



JavaScript Training





Content

- JavaScript History & Nature.
- Function & First Class Functions.
- Variables & Scopes & Hoisting.
- Objects.
- Template Literals.
- Arrow Functions
- Arrays & Array Methods.
- Spread Operator & Destructuring.
- Modules & NPM.
- Asynchronous Programming.
- Error Handling.
- Debugging.



Javascript Story

Browsers History

The early to mid-1990s was an important time for the internet. Key players like Netscape and Microsoft were in the midst of browser wars, with Netscape's Navigator and Microsoft's Internet Explorer going head to head.



Browsers History

Tales from the Browser wars: Mozilla stomps Internet Explorer



JAVASCRIPT STORY

Netscape understood this need for a programming language to run inside their browser. It would be something that would give life to it. They had a release date coming up for the Netscape 2.0 and they were also about to lose the browser war against Microsoft. They had **Ten days** before the launch of the Netscape 2.0 beta, and it was **now or never**.



Brendan Eich

JavaScript Birth

JavaScript, or Mocha, or LiveScript, or later JScript, and later ECMAScript.



The "Java" part of the name was added to capitalize on the popularity of the Java programming language at the time.



JavaScript Birth

JavaScript quickly became popular among web developers due to its ability to add interactivity and dynamic behavior to websites.





JavaScript Evolving

In November 1996, Netscape submitted JavaScript to ECMA International, as the starting point for a standard specification that all browser vendors could conform to.

- [ECMAScript Language Specification.](#)
- [JavaScript Versions.](#)



JavaScript Evolving

The JavaScript Engine is a program whose responsibility is to execute JavaScript code.

All modern browsers come with their own version of the JavaScript Engine



JavaScript Evolving

Node.js is an open-source and cross-platform JavaScript runtime environment. It is a popular tool for almost any kind of project!



Node.js runs the **V8 JavaScript engine**, the core of **Google Chrome**, outside of the browser. This allows Node.js to be very performant.



Introduction to NodeJS

Node.js® is an open-source, cross-platform JavaScript runtime environment.

Download for Windows (x64)

18.16.0 LTS

Recommended For Most Users

19.9.0 Current

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#)

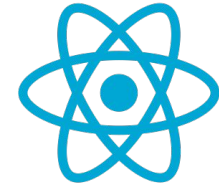
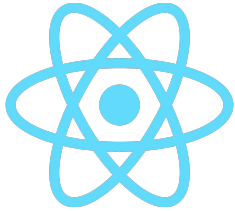
[Other Downloads](#) | [Changelog](#) | [API Docs](#)

For information about supported releases, see the [release schedule](#).

[Introduction to Node.js](#)



JavaScript is Everywhere



React Native



JavaScript Is Everywhere



JavaScript Nature

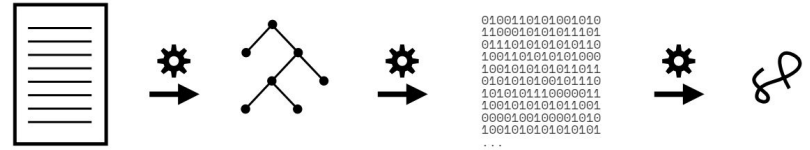
JavaScript is a **weakly typed** **interpreted**
programming and **dynamic** language

Compiled VS Interpreted

Compilation:

Turn the code into machine code without execution then after the transformation ends it execute whole the code

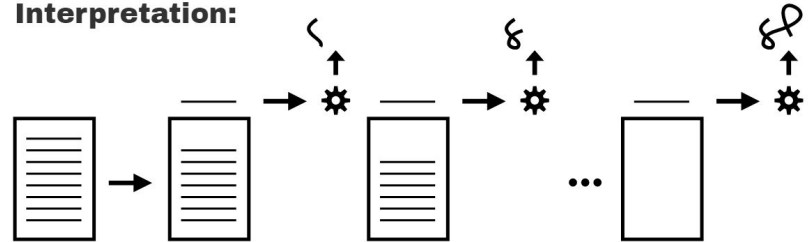
Compilation:



Interpretation:

Interpreter read the code line by line executes the line then goes to the next line reads it executes it Which means the result will appear in steps with each line

Interpretation:





Compiled VS Interpreted

Interpretation

With this approach, you have an interpreter that reads and executes the code line by line (not so efficient).

Ahead

of

Time

(AOT)

Compilation

Here, the entire program is first compiled by a compiler before being run.

Just-In-Time

Compilation

A JIT compilation strategy combines the best aspects of the AOT and interpretation strategies. It does dynamic compilation while also allowing for some optimizations, which significantly speeds up the compilation process.



Exercise

Explore how compiled (C++) and “interpreted” (JavaScript) programming languages respond to each of the following code snippets.

```
int main() {  
    int example = 10;  
    std::cout << example;  
    std::cout << another_variable;  
    return 0;  
}
```

```
var example = "Hello, World!";  
console.log(example)  
console.log(another_variable)
```



Dynamic Programming Languages

JavaScript is a dynamic language with dynamic types. Variables in JavaScript are not directly associated with any particular value type, and any variable can be assigned (and re-assigned) values of all types, the simplest way to declare a variable in js is using var:

```
var example = "Hello, World!"; // example is a string
console.log(example)           // Hello, World!
console.log(typeof example);   // Outputs: string
example = 42;                  // Now, example is a number
console.log(example)           // 42
console.log(typeof example);   // Outputs: number
```



Weakly Typed Languages

JavaScript is also a weakly typed language, which means it allows implicit type conversion when an operation involves mismatched types, instead of throwing type errors.

```
let sum = 10 + "5";           // "105" (concatenation)
let difference = 10 - "5";    // 5 (subtraction after converting "5" to number)
let product = 10 * "5";       // 50 (multiplication after converting "5" to number)
let quotient = 10 / "2";      // 5 (division after converting "2" to number)
```



Weakly Typed Languages

The Addition operator (12.7.3) - ES6 Spscs

11. If `Type(lprim)` is String or `Type(rprim)` is String, then
 - a. Let *lstr* be `ToString(lprim)`.
 - b. `ReturnIfAbrupt(lstr)`.
 - c. Let *rstr* be `ToString(rprim)`.
 - d. `ReturnIfAbrupt(rstr)`.
 - e. Return the String that is the result of concatenating *lstr* and *rstr*.



