A property descriptor object that has four properties: configurable, enumerable, writable, and value.

If you use the Object.defineProperty() method to define a property of the object, the default values of [[Configurable]], [[Enumerable]], and [[Writable]] are set to false unless otherwise specified.

The following example creates a person object with the age property:

let person = {};

person.age = 25;

Code language: JavaScript (javascript)

Since the default value of the [[Configurable]] attribute is set to true, you can remove it via the delete operator:

delete person.age;

console.log(person.age);

Code language: CSS (css)

Output:

undefined

Code language: JavaScript (javascript)

The following example creates a person object and adds the ssn property to it using the Object.defineProperty() method:

'use strict';

let person = {};

Object.defineProperty(person, 'ssn', {

configurable: false,

value: '012-38-9119'

});

delete person.ssn;

Code language: JavaScript (javascript)

Output:

TypeError: Cannot delete property 'ssn' of *#<Object>*

Code language: PHP (php)

In this example, the configurable attribute is set to false. herefore, deleting the ssn property causes an error.

Also, once you define a property as non-configurable, you cannot change it to configurable.

If you use the Object.defineProperty() method to change any attribute other than the writable, you’ll get an error. or example:

'use strict';

let person = {};

Object.defineProperty(person, 'ssn', {

configurable: false,

value: '012-38-9119'

});

Object.defineProperty(person, 'ssn', {

configurable: true

});

Code language: JavaScript (javascript)

Output:

TypeError: Cannot redefine property: ssn

Code language: JavaScript (javascript)

By default, the enumerable attribute of all the properties defined on an object is true. t means that you can iterate over all object properties using the [for...in](https://www.javascripttutorial.net/javascript-for-in/)loop like this:

let person = {};

person.age = 25;

person.ssn = '012-38-9119';

for (let property in person) {

console.log(property);

}

Code language: JavaScript (javascript)

Output:

age

ssn

The following makes the ssn property non-enumerable by setting the enumerable attribute to false.

let person = {};

person.age = 25;

person.ssn = '012-38-9119';

Object.defineProperty(person, 'ssn', {

enumerable: false

});

for (let prop in person) {

console.log(prop);

}

Code language: JavaScript (javascript)

Output

age

### **2) Accessor properties**

Similar to data properties, accessor properties also have [[Configurable]] and [[Enumerable]] attributes.

But the accessor properties have the [[Get]] and [[Set]] attributes instead of [[Value]] and [[Writable]].

When you read data from an accessor property, the [[Get]] function is called automatically to return a value. The default return value of the [[Get]] function is undefined.

If you assign a value to an accessor property, the [[Set]] function is called automatically.

To define an accessor property, you must use the Object.defineProperty() method. or example:

let person = {

firstName: 'John',

lastName: 'Doe'

}

Object.defineProperty(person, 'fullName', {

get: function () {

return this.firstName + ' ' + this.lastName;

},

set: function (value) {

let parts = value.split(' ');

if (parts.length == 2) {

this.firstName = parts[0];

this.lastName = parts[1];

} else {

throw 'Invalid name format';

}

}

});

console.log(person.fullName);

Code language: JavaScript (javascript)

Output:

'John Doe'

Code language: JavaScript (javascript)

In this example:

* First, define the person object that contains two properties: firstName and lastName.
* Then, add the fullName property to the person object as an accessor property.

In the fullname accessor property:

* The [[Get]] returns the full name that is the result of [concatenating](https://www.javascripttutorial.net/javascript-string-concat/) of firstName, space, and lastName.
* The [[Set]] method [splits](https://www.javascripttutorial.net/javascript-string-split/) the argument by the space and assigns the firstName and lastName properties the corresponding parts of the name.
* If the full name is not in the correct format i.e., first name, space, and last name, it will [throw an error](https://www.javascripttutorial.net/es6/promise-error-handling/).

## **Define multiple properties: Object.defineProperties()**

In ES5, you can define multiple properties in a single statement using the Object.defineProperties() method. or example:

var product = {};

Object.defineProperties(product, {

name: {

value: 'Smartphone'

},

price: {

value: 799

},

tax: {

value: 0.1

},

netPrice: {

get: function () {

return this.price \* (1 + this.tax);

}

}

});

console.log('The net price of a ' + product.name + ' is ' + product.netPrice.toFixed(2) + ' USD');

Code language: JavaScript (javascript)

Output:

The net price of a Smartphone is 878.90 USD

Code language: CSS (css)

In this example, we defined three data properties: name, price, and tax, and one accessor property netPrice for the product object.

## **JavaScript object property descriptor**

The  Object.getOwnPropertyDescriptor() method allows you to get the descriptor object of a property. The Object.getOwnPropertyDescriptor() method takes two arguments:

1. An object
2. A property of the object

It returns a descriptor object that describes a property. The descriptor object has four properties: configurable, enumerable, writable, and value.

The following example gets the descriptor object of the name property of the product object in the prior example.

let person = {

firstName: 'John',

lastName: 'Doe'

};

let descriptor = Object.getOwnPropertyDescriptor(person, 'firstName');

console.log(descriptor);

Code language: JavaScript (javascript)

Output:

{ value: 'John',

writable: true,

enumerable: true,

configurable: true }

Code language: CSS (css)

## **Summary**

* JavaScript objects have two types of properties: data properties and accessor properties.
* JavaScript uses internal attributes denoted by [[...]] to describe the characteristics of properties such as [[Configurable]], [[Enumerable]], [[Writable]], and  [[Value]], [[Get]], and [[Set]].
* The method Object.getOwnPropertyDescriptor() return a property descriptor of a property in an object.
* A property can be defined directly on an object or indirectly via the Object.defineProperty() or Object.defineProperties() methods. These methods can be used to change the attributes of a property.

# JavaScript for…in Loop

**Summary**: in this tutorial, you will learn how to use the JavaScript for...in loop to iterate over the enumerable properties of an object.

## **Introduction to JavaScript for...in loop**

The for...in loop over the [enumerable properties](https://www.javascripttutorial.net/javascript-enumerable-properties/) that are keyed by strings of an [object](https://www.javascripttutorial.net/javascript-objects/). Note that a property can be keyed by a string or a [symbol](https://www.javascripttutorial.net/es6/symbol/).

A property is enumerable when its internal enumerable flag is set to true.

The enumerable flag defaults to true when a property is created via a simple assignment or via a property initializer:

object.propertyName = value;

or

let obj = {

propertyName: value,

...

};

Code language: JavaScript (javascript)

The following shows the syntax of the for...in loop:

for(const propertyName in object) {

*// ...*

}

Code language: JavaScript (javascript)

The for...in  allows you to access each property and value of an object without knowing the specific name of the property. For example:

var person = {

firstName: 'John',

lastName: 'Doe',

ssn: '299-24-2351'

};

for(var prop in person) {

console.log(prop + ':' + person[prop]);

}

Code language: JavaScript (javascript)

Output:

firstName:John

lastName:Doe

ssn:299-24-2351

Code language: CSS (css)

In this example, we used the for...in loop to iterate over the properties of the person object. We accessed the value of each property using the following syntax:

object[property];

Code language: CSS (css)

## **The for...in loop & inheritance**

When you loop over properties of an object that [inherits](https://www.javascripttutorial.net/javascript-prototypal-inheritance/) from another object, the for...in statement goes up in the [prototype](https://www.javascripttutorial.net/javascript-prototype/) chain and enumerates over inherited properties. Consider the following example:

var decoration = {

color: 'red'

};

var circle = Object.create(decoration);

circle.radius = 10;

for(const prop in circle) {

console.log(prop);

}

Code language: JavaScript (javascript)

Output:

radius

color

The circle object has its own prototype that references the decoration object. Therefore, the for...in loop displays the properties of the circle object and its prototype.

If you want to enumerate only the [own properties](https://www.javascripttutorial.net/javascript-own-properties/) of an object, you use the hasOwnProperty() method:

for(const prop in circle) {

if(circle.hasOwnProperty(prop)) {

console.log(prop);

}

}

Code language: JavaScript (javascript)

Output:

radius

## **The for...in loop and Array**

It’s good practice to not use the for...in to iterate over an [array](https://www.javascripttutorial.net/javascript-array/), especially when the order of the array elements is important.

The following example works flawlessly:

const items = [10 , 20, 30];

let total = 0;

for(const item in items) {

total += items[item];

}

console.log(total);

Code language: JavaScript (javascript)

However, someone may set a property of the built-in [Array](https://www.javascripttutorial.net/javascript-array/) type in their libraries as follows:

Array.prototype.foo = 100;

Code language: JavaScript (javascript)

Hence, the for...in will not work correctly:

for(var prop in items) {

total += items[prop];

}

console.log(total);

Code language: JavaScript (javascript)

Output:

220

Or another example:

var arr = [];

*// set the third element to 3, other elements are `undefined`*

arr[2] = 3;

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

console.log(arr[i]);

}

Code language: JavaScript (javascript)

The output shows three elements of the array, which is correct:

undefined

undefined

3

Code language: JavaScript (javascript)

However, the for...in loop ignores the first two elements:

for (const key in arr) {

console.log(arr[key]);

}

Code language: JavaScript (javascript)

Output:

3

The output shows only the third element, not the first two elements.

## **Summary**

* The for...in loop iterates over the [enumerable properties](https://www.javascripttutorial.net/javascript-enumerable-properties/) of an object. It also goes up to the [prototype](https://www.javascripttutorial.net/javascript-prototype/) chain and enumerates over inherited properties.
* Avoid using for...in loop to iterate over elements of an array, especially when the index order is important.

# A Basic Guide to Enumerable Properties of an Object in JavaScript

**Summary**: in this tutorial, you will learn about JavaScript enumerable properties of an object.

## **Introduction to JavaScript enumerable properties**

In JavaScript, an [object](https://www.javascripttutorial.net/javascript-objects/) is an unordered list of key-value pairs. The key is usually a [string](https://www.javascripttutorial.net/javascript-string/) or a [symbol](https://www.javascripttutorial.net/es6/symbol/). The value can be a value of any primitive type (string, boolean, number, undefined, or null), an object, or a [function](https://www.javascripttutorial.net/javascript-function/).

The following example creates a new object using the object literal syntax:

const person = {

firstName: 'John',

lastName: 'Doe

};

Code language: PHP (php)

The person object has two properties: firstName and lastName.

An object property has several internal attributes including value, writable, enumerable and configurable. See the [Object properties](https://www.javascripttutorial.net/javascript-object-properties/) for more details.

The enumerable attribute determines whether or not a property is accessible when the object’s properties are enumerated using the [for...in](https://www.javascripttutorial.net/javascript-for-in/) loop or Object.keys() method.

By default, all properties created via a simple assignment or via a property initializer are enumerable. For example:

const person = {

firstName: 'John',

lastName: 'Doe'

};

person.age = 25;

for (const key in person) {

console.log(key);

}

Code language: JavaScript (javascript)

Output:

firstName

lastName

age

In this example:

* The firstName and lastName are enumerable properties because they are created via a property initializer.
* The age property is also enumerable because it is created via a simple assignment.

To change the internal enumerable attribute of a property, you use the Object.defineProperty() method. For example:

const person = {

firstName: 'John',

lastName: 'Doe'

};

person.age = 25;

Object.defineProperty(person, 'ssn', {

enumerable: false,

value: '123-456-7890'

});

for (const key in person) {

console.log(key);

}

Code language: JavaScript (javascript)

Output:

firstName

lastName

age

In this example, the ssn property is created with the enumerable flag sets to false, therefore it does not show up in the for...in loop.

ES6 provides a method propertyIsEnumerable() that determines whether or not a property is enumerable. It returns true if the property is enumerable; otherwise false. For example:

const person = {

firstName: 'John',

lastName: 'Doe'

};

person.age = 25;

Object.defineProperty(person, 'ssn', {

enumerable: false,

value: '123-456-7890'

});

console.log(person.propertyIsEnumerable('firstName')); *// => true*

console.log(person.propertyIsEnumerable('lastName')); *// => true*

console.log(person.propertyIsEnumerable('age')); *// => true*

console.log(person.propertyIsEnumerable('ssn')); *// => false*

Code language: JavaScript (javascript)

## **Summary**

* A property is enumerable if it has the enumerable attribute sets to true. The obj.propertyIsEnumerable() determines whether or not a property is enumerable.
* A property created via a simple assignment or a property initializer is enumerable.

# Understanding Own Properties of an Object in JavaScript

**Summary**: in this tutorial, you will learn about the **own** properties of an object in JavaScript.

In JavaScript, an [object](https://www.javascripttutorial.net/javascript-objects/) is a collection of [properties](https://www.javascripttutorial.net/javascript-object-properties/), where each property a key-value pair.

This example creates a new object called person using an object initializer:

const person = {

firstName: 'John',

lastName: 'Doe'

};

Code language: JavaScript (javascript)

The person object has two properties: firstName and lastName.

JavaScript uses [prototypal inheritance](https://www.javascripttutorial.net/javascript-prototypal-inheritance/). Therefore, a property of an object can be either **own** or **inherited**.

A property that is defined directly on an object is **own** while a property that the object receives from its prototype is **inherited**.

The following creates an object called employee that inherits from the person object:

const employee = Object.create(person, {

job: {

value: 'JS Developer',

enumerable: true

}

});

Code language: JavaScript (javascript)

The employee object has its own property job, and inherits firstName and lastName properties from its prototype person.

The hasOwnProperty() method returns true if a property is own. For example:

console.log(employee.hasOwnProperty('job')); *// => true*

console.log(employee.hasOwnProperty('firstName')); *// => false*

console.log(employee.hasOwnProperty('lastName')); *// => false*

console.log(employee.hasOwnProperty('ssn')); *// => false*

Code language: JavaScript (javascript)

## **Summary**

* A property that is directly defined on an object is an own property.
* The obj.hasOwnProperty() method determines whether or not a property is own.

JavaScript Object.values()

**Summary**: in this tutorial, you will learn how to use the JavaScript Object.values() method to access the own enumerable properties of an object.

Prior to ES2017, you use a [for...in](https://www.javascripttutorial.net/javascript-for-in/) loop and Object.hasOwnProperty()  method to access values of [own](https://www.javascripttutorial.net/javascript-own-properties/) [enumerable properties](https://www.javascripttutorial.net/javascript-enumerable-properties/) of an [object](https://www.javascripttutorial.net/javascript-objects/). For example:

const person = {

firstName: 'John',

lastName: 'Doe',

age: 25

};

for (const key in person) {

if (person.hasOwnProperty(key)) {

const value = person[key];

console.log(value);

}

}

Code language: JavaScript (javascript)

Output:

John

Doe

25

ES2017 introduces a new method called Object.values() that allows you to return an [array](https://www.javascripttutorial.net/javascript-array/) of own enumerable property’s values of an object.

The following shows the syntax of the Object.values():

Object.values(obj)

Code language: JavaScript (javascript)

The Object.values() accepts an object and returns its own enumerable property’s values as an array. See the following example:

const person = {

firstName: 'John',

lastName: 'Doe',

age: 25

};

const profile = Object.values(person);

console.log(profile);

Code language: JavaScript (javascript)

Output:

[ 'John', 'Doe', 25 ]

Code language: JSON / JSON with Comments (json)

## **Object.values() vs. for...in**

The Object.values() returns own enumerable properties while the for...in loop iterates properties in the prototype chain.

Technically, if you use the for...in loop with the Object.hasOwnProperty() method, you will get the same set of values as the Object.values().

# JavaScript Object.entries()

**Summary**: in this tutorial, you will learn how to use the JavaScript Object.entries() method.

## **Introduction to JavaScript Object.entries() method**

ES2017 introduces the Object.entries() method that accepts an object and returns own enumerable string-keyed property [key, value] pairs of the object.

Here is the syntax of the Object.entries() method:

Object.entries()

Code language: JavaScript (javascript)

See the following example:

const ssn = Symbol('ssn');

const person = {

firstName: 'John',

lastName: 'Doe',

age: 25,

[ssn]: '123-345-789'

};

const kv = Object.entries(person);

console.log(kv);

Code language: JavaScript (javascript)

Output:

[

['firstName', 'John'],

['lastName', 'Doe'],

['age', 25]

]

Code language: JSON / JSON with Comments (json)

In this example:

* The firstName, lastName, and age are own enumerable string-keyed property of the person object, therefore, they are included in the result.
* The ssn is not a string-key property of the person object, so it is not included in the result.

## **Object.entries() vs. for...in**

The main difference between the Object.entries() and the for...in loop is that the for...in loop also enumerates object [properties](https://www.javascripttutorial.net/javascript-object-properties/) in the [prototype chain](https://www.javascripttutorial.net/javascript-prototype/).

# 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:

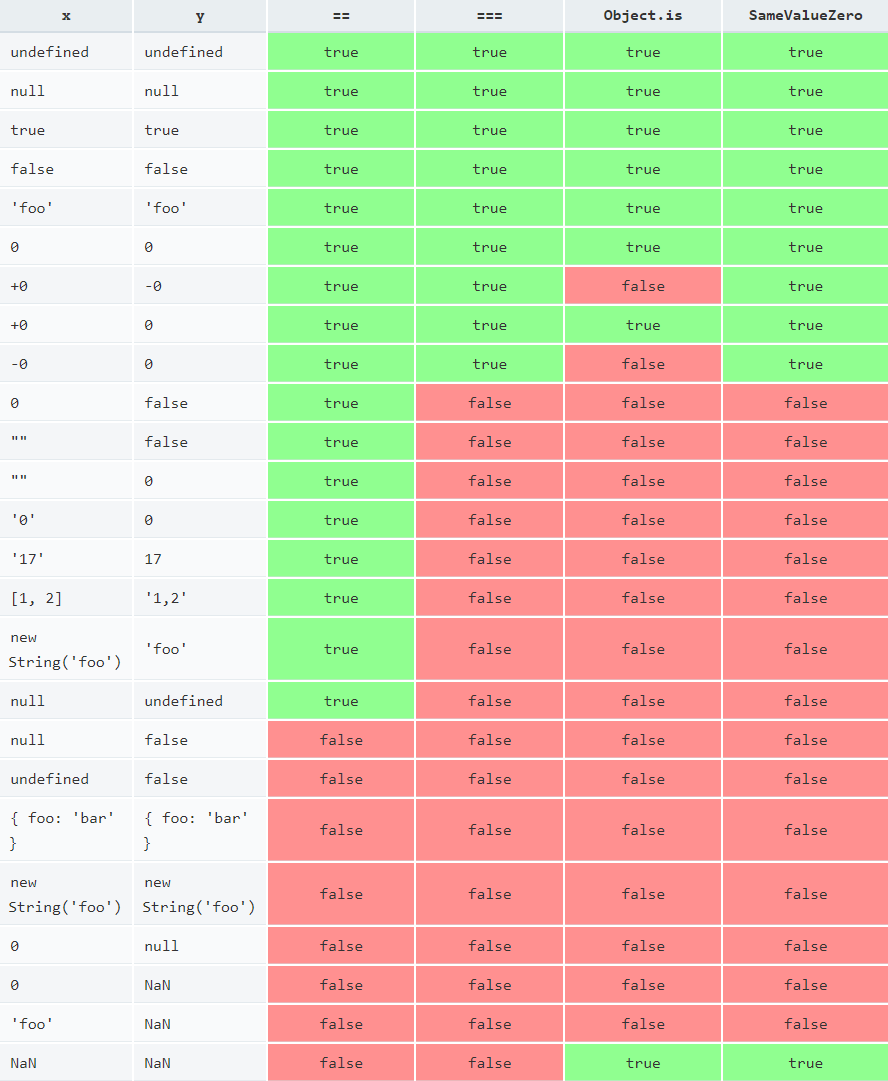
let quantity = NaN;

console.log(Object.is(quantity, quantity));

Output:

true

See the following sameness comparison table for reference:



# JavaScript Factory Functions

**Summary**: in this tutorial, you will learn about the JavaScript factory functions which are functions that return objects.

