# var, let, and const in JavaScript – the Differences Between These Keywords Explained

In JavaScript, you can declare variables with the var, let and keywords. But what are the differences between them? That's what I'll explain in this tutorial.

I have a [video version of this topic](https://youtu.be/Gd_JG3e1g4A) you can check out as well. 😇

If you're just starting out using JavaScript, a few things you may hear about these keywords are:

* var and let create variables that can be reassigned another value.
* const creates "constant" variables that cannot be reassigned another value.
* developers shouldn't use var anymore. They should use let or const instead.
* if you're not going to change the value of a variable, it is good practice to use const.

The first two points are likely pretty self-explanatory. But what about why we shouldn't use var, or when to use let vs const? As we go through this tutorial, hopefully this will all make sense to you.

## **var** vs **let** vs **const** – What's the Difference?

To analyze the differences between these keywords, I'll be using three factors:

* Scope of variables
* Redeclaration and reassignment
* Hoisting

Let's start by looking at how these factors apply to variables declared with var.

## How to Declare Variables with **var** in JavaScript

### The scope of variables declared with **var**

Variables declared with var can have a **global** or **local** scope. Global scope is for variables declared outside functions, while local scope is for variables declared inside functions.

Let's see some examples, starting from global scope:

var number = 50

function print() {

var square = number \* number

console.log(square)

}

console.log(number) // 50

print() // 2500

The number variable has a global scope – it's declared outside functions in the global space – so you can access it everywhere (inside and outside functions).

Let's see an example of local scope:

function print() {

var number = 50

var square = number \* number

console.log(square)

}

print() // 2500

console.log(number)

// ReferenceError: number is not defined

Here, we declared the number variable in the function print, so it has a local scope. This means that the variable can only be accessed inside that function. Any attempt to access the variable outside the function where it was declared will result in a **variable is not defined** reference error.

### How to redeclare and reassign variables declared with **var**

Variables declared with var can be redeclared and reassigned. I'll explain what I mean with examples.

Here's how to declare a variable with var:

var number = 50

You have the var keyword, the name of the variable number, and an initial value **50**. If an initial value is not provided, the default value will be **undefined**:

var number

console.log(number) // undefined

The var keyword allows for redeclaration. Here's an example:

var number = 50

console.log(number) // 50

var number = 100

console.log(number) // 100

As you can see, we have redeclared the variable number using the var keyword and an initial value of **100**.

The var keyword also allows for reassignment. In the code var number = 50, we assigned the **50** value to number. We can reassign another value anywhere in the code since it was declared with var. Here's what I mean:

var number = 50

console.log(number) // 50

number = 100

console.log(number) // 100

number = 200

console.log(number) // 200

Here, we're not redeclaring – rather, we're reassigning. After declaring the first time with an initial value of **50**, we reassign a new value of **100** and later on with a new value of **200**.

### How to hoist variables declared with **var**

Variables declared with var are hoisted to the top of their global or local scope, which makes them accessible before the line they are declared. Here's an example:

console.log(number) // undefined

var number = 50

console.log(number) // 50

The number variable here has a global scope. Since it is declared with var, the variable is hoisted. This means that we can access the variable before the line where it was declared without errors.

But the variable is hoisted with a default value of **undefined**. So that's the value returned from the variable (until the line where the variable is declared with an initial value gets executed).

Let's see a local scope example:

function print() {

var square1 = number \* number

console.log(square1)

var number = 50

var square2 = number \* number

console.log(square2)

}

print()

// NaN

// 2500

In the print function, number has a local scope. Due to hoisting, we can access the number variable before the line of declaration.

As we see in square1, we assign **number \* number**. Since number is hoisted with a default value of **undefined**, square1 will be **undefined \* undefined** which results in **NaN**.

After the line of declaration with an initial value is executed, number will have a value of **50**. So in square2, **number \* number** will be **50 \* 50** which results in **2500**.

## **How to Declare Variables with let in JavaScript**

### The scope of variables declared with **let**

Variables declared with let can have a **global**, **local**, or **block scope**. Block scope is for variables declared in a block. A block in JavaScript involves opening and closing curly braces:

{

// a block

}

You can find blocks in ***if, loop, switch, and a couple of other statements***. Any variables declared in such blocks with the let keyword will have a block scope. Also, you can't access these variables outside the block.

Here's an example showing a global, local, and block scope:

let number = 50

function print() {

let square = number \* number

if (number < 60) {

var largerNumber = 80

let anotherLargerNumber = 100

console.log(square)

}

console.log(largerNumber)

console.log(anotherLargerNumber)

}

print()

// 2500

// 80

// ReferenceError: anotherLargerNumber is not defined

In this example, we have a global scope variable number and a local scope variable square. There's also block scope variable anotherLargerNumber because it is declared with let in a block.

largerNumber, on the other hand – though declared in a block – does not have a block scope because it is declared with var. So largerNumber has a local scope as it is declared in the function print.

We can access number everywhere. We can only access square and largerNumber in the function because they have local scope. But accessing anotherLargerNumber outside the block throws an **anotherLargerNumber is not defined** error.

### **How to redeclare and reassign variables declared with let**

Just like var, variables declared with let can be reassigned to other values, but they cannot be redeclared. Let's see a reassignment example:

let number = 50

console.log(number) // 50

number = 100

console.log(number) // 100

Here, we reassigned another value **100** after the initial declaration of **50**.

But redeclaring a variable with let will throw an error:

let number = 50

let number = 100// SyntaxError: Identifier 'number' has already been declared

You see we get a syntax error: **Identifier 'number' has already been declared**.

### How to hoist variables declared with **let**

Variables declared with let are hoisted to the top of their global, local, or block scope, but their hoisting is a little different from the one with var.

var variables are hoisted with a default value of **undefined**, which makes them accessible before their line of declaration (as we've seen above).

But, let variables are hoisted without a default initialization. So when you try to access such variables, instead of getting **undefined**, or **variable is not defined** error, you get **cannot access variable before initialization**.

Let's see an example:

console.log(number) // ReferenceError: Cannot access 'number' before initialization

let number = 50

Here, we have a global variable, number declared with let. By trying to access this variable before the line of declaration, we get **ReferenceError: Cannot access 'number' before initialization**.

Here's another example with a local scope variable:

function print() {

let square = number \* number

let number = 50

}

print() // ReferenceError: Cannot access 'number' before initialization

Here we have a local scope variable, number, declared with let. By accessing it before the line of declaration again, we get the **cannot access 'number' before initialization** reference error

## **How to Declare Variables with const in JavaScript**

### The scope of variables declared with **const**

Variables declared with const are similar to let in regards to **scope**. Such variables can have a **global**, **local**, or **block** scope.

Here is an example:

const number = 50

function print() {

const square = number \* number

if (number < 60) {

var largerNumber = 80

const anotherLargerNumber = 100

console.log(square)

}

console.log(largerNumber)

console.log(anotherLargerNumber)

}

print()

// 2500

// 80

// ReferenceError: anotherLargerNumber is not defined

This is from our previous example, but I've replaced let with const. As you can see here, the number variable has a global scope, square has a local scope (declared in the print function), and anotherLargeNumber has a block scope (declared with const).

There's also largeNumber, declared in a block. But because it is with var, the variable only has a local scope. Therefore, it can be accessed outside the block.

Because anotherLargeNumber has a block scope, accessing it outside the block throws an **anotherLargerNumber is not defined**.

### How to redeclare and reassign variables declared with **const**

In this regard, const is different from var and let. const is used for declaring **constant** variables – which are variables with values that cannot be changed. So such variables cannot be redeclared, and neither can they be reassigned to other values. Attempting such would throw an error.

Let's see an example with redeclaration:

const number = 50

const number = 100 // SyntaxError: Identifier 'number' has already been declared

Here, you can see the **Identifier has already been declared** syntax error.

Now, let's see an example with reassignment:

const number = 50

number = 100 // TypeError: Assignment to constant variable

Here, you can see the **Assignment to constant variable** type error.

### How to hoist variables declared with **const**

Variables declared with const, just like let, are hoisted to the top of their global, local, or block scope – but without a default initialization.

var variables, as you've seen earlier, are hoisted with a default value of **undefined** so they can be accessed before declaration without errors. Accessing a variable declared with const before the line of declaration will throw a **cannot access variable before initialization** error.

Let's see an example:

console.log(number) // ReferenceError: Cannot access 'number' before initialization

const number = 50

Here, number is a globally scoped variable declared with const. By trying to access this variable before the line of declaration, we get **ReferenceError: Cannot access 'number' before initialization**. The same will occur if it was a locally scoped variable.

Here's an article to learn more about [Hoisting in JavaScript with let and const – and How it Differs from var](https://www.freecodecamp.org/news/javascript-let-and-const-hoisting/).

## Wrap up

Here's a table summary showing the differences between these keywords:

|  |  |  |  |
| --- | --- | --- | --- |
| KEYWORD | SCOPE | REDECLARATION & REASSIGNMENT | HOISTING |
| var | Global, Local | yes & yes | yes, with default value |
| let | Global, Local, Block | no & yes | yes, without default value |
| const | Global, Local, Block | no & no | yes, without default value |

These factors I've explained, play a role in determining how you declare variables in JavaScript.

If you never want a variable to change, const is the keyword to use.

If you want to reassign values:

* and you want the hoisting behavior, var is the keyword to use
* if you don't want it, let is the keyword for you

The hoisting behavior can cause unexpected bugs in your application. That's why developers are generally advised to avoid var and stick to let and cost.

# When (and why) you should use ES6 arrow functions — and when you shouldn’t

Arrow functions (also called “fat arrow functions”) are undoubtedly one of the more popular features of ES6. They introduced a

new way of writing concise functions.

Here is a function written in ES5 syntax:

function timesTwo(params) {

return params \* 2

}

timesTwo(4); // 8

Now, here is the same function expressed as an arrow function:

var timesTwo = params => params \* 2

timesTwo(4); // 8

It’s much shorter! We are able to omit(to not include) the curly braces and the return statement due to implicit returns (but only if there is no block — more on this below).

It is important to understand how the arrow function behaves differently compared to the regular ES5 functions.

### Variations

One thing you will quickly notice is the variety of syntaxes available in arrow functions. Let’s run through some of the common ones:

#### 1. No parameters

If there are no parameters, you can place an empty parentheses before =&gt;.

( ) => 42

In fact, you don’t even need the parentheses!

\_ => 42

#### 2. Single parameter

With these functions, parentheses are optional:

x => 42 || (x) => 42

#### **3. Multiple parameters**

Parentheses are required for these functions:

(x, y) => 42

#### **4. Statements (as opposed to expressions)**

In its most basic form, a function expression produces a value, while a function statement performs an action.

With the arrow function, it is important to remember that statements need to have curly braces. Once the curly braces are present, you always need to write return as well.