Data Types in JS

primitive

types:

In JavaScript, a primitive (primitive value, primitive data type) is data that is not an object and has no methods or properties, including:

- Number
- String
- Boolean
- Null
- Undefined
- Symbol



Data Types in JS

Primitive

types:

Primitives have no methods but still behave as if they do. When properties are accessed on primitives, JavaScript auto-boxes the value into a wrapper object and accesses the property on that object instead.

```
'foo'.includes('f');
```



Data Types in JS

Object

types:

object types are used to represent complex data structures that can contain multiple values and behaviors, including:

- **Object** - a collection of key-value pairs (properties), where the values can be of any type, including other objects and functions.
- **Array** - a special type of object used to represent an ordered list of values, with each value accessible by an index.
- **Function** - used to represent a block of reusable code that can be invoked by calling the function, functions are first-class objects.



Data Types in JS

Object

types:

object types are used to represent complex data structures that can contain multiple values and behaviors, including:

- **Date** - represents a specific point in time, with methods to retrieve and manipulate various aspects of the date and time.
- **RegExp** - represents a regular expression pattern used for matching and manipulating strings.
- **Error** - represents an error that can occur during the execution of JavaScript code, with a message and a stack trace providing information about the error.



VARIABLES



Variable Declaration

JavaScript has two primary keywords for **variable** declaration **"var"** and **"let"**, as well as the **"const"** keyword for declaring **constants**.

"var" represents the older method of declaring variables, while **"let"** and **"const"** were introduced in ES6 as newer alternatives for declaring variables and constants. However, these three keywords exhibit distinct differences in their behavior and usage.



Variable Declaration

By the end of this section, you should fully understand the differences mentioned in the following table and be able to explain them. ((**This is a very common interview question**)).

	var	let	const
Scope	Local/ Function	Block	Block
Redeclare	✓	×	×
Reassign	✓	✓	×
Hoisting	✓	×	×



Variable Declaration

Declaration is the process of introducing a new variable/constant to the code. In JS we have three keywords **var**, **let**, or **const** that can be used to declare variables.

```
var first_variable;  
let second_variable;  
const first_constant = 5 ; // 💡 We can't declare a constant without a value.
```



Variable Redeclaration

Redeclaration simply means to redeclare an already declared variable or identifier regardless of whether in the same block or outer scope.

```
var first_variable = 5;
var first_variable; // 💡 JavaScript will allow it but ignore it.
console.log(first_variable) // ✅ 5

let second_variable;
let second_variable; // ❌ SyntaxError: Identifier has already been declared.

const first_constant = 5 ;
const first_constant = 6; // ❌ SyntaxError: Identifier has already been declared.
```



Exercise

Test whether it's possible to redeclare a variable that was initially defined using `var` with the `const` keyword.

Test whether it's possible to redeclare a function name and use it for a variable.



Variable Assignment

Assignment is the process of giving a value to an already declared variable. The `=` operator is used for assignment.

```
var first_variable;  
first_variable = 10; // 💡 Assigning value 10 to first_variable.  
  
var second_variable = "test"; // 💡 Declaring a variable and assigning it a value.  
  
let third_variable;  
third_variable = 5; // 💡 Assigning value 5 to third_variable.  
  
const first_constant = 8; // 💡 You must assign values to constants on declaration.
```

Variable Reassignment

Reassignment is the process of changing the value of a variable after it has already been declared and assigned.

```
let first_variable = 5;
first_variable = 6 // 💡 Reassigning the value of 'first_variable '
console.log(x); // 6

var second_variable = 10;
second_variable = 5 // 💡 Reassigning the value of 'second_variable'
var second_variable = "test" // 💡 Reassigning the value of 'second_variable'

console.log(second_variable); // ✅ test

const first_constant = 5; // 💡 You can't reassign a constant
```




Functions and First Class Functions



Functions in JavaScript

Functions are declared using the function keyword, followed by a name, a list of parameters within parentheses `()`, and the function body enclosed in curly brackets `{ }`

```
function greet(name) {  
  console.log('Hello ' + name);  
}
```



Function return value

A function in JavaScript can return a value using the **return** keyword.

```
function sum(a, b) {  
    return a + b;  
}  
  
var total = sum(10, 20); // Assigns the return value of sum(10, 20) to total  
  
console.log(total); // Output: 30
```



Default Params

Default function parameters allow named parameters to be initialized with default values if no value or undefined is passed.

```
function showInfo(info = "No information provided") {  
    console.log(info);  
}  
  
showInfo();  
showInfo("Sample Information");
```



First Class Function

In JavaScript, functions are **first-class objects**, which means they can be:

- Stored in a variable, object, or array
- Passed as an argument to a function
- Returned from a function

This ability to manipulate functions as values is a key ingredient to many programming techniques in JavaScript, including **callback functions**, **higher-order functions**, and **functional programming**.



First Class Function

You create a function expression and assign it to a variable that can be called.

```
var generateIntro = function(name) {  
  return "Hi, my name is "+ name  
}
```

As you see here, we have the **function** keyword without a name for the function (**Anonymous function**). This makes it a **function expression**, which you have to assign to a variable



First Class Function

Functions are treated like other variables. If you can define a variable inside a function, you can also define a function inside another function - **Nested Function**.

```
function outer(){  
  function inner(){  
    console.log("inner")  
  }  
  console.log("outer")  
}  
  
outer() // outer - (Why didn't it print 'inner'?)  
inner() // ReferenceError: inner is not defined - (Why is it throwing an error?)
```



First Class Function

Function can be passed as arguments to other functions.

```
function addSmiley(text) {  
    return text + " 😊";  
}  
  
function processText(callback, text) {  
    var result = callback(text);  
    console.log("Result:", result);  
}  
  
var inputText = "hello";  
processText(addSmiley, inputText); // Result: hello 😊
```




First Class Function

A function can return another function.

```
function outer(){
  function inner(){
    console.log("inner function is invoked")
  }
  return inner
}

inner() // ReferenceError: inner is not defined
var returned_function = outer()
returned_function() // inner function is invoked
```



First Class Function

Higher order functions: A higher order function is a function that takes another function as an input, returns a function or does both.