## **Introduction to the factory functions in JavaScript**

A factory function is a [function](https://www.javascripttutorial.net/javascript-function/) that returns a new [object](https://www.javascripttutorial.net/javascript-objects/). The following creates a person object named person1:

let person1 = {

firstName: 'John',

lastName: 'Doe',

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

console.log(p1.getFullName());

Code language: JavaScript (javascript)

Output:

John Doe

The person1 object has two properties: firstName and lastName, and one method getFullName() that returns the full name.

Suppose that you need to create another similar object called person2, you can duplicate the code as follows:

let person2 = {

firstName: 'Jane',

lastName: 'Doe',

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

console.log(jane.getFullName());

Code language: JavaScript (javascript)

Output:

Jane Doe

In this example, the person1 and person2 objects have the same properties and methods.

The problem is that the more objects you want to create, the more duplicate code you have.

To avoid copying the same code all over again, you can define a function that creates the person object:

function createPerson(firstName, lastName) {

return {

firstName: firstName,

lastName: lastName,

getFullName() {

return firstName + ' ' + lastName;

},

};

}

Code language: JavaScript (javascript)

When a function creates and returns a new object, it is called a factory function. The createPerson() is a factory function because it returns a new person object.

The following show how to use the createPerson() factory function to create two objects person1 and person2:

function createPerson(firstName, lastName) {

return {

firstName: firstName,

lastName: lastName,

getFullName() {

return firstName + ' ' + lastName;

},

};

}

let person1 = createPerson('John', 'Doe');

let person2 = createPerson('Jane', 'Doe');

console.log(person1.getFullName());

console.log(person2.getFullName());

Code language: JavaScript (javascript)

By using the factory function, you create any number of the person objects without duplicating code.

When you create an object, the JavaScript engine allocates memory to it. If you create many person objects, the JavaScript engine needs lots of memory spaces to store these objects.

However, each person object has a copy of the same getFullName() method. It’s not efficient memory management.

To avoid duplicating the same getFullName() function in every object, you can remove the getFullName() method from the person object:

function createPerson(firstName, lastName) {

return {

firstName: firstName,

lastName: lastName

}

}

Code language: JavaScript (javascript)

And move this method to another object:

var personActions = {

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

Code language: JavaScript (javascript)

And before calling the getFullName() method on the person object, you can assign the method of the personActions object to the person object as follows:

let person1 = createPerson('John', 'Doe');

let person2 = createPerson('Jane', 'Doe');

person1.getFullName = personActions.getFullName;

person2.getFullName = personActions.getFullName;

console.log(person1.getFullName());

console.log(person2.getFullName());

Code language: JavaScript (javascript)

This approach is not scalable if the object has many methods because you have to manually assign them individually. This is why the Object.create() method comes into play.

## **The Object.create() method**

The Object.create() method creates a new object using an existing object as the [prototype](https://www.javascripttutorial.net/javascript-prototype/) of the new object:

Object.create(proto, [propertiesObject])

Code language: CSS (css)

So you can use the Object.create() as follows:

var personActions = {

getFullName() {

return this.firstName + ' ' + this.lastName;

},

};

function createPerson(firstName, lastName) {

let person = Object.create(personActions);

person.firstName = firstName;

person.lastName = lastName;

return person;

}

Code language: JavaScript (javascript)

Now, you can create person objects and call the methods of the personActions object:

let person1 = createPerson('John', 'Doe');

let person2 = createPerson('Jane', 'Doe');

console.log(person1.getFullName());

console.log(person2.getFullName());

Code language: JavaScript (javascript)

The code works perfectly fine. However, in practice, you will rarely use the factory functions. Instead, you use [classes](https://www.javascripttutorial.net/es6/javascript-class/) or [constructor/prototype](https://www.javascripttutorial.net/javascript-constructor-prototype/) patterns.

## **Summary**

* A factory function is a function that returns a new object.
* Use Object.create() to create an object using an existing object as a prototype.

# 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.

JavaScript Optional Chaining Operator

**Summary**: in this tutorial, you’ll learn about the optional chaining operator (?.) that simplifies the way to access values through connected objects.

## **Introduction to the JavaScript optional chaining operator**

The optional chaining operator (?.) allows you to access the value of a [property](https://www.javascripttutorial.net/javascript-object-properties/) located deep within a chain of [objects](https://www.javascripttutorial.net/javascript-objects/) without explicitly checking if each reference in the chain is null or undefined.

If one of the references in the chain is null or undefined, the optional chaining operator (?.) will short circuit and return undefined.

Suppose that you have a function that returns a user object:

function getUser(id) {

if(id <= 0) {

return null;

}

*// get the user from database*

*// and return null if id does not exist*

*// ...*

*// if user was found, return the user*

return {

id: id,

username: 'admin',

profile: {

avatar: '/avatar.png',

language: 'English'

}

}

}

Code language: JavaScript (javascript)

The following uses the getUser() function to access the user profile:

let user = getUser(1);

let profile = user.profile;

Code language: JavaScript (javascript)

However, if you pass the id that is less than or equal to zero or the id doesn’t exist in the database, the getUser() function will return null.

Therefore, before accessing the avatar property, you need to check if the user is not null using the [logical operator](https://www.javascripttutorial.net/javascript-logical-operators/) AND:

let user = getUser(2);

let profile = user && user.profile;

Code language: JavaScript (javascript)

In this example, we confirm that the user is not null or undefined before accessing the value of user.profile property. It prevents the error that would occur if you simply access the user.profile directly without checking the user first.

ES2020 introduced the optional chaining operator denoted by the question mark followed by a dot:

?.

Code language: JavaScript (javascript)

To access a property of an object using the optional chaining operator, you use one of the following:

objectName ?. propertyName

objectName ?. [expression]

Code language: JavaScript (javascript)

The optional chaining operator implicitly checks if the user is not null or undefined before attempting to access the user.profile:

let user = getUser(2);

let profile = user ?. profile;

Code language: JavaScript (javascript)

In this example, if the user is null or undefined, the optional chaining operator (?.) immediately returns undefined.

Technically, it is equivalent to the following:

let user = getUser(2);

let profile = (user !== null || user !== undefined)

? user.profile

: undefined;

Code language: JavaScript (javascript)

### **Stacking the optional chaining operator**

In case the user object returned by the getUser() does not have the profile property. Trying to access the avatar without checking the user.profile first will result in an error.

To avoid the error, you can use the optional chaining operator multiple times like this:

let user = getUser(-1);

let avatar = user ?. profile ?. avatar;

Code language: JavaScript (javascript)

In this case, the avatar is undefined.

### **Combining with the nullish coalescing operator**

If you want to assign a default profile to the user, you can combine the optional chaining operator (?.) with the nullish coalescing operator (??) as follows:

let defaultProfile = { default: '/default.png', language: 'English'};

let user = getUser(2);

let profile = user ?. profile ?? defaultProfile;

Code language: JavaScript (javascript)

In this example, if the user.profile is null or undefined, the profile will take the defaultProfile due to the nullish coalescing operator:

## **Using optional chaining operator with function calls**

Suppose that you have a file API as follows:

let file = {

read() {

return 'file content';

},

write(content) {

console.log(`Writing ${content} to file...`);

return true;

}

};

Code language: JavaScript (javascript)

This example calls the read() method of the file object:

let data = file.read();

console.log(data);

Code language: JavaScript (javascript)

If you call a method that doesn’t exist in the file object, you’ll get a TypeError:

let compressedData = file.compress();

Code language: JavaScript (javascript)

Error:

Uncaught TypeError: file.compress is not a function

Code language: JavaScript (javascript)

However, if you use the optional chaining operator with the method call, the expression will return undefined instead of throwing an error:

let compressedData = file.compress?.();

Code language: JavaScript (javascript)

The compressedData is now undefined.

This is useful when you use an API in which a method might be not available for some reason e.g., a specific browser or device.

The following illustrates the syntax for using the optional chaining operator with a function or method call:

functionName ?. (args)

Code language: JavaScript (javascript)

The optional chaining operator (?.) is also helpful if you have a function with an optional [callback](https://www.javascripttutorial.net/javascript-callback/):

function getUser(id, callback) {

*// get user*

*// ...*

let user = {

id: id,

username: 'admin'

};

*// test if the callback exists*

if ( callback ) {

callback(user);

}

return user;

}

Code language: JavaScript (javascript)

By using the optional chaining operator, you can skip the test if the callback exists:

function getUser(id, callback) {

*// get user*

*// ...*

let user = {

id: id,

username: 'admin'

};

*// test if the callback exists*

callback ?. (user);

return user;

}

Code language: JavaScript (javascript)

## **Summary**

* The optional chaining operator (?.) returns undefined instead of throwing an error if you attempt to access a property of an null or undefined object: obj ?. property.
* Combine the optional chaining operator (?.) with the nullish coalescing operator (??) to assign a default value.
* Use functionName ?. (args) to avoid explicitly checking if the functionName is not undefined or null before invoking it.

# 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 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 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.

# 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 Private Fields

**Summary**: in this tutorial, you’ll learn about JavaScript private fields and how to use them effectively.

## **Introduction to the JavaScript private fields**

ES2022 allows you to define private fields for a [class](https://www.javascripttutorial.net/es6/javascript-class/). To define a private field, you prefix the field name with the # sign.

For example, the following defines the Circle class with a private field radius:

class Circle {

*#radius;*

constructor(value) {

this.*#radius = value;*

}

get area() {

return Math.PI \* Math.pow(this.*#radius, 2);*

}

}

Code language: PHP (php)

In this example:

* First, define the private field #radius in the class body.
* Second, initialize the #radius field in the constructor with an argument.
* Third, calculate the area of the circle by accessing the #radius private field in the getter method.

The following creates a new instance of the Circle class and calculates its area:

let circle = new Circle(10);

console.log(circle.area); *// 314.1592653589793*

Code language: JavaScript (javascript)

Because the #radius is a private field, you can only access it inside the Circle class. In other words, the #radius field is invisible outside of the Circle class.

## **Using getter and setter to access private fields**

The following redefines the Circle class by adding the radius getter and setter to provide access to the #radius private field:

class Circle {

*#radius = 0;*

constructor(radius) {

this.radius = radius;

}

get area() {

return Math.PI \* Math.pow(this.radius, 2);

}

set radius(value) {

if (typeof value === 'number' && value > 0) {

this.*#radius = value;*

} else {

throw 'The radius must be a positive number';

}

}

get radius() {

return this.*#radius;*

}

}

Code language: PHP (php)

How it works.

* The radius setter validates the argument before assigning it to the #radius private field. If the argument is not a positive number, the radius setter throws an error.
* The radius getter returns the value of the #radius private field.
* The constructor calls the radius setter to assign the argument to the #radius private field.

## **Private fields and subclasses**

Private fields are only accessible inside the class where they’re defined. Also, they’re not accessible from the subclasses. For example, the following defines the Cylinder class that [extends](https://www.javascripttutorial.net/es6/javascript-inheritance/) the Circle class:

class Cylinder extends Circle {

*#height;*

constructor(radius, height) {

super(radius);

this.*#height = height;*

*// cannot access the #radius of the Circle class here*

}

}

Code language: PHP (php)

If you attempt to access the #radius private field in the Cylinder class, you’ll get a SyntaxError.

## **The in operator: check private fields exist**

To check if an object has a private field inside a class, you use the in operator:

fieldName in objectName

For example, the following adds the hasRadius() static method to the Circle class that uses the in operator to check if the circle object has the #radius private field:

class Circle {

*#radius = 0;*

constructor(radius) {

this.radius = radius;

}

get area() {

return Math.PI \* Math.pow(this.radius, 2);

}

set radius(value) {

if (typeof value === 'number' && value > 0) {

this.*#radius = value;*

} else {

throw 'The radius must be a positive number';

}

}

get radius() {

return this.*#radius;*

}

static hasRadius(circle) {

return *#radius in circle;*

}

}

let circle = new Circle(10);

console.log(Circle.hasRadius(circle));

Code language: PHP (php)

Output:

true

Code language: JavaScript (javascript)

## **Static private fields**

The following example shows how to use use a static private field:

class Circle {

*#radius = 0;*

static *#count = 0;*

constructor(radius) {

this.radius = radius;

Circle.*#count++;*

}

get area() {

return Math.PI \* Math.pow(this.radius, 2);

}

set radius(value) {

if (typeof value === 'number' && value > 0) {

this.*#radius = value;*

} else {

throw 'The radius must be a positive number';

}

}

get radius() {

return this.*#radius;*

}

static hasRadius(circle) {

return *#radius in circle;*

}

static getCount() {

return Circle.*#count;*

}

}

let circles = [new Circle(10), new Circle(20), new Circle(30)];

console.log(Circle.getCount());

Code language: PHP (php)

How it works.

First, add a private static field #count to the Circle class and initialize its value to zero:

static *#count = 0;*

Code language: PHP (php)

Second, increase the #count by one in the constructor:

Circle.#count++;

Code language: CSS (css)

Third, define a static method that returns the value of the #count private static field:

static getCount() {

return Circle.*#count;*

}

Code language: PHP (php)

Finally, create three instances of the Circle class and output the count value to the console:

let circles = [new Circle(10), new Circle(20), new Circle(30)];

console.log(Circle.getCount());

Code language: JavaScript (javascript)

## **Summary**

* Prefix the field name with # sign to make it private.
* Private fields are accessible only inside the class, not from outside of the class or subclasses.
* Use the in operator to check if an object has a private field.

# JavaScript Private Methods

**Summary**: in this tutorial, you’ll learn about JavaScript private methods including private instance methods, private static methods, and private getter/setter.

## **Introduction to JavaScript private methods**

By default, members of a [class](https://www.javascripttutorial.net/es6/javascript-class/) are public. ES2020 introduced the private members that include [private fields](https://www.javascripttutorial.net/javascript-private-fields/) and methods.

To make a public method private, you prefix its name with a hash #. JavaScript allows you to define private methods for instance methods, [static methods](https://www.javascripttutorial.net/es6/javascript-static-method/), and [getter/setters](https://www.javascripttutorial.net/es6/javascript-getters-and-setters/).

The following shows the syntax of defining a private instance method:

class MyClass {

#privateMethod() {

*//...*

}

}

Code language: JavaScript (javascript)

In this syntax, the #privateMethod is a private instance method. It can only be called inside the MyClass. In other words, it cannot be called from outside the class or in the subclasses of the MyClass.

To call the #privateMethod inside the MyClass, you use the this keyword as follows:

this.#privateMethod();

Code language: JavaScript (javascript)

The following illustrates the syntax of defining a private static method:

class MyClass {

static #privateStaticMethod() {

*//...*

}

}

Code language: JavaScript (javascript)

To call the #privateStaticMethod() inside the MyClass, you use the class name instead of the this keyword:

MyClass.#privateStaticMethod();

Code language: JavaScript (javascript)

The following shows the syntax of the private getters/setters:

class MyClass {

#field;

get #myField() {

return #field;

}

set #myField(value){

#field = value;

}

}

Code language: JavaScript (javascript)

In this example, the #myField is the private getter and setter that provide access to the private field #field.

In practice, you use private methods to minimize the number of methods that the object exposes.

As a rule of thumb, you should make all class methods private by default first. And then you make a method public whenever the object needs to use that method to interact with other objects.

## **JavaScript private method examples**

Let’s take some examples of using private methods

### **1) Private instance method example**

The following illustrates how to define the Person class with private instance methods:

class Person {

#firstName;

#lastName;

constructor(firstName, lastName) {

this.#firstName = firstName;

this.#lastName = lastName;

}

getFullName(format = true) {

return format ? this.#firstLast() : this.#lastFirst();

}

#firstLast() {

return `${this.#firstName} ${this.#lastName}`;

}

#lastFirst() {

return `${this.#lastName}, ${this.#firstName}`;

}

}

let person = new Person('John', 'Doe');

console.log(person.getFullName());

Code language: JavaScript (javascript)

Output:

John Doe

Code language: JavaScript (javascript)

In this example:

First, define two private fields #firstName and #lastName in the Person class body.

Second, define the private methods #firstLast() and #lastFirst(). These methods return the full name in different formats.

Third, define the public instance method getFullName() that returns a person’s full name. The getFullName() method calls the private method #firstLast() and #lastFirst() to return the full name.

Finally, create a new person object and output the full name to the console.

### **2) Private static method example**

The following adds the #validate() private static method to the Person class:

class Person {

#firstName;

#lastName;

constructor(firstName, lastName) {

this.#firstName = Person.#validate(firstName);

this.#lastName = Person.#validate(lastName);

}

getFullName(format = true) {

return format ? this.#firstLast() : this.#lastFirst();

}

static #validate(name) {

if (typeof name === 'string') {

let str = name.trim();

if (str.length === 3) {

return str;

}

}

throw 'The name must be a string with at least 3 characters';

}

#firstLast() {

return `${this.#firstName} ${this.#lastName}`;

}

#lastFirst() {

return `${this.#lastName}, ${this.#firstName}`;

}

}

let person = new Person('John', 'Doe');

console.log(person.getFullName());

Code language: JavaScript (javascript)

How it works.

First, define the static method #validate() that returns a value if it is a string with at least three characters. The method raises an exception otherwise.

Second, call the #validate() private static method in the constructor to validate the firstName and lastName arguments before assigning them to the corresponding private attributes.

### **Summary**

* Prefix a method name with the # to make it private.
* Private methods can be called inside the class, not from outside of the class or in the subclasses.

# JavaScript instanceof

**Summary**: in this tutorial, you’ll learn how to use the JavaScript instanceof operator to determine if a constructor’s prototype appears in the prototype chain of an object.

## **Introduction to the JavaScript instanceof operator**

The instanceof operator returns true if a [prototype](https://www.javascripttutorial.net/javascript-prototype/) of a constructor (constructor.prototype) appears in the prototype chain of an object.

The following shows the syntax of the instanceof operator:

object instanceof contructor

Code language: JavaScript (javascript)

In this syntax:

* object is the object to test.
* constructor is a function to test against.

## **JavaScript instanceof operator example**

The following example defines the Person type and uses the instanceof operator to check if an object is an instance of that type:

function Person(name) {

this.name = name;

}

let p1 = new Person('John');

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

How it works.

First, define a Person type using the [constructor function](https://www.javascripttutorial.net/javascript-constructor-function/) pattern:

function Person(name) {

this.name = name;

}

Code language: JavaScript (javascript)

Second, create a new object of the Person type:

let p1 = new Person('John Doe');

Code language: JavaScript (javascript)

Third, check if the person is an instance of the Person type:

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

It returns true because the Person.prototype appears on the prototype chain of the p1 object. The prototype chain of the p1 is the link between p1, Person.prototype, and Object.prototype:

The following also returns true because the Object.prototype appears on the prototype chain of the p1 object:

console.log(p1 instanceof Object); *// true*

Code language: JavaScript (javascript)

## **ES6 class and instanceof operator**

The following example defines the Person class and uses the instanceof operator to check if an object is an instance of the [class](https://www.javascripttutorial.net/es6/javascript-class/):

class Person {

constructor(name) {

this.name = name;

}

}

let p1 = new Person('John');

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

How it works.

