Lecture 014 | JavaScript Interview Prepration

Note : `JavaScript is a Synchronous Single Threaded Language`

Execution Context in Js

> Execution context is the environment in which the JavaScript code is executed. It is the place where all the variables and functions are defined.

Global Execution Context

> The global execution context is the execution context in which the code is executed when it is loaded in the browser. The global execution context is also called the global object.

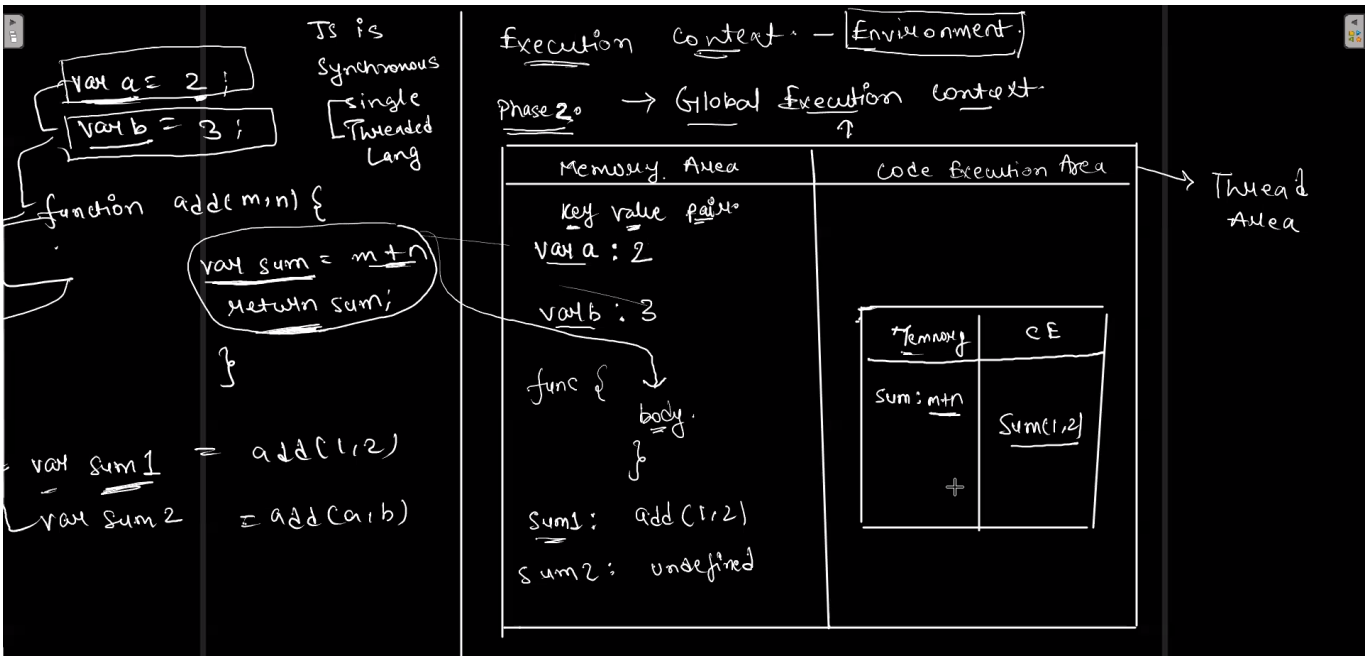
Phases in Execution Context (Code Execution)

> 1. Creation of Execution Context - Memory allocation and initialization of variables and functions.
> 2. Execution of Code - Execution of the code.
> 3. Destruction of Execution Context - Memory deallocation (Garbage Collection)

Execution Context explained w/ Code

```
var a = 2;
var b = 3;
function add(m, n) {
  var sum = m + n;
  return sum;
}
var sum1 = add(1, 2);
var sum2 = add(a, b);
console.log(sum1);
console.log(sum2);
```

Execution Context explained w/o Code



Example of Execution Context : Square Function

```
var n = 2;
function square(n) {
  return n * n;
}
var sq1 = square(n);
var sq2 = square(2);
console.log(sq1);
console.log(sq2);
```

Flow in Execution Context

```
first the global execution context is created
then the function is executed
then the local execution context is created
then the local execution context is destroyed
then the global execution context is destroyed
```

Resources to read about Execution Context

Blog 1

Hoisting : Hoisting is JavaScript's default behavior of moving declarations to the top.