Callback functions: functions passed as arguments to another function

```
// Callback Function
function greet(name) {
    console.log("Hello, " + name + "!");
}

// Higher-Order Function
function processUserInput(callback) {
    var name = prompt("Please enter your name:");
    callback(name);
}

// Using the Higher-Order Function with the Callback
processUserInput(greet);
```



Exercise

Create a calculator that performs different arithmetic operations. However, instead of defining separate functions for each operation, we'll define a single function that accepts the operation to be performed as a function argument. This approach will utilize callback functions to handle different operations.

Scope

A simplified definition of scope could be "The set of variables and functions that may be accessed from any given line of code at run-time".

Let's ask ourselves some questions:

- 🤔 Can I access any variable from any place in my code?
- 🤔 Can I have the same variable name outside a function and inside a function as well?



Global Scope or Something, I
don't know I'm not a
programmer 🤔



What do we mean by scope?

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

```
const userName = 'Sarah';  
console.log(userName); // ✓ "Sarah"
```

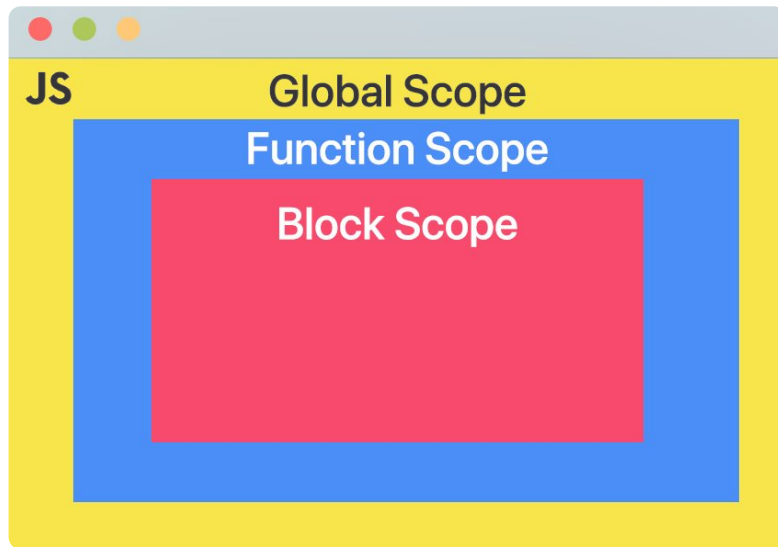
```
{const userName = 'Sarah';}  
console.log(userName); // ✗ ReferenceError: userName is not defined
```

Type of Scope

JavaScript has three different types of scope:

- Global Scope
- Function Scope
- Block scope

We will be taking a top-down approach in explaining the scope.





What is a Code Block?

A **block statement** is used to group zero or more statements. The block is delimited by a pair of braces ("curly brackets") and contains a list of zero or more statements and declarations.

```
{  
    // 💡 Example of block scope  
}  
  
if (true) {  
    // 💡 Example of block scope  
}  
  
for (i = 1; i <= 5; i++) {  
    console.log(i); // 💡 Example of block scope  
}
```

Block Scope - var

Variables declared with the **var** keyword **can NOT** have block scope. Variables declared inside a `{ }` block can be accessed from outside the block.

```
if (true) {  
    var x = 5;  
}  
console.log(x) // ✓ 5  
  
for (var y=0;y<10;y++){  
}  
console.log(y) // ✓ 10
```




Block Scope - let, const

Sometimes, we don't want that behavior. Instead, we want to restrict the variable that is declared inside a specific block from being accessed from outside of the block.

```
if (true) {  
  let x = 5;  
  const y = 10;  
  console.log(y) // ✓ 5  
}  
  
console.log(x) // ✗ ReferenceError: x is not defined  
console.log(y) // ✗ ReferenceError: y is not defined
```



Block Scope - let, const

Each block in your code has its own scope with let and const. This allows you to use the same variable twice without encountering a conflict.

```
{  
  let x = 5;  
  console.log(x) // ✓ 5  
}  
  
{  
  let x = "test";  
  console.log(x) // ✓ test  
}  
  
console.log(x) // ✗ ReferenceError: x is not defined
```



Function Scope (Local Scope)

Each and every function creates its own scope. And the variables declared inside that function are only accessible inside that function and any of its nested functions.

```
function calcAge(birthyear) {  
  const currentYear = 2023;  
  const age = currentYear - birthyear;;  
  console.log(age)  
}  
  
calcAge(1975); // ✓ 48  
console.log(currentYear); // ✗ ReferenceError: currentYear is not defined  
console.log(age); // ✗ ReferenceError: age is not defined
```

Function Scope (Local Scope)

As we discussed earlier, `var` is not block-scoped; rather, it's function-scoped. If a variable is defined using `var` inside a function, you can access it anywhere inside the function but not outside of it.

```
function exampleFunction () {  
  if (true) {  
    var varScoped = "I am var-scoped";  
    let blockScoped = "I am block-scoped";  
  }  
  
  console.log(varScoped); // ✓ Output: "I am var-scoped"  
  console.log(blockScoped); // ✗ Error: blockScoped is not defined  
}  
  
exampleFunction ();  
  
console.log(varScoped); // ✗ ReferenceError: varScoped is not defined  
console.log(blockScoped); // ✗ ReferenceError: blockScoped is not defined
```



Function Scope (Local Scope)

Function parameters in JavaScript have a local scope within the function they are defined in. This means they can only be accessed and used within the body of that function.

```
function multiply(a, b) {  
    const result = a * b;  
    return result;  
}  
  
const product = multiply(3, 4);  
  
console.log(product); // ✓ 12  
console.log(a,b); // ✗ ReferenceError: a,b are not defined
```



Global Scope.

The global scope refers to the highest level of scope that encompasses the entire codebase. Variables and functions defined in the global scope are accessible from any part of the code, both inside functions and blocks.