Here is an example of the arrow function used with an if statement:

var feedTheCat = (cat) => {

if (cat === 'hungry') {

return 'Feed the cat';

} else {

return 'Do not feed the cat';

}

}

#### **5. “Block body”**

If your function is in a block, you must also use the explicit return statement:

var addValues = (x, y) => {

return x + y

}

#### **6. Object literals**

If you are returning an object literal, it needs to be wrapped in parentheses. This forces the interpreter to evaluate what is inside the parentheses, and the object literal is returned.

x =>({ y: x })

### Syntactically anonymous

It is important to note that arrow functions are anonymous, which means that they are not named.

This anonymity creates some issues:

1. Harder to debug

When you get an error, you will not be able to trace the name of the function or the exact line number where it occurred.

2. No self-referencing

If your function needs to have a self-reference at any point (e.g. recursion, event handler that needs to unbind), it will not work.

### Main benefit: No binding of ‘this’

In classic function expressions, the this keyword is bound to different values based on the context in which it is called. With arrow functions however, this is lexically bound. It means that it uses this from the code that contains the arrow function.

For example, look at the setTimeout function below:

// ES5

var obj = {

id: 42,

counter: function counter() {

setTimeout(function() {

console.log(this.id);

}.bind(this), 1000);

}

};

In the ES5 example, .bind(this) is required to help pass the this context into the function. Otherwise, by default this would be undefined.

// ES6

var obj = {

id: 42,

counter: function counter() {

setTimeout(() => {

console.log(this.id);

}, 1000);

}

};

ES6 arrow functions can’t be bound to a this keyword, so it will lexically go up a scope, and use the value of this in the scope in which it was defined.

### When you should not use Arrow Functions

After learning a little more about arrow functions, I hope you understand that they do not replace regular functions.

Here are some instances where you probably wouldn’t want to use them:

1. Object methods

When you call cat.jumps, the number of lives does not decrease. It is because this is not bound to anything, and will inherit the value of this from its parent scope.

var cat = {

lives: 9,

jumps: () => {

this.lives--;

}

}

2. Callback functions with dynamic context

If you need your context to be dynamic, arrow functions are not the right choice. Take a look at this event handler below:

var button = document.getElementById('press');

button.addEventListener('click', () => {

this.classList.toggle('on');

});

If we click the button, we would get a TypeError. It is because this is not bound to the button, but instead bound to its parent scope.

3. When it makes your code less readable

It is worth taking into consideration the variety of syntax we covered earlier. With regular functions, people know what to expect. With arrow functions, it may be hard to decipher what you are looking at straightaway.

### When you should use them

Arrow functions shine best with anything that requires this to be bound to the context, and not the function itself.

Despite the fact that they are anonymous, I also like using them with methods such as map and reduce, because I think it makes my code more readable. To me, the pros outweigh the cons.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

JavaScript Array Methods

|  |  |
| --- | --- |
| Array length Array toString() Array pop() Array push() Array shift() Array unshift() | Array join() Array delete() Array concat() Array flat() Array splice() Array slice() |
| The methods are listed in the order they appear in this tutorial page | |

## **1] JavaScript Array length**

The length property returns the length (size) of an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

let size = fruits.length;

## **2] JavaScript Array toString()**

The JavaScript method toString() converts an array to a string of (comma separated) array values.

const fruits = ["Banana", "Orange", "Apple", "Mango"];

console.log(fruits.toString()); // Banana,Orange,Apple,Mango

## **3] JavaScript Array join()**

The **join()** method also joins all array elements into a string.

It behaves just like toString(), but in addition you can specify the separator:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

console.log(fruits.join(" \* ")); // Banana \* Orange \* Apple \* Mango

console.log(fruits.join(" - ")); // Banana - Orange - Apple - Mango

## **Popping and Pushing**

When you work with arrays, it is easy to remove elements and add new elements.

This is what popping and pushing is:

Popping items **out** of an array, or pushing items **into** an array.

## **4] JavaScript Array pop()**

The pop() method removes the last element from an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.pop();

The pop() method returns the value that was "popped out":

const fruits = ["Banana", "Orange", "Apple", "Mango"];

let fruit = fruits.pop();

console.log(fruit); //Mango

## **5] JavaScript Array push()**

The push() method adds a new element to an array (at the end):

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.push("Kiwi");

The push() method returns the new array length:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let length = fruits.push("Kiwi"); //5

## **6] JavaScript Array shift()**

The shift() method removes the first array element and "shifts" all other elements to a lower index.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.shift();

The shift() method returns the value that was "shifted out":

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
let fruit = fruits.shift();

## **7] JavaScript Array unshift()**

The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");

The unshift() method returns the new array length:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.unshift("Lemon");

## **8] Changing Elements**

Array elements are accessed using their **index number**:

Array **indexes** start with 0:

[0] is the first array element  
[1] is the second  
[2] is the third ...

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[0] = "Kiwi";

## **9] JavaScript Array length**

The length property provides an easy way to append a new element to an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits[fruits.length] = "Kiwi";

## **10] JavaScript Array delete()**

### **Warning !**

Array elements can be deleted using the JavaScript operator delete.

Using delete leaves undefined holes in the array.

Use pop() or shift() instead.

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
delete fruits[0];

## **11] Merging (Concatenating) Arrays**

The concat() method creates a new array by merging (concatenating) existing arrays:

Merging two arrays

const myGirls = ["Cecilie", "Lone"];  
const myBoys = ["Emil", "Tobias", "Linus"];  
  
const myChildren = myGirls.concat(myBoys);

The concat() method does not change the existing arrays. It always returns a new array.

The concat() method can take any number of array arguments:

### **(Merging Three Arrays)**

const arr1 = ["Cecilie", "Lone"];  
const arr2 = ["Emil", "Tobias", "Linus"];  
const arr3 = ["Robin", "Morgan"];  
const myChildren = arr1.concat(arr2, arr3);

The concat() method can also take strings as arguments:

### **Merging an Array with Values**

const arr1 = ["Emil", "Tobias", "Linus"];  
const myChildren = arr1.concat("Peter");

## **12] Flattening an Array**

Flattening an array is the process of reducing the dimensionality of an array.

The flat() method creates a new array with sub-array elements concatenated to a specified depth.

const myArr = [[1,2],[3,4],[5,6]];  
const newArr = myArr.flat();

console.log(newArr) // [1, 2, 3, 4, 5, 6]

## **Slicing and Splicing Arrays**

The slice() method slices out a piece of an array.

The splice() method adds new items to an array.

## **13] JavaScript Array slice()**

The slice() method slices out a piece of an array into a new array.

This example slices out a part of an array starting from array element 1 ("Orange"):

## const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"]; const citrus = fruits.slice(1); // ['Orange', 'Lemon', 'Apple', 'Mango']

## **Note**

The slice() method creates a new array.

The slice() method does not remove any elements from the source array.

The slice() method can take two arguments like slice(1, 3).

The method then selects elements from the start argument, and up to (but not including) the end argument.

const fruits = ["Banana", "Orange", "Lemon", "Apple", "Mango"];  
const citrus = fruits.slice(1, 3); // ['Orange', 'Lemon']

const citrus = fruits.slice(0, -2); // ['Banana', 'Orange', 'Lemon']

const citrus = fruits.slice(-1); // ['Mango']

const citrus = fruits.slice(-2); // ['Apple', 'Mango']

## **14] JavaScript Array splice()**

The splice() method can be used to add new items to an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi");

The first parameter (2) defines the position **where** new elements should be **added** (spliced in).

The second parameter (0) defines **how many** elements should be **removed**.

The rest of the parameters ("Lemon", "Kiwi") define the new elements to be **added**.

The splice() method returns an array with the deleted items:

*const* fruits = ["Banana", "Orange", "Apple", "Mango"];

console.log("Original Array:\n" + fruits); // Banana,Orange,Apple,Mango

*let* removed = fruits.splice(2, 2, "Lemon", "Kiwi");

console.log("New Array:\n" + fruits); // Banana,Orange,Apple,Mango

console.log("Removed Items:\n" + removed); // Apple,Mango

## **Using splice() to Remove Elements**

With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:

*const* fruits = ["Banana", "Orange", "Apple", "Mango"];

console.log(fruits.splice(0, 1)); *// ['Banana']*

The first parameter (0) defines the position where new elements should be **added** (spliced in).

The second parameter (1) defines **how many** elements should be **removed**.

The rest of the parameters are omitted. No new elements will be added.

Visit this link for more Array Methods

<https://www.programiz.com/javascript/library/array>

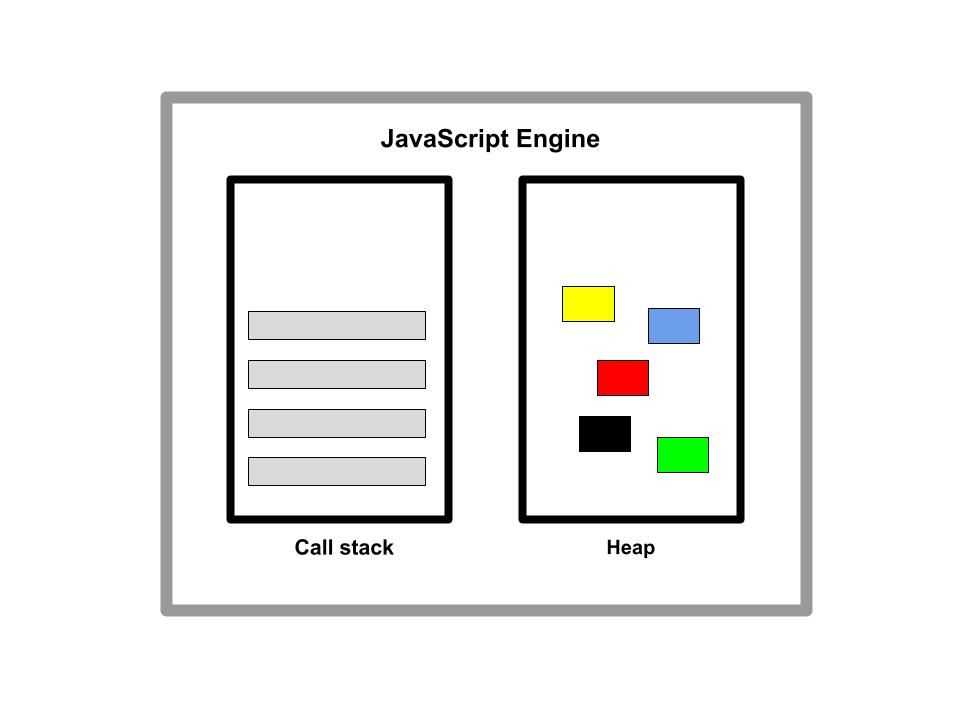
# Object Methods in JavaScript

<https://www.digitalocean.com/community/tutorials/how-to-use-object-methods-in-javascript>

<https://medium.com/geekculture/8-javascript-object-methods-every-developer-should-know-46838e6dc879>

# How Does JavaScript Work Behind the Scenes? JS Engine and Runtime Explained

## **The JavaScript Engine**



JavaScript Engine showing the Call stack and the Heap

The JavaScript engine is simply a computer program that interprets JavaScript code. The engine is responsible for executing the code.

