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.

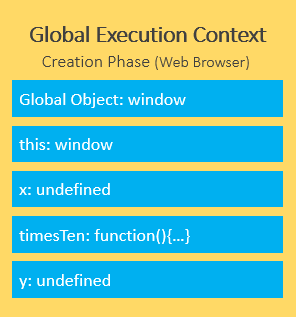
The creation phase

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:

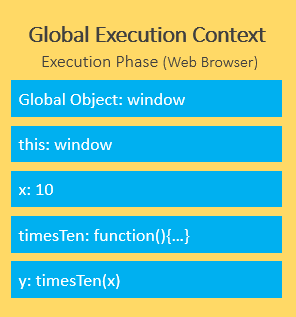
* First, store the variables x and y and function declaration timesTen() in the global execution context.
* Second, initialize the variables x and y to undefined.



After the creation phase, the global execution context moves to the execution phase.

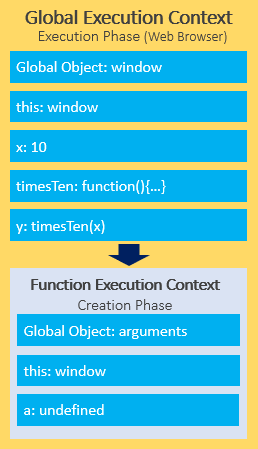
The execution phase

During the execution phase, the JavaScript engine executes the code line by line, assigns the values to variables, and executes the function calls.



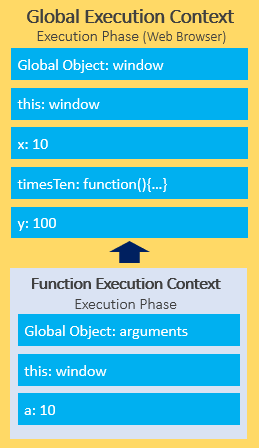
For each function call, the JavaScript engine creates a new **function execution context**.

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.

Was this tutorial helpful ?

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

A call stack is a way for the JavaScript engine to keep track of its place in code that calls multiple functions. It has the information on what function is currently being run and what functions are invoked from within that function…

Also, the JavaScript engine uses a **call stack** to manage [execution contexts](https://www.javascripttutorial.net/javascript-execution-context/):

* Global execution context
* 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 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) {

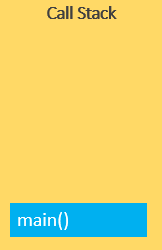
return add(a, b) / 2;

}

let x = average(10, 20);

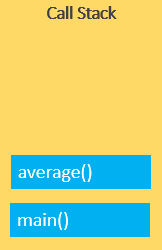
Code language: JavaScript (javascript)

When the JavaScript engine executes this script, it places the global execution context (denoted by main() or global() function on the call stack.



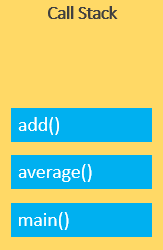
The global execution context enters the creation phase and moves to the execution phase.

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:

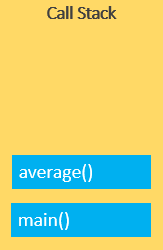


The JavaScript engine starts executing the average() since because the average() function is on the top of the call stack.

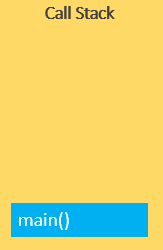
The average() function calls add() function. At this point, the JavaScript engine creates another function execution context for the add() function and places it on the top of the call stack:



JavaScript engine executes the add() function and pops it off the call stack:



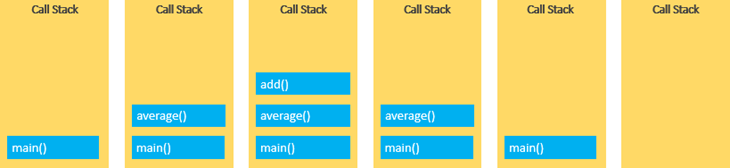
At this point, the average() function is on the top of the call stack, the JavaScript engine executes and pops it off the call stack.



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 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 a 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 be completed. For example, the JavaScript engine can:

* Request for 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](https://www.javascripttutorial.net/javascript-event-loop/), which will be covered in the following tutorial.

Summary

* JavaScript engine uses a call stack to manage execution contexts.
* The call stack uses 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...

