breaking thought barriers

Angular Training

Session -2

Outlines

- Javascript Array
- Let,const,var

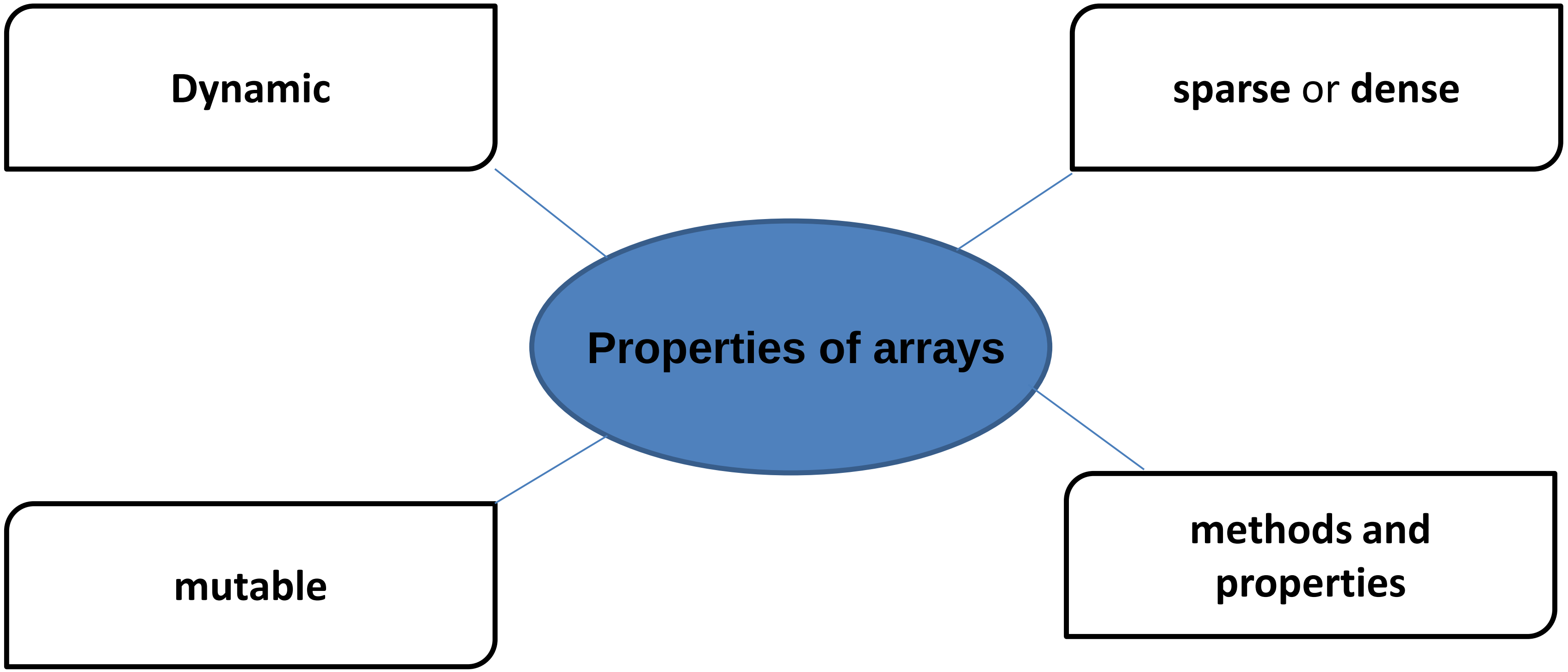
Array in JAVASCRIPT

Arrays

Arrays are *objects* with *methods* and *properties* suitable for managing data items in an orderly fashion.

They are list-like data structures that group data together, making them accessible through numerical indices.

```
var arr = [ 1, 2, 3, 34, 67, 122 ];
```


Properties of arrays
Dynamic
sparse or dense
mutable
methods and properties

Dense arrays

-- A dense array, also referred to as a contiguous array, stores elements in a sequential manner in memory.

-- A dense array is an array where the elements are all sequential starting at index 0.

```
let array = [1, 2, 3, 4, 5] array.length // Returns 5
```

Sparse array

-- A sparse array is one in which the elements are not sequential, and they don't always start at 0.

-- They are essentially Arrays with "holes", or gaps in the sequence of their indices.

```
let array = []; array[100] = "Holes now exist"; array.length // 101, but only 1 element
```

Different ways to create a Sparse array:

1. Using Array Literal

```
const arr_name = [item1,item2, , , item3];
```

2. Using Array Constructor

```
const arr_name = new Array(arraySize);
```

3. Delete any Array Item:

```
const arr_name = [item1,item2,item3];

delete arr_name[index];
```

4. By increasing the Length of Array

```
const arr_name = [item1,item2,item3];
arr_name.length = requiredLength;
```

Declaring and Mutating Arrays

1. Declaring empty arrays

```
var arr = new Array();

var arr2 = [];

var arr3 = new Array(12)
```

2.Declaring arrays with elements

```
var arr1 = new Array(1, -2, "3");
var arr2 = [4, 5, "6", true];
```

3. Accessing elements in an array

4. Array mutation

Since arrays are *objects*, they are mutable. By definition, their values are updated on *assignments*.