Every major browser has a JavaScript engine that executes JavaScript code. The most popular one is the Google Chrome [V8](https://en.wikipedia.org/wiki/JavaScript_engine) engine. Google’s V8 powers Google Chrome and [Node.js](https://nodejs.org/en/docs), a back-end JavaScript runtime environment used to build server-side applications.

Other major browser engines include:

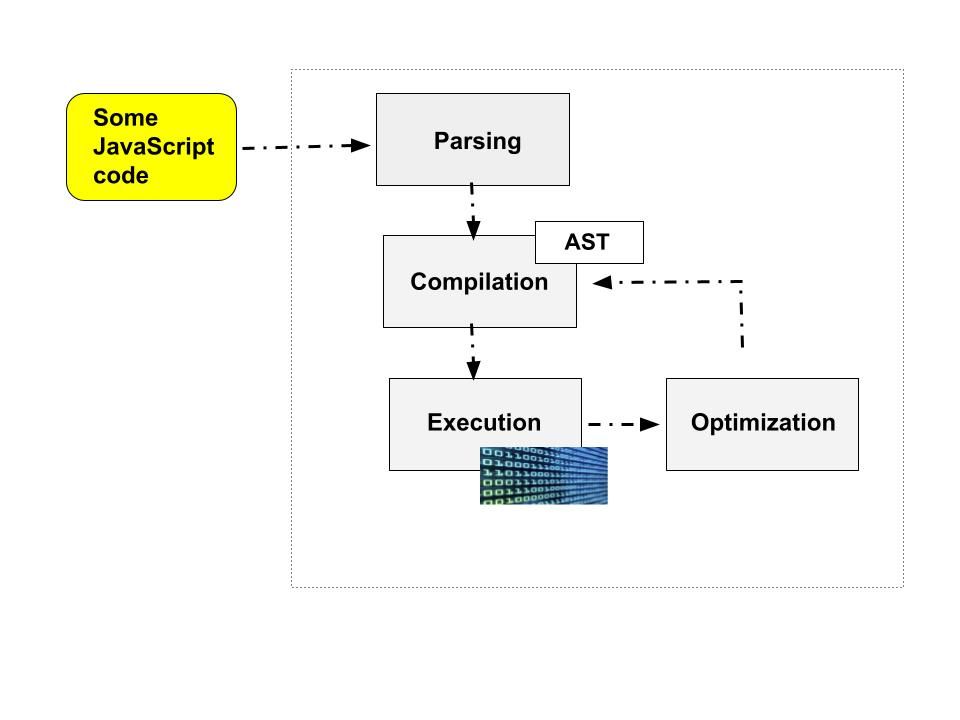
* SpiderMonkey developed by Mozilla for Firefox
* JavaScriptCore which powers the Safari browser
* Chakra which powers Internet Explorer

Any JavaScript engine typically contains a call stack and a heap. The call stack is where the code is executed. The heap is an unstructured memory pool that stores all the objects needed for the application.

Since the computer’s processor only understands binary, 0’s and 1’s, the code has to be translated to 0’s and 1’s.

When a code snippet passes into the engine, the code is initially parsed, that is read. The code is subsequently parsed to a data structure called the abstract syntax tree (AST). The resulting tree is then used to create machine codes.

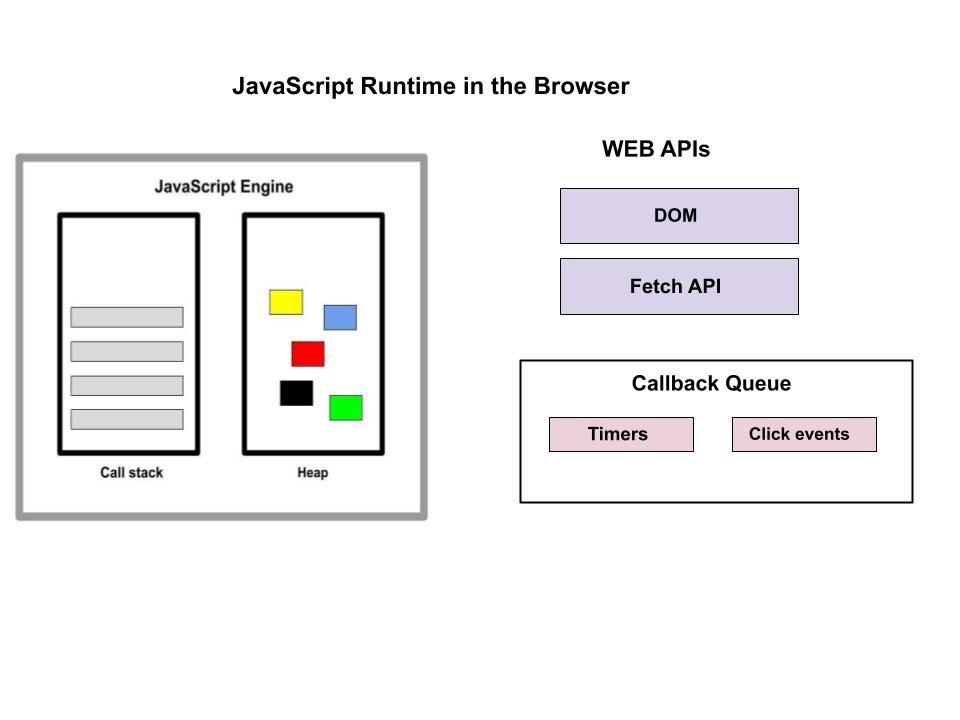
Execution happens in the JavaScript engine call stack using the execution context. This is the environment where JavaScript code is executed.

A diagram illustration showing the JavaScript execution process

## **The JavaScript Runtime**

Think of the JavaScript runtime as the house that encompasses all the components needed to run JavaScript. This house comprises the JavaScript engine, Web APIs, and the callback queue.

Web APIs are functionalities that are provided to the engine but are not part of the JavaScript language. They are accessible to the engine through the browser and help access data or enhance browser functionality. Examples are the Document Object Model (DOM) and Fetch APIs.



A diagram of JavaScript Runtime in the Browser containing the JavaScript Engine, WEB APIs, and the Callback Queue

The callback queue includes callback functions that are ready to be executed. The callback queue ensures that callbacks are executed in the First-In-First-Out (FIFO) method and they get passed into the stack when it’s empty.

The browser runtime and Node.js are examples of runtime environments.

When JavaScript executes within a web browser it is operating within the browser’s runtime environment. The browser runtime environment provides access to the DOM which enables interaction with web page elements, handling events, and manipulating the page structure.

Node.js provides a server-side runtime environment for executing JavaScript outside the browser. Because it executes JavaScript outside the browser, it does not have access to the web APIs. Instead, the Node.js runtime environment replaces it with something called C++ bindings and the thread pool.

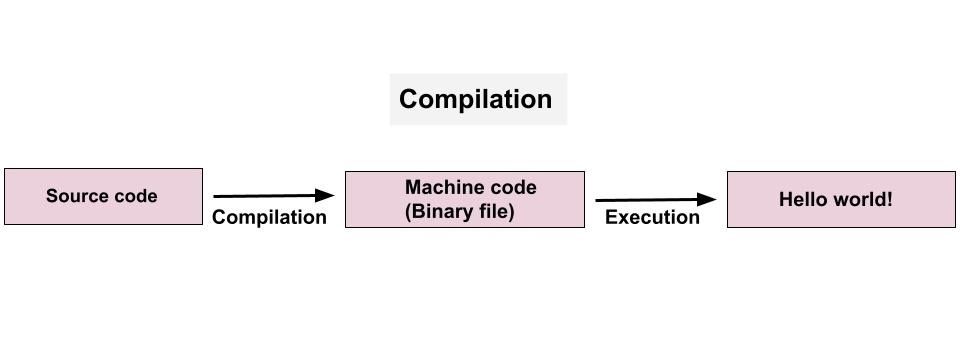
## **JavaScript Optimization Strategies**

Modern JavaScript engines have some optimization strategies put in place to enhance the performance of code execution. These optimizations occur dynamically during the execution process. Let's look at some of these strategies.

### **Just-in-Time compilation**

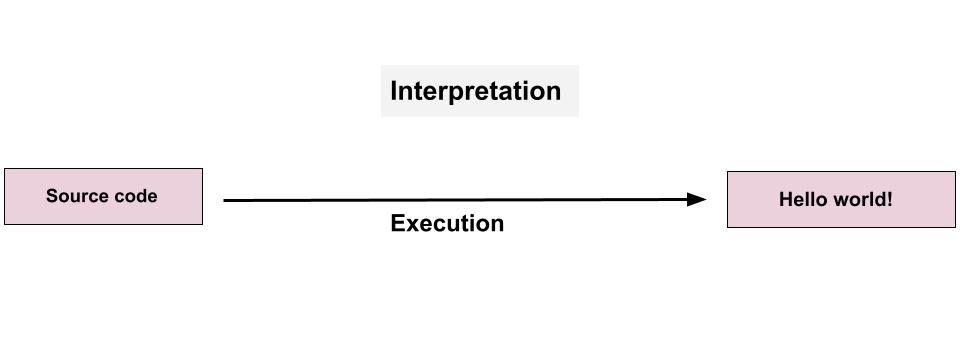
The process that involves the translation of JavaScript code into machine code occurs using compilation and interpretation.

In compilation, the entire source code is converted into machine code at once and written into a binary file to be executed by the computer.



A diagram showing the code compilation process

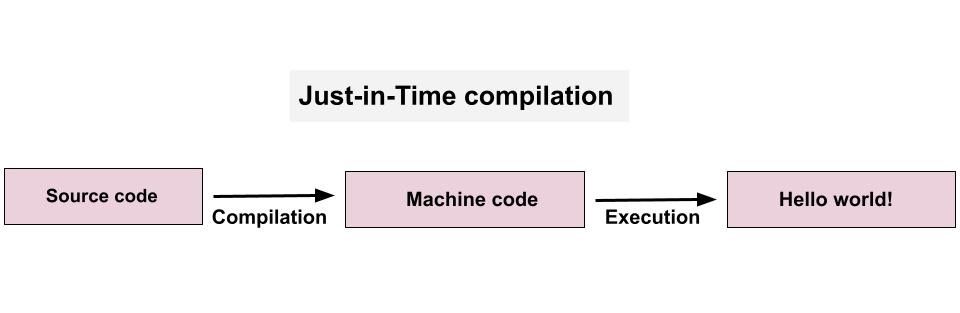
In contrast, during interpretation, the interpreter goes through the source code and interprets it line by line, executing each line as it encounters it.



A diagram showing the code interpretation process

JavaScript used to be an interpreted language, but interpreted languages are slower compared to compiled languages.

In order to optimize the performance of web applications, JavaScript combines both compilation and interpretation. This is called Just-in-Time compilation. This method compiles the entire code into machine code all at once and executes it.



A diagram showing Just-in-Time compilation of code

Just-in-Time compilation involves the same two processes as regular compilation, but here the machine code isn’t written into a binary file. The code is also executed right away after compilation.