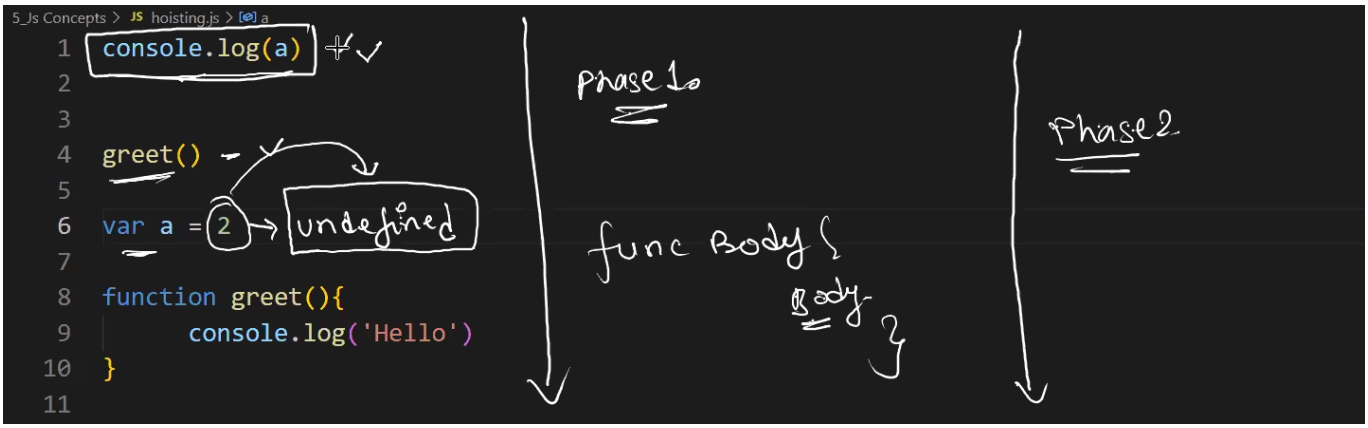
> 1. Hoisting is the process of moving declarations to the top of the current scope.
> 2. Its is relative to the current execution context.
> 3. It is a JavaScript feature that allows you to declare a variable before you use it.

Hoisting in JavaScript

```
console.log(a);
var a = 2;
greet();
function greet() {
  console.log("Hello!");
}

// ⨯> ~/D/g/pepcoding-webdev on main × node lecture-014/hoisting.js
// undefined
// Hello!
```

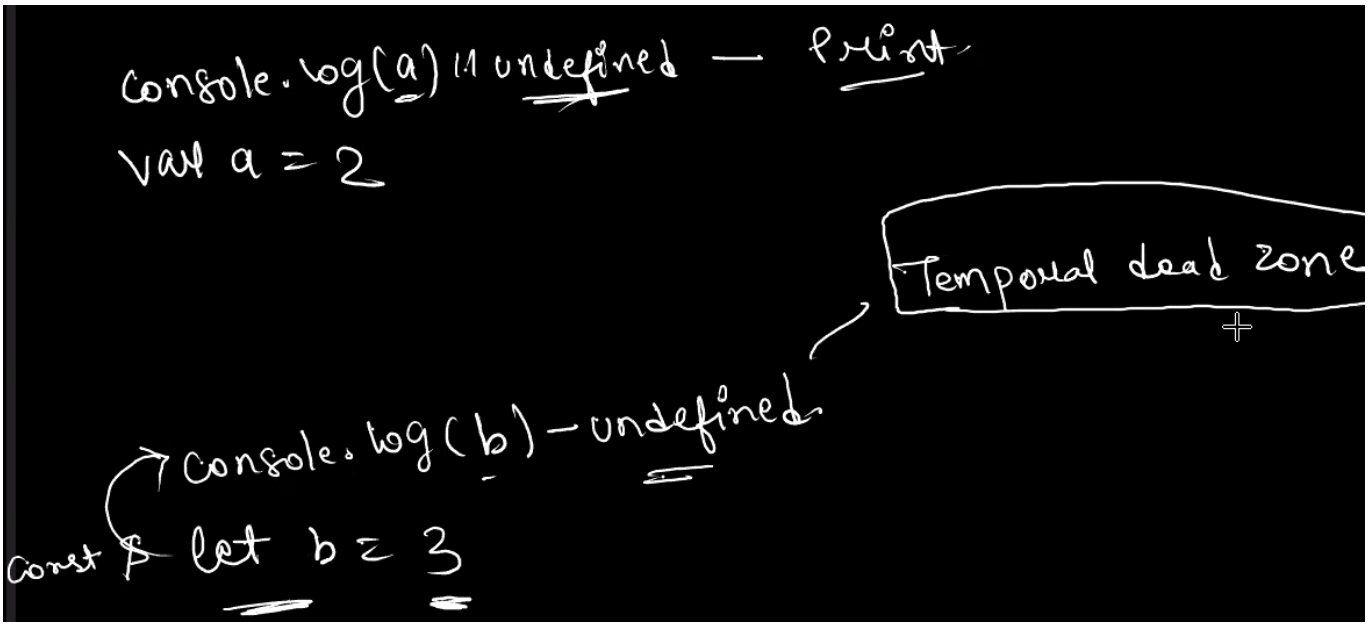
Understanding Hoisting | Example (Illustrated with Execution Context in Phase 1 and Phase 2)