- **Variables/Constants** declared using **let** or **const** outside of functions or code blocks (curly braces { }) are **globally accessible**.
- **Variables** declared using **var** outside of functions are **globally accessible**.



Global Scope.

```
let userName = "John Doe!"; //💡 This variable is globally accessible
const message = "Hi " //💡 This constant is globally accessible

if (true){
    var sample_variable = "test" //💡 This variable is globally accessible
}

function greetUser() {
    console.log(message + userName);
}

greetUser(); //✅ Hi John Doe!
console.log(sample_variable) //✅ test
```



Global Scope.

Pros of Using Global Variables:

- **Accessibility:** Global variables can be accessed from any part of the script. This makes it easier to share data among different functions without having to pass them as arguments repeatedly.
- **Reusability:** Global variables can be used across multiple functions, which makes it easier to reuse code and avoid duplication.



Global Scope.

Cons of Using Global Variables:


- **Name collisions:** Global variables are accessible from any part of the script, which means that if you define a variable with the same name as a global variable in a function, it will overwrite the global variable.
- **Security risks:** Global variables can be modified by any part of the script, which makes it difficult to keep track of changes and can potentially introduce security vulnerabilities.



What About Functions

As previously noted, functions in JavaScript are considered first-class objects. They are treated comparably to variables. The way **function declarations** scoping work is similar to the behavior of defining variables using the var keyword.

```
if (true) {  
  function sample() {  
    console.log("Function Declarations are not block scoped")  
  }  
}
```

```
sample() //  Function Declarations are not block scoped
```



What About Functions

```
function outer(){ //💡 Function Declaration
  console.log("outer");
  function inner(){ //💡 Function Declaration
    console.log("inner")
  } //💡 inner function is only accessible within outer function
}
outer() //✅ outer
inner() //❌ ReferenceError: inner is not defined

outer = "You can reassign function declarations"
console.log(outer) //✅ You can reassign function declarations
outer() //❌ TypeError: outer is not a function
```



What About Functions

Function expressions scope depends on the used keyword to define it.

```
if (true) {  
  let sample = function() {  
    console.log("The scope of Function Expressions depends on the used keyword")  
  }  
  var sample2 = sample // 💡 Function expressions now reside within the global scope.  
}  
  
sample() // ❌ ReferenceError: sample is not defined  
sample2() // ✅ The scope of Function Expressions depends on the used keyword
```



Undeclared Variable

Undeclared variables are the ones that are assigned without explicit declaration using any of the keyword tokens, **var**, **let** or **const**.

```
// undeclared variable
undeclaredVar = "Dummy Text";
console.log(undeclaredVar); // ✓ Dummy Text
```

If a variable has not been previously declared within the current scope or its parent scopes, it will be treated as a "var" and will be assigned to the global scope.



Scope Chain

The scope can be **nested**, which means that you can create:

- Functions inside another function
- Block inside another function
- Function inside another block
- Block inside a block

Or any other from of scope nesting can come to your mind 🤖

Inner and Outer Scopes

👉 The Scope contained within another scope is named inner (child) scope.

👉 The Scope that wraps another scope is named outer (parent) scope.

```
function outerScope() {  
  const outerVariable = "I'm in the outer scope"; // 💡 It's also considered inner for global scope  
  function innerScope() {  
    const innerVariable = "I'm in the inner scope";  
    console.log(innerVariable); // ✅ Accessible here  
    console.log(outerVariable); // ✅ Accessible here // 💡 Inner Scope can access outer scope  
  }  
  innerScope(); // ✅ // Call the inner function  
  console.log(outerVariable); // ✅ Accessible here  
  console.log(innerVariable); // ❌ ReferenceError // 💡 Parent Scope can't access an inner scope  
}  
outerScope(); // Call the outer function
```



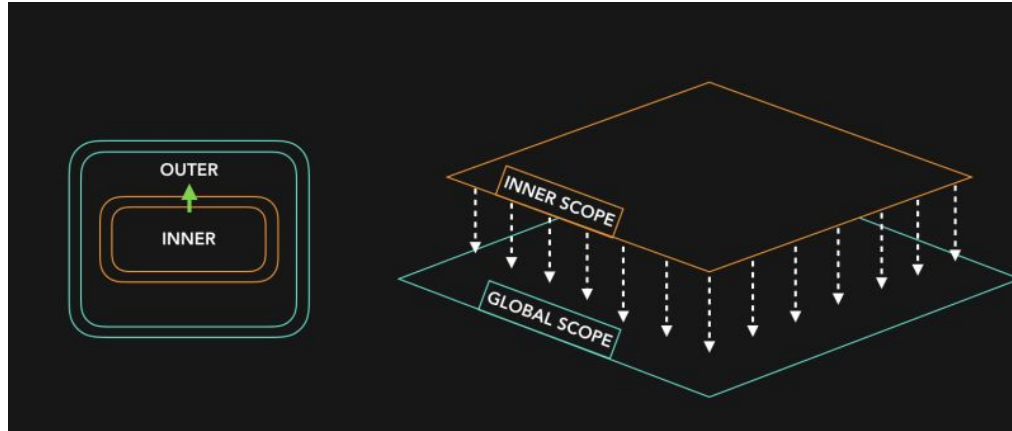
Sibling Scopes

Scopes that are defined at the same level within the code hierarchy. They exist independently of each other and do not directly share variables or data.

```
function sibling1(){  
  var x = 5;  
  sibling2() // ✓ We can call the sibling function but we can't access it's scope content  
}  
  
function sibling2(){  
  var y = 5;  
  console.log(x) // ✗ ReferenceError - We can't access a sibling scope content  
}
```