This has had a significant impact on the speed of code execution in JavaScript. So hopefully this helps dispel the notion that JavaScript is a purely interpreted language.

To fully optimize JavaScript code, the engine first creates an unoptimized version of the machine code so it can start executing immediately. While that is ongoing, the code is being re-optimized and recompiled in the background of the currently running program execution. This is done multiple times to produce the final, most optimized version.

The process of parsing, compilation, and execution happens in some special thread in the engine that can’t be accessed from the code.

# JavaScript Execution Context – How JS Works Behind the Scenes

Let's look at an example so we can learn more:

var n = 5;

function square(n) {

var ans = n \* n;

return ans;

}

var square1 = square(n);

var square2 = square(8);

console.log(square1)

console.log(square2)

In the above code,

1. n is initialized and assigned a value of 5
2. We defined a function square() that accepts an argument n and returns the square of n.
3. We call the square() function and store the returned value in the square1 variable.
4. We call the square() function and store the returned value in the square2 variable.
5. Finally, it outputs both square1 and square2

Behind the scenes, JavaScript is doing so many things. Let's understand all of it.

## **What is the Execution Context?**

When the JavaScript engine scans a script file, it makes an environment called the **Execution Context**that handles the entire transformation and execution of the code.

During the context runtime, the parser parses the source code and allocates memory for the variables and functions. The source code is generated and gets executed.

There are two types of execution contexts: **global** and **function**. The global execution context is created when a JavaScript script first starts to run, and it represents the global scope in JavaScript. A function execution context is created whenever a function is called, representing the function's local scope.

### **Phases of the JavaScript Execution Context**

There are two phases of JavaScript execution context:

1. **Creation phase**: In this phase, the JavaScript engine creates the execution context and sets up the script's environment. It determines the values of variables and functions and sets up the scope chain for the execution context.
2. **Execution phase**: In this phase, the JavaScript engine executes the code in the execution context. It processes any statements or expressions in the script and evaluates any function calls.

Everything in JS happens inside this execution context. It is divided into two components. One is memory and the other is code. It is important to remember that these phases and components are applicable to both global and functional execution contexts.

### **Creation Phase**



Execution Context

Let's take this simple example once again:

var n = 5;

function square(n) {

var ans = n \* n;

return ans;

}

var square1 = square(n);

var square2 = square(8);

console.log(square1)

console.log(square2)

At the very beginning, the JavaScript engine executes the entire source code, creates a global execution context, and then does the following things:

1. Creates a global object that is**window** in the browser and **global** in NodeJs.
2. Sets up a memory for storing variables and functions.
3. Stores the variables with values as undefined and function references.

This is called the creation phase. Here's a diagram to help explain it:



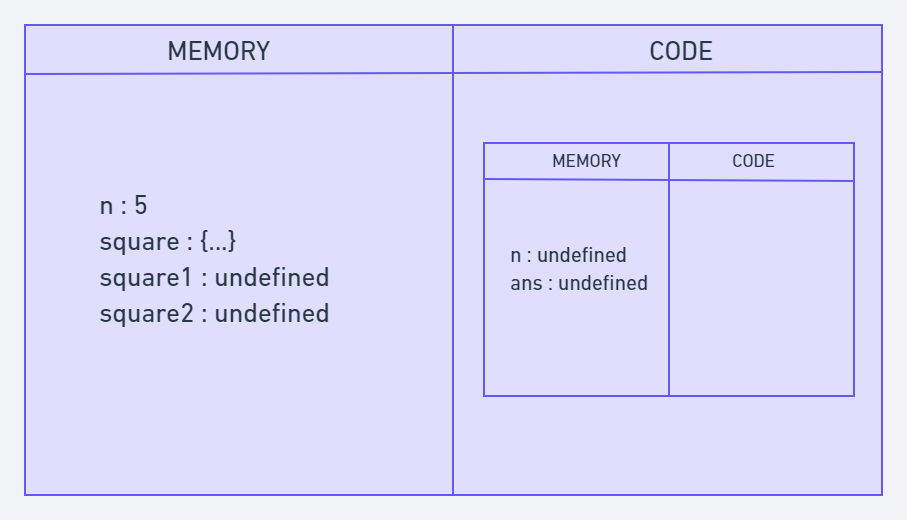
Creation Phase in Execution Context

After this creation phase, the execution context will move to the code execution phase.

### **Execution Phase**

Now, in this phase, it starts going through the entire code line by line from top to bottom. As soon as it encounters **n = 5**, it assigns the value 5 to 'n' in memory. Until now, the value of 'n' was undefined by default.

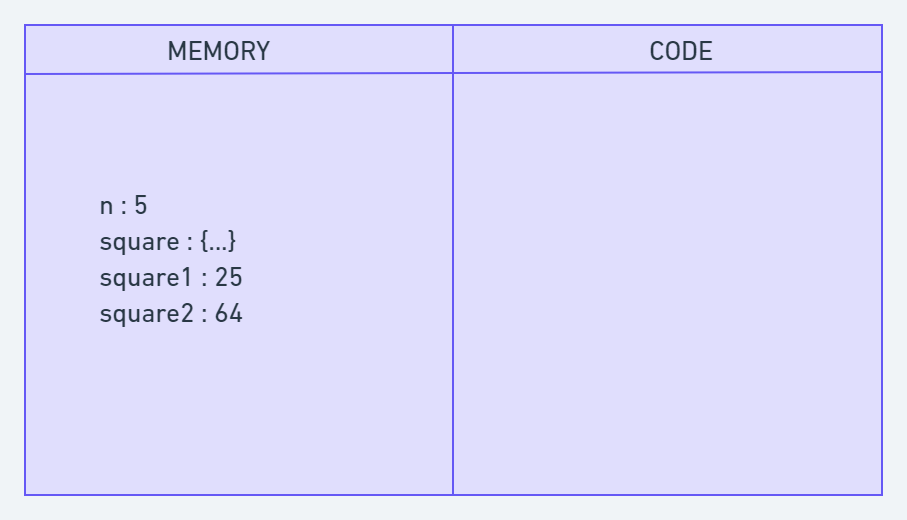
Then we get to the 'square' function. As the function has been allocated in memory, it directly jumps into the line **var square1 = square(n);**. square() will be invoked and JavaScript once again will create a new function execution context.



Code Execution Phase

Once the calculation is done, it assigns the value of square in the 'ans' variable that was undefined before. The function will return the value, and the function execution context will be destroyed.

The returned value from square() will be assigned on square1. This happens for square2 also. Once the entire code execution is done completely, the global context will look like this and it will be destroyed also.



## **What is the Call Stack?**

To keep the track of all the contexts, including global and functional, the JavaScript engine uses a **call stack**. A call stack is also known as an 'Execution Context Stack', 'Runtime Stack', or 'Machine Stack'.

It uses the LIFO principle (Last-In-First-Out). When the engine first starts executing the script, it creates a global context and pushes it on the stack. Whenever a function is invoked, similarly, the JS engine creates a function stack context for the function and pushes it to the top of the call stack and starts executing it.

When execution of the current function is complete, then the JavaScript engine will automatically remove the context from the call stack and it goes back to its parent.

Let's see the following example:

function funcA(m,n) {

return m \* n;

}

function funcB(m,n) {

return funcA(m,n);

}

function getResult(num1, num2) {

return funcB(num1, num2)

}

var res = getResult(5,6);

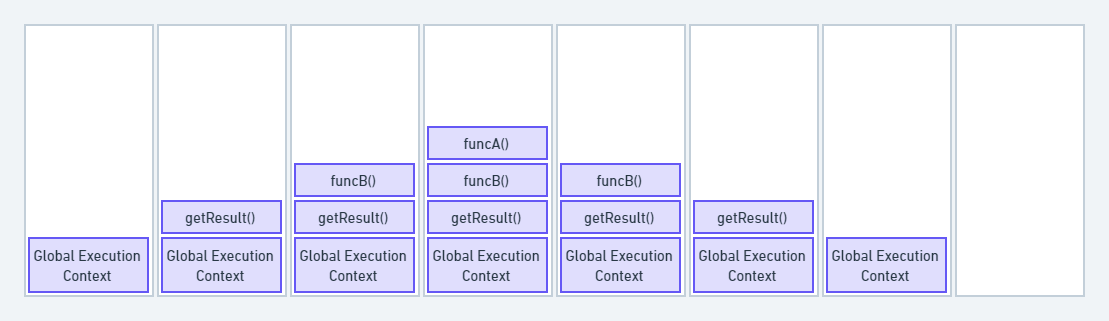
console.log(res); // 30

In this example, the JS engine creates a global execution context that enters the creation phase.

First it allocates memory for funcA, funcB, the getResult function, and the res variable. Then it invokes getResult(), which will be pushed on the call stack.

Then getResult() will call funcB(). At this point, funcB's context will be stored on the top of the stack. Then it will start executing and call another function funcA(). Similarly, funcA's context will be pushed.

Once execution of each function is done, it will be removed from the call stack. The following picture depicts the entire process of the execution:



Call Stack

The call stack has its own fixed size depending on the system or browser. If the number of contexts exceeds the limit, then a stack overflow error will occur. This happens with a recursive function that has no base condition.

function display() {

display();

}

display(); //

Output :- RangeError: Maximum call stack size exceeded

## **Conclusion**

In conclusion, JavaScript execution context is a crucial part of understanding how JavaScript works behind the scenes. It determines the environment in which code is executed and what variables and functions are available to use.

The creation phase includes creating the global and function execution contexts, creating the scope chain, and allocating memories for the variables and functions. During the execution phase, the JavaScript engine executes the code line by line. This includes evaluating and executing statements.

# Scope Chains

# Before we get to closures, we have to have an understanding of scope.

# What is **scope** and why does it matter?

# Scope is the context environment (also known as lexical environment) created when a function is written. This context defines what other data it has access to.

Put another way, scope is about access. Does the function have the ability to look up a variable for execution or manipulation, which variables are visible?

There are two types of scope: local and global. Scope resolution, or finding what variables belong where, starts at the innermost context and proceeds outward until the identifier is found. Let’s start small…

var firstNum = 1;

function number() { var secondNum = 2; return firstNum + secondNum;}

number();

The scope chain is similar to the prototype chain. If a variable or property is not found, it continues up the chain until it is either found or a error is thrown. The function creates a hidden [[scope]] property. This property links innermost scopes to outermost scopes. In this case, number’s scope chain is linked to the global window object (the containing context that holds function number). This is what allows the engine to look outside of function number to find firstNum and secondNum.