Temporal Dead Zone | Good Practices

> 1. Good practice is to avoid using the temporal dead zone. (Scoping rules, accessablity, etc.)
> 2. Temporal dead zone is a feature of JavaScript that allows you to declare a variable before you use it.
> 3. `let` and `const` are not hoisted and when accessed before declaration, it throws an error.

```
console.log(a); // undefined -> a is not defined (not hoisted)
const a = 2;
```



> The thing with `var` is that it is hoisted. It was introduced in `ES5`, which was before `let` and `const`.

Resources to read about Temporal Dead Zone

Temporal Dead Zone

Data Types in JavaScript

> There are 2 categories of data types in JavaScript
>
> Primitive and Reference types.

Primitive types are:

> 1. Number : `1`, `1.5`, `0`, `-1`, `-1.5`, `Infinity`, `-Infinity`, `NaN`
> 2. String : `"Hello"`, `'Hello'`, `"1"`, `'1'`
> 3. Boolean : `true`, `false`
> 4. Symbol : `Symbol()`
> 5. Undefined : `undefined`
> 6. Null : `null`
> 7. NaN : `NaN`

Reference types are:

> 1. Object : `{}`, `new Object()`
> 2. Array : `[]`, `new Array()`
> 3. Function : `function() {}`
> 4. Date : `new Date()`
> 5. RegExp : `/\w+/`
> 6. Error : `new Error()`
> 7. Map : `new Map()`
> 8. Set : `new Set()`
> 9. WeakMap : `new WeakMap()`

> 10. WeakSet : `new WeakSet()`

Excercise : `Data Types`

```
let a = "Hello";
console.log(a, typeof a);

let b = 2.5;
console.log(b, typeof b);

let c = true;
console.log(c, typeof c);

let d = undefined;
console.log(d, typeof d);

let e = null; // a bug in js which hasnt been fixed yet
console.log(e, typeof e); // null is a primitive type still has a typeof
object
```

```
⨯> ~/D/g/pepcoding-webdev on main × node lecture-014/datatypes.js
Hello string
2.5 number
true boolean
undefined undefined
null object
```

> A good read/research on null being a typeof object in JavaScript : the `missing object`

Refrence Data Types in JavaScript

```
let arr = [1, 2, 3];

console.log(arr, typeof arr);

let obj = {
  name: "John",
  age: 30,
};

console.log(obj, typeof obj);

function greet(name) {
  console.log("Hello", name);
}

console.log(greet, typeof greet);
```

```
> ~/D/g/pepcoding-webdev on main × node lecture-014/refrenceDataType.js
[ 1, 2, 3 ] object
{ name: 'John', age: 30 } object
[Function: greet] function
```