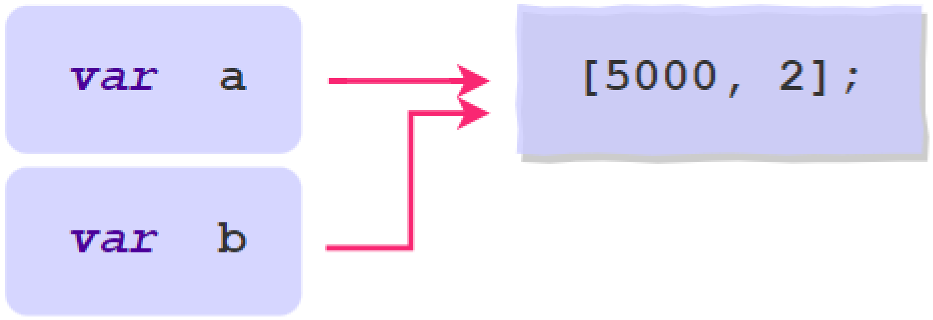
```
var arr1 = [1, 2, 3, 4];
console.log(arr1);
arr1[0] = 5;
console.log(arr1);
```

4. Array mutation

```
var arr1 = [1, 2, 3, 4];
var arr2 = arr1;
arr2[0] = 5;
console.log(arr1,arr2);
```

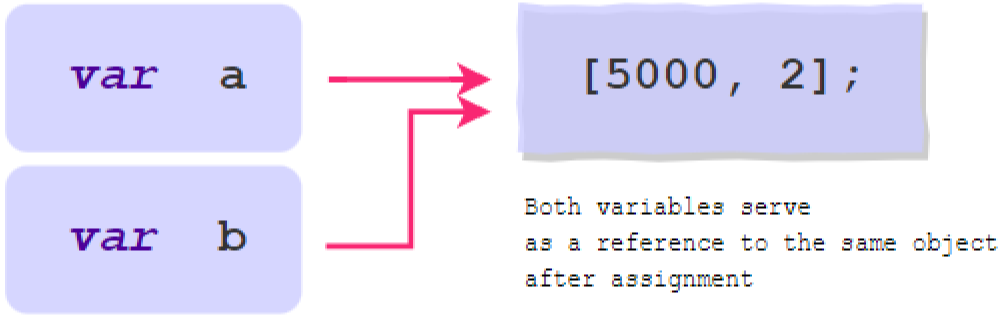
4. Array mutation

```
var a  =  [5000, 2];
var b  =  a;
```



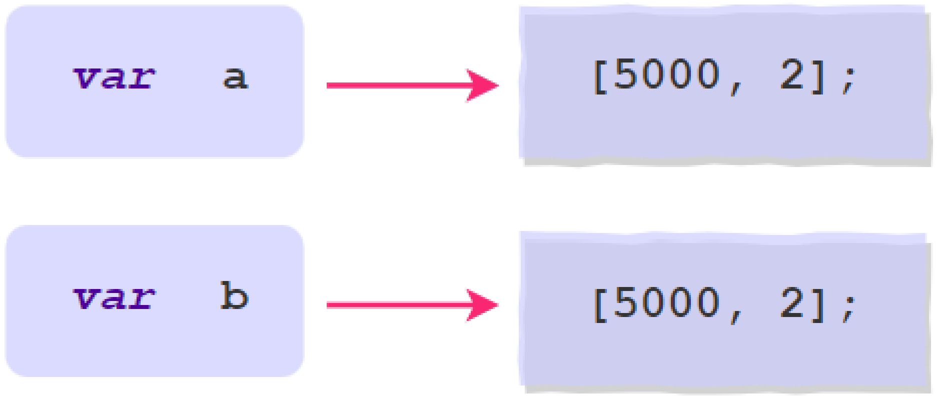
4. Array mutation

```
var a  =  [5000, 2];
var b  =   a;
```



Both variables serve
as a reference to the same object
after assignment

4. Array mutation



Both variables reference different array objects both with similar values

5. Copying arrays

To create an actual copy and assign it to a variable, use the following:

```
= arr.slice()

= [...arr]

= Array.from(arr)
```

5. Copying arrays

```
var arr1 = [1, 2, 3, 4];
var arr2 = arr1.slice();
var arr3 = [...arr1];
var arr4 = Array.from(arr1);
arr2[0] = 5;
arr3[0] = 6;
arr4[0] = 7;
console.log(arr1, arr2, arr3, arr4);
```

6. Array Methods and Properties

Properties of an array

The **property** of an array is a quality or attribute of the array.