For example, let’s take the same function number and change one thing:

*// global scope  - includes firstNum, secondNum, and the function number*

*var* firstNum = 1;

*function* number() {

*// local scope for number - only thirdNum is local to number()*

*// because it was explicitly declared. secondNum is implicitly declared in the global scope.*

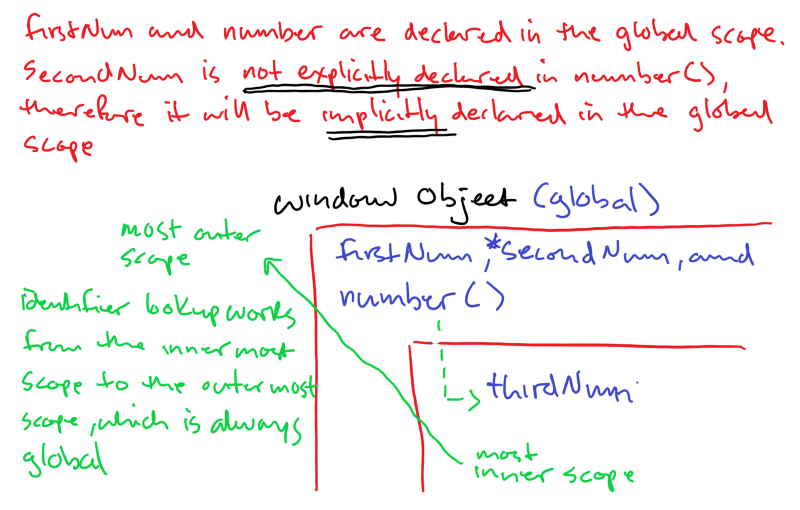
  secondNum = 2;

*var* thirdNum = 3;

  return firstNum + secondNum;

} *// what do we have access to in the global scope?number(); // 3firstNum; // 1secondNum; // 2thirdNum; // Reference Error: thirdNum is not defined*

When speaking of global scope, variable declarations, non-nested function

When speaking of global scope, variable declarations, non-nested function declarations, and function expressions (still considered a variable definition) are considered in the scope of the global window object in the browser. So as we see above, the window object has a properties firstNum, secondNum, and number added to it. If we proceed along the scope chain looking for it, we keep looking until we reach the global context’s variable object. If it’s not in there, then we get the Reference Error.

In a new tab, type "about:blank" in the search bar. A blank page will open and hit cmd-option-i to open dev tools.

Type the code above and remember, shift-enter for a new line!

Now type "window" and explore all the properties on the window object.

Look closely and you will see the properties firstNum, secondNum, and number are all available on the window object.

When we try to access thirdNum outside of where it was declared, we get a Reference Error. The engine that compiles the code failed to find an identifier in the window global scope object.

ThirdNum is only available inside of the function where it was declared. It is encapsulated or private to function number

The question you may have is “Does the global scope has access to everything inside of number?” Again, scope only works from the inside out, the innermost context, local, to the outermost context, global.

Starting with local scope, we can say that data and variables that are wrapped in a function are only accessible to members of that function. The scope chain is what links firstNum to number().

**When number() is invoked, the non-technical conversation goes like this…**

***Engine:****“Number, I’m giving you a new execution context. Let me find what you need to run”*

***Engine****: “Ok, I see that thirdNum is explicitly declared. I’m setting space aside for you, go to the top of number’s function block and wait till I call you…*

***Engine****: “Number, I see secondNum, does he belong to you?”*

***Number****: “Nope.”*

***Engine****: “Ok, I see you’re linked to the global window object, let me look outside of you.”*

***Engine****: “Window, I have an identifier named secondNum, does he belong to you?”*

***Window****: “He didn’t declare himself explicitly in Number with a var, let, or*  
*const, so I’ll take him and set space aside.”*

***Engine****: “Cool. Number, I see firstNum in your function block, does he belong to you?”*

***Number****: “Nope.”*

***Engine****: “Window, I see firstNum being used inside of Number, he needs him, does he belong to you?”*

***Window****: “Yes, he was declared.”*

***Engine****: “Everyone is accounted for, Now I’m assigning values to variables.”*

***Engine****: Number, I’m executing you, ready, go!”*

That’s pretty much it for understanding scope, The key takeaways are:

1. Identifier lookup works from the inside out and stops at the first match.
2. There are two types of scope, global and local
3. The scope chain is created at function invocation and is based on where variables and/or blocks of code are written (lexical environment). Are variables or functions nested?
4. In JavaScript, if an identifier is not proceeded with a var, let, or const, it is implicitly declared in the global scope.
5. Scope does not go 1 for 1 with a function, it goes 1 to 1 with function invocation. Execute a function 3 times, get 3 different scopes. Why? Because if the execution of a function is finished, it is popped off the execution stack and with it, its access to other variables via its scope chain. Thus, a new scope is created each time a function is executed. Closures work a little differently!

# What is Hoisting in JavaScript?

In JavaScript, hoisting allows you to use functions and variables before they're declared. In this we learn what hoisting is and how it works.

## **What is hoisting?**

Take a look at the code below and guess what happens when it runs:

console.log(foo);

var foo = 'foo';

It might surprise you that this code outputs**undefined** and doesn't fail or throw an **error** – even though foo gets assigned after we console.log it!

This is because the JavaScript interpreter splits the declaration and assignment of functions and variables: it "hoists" your declarations to the top of their containing scope before execution.

This process is called hoisting, and it allows us to use foo before its declaration in our example above.

Let's take a deeper look at functions and variable hoisting to understand what this means and how it works.

### **Variable hoisting with var**

When the interpreter hoists a variable declared with var, it initializes its value to undefined. The first line of code below will output undefined:

console.log(foo); // undefined

var foo = 'bar';

console.log(foo); // "bar"

As we defined earlier, hoisting comes from the interpreter splitting variable declaration and assignment. We can achieve this same behavior manually by splitting the declaration and assignment into two steps:

var foo;

console.log(foo); // undefined

foo = 'foo';

console.log(foo); // "foo"

Remember that the first console.log(foo) outputs undefined because foo is hoisted and given a default value (not because the variable is never declared). Using an undeclared variable will throw a ReferenceError instead:

console.log(foo); // Uncaught ReferenceError: foo is not defined

Using an undeclared variable before its assignment will also throw a ReferenceError because no declaration was hoisted:

console.log(foo); // Uncaught ReferenceError: foo is not defined

foo = 'foo'; // Assigning a variable that's not declared is valid

By now, you may be thinking, "Huh, it's kind of weird that JavaScript lets us access variables before they're declared." This behavior is an unusual part of JavaScript and can lead to errors. Using a variable before its declaration is usually not desirable.

Thankfully the let and const variables, introduced in ECMAScript 2015, behave differently.

### **Variable hoisting with let and const**

Variables declared with let and const are hoisted but not initialized with a default value. Accessing a let or const variable before it's declared will result in a ReferenceError:

console.log(foo); // Uncaught ReferenceError: Cannot access 'foo' before initialization

let foo = 'bar'; // Same behavior for variables declared with const

Notice that the interpreter still hoists foo: the error message tells us the variable is initialized somewhere.

### **The temporal dead zone**

The reason that we get a reference error when we try to access a let or const variable before its declaration is because of the temporal dead zone (TDZ).

The TDZ starts at the beginning of the variable's enclosing scope and ends when it is declared. Accessing the variable in this TDZ throws a ReferenceError.

Here's an example with an explicit [block](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/block) that shows the start and end of foo's TDZ:

{

// Start of foo's TDZ

let bar = 'bar';

console.log(bar); // "bar"

console.log(foo); // ReferenceError because we're in the TDZ

let foo = 'foo'; // End of foo's TDZ

}

The TDZ is also present in default function parameters, which are evaluated left-to-right. In the following example, bar is in the TDZ until its default value is set:

function foobar(foo = bar, bar = 'bar') {

console.log(foo);

}

foobar(); // Uncaught ReferenceError: Cannot access 'bar' before initialization

But this code works because we can access foo outside of its TDZ:

function foobar(foo = 'foo', bar = foo) {

console.log(bar);

}

foobar(); // "foo"

### **typeof** in the temporal dead zone

Using a let or const variable as an operand of the typeof operator in the TDZ will throw an error:

console.log(typeof foo); // Uncaught ReferenceError: Cannot access 'foo' before initialization

let foo = 'foo';

This behavior is consistent with the other cases of let and const in the TDZ that we've seen. The reason that we get a ReferenceError here is that foo is declared but not initialized – we should be aware that we're using it before initialization ([source: Axel Rauschmayer](https://2ality.com/2015/10/why-tdz.html)).

However, this isn't the case when using a var variable before declaration because it is initialized with undefined when it is hoisted:

console.log(typeof foo); // "undefined"

var foo = 'foo';

Furthermore, this is surprising because we can check the type of a variable that doesn't exist without an error. typeof safely returns a string:

console.log(typeof foo); // "undefined"

In fact, the introduction of let and const broke typeof's guarantee of always returning a string value for any operand.

## **Function hoisting in JavaScript**

Function declarations are hoisted, too. Function hoisting allows us to call a function before it is defined. For example, the following code runs successfully and outputs "foo":

foo(); // "foo"

function foo() {

console.log('foo');

}

Note that only function declarations are hoisted, not function expressions. This should make sense: as we just learned, variable assignments aren't hoisted.

If we try to call the variable that the function expression was assigned to, we will get a TypeError or ReferenceError, depending on the variable's scope:

foo(); // Uncaught TypeError: foo is not a function

var foo = function () { }

bar(); // Uncaught ReferenceError: Cannot access 'bar' before initialization

let bar = function () { }

baz(); // Uncaught ReferenceError: Cannot access 'baz' before initialization

const baz = function () { }

This differs from calling a function that is never declared, which throws a different ReferenceError:

foo(); // Uncaught ReferenceError: baz is not defined

### **Temporal Dead Zone**

So, if the let and const are also hoisted, why is it that they cannot be accessed before their declaration? The answer to this lies within the concept of the Temporal Dead Zone (TDZ).

Variables declared using let and the constants declared using const are hoisted but are in a TDZ. This prevents them from being accessed before their declaration has actually been executed during the step-by-step execution of the code.

**Temporal Dead Zone** is the period of time during which the let and const declarations cannot be accessed.

Temporal Dead Zone starts when the code execution enters the block which contains the let or const declaration and continues until the declaration has executed.

In our code example above, Temporal Dead Zone starts after the opening parenthesis of the printAge function and continues until after the declaration of the age variable.

Consider the following code example that illustrates an interesting point about the Temporal Dead Zone.

  function print() {

    function log() {

      console.log(age);

    }

    const age = 20;

    log();

  }

  print(); *// 20*

# The JavaScript `this` Keyword

# + 5 Key Binding Rules Explained for JS Beginners

JavaScript's this keyword is one of the hardest aspects of the language to grasp. But it is critically important for writing more advanced JavaScript code.

In JavaScript, the this keyword allows us to:

* Reuse functions in different execution contexts. It means, a function once defined can be invoked for different objects using the this keyword.
* Identifying the object in the current execution context when we invoke a method.

The this keyword is very closely associated with JavaScript functions. When it comes to this, the fundamental thing is to understand where a function is invoked. Because we don't know what is in the this keyword until the function is invoked.

The usage of this can be categorized into five different binding aspects.