First, define the Person class:

class Person {

constructor(name) {

this.name = name;

}

}

Code language: JavaScript (javascript)

Second, create a new instance of the Person class:

let p1 = new Person('John');

Code language: JavaScript (javascript)

Third, check if p1 is an instance of the Person class:

console.log(p1 instanceof Person); *// true*

Code language: JavaScript (javascript)

## **The instanceof operator and inheritance**

The following example defines the Employee class that extends the Person class:

class Person {

constructor(name) {

this.name = name;

}

}

class Employee extends Person {

constructor(name, title) {

super(name);

this.title = title;

}

}

let e1 = new Employee();

console.log(e1 instanceof Employee); *// true*

console.log(e1 instanceof Person); *// true*

console.log(e1 instanceof Object); *// true*

Code language: JavaScript (javascript)

Since e1 is an instance of the Employee class, it’s also an instance of the Person and Object classes (base classes).

## **Symbol.hasInstance**

In ES6, the instanceof operator uses the Symbol.hasInstance function to check the relationship. The Symbol.hasInstance() accepts an object and returns true if a type has that object as an instance. For example:

class Person {

constructor(name) {

this.name = name;

}

}

let p1 = new Person('John');

console.log(Person[Symbol.hasInstance](p1)); *// true*

Code language: JavaScript (javascript)

Since the Symbol.hasInstance is defined on the [Function](https://www.javascripttutorial.net/javascript-function-type/) prototype, it’s automatically available by default in all functions and classes

And you can redefine the Symbol.hasInstance on a subclass as a static method. For example:

class Person {

constructor(name) {

this.name = name;

}

}

class Android extends Person {

static [Symbol.hasInstance]() {

return false;

}

}

let a1 = new Android('Sonny');

console.log(a1 instanceof Android); *// false*

console.log(a1 instanceof Person); *// false*

Code language: JavaScript (javascript)

## **Summary**

* Use the instanceof operator to check if the constructor.protoype in object’s prototype chain.

# JavaScript Function Type

**Summary**: in this tutorial, you’ll learn about the JavaScript Function type, which is the type of all functions in JavaScript.

## **Introduction to the JavaScript Function type**

In JavaScript, all [functions](https://www.javascripttutorial.net/javascript-function/) are [objects](https://www.javascripttutorial.net/javascript-objects/). They are the instances of the Function type. Because functions are objects, they have properties and methods like other objects.

### **Functions properties**

Each function has two important properties: length and prototype.

* The length property determines the number of named arguments specified in the function declaration.
* The prototype property references the actual function object.

See the following example:

function add(x, y) {

return x + y;

}

console.log(add.length); *// 2*

console.log(add.prototype); *// Object{}*

Code language: JavaScript (javascript)

The add() function accepts two arguments x and y. Therefore, the length property returns two.

### **new.target**

Typically, you call a function normally like this:

let result = add(10,20);

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

Code language: JavaScript (javascript)

Also, you can call a function with new keyword as a constructor:

let obj = new add(10,20);

Code language: JavaScript (javascript)

ES6 introduced the [new.target](https://www.javascripttutorial.net/es6/javascript-new-target/) pseudo-property that allows you to detect whether a function or constructor was called using the new operator.

If a function is called normally, the new.target is undefined. However, if the function is called using the new keyword as a constructor, the new.target return a reference to the constructor.

For example:

function add(x, y) {

console.log(new.target);

return x + y;

}

let result = add(10, 20);

let obj = new add(10, 20);

Code language: JavaScript (javascript)

Output:

undefined

[Function: add]

Code language: JavaScript (javascript)

By using the new.target, you can control how a function will be called.

For example, to prevent the add() function from being called with the new keyword as a constructor, you can throw an error by checking the new.target like this:

function add(x, y) {

if (new.target) {

throw 'The add function cannot be called as a constructor';

}

return x + y;

}

let obj = new add(10, 20);

console.log(obj);

Code language: JavaScript (javascript)

## **Function methods: apply, call, and bind**

A function object has three important methods: [apply()](https://www.javascripttutorial.net/javascript-apply-method/), [call()](https://www.javascripttutorial.net/javascript-call/) and [bind()](https://www.javascripttutorial.net/javascript-bind/).

### **The apply() and call() methods**

The apply() and call() methods call a function with a given this value and arguments.

The difference between the apply() and call() is that you need to pass the arguments to the apply() method as an array-like object, whereas you pass the arguments to the call() function individually. For example:

let cat = { type: 'Cat', sound: 'Meow' };

let dog = { type: 'Dog', sound: 'Woof' };

const say = function (message) {

console.log(message);

console.log(this.type + ' says ' + this.sound);

};

say.apply(cat, ['What does a cat say?']);

say.apply(dog, ['What does a dog say?']);

Code language: JavaScript (javascript)

Output:

What does a cat sound?

Cat says Meow

What does a dog sound?

Dog says Woof

In this example:

First, declare two objects cat and dog with two properties:

let cat = { type: 'Cat', sound: 'Meow' };

let dog = { type: 'Dog', sound: 'Woof' };

Code language: JavaScript (javascript)

Second, define the say() function that accepts one argument:

const say = function (message) {

console.log(message);

console.log(this.type + ' says ' + this.sound);

};

Code language: JavaScript (javascript)

Third, call the say() function via the apply() method:

say.apply(cat, ['What does a cat say?']);

Code language: CSS (css)

In this example, the first argument of the  apply() method is the cat object. Therefore, the this object in the say() function references the cat object.

Fourth, call say() function and pass the dog object:

say.apply(dog, ['What does a dog say?']);

Code language: CSS (css)

In this example, the this in the say() function reference the dog object.

The call() method like the apply() method except for the way you pass the arguments to the function:

say.call(cat, 'What does a cat say?');

say.call(dog, 'What does a dog say?');

Code language: JavaScript (javascript)

### **The bind() method**

The bind() method creates a new function instance whose this value is bound to the object that you provide. For example:

First, define an object named car:

let car = {

speed: 5,

start: function() {

console.log('Start with ' + this.speed + ' km/h');

}

};

Code language: JavaScript (javascript)

Then, define another object named aircraft:

let aircraft = {

speed: 10,

fly: function() {

console.log('Flying');

}

};

Code language: JavaScript (javascript)

The aircraft has no start() method. To start an aircraft, you can use the bind() method of the start() method of the car object:

let taxiing = car.start.bind(aircraft);

Code language: JavaScript (javascript)

In this statement, we change the this value inside the start() method of the car object to the  aircraft object.  The bind() method returns a new function that is assigned to the taxiing variable.

Now, you can call the start() method via the taxiing variable:

taxiing();

It will show the following message:

Start with 10 km/h

Code language: JavaScript (javascript)

The following uses the call() method to call the start() method on the aircraft object:

car.start.call(aircraft);

Code language: CSS (css)

As you can see, the bind() method creates a new function that you can execute later while the call() method executes the function immediately. This is the main difference between the bind() and call() methods.

Technically, the aircraft object borrows the start() method of the car object via the bind(), call() or apply() method.

For this reason, the bind(), call(), and apply() methods are also known as borrowing functions.

## **Summary**

* All functions are instances of the Function type, which are the objects that have properties and methods.
* A function has two important properties: name and prototype.
* A function also has three important methods: call(), apply(), and bind().

# JavaScript call() Method

**Summary**: in this tutorial, you will learn about the JavaScript call() method and how to use it more effectively.

## **Introduction to the JavaScript call() method**

In JavaScript, a [function](https://www.javascripttutorial.net/javascript-function/) is an instance of the [Function](https://www.javascripttutorial.net/javascript-function-type/) type. For example:

function add(x, y) {

return x + y;

}

console.log(add instanceof Function); *// true*

Code language: JavaScript (javascript)

The Function.prototype type has the call() method with the following syntax:

functionName.call(thisArg, arg1, arg2, ...);

Code language: JavaScript (javascript)

In this syntax, the call() method calls a function functionName with the arguments (arg1, arg2, …) into it and the this set to the thisArg object inside the function.

* The thisArg is the object that the this object references inside the function functionName.
* The arg1, arg2, .. are the function arguments passed into the functionName.

The call() method returns the result of calling the functionName().

The following example defines the add() function and calls it normally:

function add(x, y) {

return x + y;

}

let result = add(10, 20);

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

Code language: JavaScript (javascript)

The following calls the add() function but use the call() method instead:

function add(x, y) {

return x + y;

}

let result = add.call(this, 10, 20);

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

Code language: JavaScript (javascript)

By default, the [this](https://www.javascripttutorial.net/javascript-this/) inside the function is set to the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) i.e., window in the web browsers and global in Node.js.

Note that in the strict mode, the this inside the function is set to undefined instead of the global object.

Consider the following example:

var greeting = 'Hi';

var messenger = {

greeting: 'Hello'

}

function say(name) {

console.log(this.greeting + ' ' + name);

}

Code language: JavaScript (javascript)

Inside the say() function, we reference the greeting via the this value. If you just invoke the say() function via the call() method as follows:

say.call(this,'John');

Code language: JavaScript (javascript)

It’ll show the following output to the console:

"Hi John"

Code language: JavaScript (javascript)

However, when you invoke the call() method of say function object and pass the messenger object as the this value:

say.call(messenger,'John');

Code language: JavaScript (javascript)

The output will be:

"Hello John"

Code language: JavaScript (javascript)

In this case, the this value inside the say() function references the messenger object, not the global object.

## **Using the JavaScript call() method to chain constructors for an object**

You can use the call() method for chaining constructors of an object. Consider the following example:

function Box(height, width) {

this.height = height;

this.width = width;

}

function Widget(height, width, color) {

Box.call(this, height, width);

this.color = color;

}

let widget = new Widget('red', 100, 200);

console.log(widget);

Code language: JavaScript (javascript)

Output:

Widget { height: 'red', width: 100, color: 200 }

Code language: JavaScript (javascript)

In this example:

* First, initialize the Box object with two properties: height and width.
* Second, invoke the call() method of the Box object inside the Widget object, set the this value to the Widget object.

## **Using the JavaScript call() method for function borrowing**

The following example illustrates how to use the call() method for borrowing functions:

const car = {

name: 'car',

start() {

console.log('Start the ' + this.name);

},

speedUp() {

console.log('Speed up the ' + this.name);

},

stop() {

console.log('Stop the ' + this.name);

},

};

const aircraft = {

name: 'aircraft',

fly() {

console.log('Fly');

},

};

car.start.call(aircraft);

car.speedUp.call(aircraft);

aircraft.fly();

Code language: JavaScript (javascript)

Output:

Start the aircraft

Speed up the aircraft

Fly

Code language: JavaScript (javascript)

How it works.

First, define a car object with one property name and three methods start, speedUp, and stop:

const car = {

name: 'car',

start() {

console.log('Start the ' + this.name);

},

speedUp() {

console.log('Speed up the ' + this.name);

},

stop() {

console.log('Stop the ' + this.name);

},

};

Code language: JavaScript (javascript)

Second, define the aircraft object with one property name and a method:

const aircraft = {

name: 'aircraft',

fly() {

console.log('Fly');

},

};

Code language: JavaScript (javascript)

Third, call the start() and speedUp() method of the car object and the fly() method of the aircraft object. However, pass the aircraft as the first argument into the start() and speedUp() methods:

car.start.call(aircraft);

car.speedUp.call(aircraft);

aircraft.fly();

Code language: JavaScript (javascript)

Inside the start() and speedUp() methods, the this references the aircraft object, not the car object. Therefore, the this.name returns the 'aircraf' string. Hence, the methods output the following message:

Start the aircraft

Speed up the aircraft

Code language: plaintext (plaintext)

Technically, the aircraft object borrows the start() and speedUp() method of the car object. And function borrowing refers to an object that uses a method of another object.

The following example illustrates how the arguments object borrows the filter() method of the Array.prototype via the call() function:

function isOdd(number) {

return number % 2;

}

function getOddNumbers() {

return Array.prototype.filter.call(arguments, isOdd);

}

let results = getOddNumbers(10, 1, 3, 4, 8, 9);

console.log(results);

Code language: JavaScript (javascript)

Output:

[ 1, 3, 9 ]

Code language: JavaScript (javascript)

How it works.

First, define the isOdd() function that returns true if the number is an odd number:

function isOdd(number) {

return number % 2;

}

Code language: JavaScript (javascript)

Second, define the getOddNumbers() function that accepts any number of arguments and returns an array that contains only odd numbers:

function getOddNumbers() {

return Array.prototype.filter.call(arguments, isOdd);

}

Code language: JavaScript (javascript)

In this example, the arguments object borrows the filter() method of the Array.prototype object.

Third, call the getOddNumbers() function:

let results = getOddNumbers(10, 1, 3, 4, 8, 9);

console.log(results);

Code language: JavaScript (javascript)

In this tutorial, you have learned about the JavaScript call() method and how to use it more effectively.

# JavaScript apply() method

**Summary**: in this tutorial, you’ll learn about the JavaScript apply() method of the Function type and how to use it effectively.

## **Introduction to the JavaScript apply() method**

The Function.prototype.apply() method allows you to call a [function](https://www.javascripttutorial.net/javascript-function/) with a given [this](https://www.javascripttutorial.net/javascript-this/) value and arguments provided as an [array](https://www.javascripttutorial.net/javascript-array/). Here is the syntax of the apply() method:

fn.apply(thisArg, [args]);

Code language: JavaScript (javascript)

The apply() method accepts two arguments:

* The thisArg is the value of this provided for the call to the function fn.
* The args argument is an array that specifies the arguments of the function fn. Since the ES5, the args argument can be an array-like object or array object.

THe apply() method is similar to the [call()](https://www.javascripttutorial.net/javascript-call/) method except that it takes the arguments of the function as an array instead of the individual arguments.

## **JavaScript apply() method examples**

Let’s take some examples of using the apply() method.

### **1) Simple JavaScript apply() method example**

Suppose that you have a person object:

const person = {

firstName: 'John',

lastName: 'Doe'

}

Code language: JavaScript (javascript)

…and a function named greet() as follows:

function greet(greeting, message) {

return `${greeting} ${this.firstName}. ${message}`;

}

Code language: JavaScript (javascript)

The greet() function accepts two parameters: greeting and message. Inside the greet() function, we reference an object that has the firstName property.

The following example shows how to use the apply() method to call the greet() function with the this set to the person object:

let result = greet.apply(person, ['Hello', 'How are you?']);

console.log(result);

Code language: JavaScript (javascript)

Output:

Hello John. How are you?

Code language: JavaScript (javascript)

In this example, we set the this value inside the function to the person object. The arguments of the greet() function was passed into the apply() method as an array.

The apply() method invoked the greet() function with the this value set to the person object and arguments as an array ['Hello', 'How are you?'].

If you use the call() method, you need to pass the arguments of the greet() function separately as follows:

let result = greet.call(person, Hello', 'How are you?');

Code language: JavaScript (javascript)

### **2) Function borrowing**

The apply() method allows an object to borrow the method of another object without duplicating the code.

Suppose that you have the following computer object:

const computer = {

name: 'MacBook',

isOn: false,

turnOn() {

this.isOn = true;

return `The ${this.name} is On`;

},

turnOff() {

this.isOn = false;

return `The ${this.name} is Off`;

}

};

Code language: JavaScript (javascript)

… and the following server object:

const server = {

name: 'Dell PowerEdge T30',

isOn: false

};

Code language: JavaScript (javascript)

The server object doesn’t have the turnOn() and turnOff() methods.

To execute the turnOn() method of the computer object on the server object, you can use the apply() method as follows:

let result = computer.turnOn.apply(server);

console.log(result);

Code language: JavaScript (javascript)

Output:

The Dell PowerEdge T30 is On

Code language: JavaScript (javascript)

In this example, the server object borrows the turnOn() method of the computer object.

Similarly, you can call the turnOff() method of the computer object on the server object:

let result = computer.turnOff.apply(server);

console.log(result);

Code language: JavaScript (javascript)

Output:

The Dell PowerEdge T30 is Off

Code language: JavaScript (javascript)

### **3) Using the apply() method to append an array to another**

The apply() method allows you to append elements of an array to another:

let arr = [1, 2, 3];

let numbers = [4, 5, 6];

arr.push.apply(arr, numbers);

console.log(arr);

Code language: JavaScript (javascript)

In this example, the apply() method modifies the original array arr. Note that the [Array.prototype.concat()](https://www.javascripttutorial.net/javascript-array-concat/) method also provides the same result except that it returns the new array instead of modifying the original array.

## **Summary**

* The apply() method invokes a function with a given this value and arguments provided as an array.
* The apply() method is similar to the call() method excepts that it accepts the arguments of the function as an array instead of individual arguments.

# JavaScript bind() Method

**Summary**: in this tutorial, you will learn about the JavaScript bind() method and know how to use it effectively.

## **Introduction to JavaScript bind() method**

The bind() method returns a new [function](https://www.javascripttutorial.net/javascript-function/), when invoked, has its [this](https://www.javascripttutorial.net/javascript-this/) sets to a specific value.

The following illustrates the syntax of the bind() method:

fn.bind(thisArg[, arg1[, arg2[, ...]]])

Code language: CSS (css)

In this syntax, the bind() method returns a copy of the function fn with the specific this value (thisArg) and arguments (arg1, arg2, …).

Unlike the [call()](https://www.javascripttutorial.net/javascript-call/) and [apply()](https://www.javascripttutorial.net/javascript-apply-method/) methods, the bind() method doesn’t immediately execute the function. It just returns a new version of the function whose this sets to thisArg argument.

## **Using JavaScript bind() for function binding**

When you pass a method an [object](https://www.javascripttutorial.net/javascript-objects/) is to another function as a [callback](https://www.javascripttutorial.net/javascript-callback/), the this is lost. For example:

let person = {

name: 'John Doe',

getName: function() {

console.log(this.name);

}

};

setTimeout(person.getName, 1000);

Code language: JavaScript (javascript)

Output:

undefined

Code language: JavaScript (javascript)

As you can see clearly from the output, the person.getName() returns undefined instead of 'John Doe'.

This is because [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) received the function person.getName separately from the person object.

The statement:

setTimeout(person.getName, 1000);

Code language: CSS (css)

can be rewritten as:

let f = person.getName;

setTimeout(f, 1000); *// lost person context*

Code language: JavaScript (javascript)

The this inside the setTimeout() function is set to the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) in non-strict mode and undefined in the strict mode.

Therefore, when the callback person.getName is invoked, the name does not exist in the global object, it is set to undefined.

To fix the issue, you can wrap the call to the person.getName method in an [anonymous function](https://www.javascripttutorial.net/javascript-anonymous-functions/), like this:

setTimeout(function () {

person.getName();

}, 1000);

Code language: JavaScript (javascript)

This works because it gets the person from the outer scope and then calls the method getName().

Or you can use the bind() method:

let f = person.getName.bind(person);

setTimeout(f, 1000);

Code language: JavaScript (javascript)

In this code:

* First, bind the person.getName method to the person object.
* Second, pass the bound function f with this value set to the person object to the setTimeout() function.