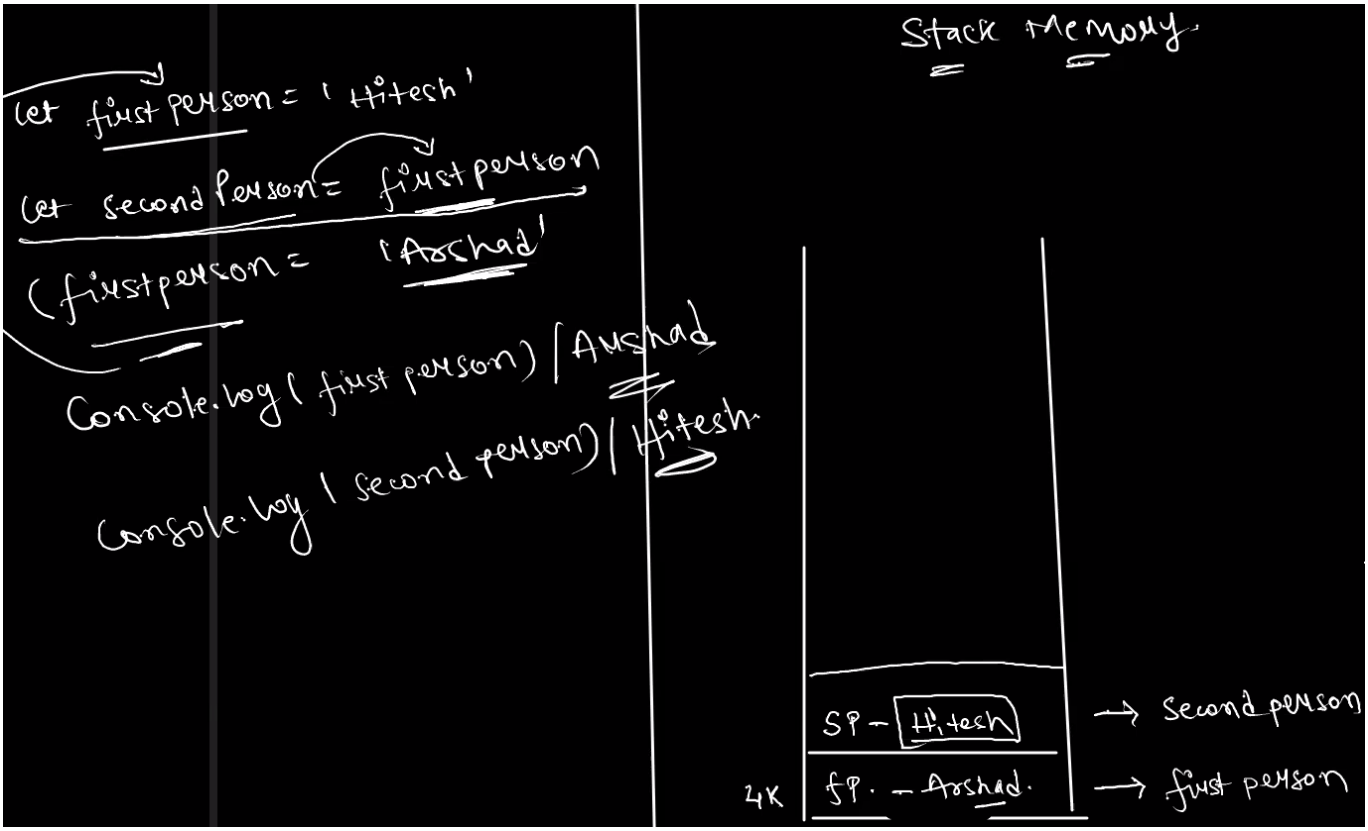
1. Primitive Datatypes gets stored in `Stack Memory`.
2. Refrence Datatypes gets stored in `Heap Memory`.