# ****First, What is Binding?****

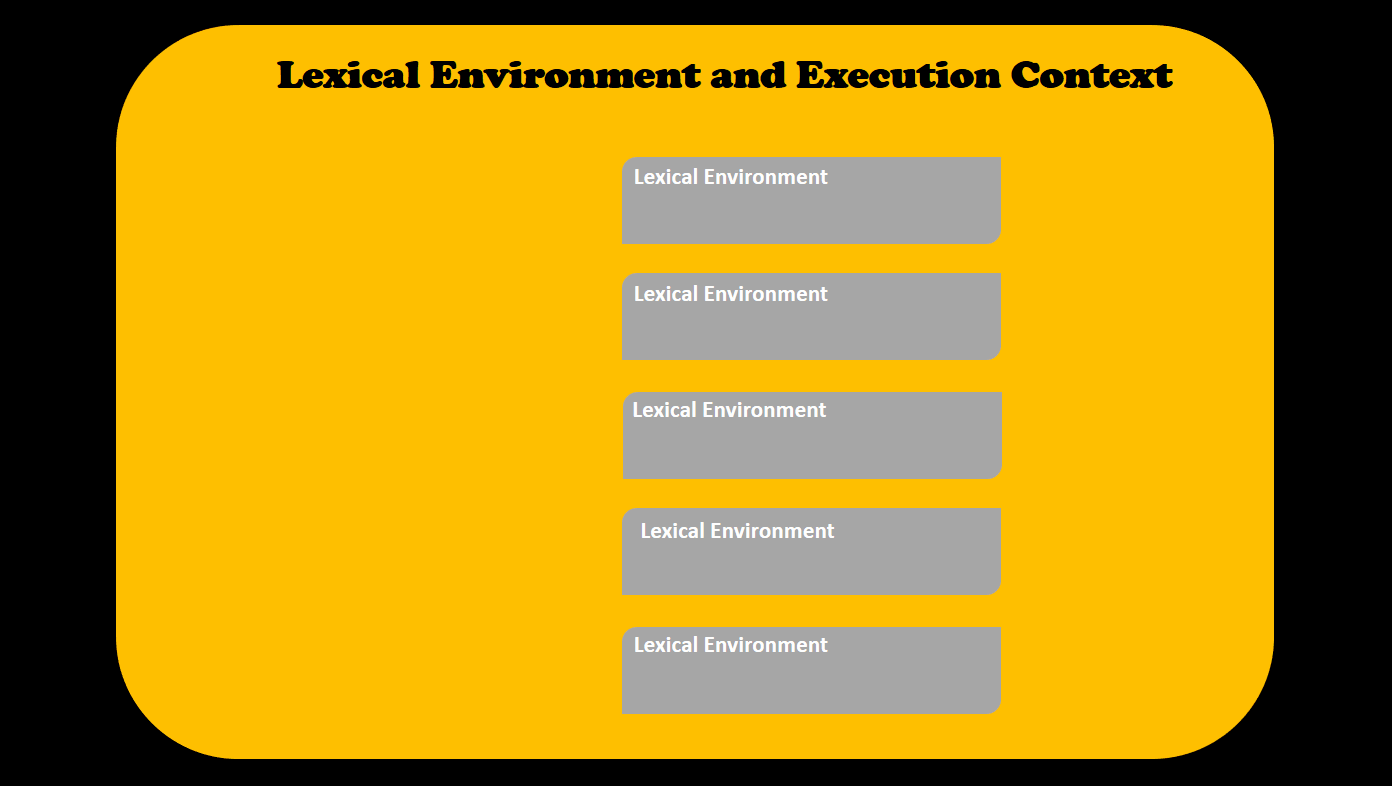
In JavaScript, a Lexical Environment is where your code is physically written. In the example below, the variable name is lexically inside the function sayName().

function sayName() {

let name = 'someName';

console.log('The name is, ', name);}

An Execution Context refers to the code that is currently running and everything else that helps run it. There can be lots of lexical environments available but the one that's currently running is managed by the [*Execution Context*](https://blog.greenroots.info/understanding-javascript-execution-context-like-never-before-ckb8x246k00f56hs1nefzpysq).



Lexical Environment vs Execution Context

Each of the Execution Context contains an Environment Record. As JavaScript engine executes the code, variables and function names gets added to the Environment Record.

This phenomenon is known as Binding in JavaScript. Binding helps associate the identifiers(variables, function names) with the this keyword for an execution context.

Don't worry if you find this a bit hard to understand now. You will get a better grasp as we proceed.

# Rule #1: How JavaScript Implicit Binding Works

Implicit binding covers most of the use-cases for dealing with the this keyword.

When we invoke a method of an object, we use the dot(.) notation to access it. In implicit binding, you need to check the object adjacent to the method at the invocation time. This determines what this is binding to.

Let's look at an example to understand it better.

let blog = {

name: 'Tapas',

address: 'freecodecamp',

message: function() {

console.log(`${this.name} blogs on ${this.address}`);

} };

blog.message(); // Tapas blogs on freecodecamp

Here this is bound to the blog object. We know this because we invoke the method message() on the blog object. So this.name is going to log Tapas and this.address is going to log  freeCodeCamp in the console.

Let's see another example to better understand this concept:

function greeting(obj) {

obj.logMessage = function() {

console.log(`${this.name} is ${this.age} years old!`);

}

};

const tom = {

name: 'Tom',

age: 7

};

const jerry = {

name: 'jerry',

age: 3

};

greeting(tom);

greeting(jerry);

tom.logMessage (); // Tom is 7 years old!

jerry.logMessage ();// jerry is 3 years old!

In this example, we have two objects, tom and jerry. We have decorated (enhanced) these objects by attaching a method called logMessage().

Notice that when we invoke tom.logMessage(), it was invoked on the tom object. So this is bound to the tom object and it logs the value tom and 7 (this.name is equal to tom and this.age is 7 here). The same applies when jerry.logMessage() is invoked.

# Rule #2: How JavaScript Explicit Binding Works

We have seen that JavaScript creates an environment to execute the code we write. It takes care of the memory creation for variables, functions, objects, and so on in the creation phase. Finally it executes the code in the execution phase. This special environment is called the Execution Context.

There can be many such environments (Execution Contexts) in a JavaScript application. Each execution context operates independently from the others.

But at times, we may want to use stuff from one execution context in another. That is where explicit binding comes into play.

In explicit binding, we can call a function with an object when the function is outside of the execution context of the object.

There are three very special methods, call(), apply() and bind() that help us achieve explicit binding.

## **How the JavaScript call() Method Works**

With the call() method, the context with which the function has to be called will be passed as a parameter to the call(). Let us see how it works with an example:

let getName = function() {

console.log(this.name);

}

let user = {

name: 'Tapas',

address: 'Freecodecamp'

};

getName.call(user); // Tapas

Here the call() method is invoked on a function called getName(). The getName() function just logs this.name. But what is this here? That gets determined by what has been passed to the call() method.

Here, this will bind to the user object because we have passed the user as a parameter to the call() method. So this.name should log the value of the name property of the user object, that is Tapas.

In the above example, we have passed just one argument to call(). But we can also pass multiple arguments to call(), like this:

let getName = function(hobby1, hobby2) {

console.log(this.name + ' likes ' + hobby1 + ' , ' + hobby2);

}

let user = {

name: 'Tapas',

address: 'Bangalore'

};

let hobbies = ['Swimming', 'Blogging'];

getName.call(user, hobbies[0], hobbies[1]); // Tapas likes Swimming , Blogging

Here we have passed multiple arguments to the call() method. The first argument must be the object context with which the function has to be invoked. Other parameters could just be values to use.

Here I am passing Swimming and Blogging as two parameters to the getName() function.

Did you notice a pain point here? In case of a call(), the arguments need to be passed one by one – which is not a smart way of doing things! That's where our next method, apply(), comes into the picture.

## **How the JavaScript apply() Method Works**

This hectic(very busy with a lot of things that you have to do quickly) way of passing arguments to the call() method can be solved by another alternate method called apply(). It is exactly the same as call() but allows you to pass the arguments more conveniently. Have a look:

let getName = function(hobby1, hobby2) {

console.log(this.name + ' likes ' + hobby1 + ' , ' + hobby2);

}

let user = {

name: 'Tapas',

address: 'Bangalore'

};

let hobbies = ['Swimming', 'Blogging'];

getName.apply(user, hobbies); // Tapas likes Swimming , Blogging

Here we are able to pass an array of arguments, which is much more convenient than passing them one by one.

*Tip: When you only have one value argument or no value arguments to pass, use call(). When you have multiple value arguments to pass, use apply().*

## **How The JavaScript bind() Method Works**

The bind() method is similar to the call() method but with one difference. Unlike the call() method of calling the function directly, bind() returns a brand new function and we can invoke that instead.

let getName = function(hobby1, hobby2) {

console.log(this.name + ' likes ' + hobby1 + ' , ' + hobby2);

}

let user = {

name: 'Tapas',

address: 'Bangalore'

};

let hobbies = ['Swimming', 'Blogging'];

let newFn = getName.bind(user, hobbies[0], hobbies[1]);

newFn(); // Tapas likes Swimming , Blogging

Here the getName.bind() doesn't invoke the function getName() directly. It returns a new function, newFn and we can invoke it as newFn().

**Rule #3: The JavaScript new Binding**

A new keyword is used to create an object from the constructor function.

let Cartoon = function(name, character) {

this.name = name;

this.character = character;

this.log = function() {

console.log(this.name + ' is a ' + this.character);

}

};

You can create objects using the new keyword  like this:

let tom = new Cartoon('Tom', 'Cat');

let jerry = new Cartoon('Jerry', 'Mouse');

When a function is invoked with the new keyword, JavaScript creates an internal this object(like, this = {}) within the function. The newly created this binds to the object being created using the new keyword.

Sounds complex? Ok, let's break it down. Take this line,

let tom = new Cartoon('Tom', 'Cat');

Here the function Cartoon is invoked with the new keyword. So the internally created this will be bound to the new object being created here, which is *tom*.

**Rule #4: JavaScript Global Object Binding**

What do you think will be the output of the code below? What is this binding to here?

let sayName = function(name) {

console.log(this.name);

};

window.name = 'Tapas';

sayName();

If the this keyword is not resolved with any of the bindings, implicit, explicit or new, then the this is bound to the window(global) object.

There is one exception though. JavaScript **strict mode** does not allow this default binding.

"use strict";

function myFunction() {

return this;

}

In the above case, this is undefined.

**Rule #5: HTML Event Element Binding in JavaScript**

In HTML event handlers, this binds to the HTML elements that receive the event.

<button onclick="console.log(this)">Click Me!</button>

The is the output log in the console when you click on the button:

"<button onclick='console.log(this)'>Click Me!</button>"

You can change the button style using the this keyword, like this:

<button onclick="this.style.color='teal'">Click Me!</button>

But be mindful when you call a function on the button click and use this inside that function.

<button onclick="changeColor()">Click Me!</button>

and the JavaScript:

function changeColor() {

this.style.color='teal';

}

The above code won't work as expected. As we have seen in the Rule 4, here this will be bound to the global object (in the 'non-strict' mode) where there is no *style* object to set the color.