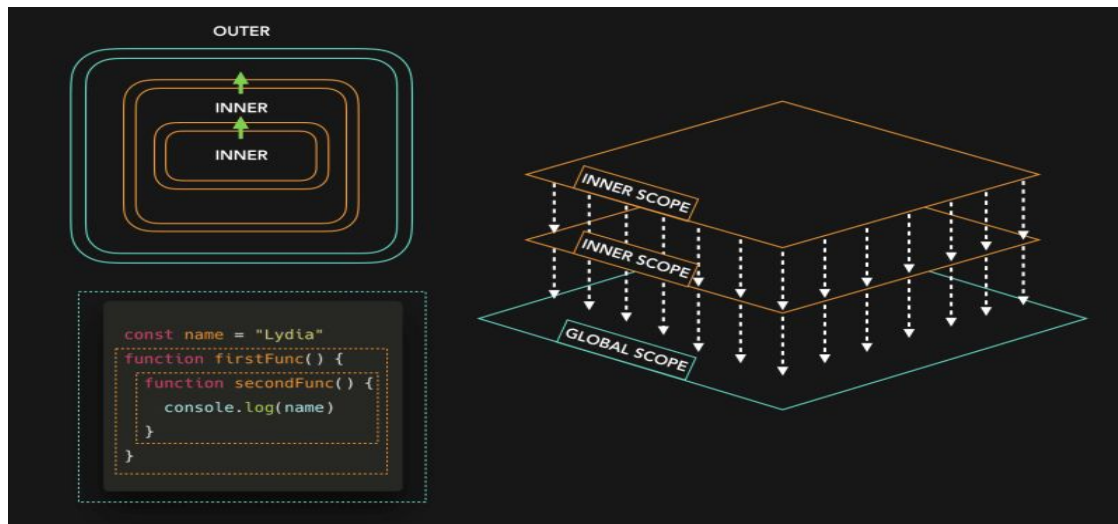

Scope Chain Rules & Notes

🔍 Inner scopes have access to variables in their containing (outer) scopes, but the reverse is not true.



Scope Chain Rules & Notes

🔍 When looking up a variable, JavaScript follows a scope chain, starting from the current scope and moving outward to higher-level scopes until it finds the variable or reaches the global scope.



Scope Chain Rules & Notes

🔍 If a variable with the same name is declared in an inner scope, it "shadows" the variable in the outer scope. The inner variable takes precedence during variable lookup.

```
const x = 10; // 💡 Outer scope variable
function shadowExample() {
  const x = 20; // 💡 Inner scope variable shadows outer scope variable
  console.log(x); // ✅ 20
}
shadowExample();
console.log(x); // ✅ 10
```



Lexical Scope

Lexical scoping means that the scope is defined at the location where the variable or function is defined, and not where they run.

```
const userName = 'Peter';
function sayUserName() {
  console.log(userName);
}
function sayUserNameAgain() {
  const userName = 'Sarah';
  sayUserName(); // Invoke the first function
}
sayUserNameAgain(); // Peter
```



Dynamic Scope

Dynamic scoping means that the scope is defined where they run and not at the location where the variable or function is defined. **This example is written using bash, run it [here](#).**

```
userName="Peter"
function sayUserName() {
    echo "$userName"
}
function sayUserNameAgain() {
    local userName="Sarah"
    sayUserName
}
sayUserNameAgain
```



Exercise

? Your task is to create a visual representation of the entire scope chain for the provided code using ExcaliDraw.

```
// Global variable
const userName = 'Peter';
// Outer function
function calcAge(birthyear) {
  const currentYear = 2021;
  const age = currentYear - birthyear;
  // inner block
  if (age <= 60) {
    var working = true;
    const message = `Peter is still employed!`;
    console.log(message);
  }
  // inner function
  function yearsToRetire() {
    const retirement = 60 - age;
    console.log(`${userName} will be retired in ${retirement} years!`);
  }
  yearsToRetire();
}
calcAge(1975);
```



Exercise

? [Bubbl.es](#) is a project that helps new JS students to understand the scoping system. Play around with this tool and try different code snippets.

The screenshot shows the 'Bubbl.es' interface with the title 'Scope Theory'. It displays a JavaScript function and its corresponding scope visualization. The code on the left is as follows:

```
1 function addOdds(...numbers) {  
2   var total = 0;  
3   for (let number of numbers) {  
4     if (number % 2 !== 0) {  
5       total += number;  
6     }  
7   }  
8   return total;  
9 }  
10  
11
```

The visualization on the right illustrates the execution context stack with three nested scopes represented by colored rectangles:

- Global Scope (Red):** The outermost region, containing the function definition.
- Function Scope (Green):** Created when the function is called, containing the `var total = 0;` declaration and the `return total;` statement.
- Block Scope (Blue):** Created for the `for` loop, containing the `let number` declaration and the `if` block.
- Block Scope (Pink):** Created for the `if` statement, containing the `total += number;` statement.

The nesting shows that the `total` variable is shared between the function and the `if` block, while `number` is only available within the `for` loop.



Hoisting

What would be the output of the following code? 🤔

```
say_hello() // ✅ hi

function say_hello(){
  console.log("hi")
}
```

It worked! But how were we able to access a function before its declaration? 🤔

Hoisting

Why didn't it throw a **ReferenceError** when we tried to access `first_variable` before its declaration? 🤔

```
console.log(first_variable) // ✅ undefined

console.log(second_variable) // ❌ ReferenceError: innerVariable is not defined

var first_variable = "Hello World!"

console.log(first_variable) // ✅ Hello World!
```

Somehow it feels like JavaScript knew about the variable before actually executing the code, similar to the function example earlier 🤖



Hoisting

In JavaScript, this behavior is referred to as **Hoisting**. So, **what exactly is hoisting?**

Before we delve into explaining hoisting, we need to understand how our code is executed in JavaScript. Let's take the following example:

```
var a = 2;  
var b = 4;  
  
var sum = a + b;  
  
console.log(sum);
```



Hoisting

This code seems straightforward, but what happens behind the scenes is what is interesting to us. Our code will go through two phases to complete:

👉 **Memory Creation Phase:** The JS Engine scans through all the code, searching for any variable or function declarations, and allocates memory for each of them."

👉 **Code Execution Phase:** The JS Engine will start going through the whole code line by line and execution it .

