Research 4

Name: *Rahma Nashaat*

Under provision of Eng. Youssef Salah

1.What is the difference between array and object?

Array

*An **Array** is a collection of data and a data structure that is stored in a **sequence of memory** locations. One can access the elements of an array by calling the index number such as 0, 1, 2, 3, ..., etc. The array can store data types like **Integer, Float, String, and Boolean** all the **primitive data types** can be stored in an array.*

***Example:** In this example, we will see the basic creation of a JavaScript Array and access the values of the array.*

```
let Arr = [1, 2, 3, 4, 5];


// Iterating through loop
for (let i = 0; i < Arr.length; i++) {
    console.log(Arr[i]);
}


// Pop an element from array
Arr.pop();
console.log("After using pop() Method: " + Arr);
```

Output

1

2

3

4

5

After using pop() Method: 1,2,3,4

An array of objects

*It stores multiple values in a **single** variable. The object can contain anything in the real world such as person names, cars, and game characters. Objects are very easy to use in some situations if you know where the data is being processed. The character set of objects is known as **Properties**. The properties of an object can be called by using **DOT notation** and **[] notation**.*

Example: *We will create a basic JavaScript array object and access its properties in this example.*

```
// Array of objects
let myObject = {
    jhon: {
        name: 'jhon',
        age: 12,
        gender: 'male'
    },
    rita: {
        name: 'rita',
        age: 32,
        gender: 'male'
    }
};

// Using DOT notation
console.log('Using DOT:' + myObject.jhon.gender);

// Using [] notation
console.log('Using []:' + myObject.rita['age']);

// Using delete keyword
delete myObject.rita;
```

```
// Iterating using for..in loop
for (let key in myObject) {

    // logs values in myObject
    console.log(myObject[key]);
}
```

Output

```
Using DOT:male
Using []:32
{ name: 'jhon', age: 12, gender: 'male' }
```

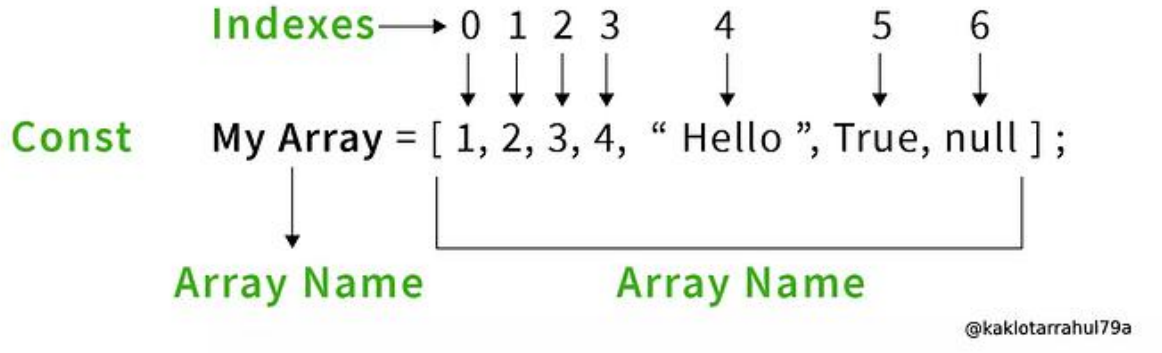
Difference between an Array and an Array of objects:

Array	*Array of objects*
*Arrays are best to use when the elements are **numbers**.*	*Objects are best to use when the elements' **strings (text)**.*
*The data inside an array is known as **Elements.***	*The data inside objects are known as **Properties** which consists of a **key** and a **value.***
The elements can be manipulated using [].	*The properties can be manipulated using both **DOT** notation and [].*
*The elements can be popped out of an array using the **pop()** function.*	*The keys or properties can be deleted by using the **delete** keyword.*
*Iterating through an array is possible using **For loop, For..in, For..of**, and **ForEach().***	*Iterating through an array of objects is possible using **For..in, For..of, and ForEach().***

1

[1] https://www.geeksforgeeks.org/javascript/difference-between-array-and-array-of-objects-in-javascript/

2.Why we use const in define array and object?



