

# HOISTING IN JAVASCRIPT

👉 **Hoisting:** Makes some types of variables accessible/usable in the code before they are actually declared. "Variables lifted to the top of their scope".



## BEHIND THE SCENES

**Before execution**, code is scanned for variable declarations, and for each variable, a new property is created in the **variable environment object**.

### EXECUTION CONTEXT

- 👉 Variable environment
- ✅ Scope chain
- 👉 this keyword

	HOISTED? 👉	INITIAL VALUE 👉	SCOPE 👉
function declarations	✅ YES	Actual function	Block
var variables	✅ YES	undefined	Function
let and const variables	🚫 NO	<uninitialized>, TDZ	Block
function expressions and arrows	🤖 Depends if using var or let/const		

Technically, yes. But not in practice

In strict mode. Otherwise: function!

Temporal Dead Zone

# TEMPORAL DEAD ZONE, LET AND CONST

```
const myName = 'Jonas';  
  
if (myName === 'Jonas') {  
  console.log(`Jonas is a ${job}`);  
  const age = 2037 - 1989;  
  console.log(age);  
  const job = 'teacher';  
  console.log(x);  
}
```

TEMPORAL DEAD ZONE FOR `job` VARIABLE

👉 Different kinds of error messages:

ReferenceError: Cannot access 'job' before initialization

ReferenceError: x is not defined

## WHY HOISTING?

- 👉 Using functions before actual declaration;
- 👉 `var` hoisting is just a byproduct.

## WHY TDZ?

- 👉 Makes it easier to avoid and catch errors: accessing variables before declaration is bad practice and should be avoided;
- 👉 Makes `const` variables actually work

# SCOPING AND SCOPE IN JAVASCRIPT: CONCEPTS

## SCOPE CONCEPTS

### EXECUTION CONTEXT

- 👉 Variable environment
- 👉 **Scope chain**
- 👉 this keyword

- 👉 **Scoping:** How our program's variables are **organized** and **accessed**. *"Where do variables live?" or "Where can we access a certain variable, and where not?"*
- 👉 **Lexical scoping:** Scoping is controlled by **placement** of functions and blocks in the code;
- 👉 **Scope:** Space or environment in which a certain variable is **declared** (*variable environment in case of functions*). There is **global** scope, **function** scope, and **block** scope;
- 👉 **Scope of a variable:** Region of our code where a certain variable can be **accessed**.

# THE 3 TYPES OF SCOPE

## GLOBAL SCOPE

```
const me = 'Jonas';  
const job = 'teacher';  
const year = 1989;
```

- 👉 Outside of **any** function or block
- 👉 Variables declared in global scope are accessible **everywhere**

## FUNCTION SCOPE

```
function calcAge(birthYear) {  
  const now = 2037;  
  const age = now - birthYear;  
  return age;  
}  
  
console.log(now); // ReferenceError
```

- 👉 Variables are accessible only **inside function**, NOT outside
- 👉 Also called local scope

## BLOCK SCOPE (ES6)

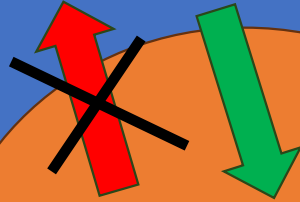
```
if (year >= 1981 && year <= 1996) {  
  const millenial = true;  
  const food = 'Avocado toast';  
} ← Example: if block, for loop block, etc.  
  
console.log(millenial); // ReferenceError
```

- 👉 Variables are accessible only **inside block** (block scoped)
- ⚠️ **HOWEVER**, this only applies to **let** and **const** variables!
- 👉 Functions are **also block scoped** (only in strict mode)

Global Scope

```
const global = 'Global';
```

```
console.log(block); // ReferenceError
```



Function and Block Scope {...}

```
const block = 'Block';
```

```
console.log(global); // 'Global'
```



## SUMMARY 🎉

- 👉 Scoping asks the question *"Where do variables live?"* or *"Where can we access a certain variable, and where not?"*;
- 👉 There are 3 types of scope in JavaScript: the global scope, scopes defined by functions, and scopes defined by blocks;
- 👉 Only `let` and `const` variables are block-scoped. Variables declared with `var` end up in the closest function scope;
- 👉 In JavaScript, we have lexical scoping, so the rules of where we can access variables are based on exactly where in the code functions and blocks are written;
- 👉 Every scope always has access to all the variables from all its outer scopes. This is the scope chain!
- 👉 When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it's looking for. This is called variable lookup;
- 👉 The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope;
- 👉 The scope chain in a certain scope is equal to adding together all the variable environments of the all parent scopes;
- 👉 The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all!