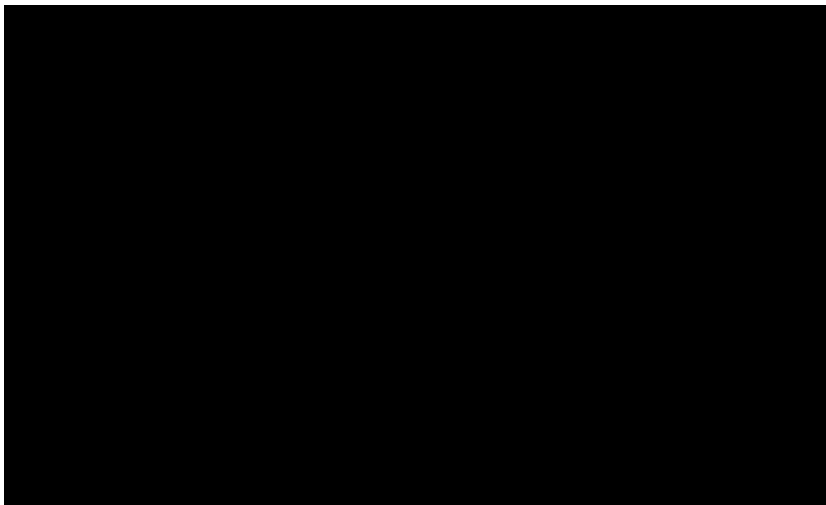


Hoisting - var

In our previous examples, the JS engine will scan through all the code and allocate memory to three variables: **a**, **b**, and **sum**.

- ❖ Variables created with **var** will have an initial value of undefined in memory.

- ❖ This type of hoisting is called **partial-hoisting** because the variable value is not hoisted.



Hoisting - var

Now that we have scanned and allocated memory to the code, we can execute the code. JavaScript will again start going through the whole code line by line.

Memory	Code
<code>a : undefined</code> <code>b : undefined</code> <code>sum : undefined</code>	



```
var a = 2;  
var b = 4;  
  
var sum = a + b;  
  
console.log(sum);
```

Hoisting - var

If we modify our code and attempt to print sum at the beginning of the code, it will output the value **undefined**.

Memory	Code
<code>a : undefined</code> <code>b : undefined</code> <code>sum : undefined</code>	



```
console.log(sum);  
  
var a = 2;  
var b = 4;  
  
var sum = a + b;
```



Hoisting - let & const

Let's check the following code example:

```
console.log(a);  
console.log(sum);  
  
var a = 2;  
let b = 4;  
const c = 6;  
  
const sum = a + b + c;
```

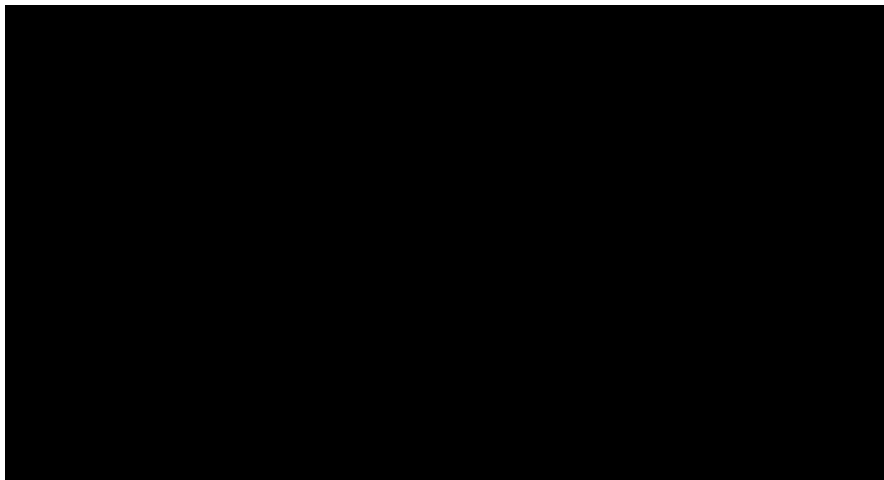


Hoisting - let & const

JS engine will scan through all the code and allocate memory to three variables: **a**, **b**, **c**, and **sum**.

❖ Variables created with **let & const** are uninitialized.

❖ Many tech blogs will simply say that variables defined with **let & const** are not hoisted. **(This is not accurate)**



Hoisting - let & const

After scanning the code, the execution starts line by line. The first line will print the value of variable 'a', which is **undefined**, to the console. However, attempting to access a variable declared with **let** or **const** before initialization will result in a thrown **ReferenceError**

main.js	Output
<pre>1 console.log(a); 2 console.log(sum); 3 4 var a = 2; 5 let b = 4; 6 const c = 6; 7 8 const sum = a + b + c; 9</pre>	<pre>node /tmp/5h9YwnJ8XY.js undefined /tmp/5h9YwnJ8XY.js:2 console.log(sum); ^ ReferenceError: sum is not initialized at Object.<anonymous> (/tmp/5h9YwnJ8XY.js:2:13) at Module._compile (internal/modules/cjs/loader.js:778:30) at Object.Module._extensions..js (internal/modules/cjs/loader.js:789:10) at Module.load (internal/modules/cjs/loader.js:653:32)</pre>



Hoisting - Function Declaration

Let's check the following code example:

```
printSum(4,5)

function printSum(a,b) {
  const c = a + b;
  console.log(c);
}
```

Hoisting - Function Declaration

When JavaScript begins scanning code and comes across a function, it stores the function's reference in memory. It is the reason why we can invoke the function before we have created it.

❖ Function declarations will have the entire function code reference stored in memory.

❖ This type of hoisting is called **complete-hoisting** because the function code reference is stored in memory.