In JavaScript, an array is a collection of values, which can be of any data type, such as numbers, strings, or objects. The ***const*** *keyword is used to declare a variable whose value cannot be reassigned once it has been initialized.*

When used together, ***const*** *and arrays create a variable whose contents cannot be reassigned, but whose values can still be modified.*

Here's an example:

```
const fruits = ["apple", "banana", "orange"];

console.log(fruits[0]); // "apple"

fruits.push("grape");

console.log(fruits); // ["apple", "banana", "orange", "grape"]
```

In the example above, we create a constant variable ***fruits*** *that contains an array of strings. We can access elements in the array using index notation (****fruits[0]*** *returns "apple").*

*Even though **fruits** is a constant variable, we can still modify its contents by using array methods like **push()**, which adds a new element to the end of the array.*

*However, we cannot reassign a new array to the **fruits** variable. For example, the following code will result in an error:*

```
const fruits = ["apple", "banana", "orange"];

fruits = ["grape", "kiwi"]; // Error: Assignment to constant variable.
```

*In summary, using **const** with arrays in JavaScript creates a variable whose contents cannot be reassigned, but whose values can still be modified using array methods.*[2]

The const keyword, when used to declare an object in many programming languages (like JavaScript or C++), signifies that the binding of the variable to that specific object cannot be reassigned. This means you cannot later make the variable point to a different object.

Here's why const is used for defining objects:

- *Preventing Reassignment: The primary reason is to ensure that the variable always refers to the same object instance throughout its scope. This prevents accidental reassignments that could lead to unexpected behavior or bugs in your program.*

JavaScript

```
const myObject = { name: "Alice" };
// myObject = { age: 30 }; // This would cause an error: Assignment to constant variable.
```

- ***Clarity and Intent:***

[2] https://medium.com/@kaklotarrahul79/javascript-array-const-2b80f8138f1

Using const clearly communicates to other developers (and your future self) that the variable's reference to the object is intended to remain constant. This improves code readability and maintainability.

- *Compiler Optimizations (in some languages):*

In languages like C++, const can enable the compiler to perform certain optimizations, as it knows the object's address won't change.

Important Distinction:

It's crucial to understand that const for objects does not make the object itself immutable (unchangeable). You can still modify the properties of the object, as long as you don't reassign the variable itself.

JavaScript

```
const person = { name: "Bob", age: 25 };
person.age = 26; // This is allowed, as we are modifying a property, not reassigning 'person'.
```

If you need to make the contents of an object truly immutable, you would need to use additional mechanisms like Object.freeze() in JavaScript or design patterns that enforce immutability in other languages.[3]

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/const

3.What is closure?

Closures

*A **closure** is the combination of a function bundled together (enclosed) with references to its surrounding state (the **lexical environment**). In other words, a closure gives a function access to its outer scope. In JavaScript, closures are created every time a function is created, at function creation time.*

Lexical scoping

Consider the following example code:

jsCopy to Clipboard

```
function init() {
  var name = "Mozilla"; // name is a local variable created by init
  function displayName() {
    // displayName() is the inner function, that forms a closure
    console.log(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

init() creates a local variable called name and a function called displayName(). The displayName() function is an inner function that is defined inside init() and is available only within the body of the init() function. Note that the displayName() function has no local variables of its own. However, since inner functions have access to the

variables of outer scopes, displayName() can access the variable name declared in the parent function, init().

If you run this code in your console, you can see that the console.log() statement within the displayName() function successfully displays the value of the name variable, which is declared in its parent function. This is an example of lexical scoping, which describes how a parser resolves variable names when functions are nested. The word lexical refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

Scoping with let and const

Traditionally (before ES6), JavaScript variables only had two kinds of scopes: function scope and global scope. Variables declared with var are either function-scoped or global-scoped, depending on whether they are declared within a function or outside a function. This can be tricky, because blocks with curly braces do not create scopes:

jsCopy to Clipboard

```
if (Math.random() > 0.5) {
  var x = 1;
} else {
  var x = 2;
}
console.log(x);
```

For people from other languages (e.g., C, Java) where blocks create scopes, the above code should throw an error on the console.log line,

because we are outside the scope of x in either block. However, because blocks don't create scopes for var, the var statements here actually create a global variable. There is also a practical example introduced below that illustrates how this can cause actual bugs when combined with closures.

In ES6, JavaScript introduced the let and const declarations, which, among other things like temporal dead zones, allow you to create block-scoped variables.

jsCopy to Clipboard