Excercise : Primitive Data Types & Stack Memory | Access by value

```
let firstPerson = "Hitesh";
let secondPerson = firstPerson; // copy
firstPerson = "Arshad"; // change
console.log(firstPerson, secondPerson); // Hitesh Arshad
```

```
⨯> ~/D/g/pepcoding-webdev on main × node lecture-014/refrenceDataType.js
Hitesh Arshad
```

Illustrating access by value : `firstPerson` is a copy of `secondPerson`



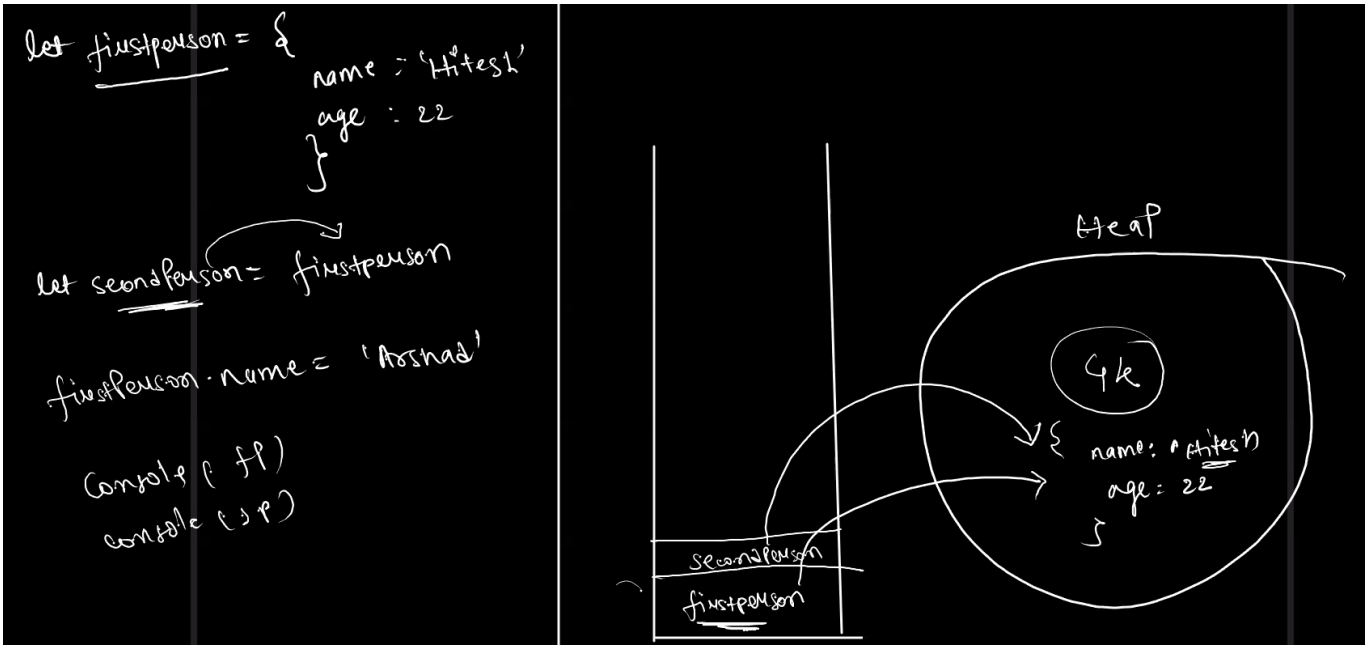
Stack is `LIFO (Last In First Out)`

Refrence Data Types & Concept of `access by reference`

```
let fp = {
  name: "Hitesh",
  age: 30,
};
let sp = fp; // copy
fp.name = "Arshad"; // change
console.log(fp, sp); // { name: 'Arshad', age: 30 } { name: 'Arshad', age:
30 }
```

```
⨯> ~/D/g/pepcoding-webdev on main ◦ node lecture-014/refrenceDataType.js
{ name: 'Arshad', age: 30 } { name: 'Arshad', age: 30 }
```

Refrence Datatypes use `Heap Memory` (Access by reference)



1. Solutions to the above problem : `spread operator`, `object.assign`, `deep copy`, `shallow copy`
2. `spread operator` : `...`
3. `object.assign` : `Object.assign()`
4. `deep copy` : `JSON.parse(JSON.stringify(obj))`
5. `shallow copy` : `Object.assign({}, obj)`

1. Altough Js is synchronous, it is possible to make asynchronous calls in JavaScript.
2. AJAX is an example of asynchronous call.
3. AJAX stands for Asynchronous JavaScript and XML.
4. Callbacks, Promises and Async/Await are some of the ways to make asynchronous calls in JavaScript.