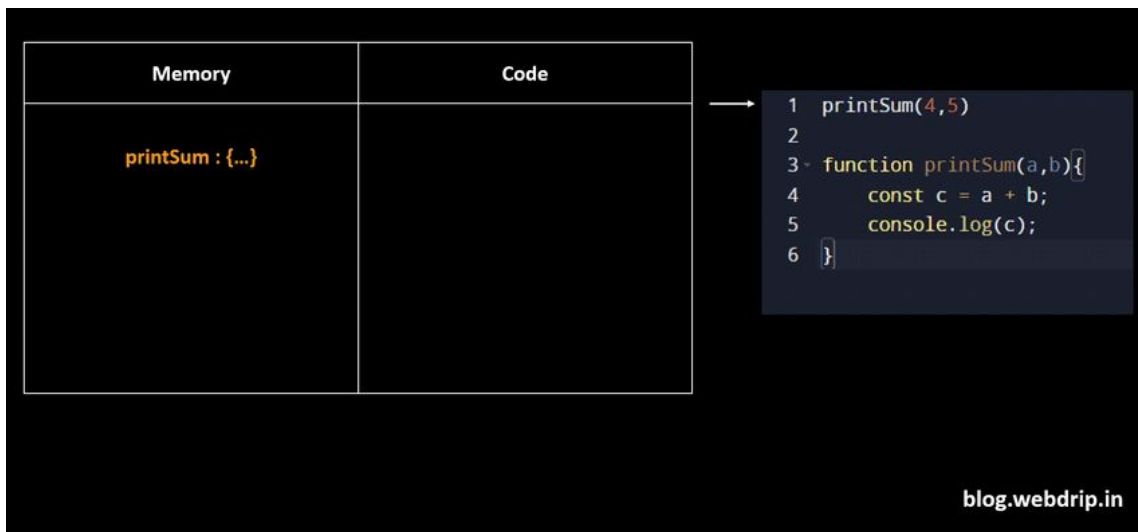
Memory	Code



```
1 printSum(4,5)
2
3 function printSum(a,b){
4     const c = a + b;
5     console.log(c);
6 }
```

Hoisting - Function Declaration

JavaScript will now execute the code line by line. It encounters a function on the first line. When a function is executed in JavaScript, it goes through the same two phases.





Final Comparison

By this point, you should have a solid grasp of all the concepts covered so far. The next time you're tasked with comparing 'var', 'let', and 'const', just remember the table below.

	var	let	const
Scope	Local/ Function	Block	Block
Redeclare	✓	×	×
Reassign	✓	✓	×
Hoisting	✓	×	×



OBJECTS

JavaScript Objects are complex data structures that allow developers to store and access data in a **key-value** format.



Objects Keys & Values

An object is an unordered collection of properties, where each property has a name (or key) and a value:

1. **Keys:** The keys are usually strings, but they can also be symbols.
2. **Values:** The values can be any data type (string, integers, arrays, functions, objects, etc).

```
let object_name = {  
  key1: value1,  
  key2: value2  
}
```



Objects Creation

JavaScript objects are a bit different. You do not need to create classes in order to create objects.

```
var emptyObject = {}; // 💡 Creating an empty Object
console.log(typeof emptyObject); // ✅ 'object'
```

The syntax to declare an object is:

```
const object_name = {
  key1: value1,
  key2: value2
}
```




Objects Creation

Here is an example of a JavaScript object:

```
let person = {  
  name: 'John',  
  age: 10  
};
```

In JavaScript, "key: value" pairs are called properties. For example:

```
let person = {  
  name: 'John',  
  age: 20  
};
```

Keys --- { } --- Values



Types of object properties

In JavaScript, object properties can have various types of values

1. Primitive Types:

String: Represents textual data.

```
let obj = { name: "John" };
```

Number: Represents both integers and floating-point numbers

```
let obj = { age: 25 };
```



Types of object properties

Boolean: Represents a true or false value.

```
let obj = { isActive: true };
```

Undefined: Represents a variable that has been declared but has not been assigned a value.

```
let obj = { something: undefined }
```

Null: Represents a deliberate absence of any value.

```
let obj = { parent: null };
```



Types of object properties

2. Reference Types:

Object: Can be used to store collections of data.

```
let obj = {  
  details: {  
    address: "123 Main St",  
    phone: "123-456-7890"  
  }  
};
```

Array: Represents a list-like collection of values.

```
let obj = { numbers: [1, 2, 3, 4, 5] };
```



Operations on Objects

For the next examples let's assume that we have the following object

```
const person = {  
  name: 'John',  
  age: 30,  
  address: {  
    city: 'New York',  
    state: 'NY',  
    country: 'USA'  
  }  
};
```

Operations on Objects

Access: We can access its properties and methods using dot notation `.` or bracket notation `[]`

```
console.log(person.name); // ✓ John
console.log(person["age"]); // ✓ 30
console.log(person.address["city"]); // ✓ New York
console.log(person["address"].state); // ✓ NY
console.log(person.address); // ✓ { city: 'New York', state: 'NY', country: 'USA' }
```

If you try to access a property that does not exist on an object, JavaScript will return **undefined**. It **won't throw an error**, just return undefined, indicating the absence of the property.

```
console.log(person.height); // ✓ undefined
```



Operations on Objects

Edit: Say we wanted to change the value of likesCoding to false. We can do that with dot notation
. or bracket notation []

```
console.log(person.name); // ✓ John
console.log(person["age"]); // ✓ 30
console.log(person.address["city"]); // ✓ New York
console.log(person["address"].state); // ✓ NY
console.log(person.address); // ✓ { city: 'New York', state: 'NY', country: 'USA' }
```



Operations on Objects

Edit: Say we wanted to change the value of likesCoding to false. We can do that with dot notation
. or bracket notation []

```
person.likesCoding = false;  
person["likesCoding"] = false;
```




Operations on Objects

Add: And if we wanted to add a new property to our person object, we could accomplish that with dot notation `.` or bracket notation `[]`

```
person.hobbies = ['hiking', 'travel', 'reading'];  
person['hobbies'] = ['hiking', 'travel', 'reading'];
```



Operations on Objects

Delete: to remove a property from an object, we use the delete keyword like so:

```
delete person.age;
```



Operations on Objects

Iterating: The most common way to loop through properties using its keys in an object is with a for...in loop:

```
for (var key in myObject) {  
  console.log(key); // ✓ logs keys in myObject  
  console.log(myObject[key]); // ✓ logs values in myObject  
}
```



Operations on Objects


Finding if object has property: You can use `hasOwnProperty` method, This method returns a boolean indicating whether the object has the specified property as its own property (as opposed to inheriting it). This method does not check the prototype chain.

```
let car = { make: 'Tesla', model: 'Model 3' };  
console.log(car.hasOwnProperty('make')); // ✓ logs true  
console.log(car.hasOwnProperty('year')); // ✓ logs false
```