```
if (Math.random() > 0.5) {
  const x = 1;
} else {
  const x = 2;
}
console.log(x); // ReferenceError: x is not defined
```

In essence, blocks are finally treated as scopes in ES6, but only if you declare variables with let or const. In addition, ES6 introduced modules, which introduced another kind of scope. Closures are able to capture variables in all these scopes, which we will introduce later.

Closure

Consider the following code example:

jsCopy to Clipboard

```
function makeFunc() {
  const name = "Mozilla";
```

```
  function displayName() {
    console.log(name);
  }
  return displayName;
}


const myFunc = makeFunc();
myFunc();
```

Running this code has exactly the same effect as the previous example of the init() function above. What's different (and interesting) is that the displayName() inner function is returned from the outer function before being executed.

At first glance, it might seem unintuitive that this code still works. In some programming languages, the local variables within a function exist for just the duration of that function's execution. Once makeFunc() finishes executing, you might expect that the name variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form closures. A closure is the combination of a function and the lexical environment within which that function was declared. This environment consists of any variables that were in-scope at the time the closure was created. In this case, myFunc is a reference to the instance of the function displayName that is created when makeFunc is run. The instance of displayName maintains a reference to its lexical environment, within which the variable name exists. For this reason,

when myFunc is invoked, the variable name remains available for use, and "Mozilla" is passed to console.log.

Here's a slightly more interesting example—a makeAdder function:

jsCopy to Clipboard

```
function makeAdder(x) {
  return function (y) {
    return x + y;
  };
}

const add5 = makeAdder(5);
const add10 = makeAdder(10);

console.log(add5(2)); // 7
console.log(add10(2)); // 12
```

In this example, we have defined a function makeAdder(x), that takes a single argument x, and returns a new function. The function it returns takes a single argument y, and returns the sum of x and y.

In essence, makeAdder is a function factory. It creates functions that can add a specific value to their argument. In the above example, the function factory creates two new functions—one that adds five to its argument, and one that adds 10.

add5 and add10 both form closures. They share the same function body definition, but store different lexical environments. In add5's lexical environment, x is 5, while in the lexical environment for add10, x is 10.

Practical closures

Closures are useful because they let you associate data (the lexical environment) with a function that operates on that data. This has obvious parallels to object-oriented programming, where objects allow you to associate data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code written in front-end JavaScript is event-based. You define some behavior, and then attach it to an event that is triggered by the user (such as a click or a keypress). The code is attached as a callback (a single function that is executed in response to the event).

For instance, suppose we want to add buttons to a page to adjust the text size. One way of doing this is to specify the font-size of the body element (in pixels), and then set the size of the other elements on the page (such as headers) using the relative em unit:

cssCopy to Clipboardplay

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}
```

```
h1 {
  font-size: 1.5em;
}

h2 {
  font-size: 1.2em;
}
```

Such interactive text size buttons can change the font-size property of the body element, and the adjustments are picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

jsCopy to Clipboardplay

```
function makeSizer(size) {
  return () => {
    document.body.style.fontSize = `${size}px`;
  };
}

const size12 = makeSizer(12);
const size14 = makeSizer(14);
const size16 = makeSizer(16);
```

size12, size14, and size16 are now functions that resize the body text to 12, 14, and 16 pixels, respectively. You can attach them to buttons as demonstrated in the following code example.

jsCopy to Clipboardplay

```
document.getElementById("size-12").onclick = size12;

document.getElementById("size-14").onclick = size14;

document.getElementById("size-16").onclick = size16;
```

htmlCopy to Clipboardplay

```
<button id="size-12">12</button>

<button id="size-14">14</button>

<button id="size-16">16</button>

<p>This is some text that will change size when you click the buttons above.</p>
```

play

Emulating private methods with closures

Languages such as Java allow you to declare methods as private, meaning that they can be called only by other methods in the same class.