**In Summary**

To summarize,

* In the case of implicit binding, this binds to the object adjacent to the dot(.) operator while invoking the method.
* In the case of explicit binding, we can call a function with an object when the function is outside of the execution context of the object. The methods call(), apply(), and bind() play a big role here.
* When a function is invoked with the new keyword, the this keyword inside the function binds to the new object being constructed.
* When the this keyword is not resolved with any of the bindings, implicit, explicit or new, then this is bound to the window(global) object. In JavaScript's strict mode, this will be undefined.
* In HTML event handlers, this binds to the HTML elements that receive the event.

# JavaScript Primitive Values vs Reference Values – Explained with Examples

# <https://www.freecodecamp.org/news/javascript-assigning-values-vs-assigning-references/>

# Array Destructuring in JavaScript

The destructuring assignment is a cool feature that came along with ES6. Destructuring is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables. That is, we can extract data from arrays and objects and assign them to variables.

Why is this necessary?

Imagine we want extract data from an array. Previously, how would this be done?

let introduction = ["Hello", "I" , "am", "Sarah"];

let greeting = introduction[0];

let name = introduction[3];

console.log(greeting);//"Hello"

console.log(name);//"Sarah"

We can see that when we want to extract data from an array, we have to do the same thing over and over again.

The ES6 destucturing assignment makes it easier to extract this data. How is this so? First, we will discuss the destructuring assignment with arrays. Then we will move on to object destructuring.

Let's get started.

## **Basic Array Destructuring**

If we want to extract data from arrays, it's quite simple using the destructuring assignment.

Let's refer to our first example for arrays. Instead of going through that repetitive process, we'd do this:

let introduction = ["Hello", "I" , "am", "Sarah"];

let [greeting, pronoun] = introduction;

console.log(greeting);//"Hello"

console.log(pronoun);//"I"

We can also do this with the same result.

let [greeting, pronoun] = ["Hello", "I" , "am", "Sarah"];

console.log(greeting);//"Hello"

console.log(pronoun);//"I"

### **Declaring Variables before Assignment**

Variables can be declared before being assigned like this:

let greeting, pronoun;

[greeting, pronoun] = ["Hello", "I" , "am", "Sarah"];

console.log(greeting);//"Hello"

console.log(pronoun);//"I"

Notice that the variables are set from left to right. So the first variable gets the first item in the array, the second variable gets the second item in the array, and so on.

### **Skipping Items in an Array**

What if we want to get the first and last item on our array instead of the first and second item, and we want to assign only two variables? This can also be done. Look at the example below:

let [greeting,,,name] = ["Hello", "I" , "am", "Sarah"];

console.log(greeting);//"Hello"

console.log(name);//"Sarah"

What just happened?

Look at the array on the left side of the variable assignment. Notice that instead of having just one comma, we have three. The comma separator is used to skip values in an array. So if you want to skip an item in an array, just use a comma.

Let's do another one. Let's skip the first and third item on the list. How would we do this?

let [,pronoun,,name] = ["Hello", "I" , "am", "Sarah"];

console.log(pronoun);//"I"

console.log(name);//"Sarah"

So the comma separator does the magic. So if we want to skip all items, we just do this:

let [,,,,] = ["Hello", "I" , "am", "Sarah"];

### **Assigning the rest of an array**

What if we want to assign some of the array to variables and the rest of the items in an array to a particular variable? In that case, we would do this:

let [greeting,...intro] = ["Hello", "I" , "am", "Sarah"];

console.log(greeting);//"Hello"

console.log(intro);//["I", "am", "Sarah"]

Using this pattern, you can unpack and assign the remaining part of an array to a variable.

### **Destructuring Assignment with Functions**

We can also extract data from an array returned from a function. Let's say we have a function that returns an array like the example below:

function getArray() {

return ["Hello", "I" , "am", "Sarah"]; }

let [greeting,pronoun] = getArray();

console.log(greeting);//"Hello"

console.log(pronoun);//"I"

We get the same results.

### **Using Default Values**

Default values can be assigned to the variables just in case the value extracted from the array is undefined:

let [greeting = "hi",name = "Sarah"] = ["hello"];

console.log(greeting);//"Hello"

console.log(name);//"Sarah"

So name falls back to "Sarah" because it is not defined in the array.

### **Swapping Values using the Destructuring Assignment**

One more thing. We can use the destructuring assignment to swap the values of variables:

let a = 3;

let b = 6;

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

console.log(a);//6

console.log(b);//3

Next, let's move on to Object Destructuring.

## **Object Destructuring**

First, let's see why there is a need for object destructuring.

Say we want to extract data from an object and assign to new variables. Prior to ES6, how would this be done?

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let name = person.name;

let country = person.country;

let job = person.job;

console.log(name);//"Sarah"

console.log(country);//"Nigeria"

console.log(job);//Developer"

See how tedious it is to extract all the data. We have to repeatedly do the same thing. ES6 destructuring really saves the day. Let's jump right into it.

### **Basic Object Destructuring**

Let's repeat the above example with ES6. Instead of assigning values one by one, we can use the object on the left to extract the data:

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let {name, country, job} = person;

console.log(name);//"Sarah"

console.log(country);//"Nigeria"

console.log(job);//Developer"

You'll get the same results. It is also valid to assign variables to an object that haven't been declared:

let {name, country, job} = {name: "Sarah", country: "Nigeria", job: "Developer"};

console.log(name);//"Sarah"

console.log(country);//"Nigeria"

console.log(job);//Developer"

### **Variables declared before being assigned**

Variables in objects can be declared before being assigned with destructuring. Let's try that:

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let name, country, job;

{name, country, job} = person;

console.log(name);// Error : "Unexpected token ="

Wait, what just happened?! Oh, we forgot to add () before the curly brackets.

The ( ) around the assignment statement is required syntax when using the object literal destructuring assignment without a declaration. This is because the {} on the left hand side is considered a block and not an object literal. So here's how to do this the right way:

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let name, country, job;

({name, country, job} = person);

console.log(name);//"Sarah"

console.log(job);//"Developer"

It is also important to note that when using this syntax, the () should be preceded by a semicolon. Otherwise it might be used to execute a function from the previous line.

Note that the variables in the object on the left hand side should have the same name as a property key in the object person. If the names are different, we'll get undefined:

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let {name, friends, job} = person;

console.log(name);//"Sarah"

console.log(friends);//undefined

But if we want to use a new variable name, well, we can.

### **Using a new Variable Name**

If we want to assign values of an object to a new variable instead of using the name of the property, we can do this:

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let {name: foo, job: bar} = person;

console.log(foo);//"Sarah"

console.log(bar);//"Developer"

So the values extracted are passed to the new variables foo and bar.

### **Using Default Values**

Default values can also be used in object destructuring, just in case a variable is undefined in an object it wants to extract data from:

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let {name = "myName", friend = "Annie"} = person;

console.log(name);//"Sarah"

console.log(friend);//"Annie"

So if the value is not undefined, the variable stores the value extracted from the object as in the case of name. Otherwise, it used the default value as it did for friend.

We can also set default values when we assign values to a new variable:

let person = {name: "Sarah", country: "Nigeria", job: "Developer"};

let {name:foo = "myName", friend: bar = "Annie"} = person;

console.log(foo);//"Sarah"

console.log(bar);//"Annie"

So  name was extracted from person and assigned to a different variable. friend, on the other hand, was undefined in person, so the new variable bar  was assigned the default value.

### **Computed Property Name**

Computed property name is another object literal feature that also works for destructuring. You can specify the name of a property via an expression if you put it in square brackets:

let prop = "name";

let {[prop] : foo} = {name: "Sarah", country: "Nigeria", job: "Developer"};

console.log(foo);//"Sarah"

### **Combining Arrays with Objects**

Arrays can also be used with objects in object destructuring:

let person = {name: "Sarah", country: "Nigeria", friends: ["Annie", "Becky"]};

let {name:foo, friends: bar} = person;

console.log(foo);//"Sarah"

console.log(bar);//["Annie", "Becky"]

### **Nesting in Object Destructuring**

Objects can also be nested when destructuring:

let person = {

name: "Sarah",

place: {

country: "Nigeria",

city: "Lagos" },

friends : ["Annie", "Becky"]

};

let {name:foo,

place: {

country : bar,

city : x}

} = person;

console.log(foo);//"Sarah"

console.log(bar);//"Nigeria"

### **Rest in Object Destructuring**

The rest syntax can also be used to pick up property keys that are not already picked up by the destructuring pattern. Those keys and their values are copied into a new object:

let person = {name: "Sarah", country: "Nigeria", job: "Developer", friends: ["Annie", "Becky"]};

let {name, friends, ...others} = person;

console.log(name);//"Sarah"

console.log(friends);//["Annie", "Becky"]

console.log(others);// {country: "Nigeria", job: "Developer"}

Here, the remaining properties whose keys where not part of the variable names listed were assigned to the variable others. The rest syntax here is ...others. others can be renamed to whatever variable you want.

One last thing – let's see how Object Destructing can be used in functions.

### **Object Destructuring and Functions**

Object Destructuring can be used to assign parameters to functions:

function person({name: x, job: y} = {}) {

console.log(x);

}

person({name: "Michelle"});//"Michelle"

person();//undefined

person(friend);//Error : friend is not defined

Notice the {} on the right hand side of the parameters object. It makes it possible for us to call the function without passing any arguments. That is why we got undefined. If we remove it, we'll get an error message.

We can also assign default values to the parameters:

function person({name: x = "Sarah", job: y = "Developer"} = {}) {

console.log(x);

}

person({name});//"Sarah"

# Spread Operator in Javascript

### [**Introduction**](https://www.digitalocean.com/community/tutorials/js-spread-operator#introduction)

The **spread operator** is a feature of JavaScript introduced with ES6 that gives you access to the insides of an [iterable object](https://www.digitalocean.com/community/tutorials/js-iterables). An “iterable object” is anything you can iterate over item by item, such as arrays, objects literals, and strings. These kinds of JavaScript types can be traversed in some sequential fashion. For example, you can use a for loop on an array, or with JavaScript objects, you can use [for...in loops](https://www.digitalocean.com/community/tutorials/for-loops-for-of-loops-and-for-in-loops-in-javascript).

## **The Basics of the Spread Operator**

The spread operator effectively gives you access to all of the items inside these iterable objects. Let’s look an example to illustrate what that means:

const foo = [

'hello',

'bonjour',

'konnichiwa'

];

const bar = [...foo]; // the three dots "..." are the spread operator syntax.

console.log(bar);

Output

['hello', 'bonjour', 'konnichiwa'];

The variable bar wound up an exact copy of the variable foo. The spread operator essentially ‘scooped’ out the insides of the foo array and **spread** the values across the new array in bar.

It’s important to note the brackets around the spread operator, [...foo]. The spread operator spreads these values within a new object of the same type; in this case, an array literal. Try running the code without the brackets:

const foo = [

'hello',

'bonjour',

'konnichiwa'

];

const bar = ...foo;

console.log(bar);

Output

Uncaught SyntaxError: expected expression, got '...'

Now that we have a basic idea, let’s look at common tasks where the spread operator might be useful.