## **Using bind() to borrow methods from a different object**

Suppose you have a runner object that has the run() method:

let runner = {

name: 'Runner',

run: function(speed) {

console.log(this.name + ' runs at ' + speed + ' mph.');

}

};

Code language: JavaScript (javascript)

And the flyer object that has the fly() method:

let flyer = {

name: 'Flyer',

fly: function(speed) {

console.log(this.name + ' flies at ' + speed + ' mph.');

}

};

Code language: JavaScript (javascript)

If you want the flyer object to be able to run, you can use the bind() method to create the run() function with the this  sets to the flyer object:

let run = runner.run.bind(flyer, 20);

run();

Code language: JavaScript (javascript)

In this statement:

* Call the bind() method of the runner.run() method and pass in the flyer object as the first argument and 20 as the second argument.
* Invoke the run() function.

Output:

Flyer runs at 20 mph.

The ability to borrow a method of an object without making a copy of that method and maintain it in two separate places is very powerful in JavaScript.

## **Summary**

* The bind() method creates a new function, when invoked, has the this sets to a provided value.
* The bind() method allows an object to borrow a method from another object without making a copy of that method. This is known as function borrowing in JavaScript.

# JavaScript Closures

**Summary**: in this tutorial, you will learn about JavaScript closures and how to use closures in your code more effectively.

## **Introduction to JavaScript closures**

In JavaScript, a closure is a [function](https://www.javascripttutorial.net/javascript-function/) that references variables in the outer scope from its inner scope. The closure preserves the outer scope inside its inner scope.

To understand the closures, you need to know how the lexical scoping works first.

### **Lexical scoping**

Lexical scoping defines the [scope of a variable](https://www.javascripttutorial.net/javascript-variable-scope/) by the position of that variable declared in the source code. For example:

let name = 'John';

function greeting() {

let message = 'Hi';

console.log(message + ' '+ name);

}

Code language: JavaScript (javascript)

In this example:

* The variable name is a global variable. It is accessible from anywhere including within the greeting() function.
* The variable message is a local variable that is accessible only within the greeting() function.

If you try to access the message variable outside the greeting() function, you will get an error.

So the JavaScript engine uses the scope to manage the variable accessibility.

According to lexical scoping, the scopes can be nested and the inner function can access the variables declared in its outer scope. For example:

function greeting() {

let message = 'Hi';

function sayHi() {

console.log(message);

}

sayHi();

}

greeting();

Code language: JavaScript (javascript)

The greeting() function creates a local variable named message and a function named sayHi().

The sayHi() is the inner function that is available only within the body of the greeting() function.

The sayHi() function can access the variables of the outer function such as the message variable of the greeting() function.

Inside the greeting() function, we call the sayHi() function to display the message Hi.

### **JavaScript closures**

Let’s modify the greeting() function:

function greeting() {

let message = 'Hi';

function sayHi() {

console.log(message);

}

return sayHi;

}

let hi = greeting();

hi(); *// still can access the message variable*

Code language: JavaScript (javascript)

Now, instead of executing the sayHi() function inside the greeting() function, the greeting() function returns the sayHi() function object.

Note that functions are the [first-class citizens in JavaScript](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/), therefore, you can return a function from another function.

Outside of the greeting() function, we assigned the hi variable the value returned by the greeting() function, which is a reference of the sayHi() function.

Then we executed the sayHi() function using the reference of that function: hi(). If you run the code, you will get the same effect as the one above.

However, the interesting point here is that, normally, a local variable only exists during the execution of the function.

It means that when the greeting() function has completed executing, the message variable is no longer accessible.

In this case, we execute the hi() function that references the sayHi() function, the message variable still exists.

The magic of this is closure. In other words, the sayHi() function is a closure.

A closure is a function that preserves the outer scope in its inner scope.

### **More JavaScript Closure example**

The following example illustrates a more practical example of closure.

function greeting(message) {

return function(name){

return message + ' ' + name;

}

}

let sayHi = greeting('Hi');

let sayHello = greeting('Hello');

console.log(sayHi('John')); *// Hi John*

console.log(sayHello('John')); *// Hello John*

Code language: JavaScript (javascript)

The greeting() function takes one argument named message and returns a function that accepts a single argument called name.

The return function returns a greeting message that is the combination of the message and name variables.

The greeting() function behaves like a function factory. It creates sayHi() and sayHello() functions with the respective messages Hi and Hello.

The sayHi() and sayHello() are closures. They share the same function body but store different scopes.

In the sayHi() closure, the message is Hi, while in the sayHello() closure the message is Hello.

## **JavaScript closures in a loop**

Consider the following example:

for (var index = 1; index <= 3; index++) {

setTimeout(function () {

console.log('after ' + index + ' second(s):' + index);

}, index \* 1000);

}

Code language: JavaScript (javascript)

Output

after 4 second(s):4

after 4 second(s):4

after 4 second(s):4

Code language: CSS (css)

The code shows the same message.

What we wanted to do in the loop is to copy the value of  i in each iteration at the time of iteration to display a message after 1, 2, and 3 seconds.

The reason you see the same message after 4 seconds is that the callback passed to the setTimeout() a closure. It remembers the value of i from the last iteration of the loop, which is 4.

In addition, all three closures created by the [for-loop](https://www.javascripttutorial.net/javascript-for-loop/) share the same global scope access the same value of i.

To fix this issue, you need to create a new closure scope in each iteration of the loop.

There are two popular solutions: IIFE & let keyword.

### **1) Using the IIFE solution**

In this solution, you use an [immediately invoked function expression](https://www.javascripttutorial.net/javascript-immediately-invoked-function-expression-iife/) (a.k.a IIFE) because an IIFE creates a new scope by declaring a function and immediately execute it.

for (var index = 1; index <= 3; index++) {

(function (index) {

setTimeout(function () {

console.log('after ' + index + ' second(s):' + index);

}, index \* 1000);

})(index);

}

Code language: JavaScript (javascript)

Output

after 1 second(s):1

after 2 second(s):2

after 3 second(s):3

Code language: CSS (css)

### **2) Using let keyword in ES6**

In ES6, you can use the [let](https://www.javascripttutorial.net/javascript-variables/#let) keyword to declare a variable that is block-scoped.

If you use the let keyword in the [for-loop](https://www.javascripttutorial.net/javascript-for-loop/), it will create a new lexical scope in each iteration. In other words, you will have a new index variable in each iteration.

In addition, the new lexical scope is chained up to the previous scope so that the previous value of the index is copied from the previous scope to the new one.

for (let index = 1; index <= 3; index++) {

setTimeout(function () {

console.log('after ' + index + ' second(s):' + index);

}, index \* 1000);

}

Code language: JavaScript (javascript)

Output

after 1 second(s):1

after 2 second(s):2

after 3 second(s):3

Code language: CSS (css)

## **Summary**

* Lexical scoping describes how the JavaScript engine uses the location of the variable in the code to determine where that variable is available.
* A closure is a combination of a function and its ability to remember variables in the outer scope.

# JavaScript Immediately Invoked Function Expression

**Summary**: in this tutorial, you will learn about JavaScript immediately invoked function expressions (IIFE).

A JavaScript immediately invoked function expression is a [function](https://www.javascripttutorial.net/javascript-function/) defined as an expression and executed immediately after creation. The following shows the syntax of defining an immediately invoked function expression:

(function(){

*//...*

})();

Code language: JavaScript (javascript)

## **Why IIFEs**

When you define a [function](https://www.javascripttutorial.net/javascript-function/), the JavaScript engine adds the function to the global object. See the following example:

function add(a,b) {

return a + b;

}

Code language: JavaScript (javascript)

In web browsers, the JavaScript engine adds the add() function to the window global object:

console.log(window.add);

Code language: JavaScript (javascript)

Likewise, if you declare a [variable](https://www.javascripttutorial.net/javascript-variables/) outside of a function using the var keyword, the JavaScript engine also adds the variable to the global object:

var counter = 10;

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

Code language: JavaScript (javascript)

If you have many global variables and functions, the JavaScript engine will only release the memory allocated for them until the global object loses its scopes.

As a result, the script may use the memory inefficiently. On top of that, having global variables and functions will likely cause the name collisions.

One way to prevent the functions and variables from polluting the global object is to use immediately invoked function expressions.

In JavaScript, you can have the following expressions:

'This is a string';

(10+20);

Code language: JavaScript (javascript)

This syntax is correct even though the expressions have no effect. A function can be also declared as an expression which is called a function expression:

let sum = function(a, b) {

return a + b;

}

Code language: JavaScript (javascript)

In this syntax, the part on the right side of the assignment operator(=) is a function expression. Because a function is an expression, you can wrap it inside parentheses:

let sum = (function(a, b) {

return a + b;

});

Code language: JavaScript (javascript)

In this example, the sum variable is referenced as the [anonymous function](https://www.javascripttutorial.net/javascript-anonymous-functions/) that adds two arguments.

In addition, you can execute the function immediately after creating it:

let sum = (function(a,b){

return a + b;

})(10, 20);

console.log(sum);

Code language: JavaScript (javascript)

In this example, the sum variable holds the result of the function call.

The following expression is called an immediately invoked function expression (IIFE) because the function is created as an expression and executed immediately:

(function(a,b){

return a + b;

})(10,20);

Code language: JavaScript (javascript)

This is the general syntax for defining an IIFE:

(function(){

*//...*

})();

Code language: JavaScript (javascript)

Note that you can use an [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) to define an IIFE:

(() => {

*//...*

})();

Code language: JavaScript (javascript)

By placing [functions](https://www.javascripttutorial.net/javascript-function/) and [variables](https://www.javascripttutorial.net/javascript-variables/) inside an immediately invoked function expression, you can avoid polluting them to the global object:

(function() {

var counter = 0;

function add(a, b) {

return a + b;

}

console.log(add(10,20)); *// 30*

}());

Code language: JavaScript (javascript)

## **Named IIFE**

An IIFE can have a name. However, it cannot be invoked again after execution:

(function namedIIFE() {

*//...*

})();

Code language: JavaScript (javascript)

## **IIFE starting with a semicolon (;)**

Sometimes, you may see an IIFE that start with a semicolon(;):

;(function() {

*/\* \*/*

})();

Code language: JavaScript (javascript)

In this syntax, the semicolon is used to terminate the statement in case two or more JavaScript files are blindly concatenated into a single file.

For example, you may have two file lib1.js and lib2.js which use IIFEs:

(function(){

*// ...*

})()

(function(){

*// ...*

})()

Code language: JavaScript (javascript)

If you use a code bundler tool to concatenate code from both files into a single file, without the semicolon (;) the concatenated JavaScript code will cause a syntax error.

## **IIFE in actions**

Suppose that you have a library called calculator.js with the following functions:

function add(a, b) {

return a + b;

}

function mutiply(a, b) {

return a \* b;

}

Code language: JavaScript (javascript)

And you load the calculator.js in an HTML document.

Later, you also want to load another JavaScript library called app.js to the same document:

<!DOCTYPE html>

**<head>**

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

**<title>**JavaScript IIFE**</title>**

**</head>**

**<body>**

**<script src="calculator.js"></script>**

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

**</body>**

**</html>**

Code language: HTML, XML (xml)

The app.js also has the add() function:

function add() {

return 'add';

}

Code language: JavaScript (javascript)

When you use the add() function in the HTML document, it returns the 'add' string instead of the sum of two numbers:

let result = add(10, 20);

console.log(result); *// 'add'*

Code language: JavaScript (javascript)

This is because the add() function in the app.js overrides the add() function in the calculator.js library.

To fix this, you can apply IIFE in the calculator.js as follows:

const calculator = (function () {

function add(a, b) {

return a + b;

}

function multiply(a, b) {

return a \* b;

}

return {

add: add,

multiply: multiply

}

})();

Code language: JavaScript (javascript)

The IIFE returns an object that contains the add and multiply methods that reference the add() and multiply() functions. In the HTML document, you can use the calculator.js library as follows:

<!DOCTYPE html>

**<head>**

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

**<title>**JavaScript IIFE**</title>**

**</head>**

**<body>**

**<script src="js/calculator.js"></script>**

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

**<script>**

let result = calculator.add(10, 20); *// add in app.js*

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

console.log(add()); *// add in the app.js*

**</script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

The calculator.add() called the add() function exported by the calculator.js while the second call to the add() function references the add() function in the app.js.

## **jQuery & IIFE**

The following HTML document uses the jQuery library:

<!DOCTYPE html>

**<head>**

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

**<title>**JavaScript IIFE - jQuery**</title>**

**</head>**

**<body>**

**<h1>**jQuery Demo**</h1>**

**<script src="https://code.jquery.com/jquery-3.4.1.slim.js"**

**integrity="sha256-BTlTdQO9/fascB1drekrDVkaKd9PkwBymMlHOiG+qLI=" crossorigin="anonymous"></script>**

**<script>**

let counter = 1;

$('h1').click(function () {

$(this).text('jQuery Demo' + ' Clicked ' + counter++);

});

**</script>**

**</body>**

**</html>**

Code language: HTML, XML (xml)

When you import the jQuery library, you can access many useful jQuery functions via the $ or jQuery object. Under the hood, jQuery uses the IIFE to expose its functionality.

By doing this, jQuery just needs to use one global variable ($) to expose a ton of functions without polluting the global object.

The following example illustrates how to change the jQuery $ object to \_ inside the IIFE:

(function (\_) {

let counter = 1;

\_('h1').click(function () {

\_(this).text('jQuery Demo' + ' Clicked ' + counter++);

});

})(jQuery);

Code language: JavaScript (javascript)

In this example, we passed the jQuery object into the IIFE. In the function inside the IIFE used the \_ argument instead.

In this tutorial, you will have learned about the JavaScript immediately invoked function expressions (IIFE) and their purposes.

# Returning Multiple Values from a Function

**Summary**: in this tutorial, you will learn to define JavaScript functions that return multiple values.

[JavaScript functions](https://www.javascripttutorial.net/javascript-function/) can return a single value. To return multiple values from a function, you can pack the return values as elements of an [array](https://www.javascripttutorial.net/javascript-array/) or as properties of an [object](https://www.javascripttutorial.net/javascript-objects/).

## **Returning multiple values from a function using an array**

Suppose the following getNames() function retrieves the first name and last name from a database in the backend or from the result of a third-party API call and returns them as elements of an array:

function getNames() {

*// get names from the database or API*

let firstName = 'John',

lastName = 'Doe';

*// return as an array*

return [firstName, lastName];

}

Code language: JavaScript (javascript)

The following shows how to get the return value from the getNames() function:

let names = getNames();

Code language: JavaScript (javascript)

Because the names variable is an array, you can reference its elements using the square brackets, like this:

const firstName = names[0],

lastName = names[1];

Code language: JavaScript (javascript)

In ES6, you can use the [destructuring assignment](https://www.javascripttutorial.net/es6/destructuring/) syntax to unpack values from an array more intuitively, like this:

const [firstName, lastName] = getNames();

Code language: JavaScript (javascript)

In this code, the firstName and lastName variables will take the first and second elements of the return array.

## **Returning multiple values from an function using an object**

If you want to assign a name to each returned value to make it more readable and easier to maintain, you can use an [object](https://www.javascripttutorial.net/javascript-objects/):

function getNames() {

*// get names from the database or API*

let firstName = 'John',

lastName = 'Doe';

*// return values*

return {

'firstName': firstName,

'lastName': lastName

};

}

Code language: JavaScript (javascript)

Since the names of the properties are the same as the variables, you can shorten it using the [object literal syntax extensions in ES6](https://www.javascripttutorial.net/es6/object-literal-extensions/) as follows:

function getNames() {

*// get names from the database or API*

let firstName = 'John',

lastName = 'Doe';

return { firstName, lastName };

}

Code language: JavaScript (javascript)

And you can get the return value as an object like this:

let names = getNames();

let firstName = names.firstName,

lastName = names.lastName;

Code language: JavaScript (javascript)

If you want to unpack properties from an object, you can use the [object destructuring syntax](https://www.javascripttutorial.net/es6/destructuring/) as follows:

let { firstName, lastName } = getNames();

Code language: JavaScript (javascript)

## **Summary**

* JavaScript doesn’t support functions that return multiple values. However, you can wrap multiple values into an array or an object and return the array or the object.
* Use destructuring assignment syntax to unpack values from the array, or properties from objects.

# 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.

# 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/" \l "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 Callbacks

**Summary**: in this tutorial, you will learn about JavaScript callback functions including synchronous and asynchronous callbacks.

## **What are callbacks**

In JavaScript, [functions are first-class citizens](https://www.javascripttutorial.net/javascript-functions-are-first-class-citizens/). Therefore, you can pass a [function](https://www.javascripttutorial.net/javascript-function/) to another function as an argument.

By definition, a callback is a [function](https://www.javascripttutorial.net/javascript-function/) that you pass into another function as an argument for executing later.

The following defines a filter() function that accepts an [array](https://www.javascripttutorial.net/array/) of numbers and returns a new array of odd numbers:

function filter(numbers) {

let results = [];

for (const number of numbers) {

if (number % 2 != 0) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers));

Code language: JavaScript (javascript)

How it works.

* First, define the filter() function that accepts an array of numbers and returns a new array of the odd numbers.
* Second, define the numbers array that has both odd and even numbers.
* Third, call the filter() function to get the odd numbers out of the numbers array and output the result.

If you want to return an array that contains even numbers, you need to modify the filter() function. To make the filter() function more generic and reusable, you can:

* First, extract the logic in the if block and wrap it in a separate function.
* Second, pass the function to the filter() function as an argument.

Here’s the updated code:

function isOdd(number) {

return number % 2 != 0;

}

function filter(numbers, fn) {

let results = [];

for (const number of numbers) {

if (fn(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));

Code language: JavaScript (javascript)

The result is the same. However, you can pass any function that accepts an argument and returns a boolean value to the second argument of the filter() function.

For example, you can use the filter() function to return an array of even numbers like this:

function isOdd(number) {

return number % 2 != 0;

}

function isEven(number) {

return number % 2 == 0;

}

function filter(numbers, fn) {

let results = [];

for (const number of numbers) {

if (fn(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

console.log(filter(numbers, isOdd));

console.log(filter(numbers, isEven));

Code language: JavaScript (javascript)

By definition, the isOdd and isEven are callback functions or callbacks. Because the filter() function accepts a function as an argument, it’s called a high-order function.

A callback can be an anonymous function, which is a function without a name like this:

function filter(numbers, callback) {

let results = [];

for (const number of numbers) {

if (callback(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, function (number) {

return number % 2 != 0;

});

console.log(oddNumbers);

Code language: JavaScript (javascript)

In this example, we pass an anonymous function to the filter() function instead of using a separate function.

In ES6, you can use an [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/) like this:

function filter(numbers, callback) {

let results = [];

for (const number of numbers) {

if (callback(number)) {

results.push(number);

}

}

return results;

}

let numbers = [1, 2, 4, 7, 3, 5, 6];

let oddNumbers = filter(numbers, (number) => number % 2 != 0);

console.log(oddNumbers);

Code language: JavaScript (javascript)

There are two types of callbacks: synchronous and asynchronous callbacks.

## **Synchronous callbacks**

A synchronous callback is executed during the execution of the high-order function that uses the callback. The isOdd and isEven are examples of synchronous callbacks because they execute during the execution of the filter() function.