JavaScript, prior to classes, didn't have a native way of declaring private methods, but it was possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code. They also provide a powerful way of managing your global namespace.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Note that these closures follow the Module Design Pattern.

jsCopy to Clipboard

```
const counter = (function () {
  let privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }

  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
})();
```

console.log(counter.value()); // 0.

counter.increment();

counter.increment();

console.log(counter.value()); // 2.

counter.decrement();

console.log(counter.value()); // 1.

In previous examples, each closure had its own lexical environment. Here though, there is a single lexical environment that is shared by the three functions: counter.increment, counter.decrement, and counter.value.

The shared lexical environment is created in the body of an anonymous function, which is executed as soon as it has been defined (also known as an IIFE). The lexical environment contains two private items: a variable called privateCounter, and a function called changeBy. You can't access either of these private members from outside the anonymous function. Instead, you indirectly access them using the three public functions that are returned from the anonymous wrapper.

Those three public functions form closures that share the same lexical environment. Thanks to JavaScript's lexical scoping, they each have access to the privateCounter variable and the changeBy function.

jsCopy to Clipboard

function makeCounter() {

let privateCounter = 0;

```
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment() {
      changeBy(1);
    },

    decrement() {
      changeBy(-1);
    },

    value() {
      return privateCounter;
    },
  };
}

const counter1 = makeCounter();
const counter2 = makeCounter();

console.log(counter1.value()); // 0.
```

counter1.increment();

counter1.increment();

console.log(counter1.value()); // 2.

counter1.decrement();

console.log(counter1.value()); // 1.

console.log(counter2.value()); // 0.

Notice how the two counters maintain their independence from one another. Each closure references a different version of the privateCounter variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable. Changes to the variable value in one closure don't affect the value in the other closure.

Note: *Using closures in this way provides benefits that are normally associated with object-oriented programming. In particular, data hiding and encapsulation.*

Closure scope chain

A nested function's access to the outer function's scope includes the enclosing scope of the outer function—effectively creating a chain of function scopes. To demonstrate, consider the following example code.

jsCopy to Clipboard

// global scope

const e = 10;

```
function sum(a) {
  return function (b) {
    return function (c) {
      // outer functions scope
      return function (d) {
        // local scope
        return a + b + c + d + e;
      };
    };
  };
}

console.log(sum(1)(2)(3)(4)); // 20
```

You can also write without anonymous functions:

jsCopy to Clipboard

```
// global scope
const e = 10;
function sum(a) {
  return function sum2(b) {
    return function sum3(c) {
      // outer functions scope
      return function sum4(d) {
```

```
      // local scope
      return a + b + c + d + e;
    };
  };
};
}

const sum2 = sum(1);
const sum3 = sum2(2);
const sum4 = sum3(3);
const result = sum4(4);
console.log(result); // 20
```

In the example above, there's a series of nested functions, all of which have access to the outer functions' scope. In this context, we can say that closures have access to all outer scopes.

Closures can capture variables in block scopes and module scopes as well. For example, the following creates a closure over the block-scoped variable y:

jsCopy to Clipboard

```
function outer() {
  let getY;
  {
    const y = 6;
```

```
  getY = () => y;
 }
 console.log(typeof y); // undefined
 console.log(getY()); // 6
}

outer();
```

Closures over modules can be more interesting.

jsCopy to Clipboard

```
// myModule.js
let x = 5;
export const getX = () => x;
export const setX = (val) => {
 x = val;
};
```

Here, the module exports a pair of getter-setter functions, which close over the module-scoped variable x. Even when x is not directly accessible from other modules, it can be read and written with the functions.

jsCopy to Clipboard

```
import { getX, setX } from "./myModule.js";

console.log(getX()); // 5
```

setX(6);

console.log(getX()); // 6

Closures can close over imported values as well, which are regarded as live bindings, because when the original value changes, the imported one changes accordingly.