Operations on Objects

Using non-existing property: if you try to access a property that does not exist on an object, JavaScript will return **undefined**. It won't throw an error, just return **undefined**, indicating the absence of the property.

```
let car = { make: 'Tesla', model: 'Model 3' };  
console.log(car.year); //  logs undefined
```



Operations on Objects

Iterating: There is another way to access the values of the properties of object by using a for...of loop:



```
for (var value in myObject) {  
  console.log(value); //  logs value in myObject  
}
```



Object methods

an object method is a property of an object that is a function

```
let car = {  
  make: 'Tesla',  
  model: 'Model 3',  
  start: function() {  
    console.log('The car starts');  
  },  
  drive: function() {  
    console.log('The car is driving');  
  }  
};
```

```
car.start(); //  logs 'The car starts'  
car.drive(); //  logs 'The car is driving'
```



Object methods

Note that starting with ES6, there is a shorthand syntax for method definitions:

```
let car = {  
  make: 'Tesla',  
  model: 'Model 3',  
  start() {  
    console.log('The car starts');  
  },  
  drive() {  
    console.log('The car is driving');  
  }  
};  
  
car.start(); // ✓ logs 'The car starts'  
car.drive(); // ✓ logs 'The car is driving'
```




Object methods

object properties can be assigned to be methods. A method in this context is just a function that's assigned as a property in an object

```
let car = {  
  make: 'Tesla',  
  model: 'Model 3',  
};  
  
car.start = function() {  
  console.log('The car starts');  
};  
  
car.drive = function() {  
  console.log('The car is driving');  
};
```



Objects are assigned by reference

In JavaScript, objects are assigned by reference. This means that when you assign an object to a new variable, you're not creating a copy of that object. Instead, you're creating a new reference to the same object.

```
let car1 = { make: 'Tesla', model: 'Model 3' };  
let car2 = car1;
```

```
console.log(car1.make); // ✓ logs 'Tesla'  
console.log(car2.make); // ✓ logs 'Tesla'
```

```
car1.make = 'Toyota';
```

```
console.log(car1.make); // ✓ logs 'Toyota'  
console.log(car2.make); // ✓ logs 'Toyota'
```



Objects with const

In JavaScript, variables declared with **const** are **read-only**, which means once a value is **assigned**, it can't be **reassigned**. However, when it comes to objects, **const** is a little more nuanced since they are assigned by reference as we have explained before .

```
const car = { make: 'Tesla', model: 'Model 3' };

car.make = 'Toyota'; // ✓ This is fine
console.log(car.make); // ✓ logs 'Toyota'

car.color = 'Red'; // ✓ Adding new properties is also fine
console.log(car.color); // ✓ logs 'Red'
```



Objects with const

Even though `car` is a **const**, this doesn't make the object itself **immutable**. It means that the `car` variable will always refer **to the same object**, but the **properties of that object** can still be modified:

```
const car = { make: 'Tesla', model: 'Model 3' };

car.make = 'Toyota'; // ✓ This is fine
console.log(car.make); // ✓ logs 'Toyota'

car.color = 'Red'; // ✓ Adding new properties is also fine
console.log(car.color); // ✓ logs 'Red'
```



Objects with const


Even though `car` is a `const`, this doesn't make the object itself `immutable`. It means that the `car` variable will always refer `to the same object`, but the `properties of that object` can still be modified:

```
car = { make: 'Ford', model: 'Mustang' }; // ✖ TypeError: Assignment to constant variable.
```



Objects passed as a reference

when you pass an object to a function in JavaScript, you're passing a reference to that object. This means that if you modify the object within the function, those changes will be reflected outside the function as well, because both the outside and the inside are referring to the same object. Here's an example:

```
function paintCar(car, color) {  
  car.color = color;  
}  
  
const myCar = { make: 'Tesla', model: 'Model 3' };  
paintCar(myCar, 'Red');  
  
console.log(myCar.color); //  logs 'Red'
```



Exercise

Suppose you have an object that represents a student's grade in various courses, like this

```
let grades = {  
  math: 90,  
  science: 80,  
  english: 85  
};
```



Exercise

Your tasks:

1. Add a course: Write a function `addCourse` that takes the course name and grade, and adds the new course to the grades object.
2. Find a grade: Write a function `findGrade` that takes a course name and returns the grade for that course.
3. Update a grade: Write a function `updateGrade` that takes a course name and a new grade, and updates the grade for that course.
4. Calculate GPA: Write a function `calculateGPA` that calculates and returns the GPA (Grade Point



Exercise

Once you've written the functions, you can test them by doing the following:

- Add a new course to the grades object.
- Find a grade for a course.
- Update a grade for a course.
- Calculate the GPA.



TEMPLATE LITERALS



Template literals

Template literals provide a way to create strings that include expressions and variables in a more concise and readable manner. They allow you to create multi-line strings, interpolate variables and expressions.



Template literals

we use backticks (``) to define the template literal. Inside the template literal, we can use the `${}` syntax to interpolate the name variable. When we run this code, it will log the string "Hello, John!" to the console.

```
const name = 'John';  
// with template literals  
console.log(`Hello, ${name}!`);  
  
//without template literals  
console.log("Hello, " + name + "!");
```



Template literals

Taking it further we can include expressions inside the ``` using the template literals

```
const name = 'John';  
// with template literals  
console.log(`Hello, ${name}!`);  
  
// include expression  
console.log(`${name} is ${name === "John" ? 170 : "unknown" }cm and ${10 + 14}years old`);
```



Template literals

More over Template literals `` can also be used to create multi-line strings:

```
const message = `This is a
multi-line string`;
console.log(message);
/*
Output: ✓
This is a
multi-line string
*/
```



Template literals

Where it wouldn't work with the normal `'` or `"`

```
const message = "This is a  
multi-line string";  
//notice how its not colored which means its not recognized as the same line by the code editor  
// ❌ Uncaught SyntaxError: Invalid or unexpected token -> we will not reach the console statement  
console.log(message);
```