## **Duplicating Iterable Objects**

As we saw earlier, the spread operator is one of the best ways for duplicating an iterable object. There are more complex ways to do this, but the conciseness of the spread operator makes it delightfully easy. Using the spread operator to duplicate object literals isn’t much different than for arrays. For example:

const foo = {

english: 'hello',

french: 'bonjour',

japanese: 'konnichiwa'

};

const bar = {...foo};

console.log(bar);

Output

{ english: 'hello', french: 'bonjour', japanese: 'konnichiwa' }

## **Merging Iterable Objects**

The spread operator can also be used to compose a single iterable object from several others.

const foo = ['hello', 'bonjour', 'konnichiwa'];

const bar = ['gutentag', 'saluton'];

const baz = [...foo, ...bar];

console.log(baz);

This will output the contents of foo and bar which are now contained in baz:

Output

['hello', 'bonjour', 'konnichiwa', 'gutentag', 'saluton']

You can also place a **spreaded** array inside another array as you would any other item:

const foo = ['hello', 'bonjour', 'konnichiwa'];

const bar = [...foo, 'gutentag', 'saluton'];

console.log(bar);

Now bar contains some additions to foo:

Output

['hello', 'bonjour', 'konnichiwa', 'gutentag', 'hello-ey']

A good way to think about it is that the spread operator just holds the items within the iterable object, rather the objects themselves.

How about object literals? It’s very similar to merging arrays:

const foo = {

english: 'hello',

french: 'bonjour',

japanese: 'konnichiwa'

};

const bar = {

german: 'gutentag',

esperanto: 'saluton'

};

const baz = {...foo, ...bar};

console.log(baz);

As the spread operator for the two objects contains those objects’ internals, using them in the context of a new object literal will ensure that they have those contents as well.

Output

{ english: 'hello', french: 'bonjour', japanese: 'konnichiwa', german: 'gutentag', esperanto: 'saluton' }

This is a common task for [Object.assign()](https://www.digitalocean.com/community/tutorials/js-dealing-with-objects#object-assign) but the spread syntax makes this far more concise.

**Note:** While you can merge iterable objects of different types using the spread operator, this may lead to some unwanted behaviors. For arrays and strings, you can think of them as objects where the keys are item or letter’s index in the array (e.g: {0: 'a', 1: 'b', 2: 'c'}. Should you use the spread operator for an array or string in the context of an object literal, your results will contain these key/value pairs. However, since objects have keys that are not numbers, you will not be able to use their values in an array context.

What happens when there are duplicate keys?

const foo = {

english: 'hello',

french: 'bonjour',

japanese: 'konnichiwa'

};

const bar = {

english: 'howdy',

german: 'gutentag'

};

const baz = {

...foo,

...bar,

esperanto: 'saluton',

korean: 'annyeong'

};

console.log(baz);

Here, we’re merging two existing objects into a third, both of which contain an entry for english.

Output

{ english: 'howdy', french: 'bonjour', japanese: 'konnichiwa', german: 'gutentag', esperanto: 'saluton', korean: 'annyeong' }

The duplicate keys are overwritten in the order they’re applied. It’s important to take into consideration whether or not valuable data will be lost in the process of using the spread operator to merge iterable objects.

## [**Feeding Arguments into Functions**](https://www.digitalocean.com/community/tutorials/js-spread-operator#feeding-arguments-into-functions)

The spread operator can be used in many instances where one might choose to use the apply method, which passes the values of a variable to a function in a similar way.

function calcVolume(width, height, depth) {

return width \* height \* depth;

};

calcVolume(12, 30, 14); // basic

// Passing arguments to the function from a variable:

const cube = [12, 30, 14];

calcVolume.apply(null, cube); // using "apply"

calcVolume(...cube); // using "spread operator"

The spread operator makes it easy to feed a series of arguments into functions in cases where apply may not be totally applicable.

## [**Using the Spread Operator With Strings**](https://www.digitalocean.com/community/tutorials/js-spread-operator#using-the-spread-operator-with-strings)

Lastly, you can also use the spread operator with strings since they’re also considered an iterable object.

const foo = "jumanji";

const bar = [...foo];

console.log(bar);

This will break the string jumanji up into its individual characters.

Output

// [ "j", "u", "m", "a", "n", "j", "i" ]

# Nullish Coalescing Operator in JavaScript

***The Nullish Coalescing Operator is a new logical operator in JavaScript introduced in ES 2020.***

There are over four logical operators in JavaScript: the AND &&, OR ||, NOT !, and the Nullish Coalescing Operator ??.

Sometimes called the **Nullish Operator**, this operator is used between two operands:

operand1 ?? operand2

To understand this operator, we'll have to understand what "nullish", "coalescing" and **short-circuiting** mean.

## **What are "nullish" values?**

Nullish values in JavaScript are **null and undefined**. These values fall under **falsy values** but are more specifically referred to as **null values**. All nullish values are falsy, but not all falsy values (for example, 0) are nullish.

So, the nullish operator is related to **null**and**undefined** values while other logical operators are related to truthy and falsy values in general.

## **What does "coalescing" mean?**

Coalescing, according to the dictionary means "***coming together to form one whole***". How does this apply to programming? It means you bring multiple values together, to make one value out of it.

Coalescing in programming does not specifically mean "joining the values together", but more about "deciding what value is made out of the provided values".

We'll see how this works with examples later in this.

## **Short-Circuiting**

The concept of short-circuiting applies to many programming languages. It occurs when interpreters execute a boolean-related expression and skips the irrelevant part of the expression.

For example, a boolean expression like "I am 40 years old, and I am in tech".

This expression would only be **true** if I am 40 years old, and not just that, I am in tech.

After the interpreter executes "I am 40 years old", it cannot yet conclude that the whole expression is true, because in the case that "I am **NOT** in tech", the expression will be false. The second part of the expression is relevant because it can change the result.

But, in the case that "I am **NOT** 40 years old", short-circuiting will happen. Since the first part of the expression returns false, the interpreter knows that there is no point evaluating the second expression. The second part is irrelevant, as the value from this expression does not change the result. So the interpreter skips the second part (thereby saving resources--time, power).

This also applies to the nullish operator.

You can learn more about short-circuiting [in this article](https://dillionmegida.com/p/short-circuit-in-programming-simplified/)

## **The Nullish Coalescing Operator**

Now that we've looked at the fundamentals of this operator, let's understand what this operator does.

When used in an expression, the Nullish Operator checks the first operand (on the left) to see if its value is **null** or **undefined**. If it isn't, the operator returns it from the expression. But if the first operand is any of those values, the operator returns the second operand from the expression.

Let's see a quick example:

function expression1() {

return null

}

const expression2 = 4 \* 5

const result = expression1() ?? expression2

console.log(result)

// 20

Here, we have a function called expression1 which when called returns null. And we also have expression2 which holds the value from the expression **4 \* 5**.

For the result variable, we use the nullish operator and pass expression1() and expression2 as the operands.

The first operand (the function call expression) returns null. The operator confirms that the first operand is null, so it returns the value from the second expression: expression2.

Let's see another example:

function expression1() {

console.log("expression1")

return false

}

function expression2() {

console.log("expression2")

return "Dillion"

}const result = expression1() ?? expression2()

console.log(result)

// expression1

// false

Here, we have expression1, a function which when called executes console.log("expression1"), then returns false. And we have expression2, which is a function that when called, executes console.log("expression2"), and returns "Dillion".

Using the nullish operator, we have the first operand as expression1( ), and the second operand as expression2( ) and assign the value from the expression to result.

When we run this code, you see that we have "expression1" logged, which calls from the execution of expression1. And you see that result is logged as false. This means, expression1( ) is the returned expression from the nullish operator.

The operator checks if the first expression returns null or undefined, for which it would return the second expression. But in this case, the first expression returns false, therefore, the operator would return the first expression.

Another thing you notice is that "expression2" is not logged. This means that expression2( ) is not executed at all. **Short-circuiting** happens here.

Since the operator has already confirmed that the first operand is NOT null or undefined, it doesn't bother about the second expression, because the value of the second expression does not change what the operator would return.

## **Nullish vs OR Operator**

The Nullish and OR operators have some similarities, but they work a bit differently.

The OR operator **checks if the first operand is a truthy value**. If the first operand is one, it returns it, else, it returns the second operand.

But, the Nullish operator **checks if the first operand is a nullish value**. If the first operand isn't one, it returns it, else, it returns the second operand

Here's an OR example:

const expression1 = ""

const expression2 = "Dillion"

const result = expression1 || expression2

console.log(result)

// "Dillion"

Since the first operand, expression1, is a falsy value (empty string), the operator returns the second operand. If expression1 was 20, for example (which is a truthy value), it would have been returned, and short-circuiting would have happened.

Here's a Nullish example:

const expression1 = undefined

const expression2 = "Dillion"

const result = expression1 ?? expression2

console.log(result)

// "Dillion"

Using the nullish operator here, the first operand is undefined, a nullish value, so the operator returns the second operand. If expression1 was false, 20, or some other non-nullish value, it would have been returned, and short-circuiting would have happened.

## **Using the nullish operator directly with AND/OR**

You can directly mix the AND and OR operators in expressions but you cannot do that for the nullish operator. Here's what I mean:

**exp1 && exp2 || exp3 && exp4**

Here, we combine AND and OR. The order in this expression is:

1. "exp1 AND exp2"
2. "the result of that OR exp3"
3. "the result of that AND exp4"

*Don't forget that due to short-circuiting, step 2 or step 3 may never be reached.*

But you cannot do these combinations directly with the nullish operator. For example:

**exp1 && exp2 ?? exp3 || exp4**

We're mixing AND, Nullish, and OR here: this will throw a syntax error. Let's see an actual example:

function expression1() {

return null

}

const expression2 = 20 < 10

const expression3 ="Dillion"

const result = expression1() ?? expression2 || expression3

// SyntaxError: Unexpected token '||'

We have expression1 a function which when called returns null, expression2 which holds the returned value from **20 < 10**, and expression3 which holds the string value "Dillion".

Use the nullish and OR operators with these three expressions, what I would expect is that:

1. expression1() returns null, so the nullish operator returns the right side of the expression which is expression2 || expression3
2. on the right side, the OR operator is used, which checks if the left side, expression2 is truthy; since it is a falsy value, the operator returns the right side

But, by executing this, we get an error: **SyntaxError: Unexpected token '||'**. This means you cannot use these operators directly. The only way to combine them is to add parenthesis like this:

**const result = (expression1() ?? expression2) || expression3**

console.log(result)

// Dillion

By surrounding **expression1() ?? expression2** with parentheses, we can then use the returned result as the first operand for the OR operator, and add expression3 as the second operand.

## Wrap up

The nullish operator is very useful in declaring default values for potential null or undefined values. Say, you're expecting an object from an API. If that object does not contain an expected property, that property may either hold null or be undefined like this:

const obj = {}

console.log(obj.type) // undefined

Using the nullish operator, we can provide a default value:

const obj = {}

console.log(obj.type ?? "default") // "default"

There are many other ways you can use this operator for default values or safe checks.