## **Asynchronous callbacks**

An asynchronous callback is executed after the execution of the high-order function that uses the callback.

Asynchronicity means that if JavaScript has to wait for an operation to complete, it will execute the rest of the code while waiting.

Note that JavaScript is a single-threaded programming language. It carries asynchronous operations via the callback queue and[event loop](https://www.javascripttutorial.net/javascript-event-loop/).

Suppose that you need to develop a script that downloads a picture from a remote server and process it after the download completes:

function download(url) {

*// ...*

}

function process(picture) {

*// ...*

}

download(url);

process(picture);

Code language: JavaScript (javascript)

However, downloading a picture from a remote server takes time depending on the network speed and the size of the picture.

The following download() function uses the setTimeout() function to simulate the network request:

function download(url) {

setTimeout(() => {

*// script to download the picture here*

console.log(`Downloading ${url} ...`);

},1000);

}

Code language: JavaScript (javascript)

And this code emulates the process() function:

function process(picture) {

console.log(`Processing ${picture}`);

}

Code language: JavaScript (javascript)

When you execute the following code:

let url = 'https://www.javascripttutorial.net/pic.jpg';

download(url);

process(url);

Code language: JavaScript (javascript)

you will get the following output:

Processing https:*//javascripttutorial.net/pic.jpg*

Downloading https:*//javascripttutorial.net/pic.jpg ...*

Code language: JavaScript (javascript)

This is not what you expected because the process() function executes before the download() function. The correct sequence should be:

* Download the picture and wait for the download completes.
* Process the picture.

To resolve this issue, you can pass the process() function to the download() function and execute the process() function inside the download() function once the download completes, like this:

function download(url, callback) {

setTimeout(() => {

*// script to download the picture here*

console.log(`Downloading ${url} ...`);

*// process the picture once it is completed*

callback(url);

}, 1000);

}

function process(picture) {

console.log(`Processing ${picture}`);

}

let url = 'https://wwww.javascripttutorial.net/pic.jpg';

download(url, process);

Code language: JavaScript (javascript)

Output:

Downloading https:*//www.javascripttutorial.net/pic.jpg ...*

Processing https:*//www.javascripttutorial.net/pic.jpg*

Code language: JavaScript (javascript)

Now, it works as expected.

In this example, the process() is a callback passed into an asynchronous function.

When you use a callback to continue code execution after an asynchronous operation, the callback is called an asynchronous callback.

To make the code more concise, you can define the process() function as an anonymous function:

function download(url, callback) {

setTimeout(() => {

*// script to download the picture here*

console.log(`Downloading ${url} ...`);

*// process the picture once it is completed*

callback(url);

}, 1000);

}

let url = 'https://www.javascripttutorial.net/pic.jpg';

download(url, function(picture) {

console.log(`Processing ${picture}`);

});

Code language: JavaScript (javascript)

### **Handling errors**

The download() function assumes that everything works fine and does not consider any exceptions. The following code introduces two callbacks: success and failure to handle the success and failure cases respectively:

function download(url, success, failure) {

setTimeout(() => {

console.log(`Downloading the picture from ${url} ...`);

!url ? failure(url) : success(url);

}, 1000);

}

download(

'',

(url) => console.log(`Processing the picture ${url}`),

(url) => console.log(`The '${url}' is not valid`)

);

Code language: JavaScript (javascript)

### **Nesting callbacks and the Pyramid of Doom**

How do you download three pictures and process them sequentially? A typical approach is to call the download() function inside the callback function, like this:

function download(url, callback) {

setTimeout(() => {

console.log(`Downloading ${url} ...`);

callback(url);

}, 1000);

}

const url1 = 'https://www.javascripttutorial.net/pic1.jpg';

const url2 = 'https://www.javascripttutorial.net/pic2.jpg';

const url3 = 'https://www.javascripttutorial.net/pic3.jpg';

download(url1, function (url) {

console.log(`Processing ${url}`);

download(url2, function (url) {

console.log(`Processing ${url}`);

download(url3, function (url) {

console.log(`Processing ${url}`);

});

});

});

Code language: JavaScript (javascript)

Output:

Downloading https:*//www.javascripttutorial.net/pic1.jpg ...*

Processing https:*//www.javascripttutorial.net/pic1.jpg*

Downloading https:*//www.javascripttutorial.net/pic2.jpg ...*

Processing https:*//www.javascripttutorial.net/pic2.jpg*

Downloading https:*//www.javascripttutorial.net/pic3.jpg ...*

Processing https:*//www.javascripttutorial.net/pic3.jpg*

Code language: JavaScript (javascript)

The script works perfectly fine.

However, this callback strategy does not scale well when the complexity grows significantly.

Nesting many asynchronous functions inside callbacks is known as the **pyramid of doom** or the **callback hell**:

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

asyncFunction(function(){

....

});

});

});

});

});

Code language: JavaScript (javascript)

To avoid the pyramid of doom, you use [promises](https://www.javascripttutorial.net/es6/javascript-promises/) or [async/await](https://www.javascripttutorial.net/es-next/javascript-async-await/) functions.

## **Summary**

* A callback is a function passed into another function as an argument to be executed later.
* A high-order function is a function that accepts another function as an argument.
* Callback functions can be synchronous or asynchronous.

# 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:

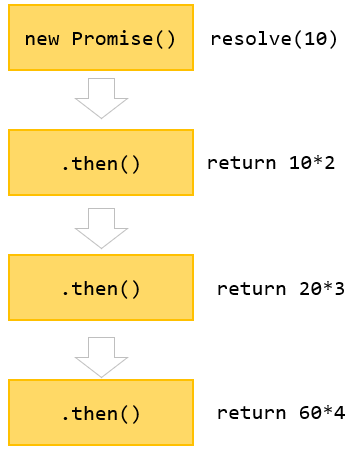
10

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:

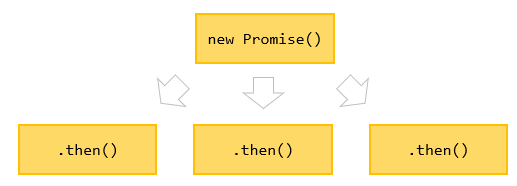
10

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.

# JavaScript Promise.any()

**Summary**: in this tutorial, you’ll learn how to use the JavaScript Promise.any() method to compose promises.

## **Introduction to JavaScript Promise.any() method**

The Promise.any() method accepts a list of [Promise](https://www.javascripttutorial.net/es6/javascript-promises/) objects as an [iterable object](https://www.javascripttutorial.net/es-next/javascript-asynchronous-iterators/):

Promise.any(iterable);

Code language: JavaScript (javascript)

If one of the promises in the iterable object is fulfilled, the Promise.any() returns a single promise that resolves to a value which is the result of the fulfilled promise:

In this diagram:

* The promise1 resolves to a value v1 at t1.
* The promise2 resolves to a value v2 at t2.
* The Promise.any() returns a promise that resolves to a value v1, which is the result of the promise1, at t1

The Promise.any() returns a promise that is fulfilled with any first fulfilled promise even if some promises in the iterable object are rejected:

In this diagram:

* The promise1 is rejected with an error at t1.
* The promise2 is fulfilled to value v2 at t2.
* The Promise.any() returns the a promise that resolves to a value v2 which is the result of the promise2. Note that the Promise.any() method ignores the rejected promise (promise1).

If all promises in the iterable object are rejected or if the iterable object is empty, the Promise.any() return a promise that rejects with an AggregateError containing all the rejection reasons. The AggregateError is a subclass of Error.

In this diagram:

* The promise1 is rejected for an error1 at t1.
* The promise2 is rejected for an error2 at t2.
* The Promise.any() returns a promise that is rejected at t2 with an AggregateError containing the error1 and error2 of all the rejected promises.

## **JavaScript Promise.any() examples**

Let’s take some examples of using the Promise.any() method.

### **1) All promises fulfilled example**

The following example demonstrates the Promise.any() method with all promises fulfilled:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 1 fulfilled');

resolve(1);

}, 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 2 fulfilled');

resolve(2);

}, 2000);

});

const p = Promise.any([p1, p2]);

p.then((value) => {

console.log('Returned Promise');

console.log(value);

});

Code language: JavaScript (javascript)

Output:

Promise 1 fulfilled

Returned Promise

1

Promise 2 fulfilled

Code language: JavaScript (javascript)

How it works.

* First, create a new promise p1 that will resolve to a value 1 after one second.
* Second, create a new promise p2 that will resolve to a value 2 after two seconds.
* Third, use the Promise.any() method that uses two promises p1 and p2. The Promise.any() returns a promise p that will resolve to the value 1 of the first fulfilled promise (p1) after one second.

### **2) One promise rejected example**

The following example uses the Promise.any() method with list of promises that has a rejected promise:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 1 rejected');

reject('error');

}, 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 2 fulfilled');

resolve(2);

}, 2000);

});

const p = Promise.any([p1, p2]);

p.then((value) => {

console.log('Returned Promise');

console.log(value);

});

Code language: JavaScript (javascript)

Output:

Promise 1 rejected

Promise 2 fulfilled

Returned Promise

2

Code language: JavaScript (javascript)

In this example, the Promise.any() ignores the rejected promise. When the p2 resolves with the value 2, the Promise.any() returns a promise that resolves to the same value of the result of the p2.

### **3) All promises rejected example**

The following example demonstrates how to use the Promise.any() method with all promises rejected:

const p1 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 1 rejected');

reject('error1');

}, 1000);

});

const p2 = new Promise((resolve, reject) => {

setTimeout(() => {

console.log('Promise 2 rejected');

reject('error2');

}, 2000);

});

const p = Promise.any([p1, p2]);

p.catch((e) => {

console.log('Returned Promise');

console.log(e, e.errors);

});

Code language: JavaScript (javascript)

Output:

Promise 1 rejected

Promise 2 rejected

Returned Promise

[AggregateError: All promises were rejected] [ 'error1', 'error2' ]

Code language: JavaScript (javascript)

In this example, both p1 and p2 were rejected with the string error1 and error2. Therefore, the Promise.any() method was rejected with an AggregateError object that has the errors property containing all the errors of the rejected promises.

## **When to use the JavaScript Promise.any() method**

In practice, you use the Promise.any() to return the first fulfilled promise. Once a promise is fulfilled, the Promise.any() method does not wait for other promises to be complete. In other words, the Promise.any() short-circuits after a promise is fulfilled.

For example, you have a resource served by two or more content delivery networks (CDN). To dynamically load the first available resource, you can use the Promise.any() method.

The following example uses the Promise.any() method to fetch two images and displays the first available image.

### **The index.html file**

<!DOCTYPE html>

**<html lang="en">**

**<head>**

**<meta charset="UTF-8" />**

**<meta name="viewport" content="width=device-width, initial-scale=1.0" />**

**<title>**JavaScript Promise.any() Demo**</title>**

**</head>**

**<body>**

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

**</body>**

**</html>**

Code language: JavaScript (javascript)

### **The app.js file**

function getImageBlob(url) {

return fetch(url).then((response) => {

if (!response.ok) {

throw new Error(`HTTP status: ${response.status}`);

}

return response.blob();

});

}

let cat = getImageBlob(

'https://upload.wikimedia.org/wikipedia/commons/4/43/Siberian\_black\_tabby\_blotched\_cat\_03.jpg'

);

let dog = getImageBlob(

'https://upload.wikimedia.org/wikipedia/commons/a/af/Golden\_retriever\_eating\_pigs\_foot.jpg'

);

Promise.any([cat, dog])

.then((data) => {

let objectURL = URL.createObjectURL(data);

let image = document.createElement('img');

image.src = objectURL;

document.body.appendChild(image);

})

.catch((e) => {

console.log(e.message);

});

Code language: JavaScript (javascript)

How it works.

* First, define the getImageBlob() function that uses the [fetch API](https://www.javascripttutorial.net/javascript-fetch-api/) to get the image’s blob from an URL. The getImageBlob() returns a Promise object that resolves to the image blob.
* Second, define two promises that load the images.
* Third, show the first available image by using the Promise.any() method.

## **Summary**

* Use the JavaScript Promise.any() method to take a list of promises and return a promise that fulfills first.

# JavaScript Promise.allSettled()

**Summary**: in this tutorial, you’ll learn about the Promise.allSettled() method to compose promises.

## **Introduction to the Promise.allSettled() method**

ES2020 introduced the Promise.allSettled() method that accepts a list of [Promises](https://www.javascripttutorial.net/es6/javascript-promises/) and returns a new promise that resolves after all the input promises have settled, either resolved or rejected.

The following shows the syntax of the Promise.allSettled() method:

Promise.allSettled(iterable);

Code language: JavaScript (javascript)

The iterable contains the promises. The Promise.allSettled() returns a pending promise that will be asynchronously fulfilled once every input promise has settled.

The Promise.allSettled() method returns a promise that resolves to an [array](https://www.javascripttutorial.net/javascript-array/) of [objects](https://www.javascripttutorial.net/javascript-objects/) that each describes the result of the input promise.

Each object has two properties: status and value (or reason).

* The status can be either fulfilled or rejected.
* The value if case the promise is fulfilled or reason) if the promise is rejected.

The following diagram illustrates how the Promise.allSettled() method works:

In this diagram:

* The promise1 rejects to the error at t1.
* The promise2 resolves to a value at t2.
* The Promise.allSettled() method resolves to a array containing objects that describe the statuses and outcomes of the promise1 and promise2.

## **JavaScript Promise.allSettled() example**

The following example uses the Promise.allSettled() to wait for all the input Promises to settle:

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.allSettled([p1, p2])

.then((result) => {

console.log(result);

});

Code language: JavaScript (javascript)

Output:

Graphical user interface, text, application

Description automatically generated

How it works:

* The first promise p1 resolves to the value 10 after one second
* The second promise p2 rejects for a reason with a value 20 after two seconds.
* The Promise.allSettled() returns a promise that resolves to the result array that has two elements. The first element is an object resolved by the p1 promise and the second one is another object which is rejected by the p2 promise.

## **Summary**

* The Promise.allSettled() method accepts an iterable of promises and returns a new promise that resolves when every input promise has settled with an array of objects that describes the result of each promise in the iterable object.

# JavaScript Promise finally()

**Summary**: in this tutorial, you will learn how to use the JavaScript Promise finally() method to execute the code once the promise is settled, regardless of its outcome.

## **Introduction to the JavaScript Promise finally() method**

Suppose that you have a [promise](https://www.javascripttutorial.net/es6/javascript-promises/):

promise

.then(result => { ...})

.catch(error => { ... })

.finally(() => { ... })

Code language: JavaScript (javascript)

The finally() method is always executed whether the promise is fulfilled or rejected. In other words, the finally() method is executed when the promise is settled.

The finally() method was introduced in ES2018. In the finally() method, you can place the code that cleans up the resource when the promise is settled, regardless of its outcome. By using the finally() method, you can avoid duplicate code in the then() and catch() methods like this:

promise

.then(result => {

*// process the result*

*// clean up the resources*

})

.catch(error => {

*// handle the error*

*// clean up the resources*

});

Code language: JavaScript (javascript)

Now, you can move the clean up the resources part to the finally() method as follows:

promise

.then(result => {

*// process the result*

})

.catch(error => {

*// handle the error*

})

.finally(() => {

*// clean up the resources*

});

Code language: JavaScript (javascript)

The finally() method is similar to the finally block in the [try...catch...finally](https://www.javascripttutorial.net/javascript-try-catch-finally/) statement. In synchronous code, you use the finally block to clean up the resources. In asynchronous code, you use the finally() method instead.

## **The JavaScript Promise finally() method example**

The following defines a Connection class:

class Connection {

execute(query) {

if (query != 'Insert' && query != 'Update' && query != 'Delete') {

throw new Error(`The ${query} is not supported`);

}

console.log(`Execute the ${query}`);

return this;

}

close() {

console.log('Close the connection')

}

}

Code language: JavaScript (javascript)

The Connection class has two methods: execute() and close():

* The execute() method will only execute the insert, update, or delete query. It will issue an error if you pass into another query that is not in the list.
* The close() method closes the connection, kind of cleaning up the resource.

The following connect() function returns a promise that resolves to a new Connection if the success flag is set to true:

const success = true;

function connect() {

return new Promise((resolve, reject) => {

if (success)

resolve(new Connection());

else

reject('Could not open the database connection');

});

}

Code language: JavaScript (javascript)

The following example uses the finally() method to close the connection:

let globalConnection;

connect()

.then((connection) => {

globalConnection = connection;

return globalConnection.execute('Insert');

})

.then((connection) => {

globalConnection = connection;

return connection.execute('Select');

})

.catch(console.log)

.finally(() => {

if (globalConnection) {

globalConnection.`close()`;

}

});

Code language: JavaScript (javascript)

In this example:

* The connect() function resolves to a new Connection  object because the success flag is set to true.
* The first then() method executes the Insert query and returns a Connection object. The globalConnection is used to save the connection.
* The second then() method executes the Select query and issues an error. The catch() method shows the error message and the finally() method closes the connection.

## **Summary**

* The finally() method schedule a function to execute when the promise is settled, either fulfilled or rejected.
* It’s good practice to place the code that cleans up the resources in the finally() method once the promise is settled, regardless of its outcome.

# 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 async/await

**Summary**: in this tutorial, you will learn how to write asynchronous code  using JavaScript  async/  await keywords.

Note that to understand how the async / await works, you need to know how [promises](https://www.javascripttutorial.net/es6/javascript-promises/) work.

## **Introduction to JavaScript async / await keywords**

In the past, to deal with asynchronous operations, you often used the [callback functions](https://www.javascripttutorial.net/javascript-callback/). However, when you nest many callback functions, the code will be more difficult to maintain. And you end up with a notorious issue which is known as the callback hell.

Suppose that you need to perform three asynchronous operations in the following sequence:

1. Select user from the database.
2. Get services of the user from an API.
3. Calculate the service cost based on the services from the server.

The following functions illustrates the three tasks. Note that we use the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function to simulate the asynchronous operation.

function getUser(userId, callback) {

console.log('Get user from the database.');

setTimeout(() => {

callback({

userId: userId,

username: 'john'

});

}, 1000);

}

function getServices(user, callback) {

console.log(`Get services of ${user.username} from the API.`);

setTimeout(() => {

callback(['Email', 'VPN', 'CDN']);

}, 2 \* 1000);

}

function getServiceCost(services, callback) {

console.log(`Calculate service costs of ${services}.`);

setTimeout(() => {

callback(services.length \* 100);

}, 3 \* 1000);

}

Code language: JavaScript (javascript)

The following shows the nested callback functions:

getUser(100, (user) => {

getServices(user, (services) => {

getServiceCost(services, (cost) => {

console.log(`The service cost is ${cost}`);

});

});

});

Code language: JavaScript (javascript)

Output:

Get user from the database.

Get services of john from the API.

Calculate service costs of Email,VPN,CDN.

The service cost is 300

Code language: JavaScript (javascript)

To avoid this callback hell issue, ES6 introduced the [promises](https://www.javascripttutorial.net/es6/javascript-promises/) that allow you to write asynchronous code in more manageable ways.

