Hoisting in JavaScript

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their containing scope during the compile phase.

This means that you can use variables and functions before they are declared in the code.  
  
This creates a two-phase process:

1. **Memory Allocation Phase:** During this phase, the JavaScript interpreter scans your code and allocates memory for variables and functions based on their declarations. For var declarations (not recommended in modern JavaScript due to potential scoping issues), the variable is initialized with undefined. Function declarations are treated as complete entities, ready to be called.
2. **Code Execution Phase:** In this phase, the actual code is executed line by line

**Key Points about Hoisting:**

* **Functions:** Function declarations are hoisted entirely, meaning you can call a function before it's declared in the code. This can be surprising behavior for beginners.
* **Variables with**var**:** For var declarations, only the declaration is hoisted, not the assignment. So, accessing a var variable before its initialization will result   
  in undefined.
* **Let and Const:** Variables declared with let and const are hoisted without default behaviour. Using them before declaration will result in a ReferenceError.
* **Scope:** Hoisting applies within a specific scope, which can be global (entire script) or local (function). Variables and functions are hoisted to the top of their respective scopes.

**Why can you call a function in JavaScript before it's defined?**

This question tests your understanding of hoisting. The answer is that function declarations are hoisted to the top of their scope during the memory allocation phase. So, even if the function appears later in the code, its declaration is available for calling beforehand.

Hoisting Example

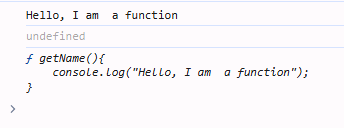
Before executing the program memory is allocated for variable and function  
then two phase is created 1) memory component 2) code component  
then again same process…

Ex:1  
getName(); // Hello, I am  a function  
console.log(x); // undefined because we accessing before initialization  
console.log(getName) // we will get actual copy of function  
var x = 10; // hoisted to the top of the scope  
// function declaration hoisting

function getName(){

    console.log("Hello, I am  a function");

}  
//output



Ex:2

// function expression hoisting  
getName(); //get error because it is behave like another variable

const getName = () =>{

    console.log("Hello, I am  a function");

}

const getName1 = function() {

    console.log("Hello, I am a function");

}

//output



Hoisting with let and const behave same as it is

1)

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

let name = 'Mukul Latiyan';

**Output:**

ReferenceError: Cannot access 'name' before initialization

2)

let name;

console.log(name); // undefined

name = 'Mukul Latiyan';

3)

// Function scoped

**function** fun() {

  console.log(name);

  let name = 'Mukul Latiyan';

}

fun(); // ReferenceError: Cannot access 'name' before initialization

4)

function codeHoist() {

a = 10;

let b = 50;

}

codeHoist();

console.log(a); // 10

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

5) function fun() {

let name;

console.log(name);

name = 'Mukul Latiyan';

}

fun(); // undefined

6) //let example(global)

console.log(name);

let name = 'Mukul Latiyan'; // ReferenceError: Cannot access 'name' before initialization

7) fun() // Calling the expression

let fun = () =>{ // Declaring

let name = 'Mukul Latiyan';

console.log(name);

}// ReferenceError: Cannot access 'fun' before initialization

8) fun() // Calling the expression

var fun = () =>{ // Declaring

let name = 'Mukul Latiyan';

console.log(name);

}  
// TypeError: fun is not a function

9) printHello()

// hello

printDillion()

// ReferenceError: printDillion is not defined

function printHello() {

console.log('hello')

function printDillion() {

console.log("dillion")

}

}

10)Hoisting with var

console.log(name)

// undefined

var myName = "Dillion"

11)print()

console.log(name)

// ReferenceError: name is not defined

function print() {

var name = "Dillion"

}

12) print()

function print() {

console.log(name)

// undefined

var name = "Dillion"

}

### 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.

Ex:1)

{

// 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

}

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

console.log(foo);

}

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

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

console.log(bar);

}

foobar(); // "foo"

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

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

Ex: 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.   
It also follow the not initialized yet behavior  
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

resetScore();

drawGameBoard();

populateGameBoard();

startGame();

function resetScore() {

console.log("Resetting score");

}

function drawGameBoard() {

console.log("Drawing board");

}

function populateGameBoard() {

console.log("Populating board");

}

function startGame() {

console.log("Starting game");

}