jsCopy to Clipboard

// myModule.js

export let x = 1;

export const setX = (val) => {

* x = val;*

};

jsCopy to Clipboard

// closureCreator.js

import { x } from "./myModule.js";

export const getX = () => x; // Close over an imported live binding

jsCopy to Clipboard

import { getX } from "./closureCreator.js";

import { setX } from "./myModule.js";

console.log(getX()); // 1

setX(2);

console.log(getX()); // 2

Creating closures in loops: A common mistake

Prior to the introduction of the let keyword, a common problem with closures occurred when you created them inside a loop. To demonstrate, consider the following example code.

htmlCopy to Clipboardplay

```
<p id="help">Helpful notes will appear here</p>
<p>Email: <input type="text" id="email" name="email" /></p>
<p>Name: <input type="text" id="name" name="name" /></p>
<p>Age: <input type="text" id="age" name="age" /></p>
```

jsCopy to Clipboardplay

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
    // Culprit is the use of `var` on this line
```

```
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function () {
      showHelp(item.help);
    };
  }
}

setupHelp();

play
```

The helpText array defines three helpful hints, each associated with the ID of an input field in the document. The loop cycles through these definitions, hooking up an onfocus event to each one that shows the associated help method.

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to onfocus form closures; they consist of the function definition and the captured environment from the setupHelp function's scope. Three closures have been created by the loop, but each one shares the same single lexical environment, which has a variable with changing values (item). This is because the variable item is declared with var and thus has function scope due to hoisting. The value of item.help is determined when the onfocus callbacks are executed. Because the loop has already run its course by that time, the item variable object (shared by all three closures) has been left pointing to the last entry in the helpText list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier:

jsCopy to Clipboardplay

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function makeHelpCallback(help) {
  return function () {
    showHelp(help);
  };
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
```

```
    document.getElementById(item.id).onfocus =
makeHelpCallback(item.help);
  }
}

setupHelp();
play
```

This works as expected. Rather than the callbacks all sharing a single lexical environment, the makeHelpCallback function creates a new lexical environment for each callback, in which help refers to the corresponding string from the helpText array.

One other way to write the above using anonymous closures is:

```
jsCopy to Clipboard
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];
```

```
  for (var i = 0; i < helpText.length; i++) {
    (function () {
      var item = helpText[i];
      document.getElementById(item.id).onfocus = function () {
        showHelp(item.help);
      };
    })(); // Immediate event listener attachment with the current value of item (preserved until iteration).
  }
}

setupHelp();
```

If you don't want to use more closures, you can use the let or const keyword:

```
jsCopy to Clipboard
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
  const helpText = [
    { id: "email", help: "Your email address" },
```

```
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  for (let i = 0; i < helpText.length; i++) {
    const item = helpText[i];
    document.getElementById(item.id).onfocus = () => {
      showHelp(item.help);
    };
  }
}

setupHelp();
```

This example uses const instead of var, so every closure binds the block-scoped variable, meaning that no additional closures are required.

Another alternative could be to use forEach() to iterate over the helpText array and attach a listener to each <input>, as shown:

jsCopy to Clipboard

```
function showHelp(help) {
  document.getElementById("help").textContent = help;
}

function setupHelp() {
```

```
  var helpText = [
    { id: "email", help: "Your email address" },
    { id: "name", help: "Your full name" },
    { id: "age", help: "Your age (you must be over 16)" },
  ];

  helpText.forEach(function (text) {
    document.getElementById(text.id).onfocus = function () {
      showHelp(text.help);
    };
  });
}

setupHelp();
```

Performance considerations

As mentioned previously, each function instance manages its own scope and closure. Therefore, it is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is

called, the methods would get reassigned (that is, for every object creation).

Consider the following case:

jsCopy to Clipboard

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function () {
    return this.name;
  };

  this.getMessage = function () {
    return this.message;
  };
}
```

Because the previous code does not take advantage of the benefits of using closures in this particular instance, we could instead rewrite it to avoid using closures as follows:

jsCopy to Clipboard

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
```

```
MyObject.prototype = {
  getName() {
    return this.name;
  },
  getMessage() {
    return this.message;
  },
};
```