First, you need to return a Promise in each function:

function getUser(userId) {

return new Promise((resolve, reject) => {

console.log('Get user from the database.');

setTimeout(() => {

resolve({

userId: userId,

username: 'john'

});

}, 1000);

})

}

function getServices(user) {

return new Promise((resolve, reject) => {

console.log(`Get services of ${user.username} from the API.`);

setTimeout(() => {

resolve(['Email', 'VPN', 'CDN']);

}, 2 \* 1000);

});

}

function getServiceCost(services) {

return new Promise((resolve, reject) => {

console.log(`Calculate service costs of ${services}.`);

setTimeout(() => {

resolve(services.length \* 100);

}, 3 \* 1000);

});

}

Code language: JavaScript (javascript)

Then, you [chain the promises](https://www.javascripttutorial.net/es6/promise-chaining/):

getUser(100)

.then(getServices)

.then(getServiceCost)

.then(console.log);

Code language: CSS (css)

ES2017 introduced the async/await keywords that build on top of promises, allowing you to write asynchronous code that looks more like synchronous code and more readable. Technically speaking, the async / await is syntactic sugar for promises.

If a function returns a Promise, you can place the await keyword in front of the function call, like this:

let result = await f();

Code language: JavaScript (javascript)

The await will wait for the Promise returned from the f() to settle. The await keyword can be used only inside the async functions.

The following defines an async function that calls the three asynchronous operations in sequence:

async function showServiceCost() {

let user = await getUser(100);

let services = await getServices(user);

let cost = await getServiceCost(services);

console.log(`The service cost is ${cost}`);

}

showServiceCost();

Code language: JavaScript (javascript)

As you can see, the asynchronous code now looks like the synchronous code.

Let’s dive in the async / await keywords.

## **The async keyword**

The async keyword allows you to define a function that handles asynchronous operations.

To define an async function, you place the async keyword in front of the function keyword as follows:

async function sayHi() {

return 'Hi';

}

Code language: JavaScript (javascript)

Asynchronous functions execute asynchronously via the [event loop](https://www.javascripttutorial.net/javascript-event-loop/). It always returns a Promise.

In this example, because the sayHi() function returns a Promise, you can consume it, like this:

sayHi().then(console.log);

Code language: CSS (css)

You can also explicitly return a Promise from the sayHi() function as shown in the following code:

async function sayHi() {

return Promise.resolve('Hi');

}

Code language: JavaScript (javascript)

The effect is the same.

Besides the regular functions, you can use the async keyword in the function expressions:

let sayHi = async function () {

return 'Hi';

}

Code language: JavaScript (javascript)

[arrow functions](https://www.javascripttutorial.net/es6/javascript-arrow-function/):

let sayHi = async () => 'Hi';

Code language: JavaScript (javascript)

and methods of classes:

class Greeter {

async sayHi() {

return 'Hi';

}

}

Code language: JavaScript (javascript)

## **The await keyword**

You use the await keyword to wait for a Promise to settle either in resolved or rejected state. And you can use the await keyword only inside an async function:

async function display() {

let result = await sayHi();

console.log(result);

}

Code language: JavaScript (javascript)

In this example, the await keyword instructs JavaScript engine to wait for the sayHi() function to complete before displaying the message.

Note that if you use the await operator outside of an async function, you will get an error.

## **Error handling**

If a promise resolves, the await promise returns the result. However, when the promise rejects, the await promise will throw an error as if there were a throw statement.

The following code:

async function getUser(userId) {

await Promise.reject(new Error('Invalid User Id'));

}

Code language: JavaScript (javascript)

… is the same as this:

async function getUser(userId) {

throw new Error('Invalid User Id');

}

Code language: JavaScript (javascript)

In the real scenario, it will take a while for the promise to throw an error.

You can catch the error by using the [try...catch](https://www.javascripttutorial.net/javascript-try-catch/) statement, the same way as a regular throw statement:

async function getUser(userId) {

try {

const user = await Promise.reject(new Error('Invalid User Id'));

} catch(error) {

console.log(error);

}

}

Code language: JavaScript (javascript)

It’s possible to catch errors caused by one or more await promise‘s:

async function showServiceCost() {

try {

let user = await getUser(100);

let services = await getServices(user);

let cost = await getServiceCost(services);

console.log(`The service cost is ${cost}`);

} catch(error) {

console.log(error);

}

}

Code language: JavaScript (javascript)

In this tutorial, you have learned how to use the JavaScript async / await keyword to write asynchronous code look like synchronous code.

# 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 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.

# JavaScript Asynchronous Iterators

**Summary**: in this tutorial, you will learn about the JavaScript asynchronous iterators that allow you to access asynchronous data sequentially.

Introduction to JavaScript Asynchronous Iterators

[ES6](https://www.javascripttutorial.net/es6/) introduced the [iterator](https://www.javascripttutorial.net/es6/javascript-iterator/) interface that allows you to access data sequentially. The iterator is well-suited for accessing the synchronous data sources like [arrays](https://www.javascripttutorial.net/javascript-array/), [sets](https://www.javascripttutorial.net/es6/javascript-set/), and [maps](https://www.javascripttutorial.net/es6/javascript-map/).

The main method of an iterator interface is the next() that returns the {value, done} object, where done is a boolean indicating whether the end of the sequence is reached and value is the yielded value in the sequence.

The synchronous data means that the next value in the sequence and the done state is known at the time the next() method returns.

Besides the synchronous data sources, JavaScript often has to access asynchronous data sources like I/O access. For the asynchronous data sources, the value and done state of the iterator is often unknown at the time the next() method returns.

To deal with the asynchronous data sources, ES2018 introduced the asynchronous iterator (or async iterator) interface.

An async iterator is like an iterator except that its next() method returns a [promise](https://www.javascripttutorial.net/es6/javascript-promises/) that resolves to the {value, done} object.

The following illustrates the Sequence class that implements the iterator interface. (Check it out the [iterator tutorial](https://www.javascripttutorial.net/es6/javascript-iterator/) for more information on how to implement Sequence class.)

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)

To make this Sequence class asynchronously, you need to modify it as follows:

* Use the Symbol.asyncIterator instead of the Symbol.iterator
* Return a Promise from the next() method.

The following code transforms the Sequence class to the AsyncSequence class:

class AsyncSequence {

constructor(start = 0, end = Infinity, interval = 1) {

this.start = start;

this.end = end;

this.interval = interval;

}

[Symbol.asyncIterator]() {

let counter = 0;

let nextIndex = this.start;

return {

next: async () => {

if (nextIndex <= this.end) {

let result = {

value: nextIndex,

done: false

}

nextIndex += this.interval;

counter++;

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve(result);

}, 1000);

});

}

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve({

value: counter,

done: true

});

}, 1000);

});

}

}

}

}

Code language: JavaScript (javascript)

The AsyncSequence returns the next number in the sequence after every 1 second.

The for await...of statement

To iterate over an asynchronous iterable object, ES2018 introduced the for await...of statement:

for await (variable of iterable) {

*// statement*

}

Code language: JavaScript (javascript)

Since we can use the await keyword in an async function only, we can create an async [IIFE](https://www.javascripttutorial.net/javascript-immediately-invoked-function-expression-iife/) as that uses the AsyncSequence class as follows:

(async () => {

let seq = new AsyncSequence(1, 10, 1);

for await (let value of seq) {

console.log(value);

}

})();

Code language: JavaScript (javascript)

Output (each number is returned after every second)

1

2

3

4

5

6

7

8

9

10

Code language: JavaScript (javascript)

The following table illustrates the differences between the iterators and async iterators:

| **#** | **Iterators** | **Async iterators** |
| --- | --- | --- |
| Well-known Symbol | Symbol.iterator | Symbol.asyncIterator |
| next() return value is | {value, done } | [Promise](https://www.javascripttutorial.net/es6/javascript-promises/) that resolves to {value, done} |
| Loop statement | [for...of](http://for..of/) | for await...of |

# JavaScript Async Generators

**Summary**: in this tutorial, you’ll learn about the JavaScript async generators that iterate over data which comes asynchronously.

## **What is an async generator**

An async generator is similar to a regular [generator](https://www.javascripttutorial.net/es6/javascript-generators/) except that its next() method returns a [Promise](https://www.javascripttutorial.net/es6/javascript-promises/). To iterate over an async generator, you use the for await...of statement.

## **Introduction to the JavaScript async generators**

A regular generator is a function that can pause midway and continues from where it paused. See the following example:

function\* sequence(start, end) {

for (let i = start; i <= end; i++) {

yield i;

}

}

let seq = sequence(1, 5);

for (const num of seq) {

console.log(num);

}

Code language: JavaScript (javascript)

The sequence is a generator that returns a number from the start to the end.

An async generator is similar to a regular generator with the following differences:

* The async keyword is placed in front of the function keyword.
* The yield returns a Promise , instead of a value. The Promise is typically a wrapper of an asynchronous operation.

The following illustrates how to convert the generator sequence to the async generator asyncSequence:

async function\* asyncSequence(start, end) {

for (let i = start; i <= end; i++) {

yield new Promise((resolve, reject) => {

setTimeout(() => {

resolve(i);

}, 1000);

});

}

}

Code language: JavaScript (javascript)

Note that we used the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) to simulate an asynchronous operation.

To iterate over the entire async generator, you use the for await...of statement.

Since we only can use the await keyword inside an async function, we wrap the code inside an async [IIFE](https://www.javascripttutorial.net/javascript-immediately-invoked-function-expression-iife/) as follows:

(async () => {

let seq = asyncSequence(1, 5);

for await (let num of seq) {

console.log(num);

}

})();

Code language: JavaScript (javascript)

The code returns a sequence from 1 to 5 after every second:

1

2

3

4

5

Code language: JavaScript (javascript)

The async generators can be very useful when you access a stream of data and want to report the progress like using a progress bar.

# 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>**

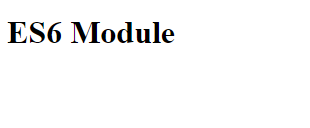
**<script type="module" src="./app.js"></script>**

**</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 import

**Summary**: in this tutorial, you’ll learn how to dynamically import modules by using the function-like import() in ES2020.

## **Introduction to the JavaScript import()**

ES6 introduced the module concept that allows you to develop modular JavaScript code. Suppose you have the following simple HTML document that has one button:

<!DOCTYPE html>

**<html>**

**<head>**

**<title>**Module Dynamic Import**</title>**

**</head>**

**<body>**

**<button id="show">**Show Dialog**</button>**

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

**</body>**

**</html>**

Code language: JavaScript (javascript)

When users click the button, you want to show a dialog. To make the code more organized, you develop a module called dialog.js:

export function show(message) {

alert(message);

}

Code language: JavaScript (javascript)

And use the show() function in the app.js:

import {show} from './dialog.js';

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

show('Hi');

});

Code language: JavaScript (javascript)

Prior to ES2020, it was not possible to dynamically load the dialog.js module when needed. The following will cause an error:

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

import {show} from './dialog.js';

show('Hi');

});

Code language: JavaScript (javascript)

The above code attempts to load the dialog.js module only when the button is clicked.

ES2020 introduced the dynamic import of the module via the function-like import() with the following syntax:

import(moduleSpecifier);

Code language: JavaScript (javascript)

The import() allows you to dynamically import a module when needed. Here is how the import() works:

* The import() accepts a module specifier (moduleSpecifier) that has the same format as the module specifier used for the import statement. In addition, the moduleSpecifier can be an expression that evaluates to a string.
* The import() returns a [Promise](https://www.javascripttutorial.net/es6/javascript-promises/) that will be fulfilled once the module is loaded completely.

To load the dialog.js dynamically, you can use the import() as follows:

let btn = document.querySelector('#show');

btn.addEventListener('click', function() {

import('./dialog.js')

.then(( dialog ) => {

dialog.show();

})

.catch( error => {

*// handle error here*

});

});

Code language: JavaScript (javascript)

Since the import() returns a [Promise](https://www.javascripttutorial.net/es6/javascript-promises/), you can use the [async/await](https://www.javascripttutorial.net/es-next/javascript-async-await/) in the app.js module like this:

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

(async () => {

try {

let dialog = await import('./dialog.js');

dialog.show('Hi')

} catch (error) {

console.log(error);

}

})();

});

Code language: JavaScript (javascript)

## **Some practical use cases of JavaScript import()**

The import() has the following practical use cases:

### **1) Loading module on demand**

Some functionality may not need to be available when applications start. To decrease the loading time, you can place such functionality in modules and use the import() to load them on demand like this:

function eventHandler() {

import('./module1.js')

.then((ns) => {

*// use the module*

ns.func();

})

.catch((error) => {

*// handle error*

});

}

Code language: JavaScript (javascript)

### **2) Loading modules based on conditions**

When placing the import() inside the conditional statement such as [if-else](https://www.javascripttutorial.net/javascript-if-else/), you can load modules based on a specific condition. The following example loads a module that targets a specific platform:

if( isSpecificPlatform() ) {

import('./platform.js')

.then((ns) => {

ns=>f();

});

}

Code language: JavaScript (javascript)

### **3) Computed module specifiers**

The module specifier is an expression that allows you to decide which module to load at runtime.

For example, you can load a module based on the user’s locale to show the message in the user’s specific language:

let lang = `message\_${getUserLocale()}.js`;

import(lang)

.then(...);

Code language: JavaScript (javascript)

## **More on the JavaScript import()**

### **Using object destructuring**

If a module has multiple exports, you can use the [object destructuring](https://www.javascripttutorial.net/es6/javascript-object-destructuring/) to receive the exporting objects. Suppose the dialog.js has two functions:

export function show(message) {

alert(message);

}

export function hide(message) {

console.log('Hide it...');

}

Code language: JavaScript (javascript)

In the app.js, you can use the object destructuring ass follows:

let btn = document.querySelector('#show');

btn.addEventListener('click', function () {

(async () => {

try {

*// use object destructuring*

let {

show,

hide

} = await import('./dialog.js');

*// use the functions*

show('Hi');

hide();

} catch (err) {

console.log(err);

}

})();

});

Code language: JavaScript (javascript)

### **Dynamically loading multiple modules**

To load multiple modules dynamically, you can use the [Promise.all()](https://www.javascripttutorial.net/es6/javascript-promise-all/) method:

Promise.all([

import(module1),

import(module2),

...])

.then(([module1,module2,module3]) => {

*// use the modules*

});

Code language: JavaScript (javascript)

### **Accessing the default export**

If a module has a default export, you can access it using the default keyword. For example:

import(moduleSpecifier)

.then((module) => {

*// access the default export*

console.log(module.default);

});

Code language: JavaScript (javascript)

## **Summary**

* Use the JavaScript import() to dynamically load a module. The import() returns a Promise that will be fulfilled once the module is loaded completely.
* Use the async / await to handle the result of the import().
* Use the Promise.all() method to load multiple modules at once.
* Use the object destructuring to assign variables to the exporting objects of a module.
* Use the default keyword to access the default export.

# JavaScript Top-level await

**Summary**: in this tutorial, you’ll learn about the JavaScript top-level await and its use cases.

## **Introduction to the JavaScript top-level await**

ES2020 introduced the top-level await feature that allows a module to behave like an async function. A [module](https://www.javascripttutorial.net/es6/es6-modules/) that imports the top-level await module will wait for it to load before evaluating its body.

To better understand the top-level await feature, we’ll take an example:

In this example, we’ll have three files: index.html, app.mjs, and user.mjs:

* The index.html uses the app.mjs file.
* The app.mjs imports the user.mjs file.
* The user.mjs fetches the user data in JSON format from an API with the URL endpoint <https://jsonplaceholder.typicode.com/users>

Here’s the index file that uses the app.mjs module:

<!DOCTYPE html>

**<html lang="en">**

**<head>**

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

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<title>**JavaScript Top-Level Await Demo**</title>**

**</head>**

**<body>**

**<div class="container"></div>**

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

**</body>**

**</html>**

Code language: HTML, XML (xml)

The following shows the user.mjs file:

let users;

(async () => {

const url = 'https://jsonplaceholder.typicode.com/users';

const response = await fetch(url);

users = await response.json();

})();

export { users };

Code language: JavaScript (javascript)

The user.mjs module uses the [fetch API](https://www.javascripttutorial.net/javascript-fetch-api/) to get the users in JSON format from an API and export it.

Because we can only use the await keyword inside an async function (before ES2020), we need to wrap the API call inside an immediately invoked async function expression (IIAFE).

The following shows the app.mjs module:

import { users } from './user.mjs';

function render(users) {

if (!users) {

throw 'The user list is not available';

}

const list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join('');

return `<ol>${list}</ol>`;

}

const container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (e) {

container.innerHTML = e;

}

Code language: JavaScript (javascript)

How it works.

First, import users from the user.mjs module:

import { users } from './user.mjs';

Code language: JavaScript (javascript)

Second, create a render() function that renders the user list to an ordered list in HTML format:

function render(users) {

if (!users) {

throw 'The user list is not available.';

}

const list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join('');

return `<ol>${list}</ol>`;

}

Code language: JavaScript (javascript)

Third, add the user list to the HTML element with the class .container:

const container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (e) {

container.innerHTML = e;

}

Code language: JavaScript (javascript)

If you open the index.html, you’ll see the following message:

The user list is not available.

Code language: PHP (php)

The following shows the main flow:

In this flow:

* First, the app.mjs imports the user.mjs module.
* Second, the user.mjs module executes and make an API call.
* Third, while the second step is still on-going, the app.mjs starts using the users data imported from the user.mjs module.

Since the step 2 has not completed, the users variable was undefined. Therefore, you saw the error message on the page.

### **Workaround**

To fix the issue, you can export a Promise from the user.mjs module and wait for the API call to complete before using its result.

The following shows the new version of the user.mjs module:

let users;

export default (async () => {

const url = 'https://jsonplaceholder.typicode.com/users';

const response = await fetch(url);

users = await response.json();

})();

export { users };

Code language: JavaScript (javascript)

In this new version, the user.mjs model exports the users and a Promise as a default export.

In the app.mjs imports the promise and users from the user.mjs file and calls then then() method of the promise as follows:

import promise, { users } from './user.mjs';

function render(users) {

if (!users) {

throw 'The user list is not available.';

}

let list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join(' ');

return `<ol>${list}</ol>`;

}

promise.then(() => {

let container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (error) {

container.innerHTML = error;

}

});

Code language: JavaScript (javascript)

How it works.

First, import promise and users from the user.mjs module:

import promise, { users } from './user.mjs';

Code language: JavaScript (javascript)

Second, call the then() method of the promise and wait for the API call to complete to use its results:

promise.then(() => {

let container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (error) {

container.innerHTML = error;

}

});

Code language: JavaScript (javascript)

Now, if you open the index.html, you’ll see a list of users. However, you need to know the right promise to wait for it when you use the module.

ES2022 introduc in this workaroundes the top-level await module to resolve this issue.

### **Using the top-level await**

First, change the user.mjs to the following:

const url = 'https://jsonplaceholder.typicode.com/users';

const response = await fetch(url);

let users = await response.json();

export { users };

Code language: JavaScript (javascript)

In this module, you can use the await keyword without placing a statement inside an async function.

Second, import the users from the user.mjs module and use it:

import { users } from './user.mjs';

function render(users) {

if (!users) {

throw 'The user list is not available.';

}

let list = users

.map((user) => {

return `<li> ${user.name}(<a href="email:${user.email}">${user.email}</a>)</li>`;

})

.join(' ');

return `<ol>${list}</ol>`;

}

let container = document.querySelector('.container');

try {

container.innerHTML = render(users);

} catch (error) {

container.innerHTML = error;

}

Code language: JavaScript (javascript)

In this case, the app.mjs module will wait for the user.mjs module to complete before executing its body.