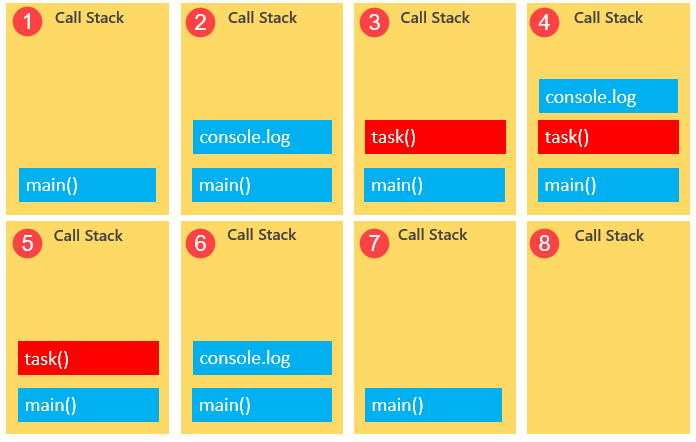
Download a file.

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.

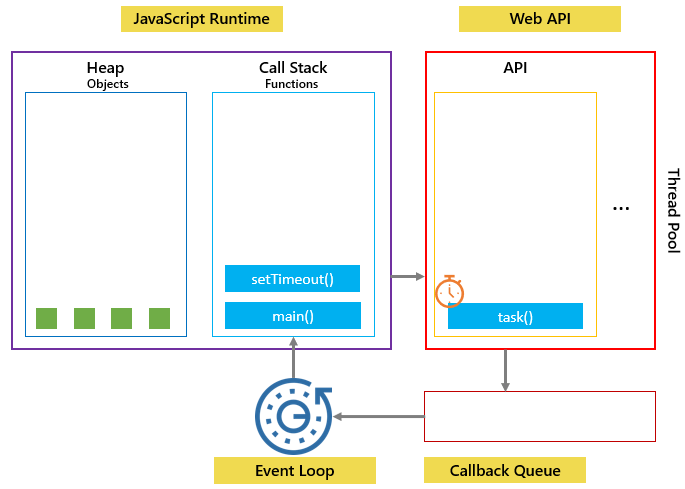
As mentioned earlier, the JavaScript engine can do only one thing at a time. However, it’s more precise to say that the JavaScript runtime can do one thing at a time.

The web browser also has other components, not just the JavaScript engine.

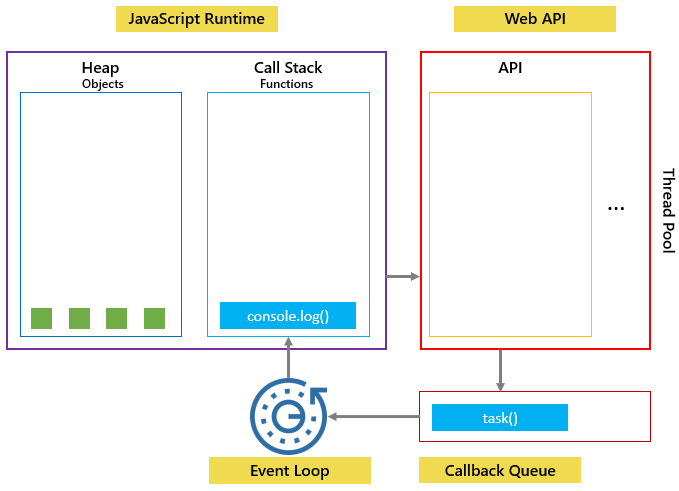
When you call the [setTimeout()](https://www.javascripttutorial.net/javascript-bom/javascript-settimeout/) function, make a [fetch request](https://www.javascripttutorial.net/javascript-fetch-api/), or click a button, the web browser can do these activities concurrently and asynchronously.

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.

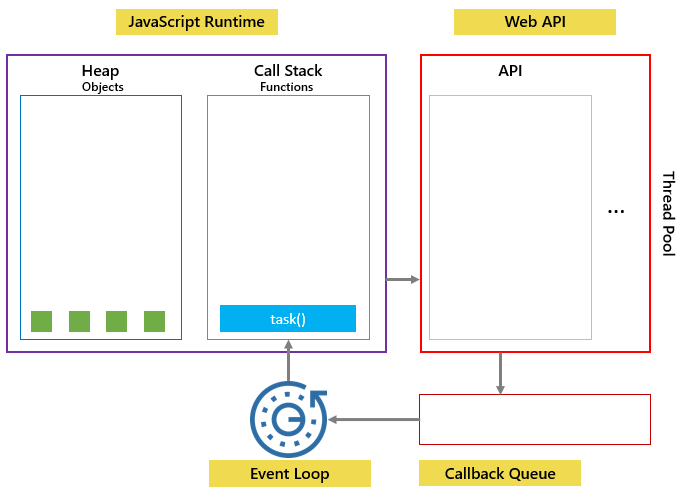


Then JavaScript engine place the task() function is into a queue called a callback queue or a task queue:



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)

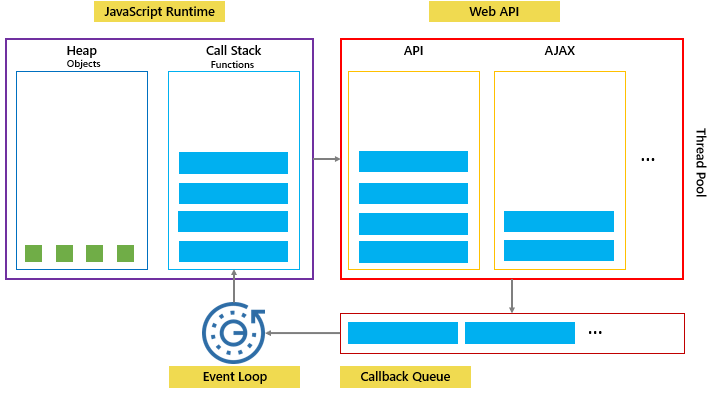
Here’s the output:

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
* Execution

During the creation phase, the JavaScript engine moves the variable and function declarations to the top of your code. This 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](https://www.javascripttutorial.net/javascript-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); *// 👉 30*

function add(a, b) {

return a + b;

}

Code language: JavaScript (javascript)

Output:

30

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); *// 👉 30*

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); *// ❌ Uncaught ReferenceError: add is not defined*

console.log(result);

let add = function(x, y) {

return x + y;

}

Code language: JavaScript (javascript)

If you execute the code, the following error will occur:

Uncaught ReferenceError: add is not defined

Code language: JavaScript (javascript)

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); *// ❌ Uncaught ReferenceError: add is not defined*

console.log(result);

let 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.

Uncaught ReferenceError: add is not defined

Code language: JavaScript (javascript)

Similar to the functions expressions, arrow functions are not 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);

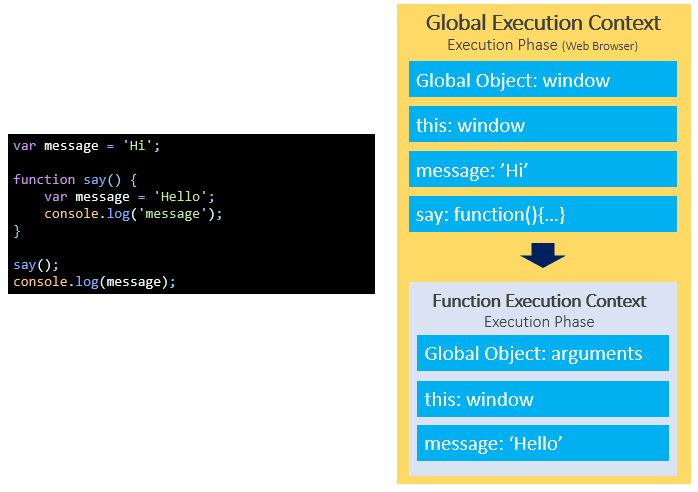
Code language: JavaScript (javascript)

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)

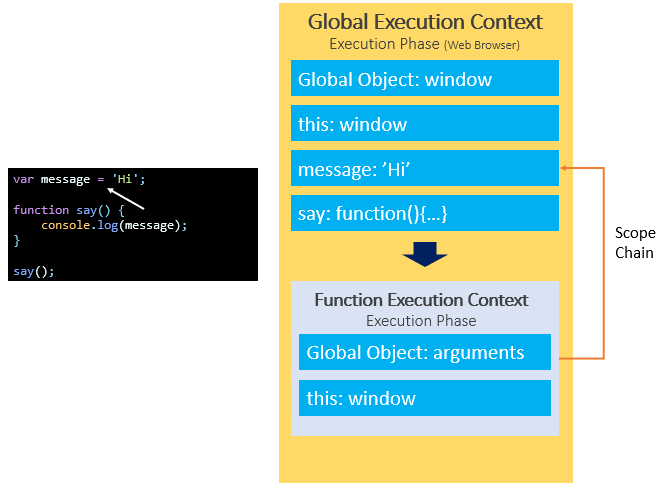
Output:

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.