However, redefining the prototype is not recommended. The following example instead appends to the existing prototype:

jsCopy to Clipboard

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function () {
  return this.name;
};
MyObject.prototype.getMessage = function () {
  return this.message;
};
```

In the two previous examples, the inherited prototype can be shared by all objects and the method definitions need not occur at every object creation. See Inheritance and the prototype chain for more.[4]

[4] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

4.What are scopes in js?

Scope

*The **scope** is the current context of execution in which values and expressions are "visible" or can be referenced. If a variable or expression is not in the current scope, it will not be available for use. Scopes can also be layered in a hierarchy, so that child scopes have access to parent scopes, but not vice versa.*

JavaScript has the following kinds of scopes:

- ***Global scope:** The default scope for all code running in script mode.*
- ***Module scope:** The scope for code running in module mode.*
- ***Function scope:** The scope created with a function.*

In addition, identifiers declared with certain syntaxes, including let, const, class, or (in strict mode) function, can belong to an additional scope:

- *Block scope: The scope created with a pair of curly braces (a block).*

A function creates a scope, so that (for example) a variable defined exclusively within the function cannot be accessed from outside the function or within other functions. For instance, the following is invalid:

jsCopy to Clipboard

```
function exampleFunction() {
  const x = "declared inside function"; // x can only be used in exampleFunction
  console.log("Inside function");
```

```
  console.log(x);
}

console.log(x); // Causes error
```

However, the following code is valid due to the variable being declared outside the function, making it global:

jsCopy to Clipboard

```
const x = "declared outside function";

exampleFunction();

function exampleFunction() {
  console.log("Inside function");
  console.log(x);
}

console.log("Outside function");
console.log(x);
```

Blocks only scope let and const declarations, but not var declarations.

jsCopy to Clipboard

```
{
  var x = 1;
```

```
}
console.log(x); // 1
jsCopy to Clipboard
{
  const x = 1;
}
console.log(x); // ReferenceError: x is not defined[5]
```

[5] https://developer.mozilla.org/en-US/docs/Glossary/Scope

5.What is hoisting in js?

Hoisting

*JavaScript **Hoisting** refers to the process whereby the interpreter appears to move the declaration of functions, variables, classes, or imports to the top of their scope, prior to execution of the code.*

Hoisting is not a term normatively defined in the ECMAScript specification. The spec does define a group of declarations as HoistableDeclaration, but this only includes function, function, async function, and async function* declarations. Hoisting is often considered a feature of var declarations as well, although in a different way. In colloquial terms, any of the following behaviors may be regarded as hoisting:*

1. *Being able to use a variable's value in its scope before the line it is declared. ("Value hoisting")*
2. *Being able to reference a variable in its scope before the line it is declared, without throwing a ReferenceError, but the value is always undefined. ("Declaration hoisting")*
3. *The declaration of the variable causes behavior changes in its scope before the line in which it is declared.*
4. *The side effects of a declaration are produced before evaluating the rest of the code that contains it.*

The four function declarations above are hoisted with type 1 behavior; var declaration is hoisted with type 2 behavior; let, const, and class declarations (also collectively called lexical declarations) are hoisted with type 3 behavior; import declarations are hoisted with type 1 and type 4 behavior.

Some prefer to see let, const, and class as non-hoisting, because the temporal dead zone strictly forbids any use of the variable before its declaration. This dissent is fine, since hoisting is not a universally-agreed term. However, the temporal dead zone can cause other observable changes in its scope, which suggests there's some form of hoisting:

jsCopy to Clipboard

```
const x = 1;

{
  console.log(x); // ReferenceError
  const x = 2;
}
```

If the const x = 2 declaration is not hoisted at all (as in, it only comes into effect when it's executed), then the console.log(x) statement should be able to read the x value from the upper scope. However, because the const declaration still "taints" the entire scope it's defined in, the console.log(x) statement reads the x from the const x = 2 declaration instead, which is not yet initialized, and throws a ReferenceError. Still, it may be more useful to characterize lexical declarations as non-hoisting, because from a utilitarian perspective, the hoisting of these declarations doesn't bring any meaningful features.

Note that the following is not a form of hoisting:

jsCopy to Clipboard

```
{
  var x = 1;
}
```

console.log(x); // 1

There's no "access before declaration" here; it's simply because var declarations are not scoped to blocks.[6]

[6] https://developer.mozilla.org/en-US/docs/Glossary/Hoisting