## **JavaScript top-level await use cases**

When do you use the top-level await? Here are some use cases.

### **Dynamic dependency pathing**

const words = await import(`/i18n/${navigator.language}`);

Code language: JavaScript (javascript)

In this example, the top-level await allows modules to use runtime values to decide the dependencies, which is useful for the following scenarios:

* Internationalization (i18n)
* Development / production environment splits.

### **Dependency fallback**

In this case, you can use the top-level await to load a module from a server (cdn1). And if it fails, you can load it from a backup server (cdn2):

let module;

try {

module = await import('https://cdn1.com/module');

} catch {

module = await import('https://cdn2.com/module');

}

Code language: JavaScript (javascript)

## **Summary**

* A top-level await module acts like an async function.
* When a module imports a top-level await module, it waits for the top-level await module to complete before evaluating its body.

# JavaScript try…catch

**Summary**: in this tutorial, you will learn how to use the JavaScript try...catch statement to handle exceptions.

## **Introduction to JavaScript try…catch statement**

The following example attempts to call the add() function that doesn’t exist:

let result = add(10, 20);

console.log(result);

console.log('Bye');

Code language: JavaScript (javascript)

And the JavaScript engine issues the following error:

Uncaught TypeError: add is not a function

Code language: JavaScript (javascript)

The error message states that the add is not a [function](https://www.javascripttutorial.net/javascript-function/) and the error type is TypeError.

When the JavaScript engine encounters an error, it issues that error and immediately terminates the execution of the entire script. In the above example, the code execution stops at the first line.

Sometimes, you want to handle the error and continue the execution. To do that, you use the try...catch statement with the following syntax:

try {

*// code may cause error*

} catch(error){

*// code to handle error*

}

Code language: JavaScript (javascript)

In this syntax:

* First, place the code that may cause an error in the try block.
* Second, implement the logic to handle the error in the catch block.

If an error occurs in the try block, the JavaScript engine immediately executes the code in the catch block. Also, the JavaScript engine provides you with an error object that contains detailed information about the error.

Basically, the error object has at least two properties:

* name specifies the error name.
* message explains the error in detail.

If no error occurs in the try block, the JavaScript engine ignores the catch block.

Note that web browsers may add more properties to the error object. For example, Firefox adds filename, lineNumber, and stack properties to the error object.

It’s a good practice to place only the code that may cause an exception in the try block.

The following flowchart illustrates how the try...catch statement works:

## **JavaScript try…catch statement examples**

The following example uses the try...catch statement to handle the error:

try {

let result = add(10, 20);

console.log(result);

} catch (e) {

console.log({ name: e.name, message: e.message });

}

console.log('Bye');

Code language: JavaScript (javascript)

Output

{name: 'TypeError', message: 'add is not a function'}

Bye

Code language: CSS (css)

In this example, we call the add() function and assign the return value to the result variable. Because the add() function doesn’t exist, the JavaScript engine skips the statement that outputs the result to the console:

console.log(result);

Code language: JavaScript (javascript)

And it immediately executes the statement in the catch block that outputs the error name and message:

console.log({ name: e.name, message: e.message });

Code language: CSS (css)

Since we already handled the error, the JavaScript engine continues to execute the last statement:

console.log('Bye');

Code language: JavaScript (javascript)

### **Ingoring the catch block**

The following example defines the add() function that returns the sum of two arguments:

const add = (x, y) => x + y;

try {

let result = add(10, 20);

console.log(result);

} catch (e) {

console.log({ name: e.name, message: e.message });

}

console.log('Bye');

Code language: JavaScript (javascript)

Output:

30

Bye

In this example, no error occurs because the add() function exists. Therefore, the JavaScript engine skips the catch block.

## **The exception identifier**

When an exception occurs in the try block, the exception variable (e) in the catch block store the exception object.

If you don’t want to use the exception variable, you can omit it like this:

try {

*//...*

} catch {

*//...*

}

Code language: JavaScript (javascript)

For example, the following uses the try…catch statement without the exception variable:

const isValidJSON = (str) => {

try {

JSON.parse(str);

return true;

} catch {

return false;

}

};

let valid = isValidJSON(`{"language":"JavaScript"}`);

console.log(valid);

Code language: JavaScript (javascript)

How it works.

First, define the isValidJSON() function that accepts a string and returns true if that string is a valid JSON or false otherwise.

To validate JSON, the isValidJSON() function uses the JSON.parse() method and try...catch statement.

The JSON.parse() method parses a JSON string and returns an object. If the input string is not valid JSON, the JSON.parse() throws an exception.

If no exception occurs, the function returns true in the try block. Otherwise, it returns false in the catch block.

Second, call the isValidJSON() function and pass a JSON string into it:

let valid = isValidJSON(`{"language":"JavaScript"}`);

Code language: JavaScript (javascript)

Since the input string is valid JSON format, the function returns true.

Third, output the result to the console:

console.log(valid);

Code language: JavaScript (javascript)

## **Summary**

* Use the try...catch statement to handle exceptions in JavaScript.
* Place only the code that may cause an exception in the try block.

# JavaScript Throw Exception

**Summary**: in this tutorial, you’ll learn how to use the JavaScript throw statement to throw an exception.

## **Introduction to the JavaScript throw statement**

The throw statement allows you to throw an exception. Here’s the syntax of the throw statement:

throw expression;

Code language: JavaScript (javascript)

In this syntax, the expression specifies the value of the exception. Typically, you’ll use a new instance of the Error class or its subclasses.

When encountering the throw statement, the JavaScript engine stops executing and passes the control to the first [catch](https://www.javascripttutorial.net/javascript-try-catch/) block in the [call stack](https://www.javascripttutorial.net/javascript-call-stack/). If no catch block exists, the JavaScript engine terminates the script.

## **JavaScript throw exception examples**

Let’s take some examples of using the throw statement.

### **1) Using the JavaScript throw statement to throw an exception**

The following example uses the throw statement to throw an exception in a [function](https://www.javascripttutorial.net/javascript-function/):

function add(x, y) {

if (typeof x !== 'number') {

throw 'The first argument must be a number';

}

if (typeof y !== 'number') {

throw 'The second argument must be a number';

}

return x + y;

}

const result = add('a', 10);

console.log(result);

Code language: JavaScript (javascript)

How it works.

First, define the add() function that accepts two arguments and returns the sum of them. The add() function uses the [typeof](https://www.javascripttutorial.net/javascript-typeof/) operator to check the type of each argument and throws an exception if the type is not number.

Second, call the add() function and pass a string and a number into it.

Third, show the result to the console.

The script causes an error because the first argument ("a") is not a number:

Uncaught The first argument must be a number

To handle the exception, you can use the [try...catch](https://www.javascripttutorial.net/javascript-try-catch/) statement. For example:

function add(x, y) {

if (typeof x !== 'number') {

throw 'The first argument must be a number';

}

if (typeof y !== 'number') {

throw 'The second argument must be a number';

}

return x + y;

}

try {

const result = add('a', 10);

console.log(result);

} catch (e) {

console.log(e);

}

Code language: JavaScript (javascript)

Output:

The first argument must be a number

In this example, we place the call to the add() function in a try block. Because the expression in the throw statement is a string, the exception in the catch block is a string as shown in the output.

### **2) Using JavaScript throw statement to throw an instance of the Error class**

In the following example, we throw an instance of the Error class rather than a string in the add() function;

function add(x, y) {

if (typeof x !== 'number') {

throw new Error('The first argument must be a number');

}

if (typeof y !== 'number') {

throw new Error('The second argument must be a number');

}

return x + y;

}

try {

const result = add('a', 10);

console.log(result);

} catch (e) {

console.log(e.name, ':', e.message);

}

Code language: JavaScript (javascript)

Output:

Error : The first argument must be a number

Code language: JavaScript (javascript)

As shown in the output, the exception object in the catch block has the name as Error and the message as the one that we pass to the Error() constructor.

### **3) Using JavaScript throw statement to throw a user-defined exception**

Sometimes, you want to throw a custom error rather than the built-in Error. To do that, you can define a custom error class that extends the Error class and throw a new instance of that class. For example:

First, define the NumberError that extends the Error class:

class NumberError extends Error {

constructor(value) {

super(`"${value}" is not a valid number`);

this.name = 'InvalidNumber';

}

}

Code language: JavaScript (javascript)

The constructor() of the NumberError class accepts a value that you’ll pass into it when creating a new instance of the class.

In the constructor() of the NunberError class, we call the constructor of the Error class via the super and pass a string to it. Also, we override the name of the error to the literal string NumberError. If we don’t do this, the name of the NumberError will be Error.

Second, use the NumberError class in the add() function:

function add(x, y) {

if (typeof x !== 'number') {

throw new NumberError(x);

}

if (typeof y !== 'number') {

throw new NumberError(y);

}

return x + y;

}

Code language: JavaScript (javascript)

In the add() function, we throw an instance of the NumberError class if the argument is not a valid number.

Third, catch the exception thrown by the add() function:

try {

const result = add('a', 10);

console.log(result);

} catch (e) {

console.log(e.name, ':', e.message);

}

Code language: JavaScript (javascript)

Output:

NumberError : "a" is not a valid number

Code language: JavaScript (javascript)

In this example, the exception name is NumberError and the message is the one that we pass into the super() in the constructor() of the NumberError class.

## **Summary**

* Use the JavaScript throw statement to throw a user-define exception.

# Optional catch Binding

**Summary**: in this tutorial, you will learn how to use the optional catch binding in the try...catch statement.

## **Introduction to the optional catch binding**

The [try...catch](https://www.javascripttutorial.net/javascript-try-catch/) statement is used to handle any errors that may occur. Generally, you place the code that may cause an error in the try block and the code that handles the error in the catch block, like this:

try {

*// code that may cause an error*

} catch (error) {

*// code that handles the error*

}

Code language: JavaScript (javascript)

In the catch block, you can access the Error object that contains detailed information on the error.

In practice, you may want to use the try...catch statement to check if a feature is implemented in the web browser. If it isn’t, you want to fall back to a less desirable feature with broader support, for example:

try {

*// check if a feature is implemented*

} catch (error) {

*// fall back to a less desirable feature*

}

Code language: JavaScript (javascript)

In this case, the error object is declared but never used.

ES2019 introduced the optional catch binding that allows you to omit the catch binding and its surrounding parentheses, like this:

try {

} catch {

}

Code language: JavaScript (javascript)

## **Summary**

* Since ES2019, you can omit the catch binding in the try...catch statement.

# JavaScript Execution Context

**Summary**: in this tutorial, you will learn about the JavaScript execution context to deeply understand how JavaScript code gets executed.

## **Introduction to the JavaScript execution context**

Let’s start with the following example:

let x = 10;

function timesTen(a){

return a \* 10;

}

let y = timesTen(x);

console.log(y); *// 100*

Code language: JavaScript (javascript)

In this example:

* First, declare the x variable and initialize its value with 10.
* Second, declare the timesTen() function that accepts an argument and returns a value that is the result of multiplication of the argument with 10.
* Third, call the timesTen() function with the argument as the value of the x variable and store result in the variable y.
* Finally, output the variable y to the [Console](https://www.javascripttutorial.net/web-development-tools/).

Behind the scene, JavaScript does many things. in this tutorial, you will focus on execution contexts.

When the JavaScript engine executes the JavaScript code, it creates execution contexts.

Each execution context has two phases: the creation phase and the execution phase.

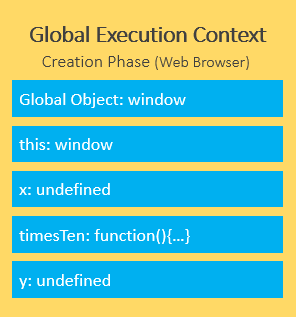
## **The creation phase**

When the JavaScript engine executes a script for the first time, it creates the global execution context. During this phase, the JavaScript engine performs the following tasks:

* Create the [global object](https://www.javascripttutorial.net/es-next/javascript-globalthis/) i.e., window in the web browser or global in Node.js.
* Create the this object and bind it to the global object.
* Setup a memory heap for storing [variables](https://www.javascripttutorial.net/javascript-variables/) and [function](https://www.javascripttutorial.net/javascript-function/) references.
* Store the function declarations in the memory heap and variables within the global execution context with the initial values as undefined.

When the JavaScript engine executes the code example above, it does the following in the creation phase:

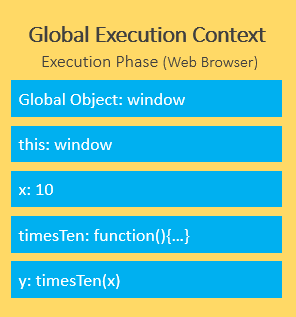
* First, store the variables x and y and function declaration timesTen() in the global execution context.
* Second, initialize the variables x and y to undefined.



After the creation phase, the global execution context moves to the execution phase.

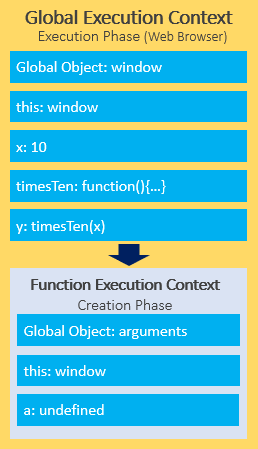
## **The execution phase**

During the execution phase, the JavaScript engine executes the code line by line, assigns the values to variables, and executes the function calls.



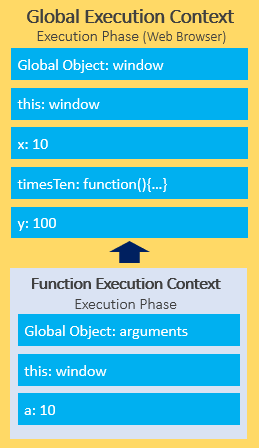
For each function call, the JavaScript engine creates a new **function execution context**.

The function execution context is similar to the global execution context. But instead of creating the global object, the JavaScript engine creates the arguments object that is a reference to all the parameters of the function:



In our example, the function execution context creates the arguments object that references all parameters passed into the function, sets this value to the global object, and initializes the a parameter to undefined.

During the execution phase of the function execution context, the JavaScript engine assigns 10 to the parameter a and returns the result (100) to the global execution context:



To keep track of all the execution contexts, including the global execution context and function execution contexts, the JavaScript engine uses the [call stack](https://www.javascripttutorial.net/javascript-call-stack/), which you will learn in the next tutorial.

In this tutorial, you have learned about the JavaScript execution contexts, including the global execution context and function execution contexts.

# JavaScript Call Stack

**Summary**: in this tutorial, you will learn about the JavaScript Call Stack which is a mechanism to keep track of the function calls.

## **Introduction to JavaScript Call Stack**

JavaScript engine uses a **call stack** to manage [execution contexts](https://www.javascripttutorial.net/javascript-execution-context/): the Global Execution Context and Function Execution Contexts.

The call stack works based on the LIFO principle i.e., last-in-first-out.

When you execute a script, the JavaScript engine creates a Global Execution Context and pushes it on top of the call stack.

Whenever a function is called, the JavaScript engine creates a function execution context for the function, pushes it on top of the call stack, and starts executing the function.

If a function calls another function, the JavaScript engine creates a new function execution context for the function that is being called and pushes it on top of the call stack.

When the current function completes, the JavaScript engine pops it off the call stack and resumes the execution where it left off.

The script will stop when the call stack is empty.

## **JavaScript call stack example**

Let’s start with the following example:

function add(a, b) {

return a + b;

}

function average(a, b) {

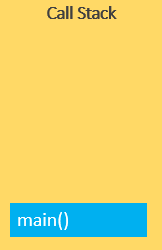
return add(a, b) / 2;

}

let x = average(10, 20);

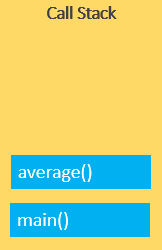
Code language: JavaScript (javascript)

When the JavaScript engine executes this script, it places the global execution context (denoted by main() or global() function on the call stack.



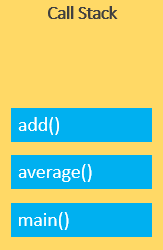
The global execution context enters the creation phase and moves to the execution phase.

The JavaScript engine executes the call to the average(10, 20) function and creates a function execution context for the average() function and pushes it on top of the call stack:

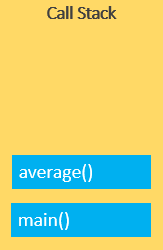


The JavaScript engine starts executing the average() since because the average() function is on the top of the call stack.

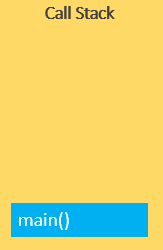
The average() function calls add() function. At this point, the JavaScript engine creates another function execution context for the add() function and places it on the top of the call stack:



JavaScript engine executes the add() function and pops it off the call stack:



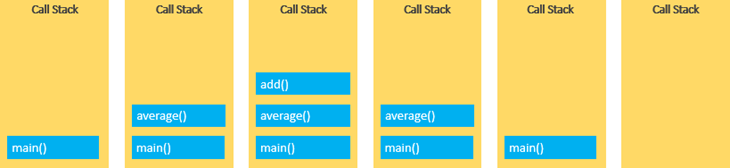
At this point, the average() function is on the top of the call stack, JavaScript engine executes and pops it off the call stack.



Now, the call stack is empty so the script stops executing:



The following picture illustrates the overall status of the Call Stack in all steps:



## **Stack overflow**

The call stack has a fixed size, depending on the implementation of the host environment, either the web browser or Node.js.

If the number of the execution contexts exceeds the size of the stack, a stack overflow error will occur.

For example, when you execute a [recursive function](https://www.javascripttutorial.net/javascript-recursive-function/) that has no exit condition, the JavaScript engine will issue a stack overflow error:

function fn() {

fn();

}

fn(); *// stack overflow*

Code language: JavaScript (javascript)

## **Asynchronous JavaScript**

JavaScript is the single-threaded programming language. This means that the JavaScript engine has only one call stack. Therefore, it only can do one thing at a time.

When executing a script, the JavaScript engine executes code from top to bottom, line by line. In other words, it is synchronous.

Asynchronous means the JavaScript engine can execute other tasks while waiting for another task to complete. For example, the JavaScript engine can:

* Request for a data from a remote server.
* Display a spinner
* When the data is available, display it on the webpage.

To do this, the JavaScript engine uses an event loop, which will be covered in the next tutorial.

## **Summary**

* JavaScript engine uses a call stack to manage execution contexts.
* The call stack use the stack data structure that works based on the LIFO (last-in first-out) principle.

# JavaScript Event Loop

**Summary**: in this tutorial, you’ll learn about the event loop in JavaScript and how JavaScript achieves the concurrency model based on the event loop.

## **JavaScript single-threaded model**

JavaScript is a single-threaded programming language. This means that JavaScript can do only one thing at a single point in time.

The JavaScript engine executes a script from the top of the file and works its way down. It creates the [execution contexts](https://www.javascripttutorial.net/javascript-execution-context/), pushes, and pops functions onto and off the [call stack](https://www.javascripttutorial.net/javascript-call-stack/) in the execution phase.

If a function takes a long time to execute, you cannot interact with the web browser during the function’s execution because the page hangs.

A function that takes a long time to complete is called a blocking function. Technically, a blocking function blocks all the interactions on the webpage, such as mouse click.

An example of a blocking function is a function that calls an API from a remote server.

The following example uses a big loop to simulate a blocking function:

function task(message) {

*// emulate time consuming task*

let n = 10000000000;

while (n > 0){

n--;

}

console.log(message);

}

console.log('Start script...');

task('Call an API');

console.log('Done!');

Code language: JavaScript (javascript)

In this example, we have a big [while](https://www.javascripttutorial.net/javascript-while-loop/) loop inside the task() function that emulates a time-consuming task. The task() function is a blocking function.

The script hangs for a few seconds (depending on how fast the computer is) and issues the following output:

Start script...

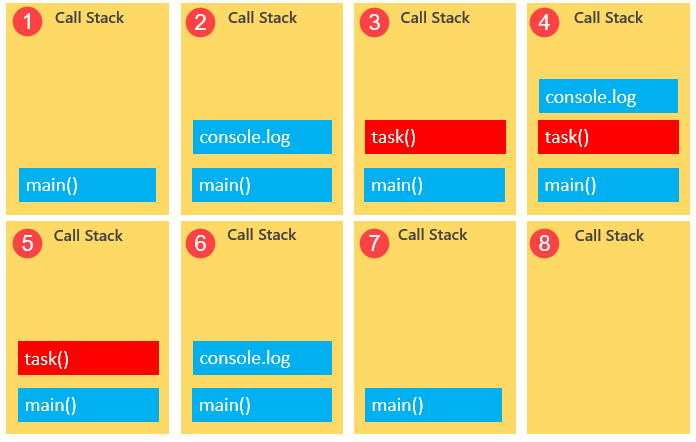
Download a file.

Done!

To execute the script, the JavaScript engine places the first call console.log() on top of the call stack and executes it. Then, it places the task() function on top of the call stack and executes the function.

However, it’ll take a while to complete the task() function. Therefore, you’ll see the message 'Download a file.' a little time later. After the task() function completes, the JavaScript engine pops it off the call stack.

Finally, the JavaScript engine places the last call to the console.log('Done!') function and executes it, which will be very fast.



## **Callbacks to the rescue**

To prevent a blocking function from blocking other activities, you typically put it in a [callback function](https://www.javascripttutorial.net/javascript-callback/) for execution later. For example:

console.log('Start script...');

setTimeout(() => {

task('Download a file.');

}, 1000);

console.log('Done!');

Code language: JavaScript (javascript)

In this example, you’ll see the message 'Start script...' and 'Done!' immediately. And after that, you’ll see the message 'Download a file'.

Here’s the output:

Start script...

Done!

Download a file.

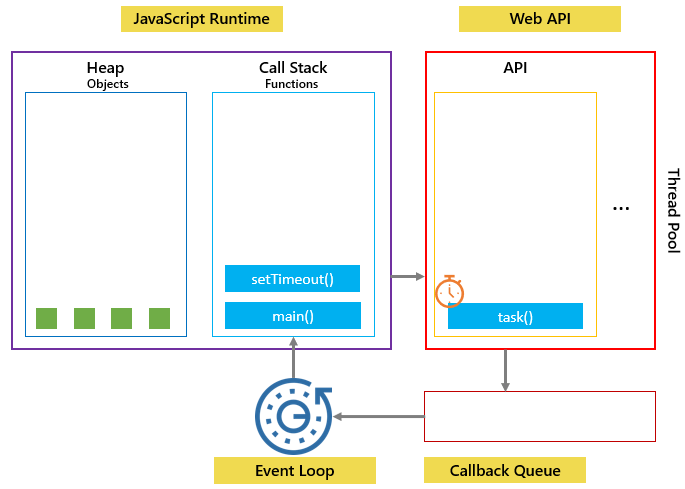
As mentioned earlier, the JavaScript engine can do only one thing at a time. However, it’s more precise to say that the JavaScript runtime can do one thing at a time.

The web browser also has other components, not just the JavaScript engine.

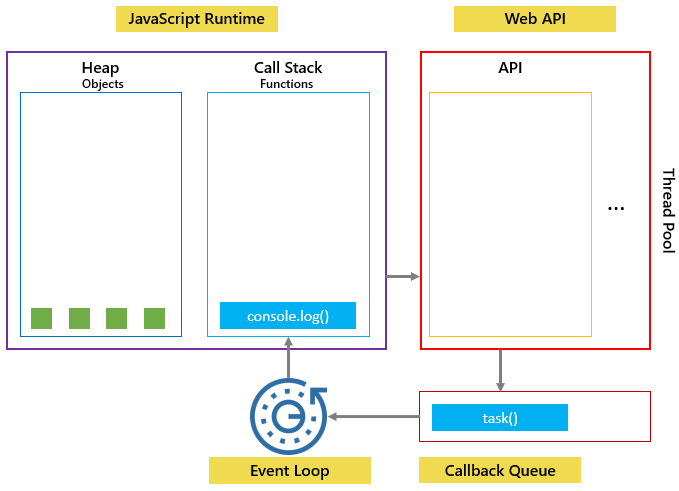
When you call the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function, make a [fetch request](https://www.javascripttutorial.net/javascript-fetch-api/), or click a button, the web browser can do these activities concurrently and asynchronously.

The [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/), fetch requests, and [DOM](https://www.javascripttutorial.net/javascript-dom/) events are parts of the [Web APIs](https://www.javascripttutorial.net/web-apis/) of the web browser.

In our example, when calling the setTimeout() function, the JavaScript engine places it on the call stack, and the Web API creates a timer that expires in 1 second.

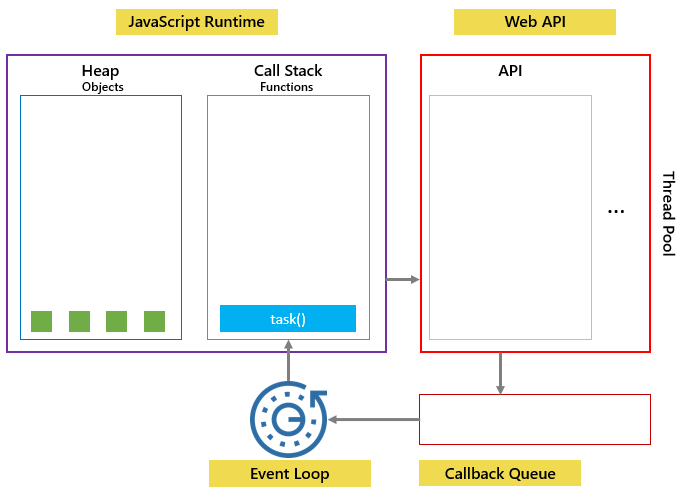


Then JavaScript engine place the task() function is into a queue called a callback queue or a task queue:



The event loop is a constantly running process that monitors both the callback queue and the call stack.

If the call stack is not empty, the event loop waits until it is empty and places the next function from the callback queue to the call stack. If the callback queue is empty, nothing will happen:



See another example:

console.log('Hi!');

setTimeout(() => {

console.log('Execute immediately.');

}, 0);

console.log('Bye!');

Code language: JavaScript (javascript)

In this example, the timeout is 0 second, so the message 'Execute immediately.' should appear before the message 'Bye!'. However, it doesn’t work like that.

The JavaScript engine places the following function call on the callback queue and executes it when the call stack is empty. In other words, the JavaScript engine executes it after the console.log('Bye!').

console.log('Execute immediately.');

Code language: JavaScript (javascript)

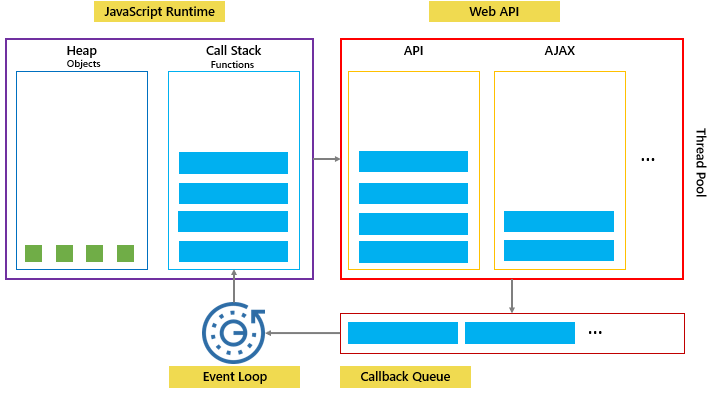
Here’s the output:

Hi!

Bye!

Execute immediately.

The following picture illustrates JavaScript runtime, Web API, Call stack, and Event loop:



In this tutorial, you have learned about the JavaScript event loop, a constantly running process that coordinates the tasks between the call stack and callback queue to achieve concurrency.

# JavaScript Hoisting

**Summary**: in this tutorial, you’ll learn about JavaScript hoisting and how it works under the hood.

## **Introduction to the JavaScript hoisting**

When the JavaScript engine executes the JavaScript code, it creates the [global execution context](https://www.javascripttutorial.net/javascript-execution-context/). The global execution context has two phases: creation and execution.

During the creation phase, the JavaScript engine moves the variable and function declarations to the top of your code. This feature is known as hoisting in JavaScript.

## **Variable hoisting**

Variable hoisting means the JavaScript engine moves the [variable declarations](https://www.javascripttutorial.net/javascript-variable-scope/) to the top of the script. For example, the following example declares the counter variable and initialize its value to 1:

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

var counter = 1;

Code language: JavaScript (javascript)

In this example, we reference the counter variable before the declaration.

However, the first line of code doesn’t cause an error. The reason is that the JavaScript engine moves the variable declaration to the top of the script.

Technically, the code looks like the following in the execution phase:

var counter;

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

counter = 1;

Code language: JavaScript (javascript)

During the creation phase of the global execution context, the JavaScript engine places the variable counter in the memory and initializes its value to [undefined](https://www.javascripttutorial.net/javascript-undefined/).

### **The let keyword**

The following declares the variable counter with the [let](https://www.javascripttutorial.net/es6/javascript-let/) keyword:

console.log(counter);

let counter = 1;

Code language: JavaScript (javascript)

The JavaScript issues the following error:

"ReferenceError: Cannot access 'counter' before initialization

The error message explains that the counter variable is already in the heap memory. However, it hasn’t been initialized.

Behind the scenes, the JavaScript engine hoists the variable declarations that use the let keyword. However, it doesn’t initialize the let variables.

Notice that if you access a variable that doesn’t exist, the JavaScript will throw a different error:

console.log(alien);

let counter = 1;

Code language: JavaScript (javascript)

Here is the error:

"ReferenceError: alien is not defined

## **Function hoisting**

Like variables, the JavaScript engine also hoists the function declarations. This means that the JavaScript engine also moves the function declarations to the top of the script.

For example:

let x = 20,

y = 10;

let result = add(x, y);

console.log(result);

function add(a, b) {

return a + b;

}

Code language: JavaScript (javascript)

In this example, we called the add() function before defining it. The above code is equivalent to the following:

function add(a, b){

return a + b;

}

let x = 20,

y = 10;

let result = add(x,y);

console.log(result);

Code language: JavaScript (javascript)

During the creation phase of the execution context, the JavaScript engine places the add() function declaration in the heap memory. To be precise, the JavaScript engine creates an object of the [Function](https://www.javascripttutorial.net/javascript-function-type/) type and a function reference called add that refers to the function object.

### **Function expressions**

The following example changes the add from a regular function to a function expression:

let x = 20,

y = 10;

let result = add(x,y);

console.log(result);

var add = function(x, y) {

return x + y;

}

Code language: JavaScript (javascript)

If you execute the code, the following error will occur:

"TypeError: add is not a function

During the creation phase of the global execution context, the JavaScript engine creates the add variable in the memory and initializes its value to undefined.

When executing the following code, the add is undefined, hence, it isn’t a function:

let result = add(x,y);

Code language: JavaScript (javascript)

The add variable is assigned to an [anonymous function](https://www.javascripttutorial.net/javascript-anonymous-functions/) only during the execution phase of the global execution context.

### **Arrow functions**

The following example changes the add function expression to the [arrow function](https://www.javascripttutorial.net/es6/javascript-arrow-function/):

let x = 20,

y = 10;

let result = add(x,y);

console.log(result);

var add = (x, y) => x + y;

Code language: JavaScript (javascript)

The code also issues the same error as the function expression example because arrow functions are syntactic sugar for defining function expressions.

"TypeError: add is not a function

Similar to the functions expressions, the arrow functions aren’t hoisted.

## **Summary**

* JavaScript hoisting occurs during the creation phase of the execution context that moves the variable and function declarations to the top of the script.
* The JavaScript engine hoists the variables declared using the let keyword, but it doesn’t initialize them as the variables declared with the var keyword.
* The JavaScript engine doesn’t hoist the function expressions and arrow functions.

# JavaScript Variable Scopes

**Summary**: in this tutorial, you will learn about the JavaScript variable scope that determines the visibility and accessibility of variables.

## **What is variable scope**

Scope determines the visibility and accessibility of a [variable](https://www.javascripttutorial.net/javascript-variables/). JavaScript has three scopes:

* The global scope
* Local scope
* Block scope (started from ES6)

## **The global scope**

When the JavaScript engine executes a script, it creates a global execution context.

Also, it also assigns variables that you declare outside of functions to the [global execution context](https://www.javascripttutorial.net/javascript-execution-context/). These variables are in the global scope. They are also known as global variables.

See the following example:

var message = 'Hi';

Code language: JavaScript (javascript)

The variable message is global-scoped. It can be accessible everywhere in the script.



## **Local scope**

The variables that you declare inside a function are local to the function. They are called local variables. For example:

var message = 'Hi';

function say() {

var message = 'Hello';

console.log('message');

}

say();

console.log(message);

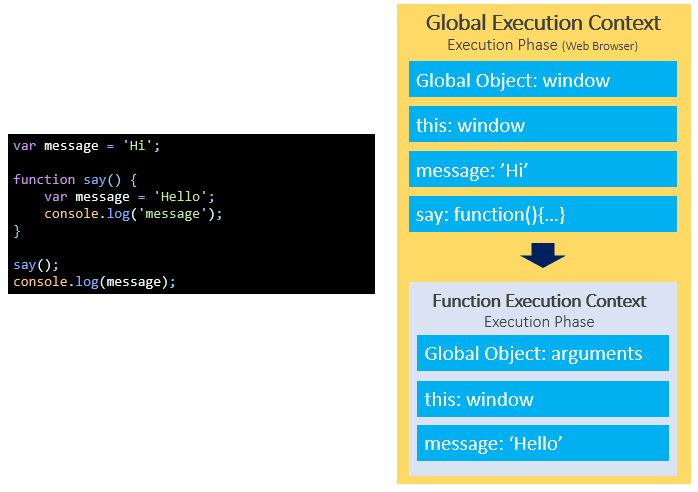
Code language: JavaScript (javascript)

Output:

Hello

Hi

When the JavaScript engine executes the say() function, it creates a function execution context. The variable message declared inside the say() function is bound to the function execution context of the function, not the global execution context.



## **Scope chain**

Consider the following example:

var message = 'Hi';

function say() {

console.log(message);

}

say();

Code language: JavaScript (javascript)

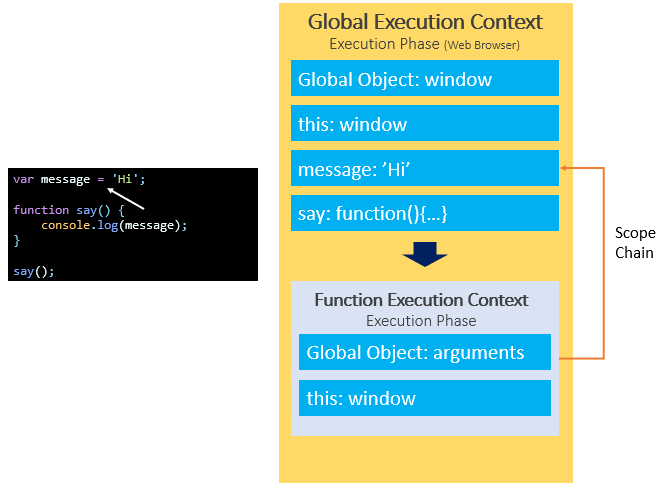
Output:

Hi

In this example, we reference the variable message inside the say() function. Behind the scenes, JavaScript performs the following:

* Look up the variable message in the current context (function execution context) of the say() function. It cannot find any.
* Find the variable message in the outer execution context which is the global execution context. It finds the variable message.

The way that JavaScript resolves a variable is by looking at it in its current scope, if it cannot find the variable, it goes up to the outer scope, which is called the scope chain.



### **More scope chain example**

Consider the following example:

var y = 20;

function bar() {

var y = 200;

function baz() {

console.log(y);

}

baz();

}

bar();

Code language: JavaScript (javascript)

Output:

200

In this example:

* First, the JavaScript engine finds the variable y in the scope of the baz() function. It cannot find any. So it goes out of this scope.
* Then, the JavaScript engine finds the variable y in the bar() function. It can find the variable y in the scope of the bar() function so it stops searching.

## **Global variable leaks: the weird part of JavaScript**

See the following example:

function getCounter() {

counter = 10;

return counter;

}

console.log(getCounter());

Code language: JavaScript (javascript)

Output:

10

In this example, we assigned 10 to the counter variable without the var, let, or const keyword and then returned it.

Outside the function, we called the getCounter() function and showed the result in the console.

This issue is known as the leaks of the global variables.

Under the hood, the JavaScript engine first looks up the counter variable in the local scope of the getCounter() function. Because there is no var, let, or const keyword, the counter variable is not available in the local scope. It hasn’t been created.

Then, the JavaScript engine follows the scope chain and looks up the counter variable in the global scope. The global scope also doesn’t have the counter variable, so the JavaScript engine creates the counter variable in the global scope.

To fix this “weird” behavior, you use the 'use strict' at the top of the script or at the top of the function:

'use strict'

function getCounter() {

counter = 10;

return counter;

}

console.log(getCounter());

Code language: JavaScript (javascript)

Now, the code throws an error:

ReferenceError: counter is not defined

Code language: JavaScript (javascript)

The following shows how to use the 'use strict' in the function:

function getCounter() {

'use strict'

counter = 10;

return counter;

}

console.log(getCounter());

Code language: JavaScript (javascript)

## **Block scope**

ES6 provides the [let](https://www.javascripttutorial.net/es6/javascript-let/) and [const](https://www.javascripttutorial.net/es6/javascript-const/) keywords that allow you to declare variables in block scope.

Generally, whenever you see curly brackets {}, it is a block. It can be the area within the [if](https://www.javascripttutorial.net/javascript-if-else/), else, [switch](https://www.javascripttutorial.net/javascript-switch-case/) conditions or [for](https://www.javascripttutorial.net/javascript-for-loop/), [do while](https://www.javascripttutorial.net/javascript-do-while/), and [while](https://www.javascripttutorial.net/javascript-while-loop/) loops.

See the following example:

function say(message) {

if(!message) {

let greeting = 'Hello'; *// block scope*

console.log(greeting);

}

*// say it again ?*

console.log(greeting); *// ReferenceError*

}

say();

Code language: JavaScript (javascript)

In this example, we reference the variable greeting outside the if block that results in an error.

In this tutorial, you have learned about the JavaScript variable scopes including function scope, global scope, and block scope.