- ➢ **length**
- ➢ **Prototype**

The prototype property is the property of the global Array constructor itself.

It allows you to add custom properties and methods to the global Array.

NOTE: The prototype property is available for all objects in JavaScript.

flat()

Methods in Array : (39)

at	filter	flatMap	lastIndexOf	shift	toSpliced
concat	find	forEach	map	slice	toString
copyWithin	findIndex	includes	pop	some	unshift
entries	findLast	indexOf	push	sort	values
every	findLastIndex	join	reduce	splice	with
fill	flat	keys	reduceRight	toLocaleString	Array
			reverse	toReversed	toSorted

7. Methods of an array

push() and pop()

unshift() and shift()

toString()

8. Replication with Array Methods

slice()

The slice() method gets a copy of the array.

This method can *optionally* pass values to specify the portion of the array to copy.

By default, if no indices are specified, it copies the entire array.

This method does not make any changes to the original array.

```
var arr = [1, 2, 3, 4]; // Assign an array of elements to arr

// Print array along with it's type
console.log('Before Slice:');
console.log('arr:',arr);

// Call toString method and assign to str_arr
var arr1 = arr.slice();
var arr2 = arr.slice(1);
var arr3 = arr.slice(1,1);
var arr4 = arr.slice(0, -1);

// Print the type and values of arr and str_arr after calling toString
method
console.log('After Slice:');
console.log(`original arr: ${arr}`);
console.log('arr1:', arr1);
console.log('arr2:', arr2);
console.log('arr3:', arr3);
console.log('arr4:', arr4);
```

8. Replication with Array Methods

splice()

The splice() method is a method used to add or remove a sub-array of elements from the array.

The splice() method returns the removed elements from the array after modifying the original array.

```
var arr = [1, 2, 3, 4]; // Assign an array of elements to arr
console.log("intial arr:", arr);

// Add zero to list without deleting
var add = arr.splice(0 , 0, 0);
console.log("arr after addition:",arr);
console.log("elements removed in addition:",add);

// remove 3 elements from index 1 onwards
var remove = arr.splice(1, 3);
console.log("arr after removal:",arr);
console.log("elements removed in removal:",remove);

// remove 1 element from index 1 and add 1, 2 and 3
var replace = arr.splice(1, 1, 1, 2, 3);
console.log("arr after replacement:",arr);
console.log("elements removed in replacement:",replace);

console.log("arr:",arr);
```

8. Replication with Array Methods

concat()

The concat() method merges two arrays and returns a copy of the merged arrays.

```
var arr1 = [1, 2, 3]; // Assign an array of 3 elements to arr1
var arr2 = [4, 5]; // Assign an array of 2 elements to arr2
// Print the arrays
console.log("initial arrays:");
console.log(arr1);
console.log(arr2);
// Merge arr1 and arr2 and assign to corresponding variables
var merged1 = arr1.concat(arr2);
var merged2 = arr2.concat(arr1);
var merged3 = Array.prototype.concat(arr1, arr2);
// Print each array after merging
console.log("After concat:");
console.log("arr1:", arr1, "arr2:", arr2);
console.log("merged1:", merged1);
console.log("merged2:", merged2);
console.log("merged3:", merged3);
```

Callbacks

- ❑ Callbacks are functions that are passed to other functions as parameters.
- ❑ They're necessary for asynchronous code, encapsulation, eliminating code repetition, and so much more.
- ❑ One of the most powerful properties of JavaScript is that functions are first-class objects.
- ❑ This means that they are like any other object and have the same properties as standard objects.
- ❑ In fact, we should think of them as nothing more than callable objects.

```javascript
function fnGenerator() {
    return function() {
        console.log('Ran the inner function');
    }
}

const fnReturned = fnGenerator();
console.log(fnReturned); // -> [Function]
fnReturned(); // -> Ran the inner function
```

We can also pass functions *in* to other functions.

```
function fnCaller(fn) {
    fn();
}

function log() {
    console.log('Calling log');
}

fnCaller(log); // -> Calling log
```

❑ Callback functions allow us to do interesting things with our code. Passing functions as parameters and returning functions allow us to:

- Eliminate code repetition
- Allow asynchronous execution
- Allow event binding, such as with user interaction
- Hide data and create a pleasant user interface
- Prevent scope pollution

```
function multiply(x, y) {
    return x * y;
}

function callMultiply(fn, val1, val2)
{
    // Your code here
}
```

Map() Method :

- The map() method is used to get a modified version of the array using callback functions
- map() applies a function to each array element and creates a new array of the returned values.
- The syntax of the method is as follows:

```
array.map(function( currentValue, index, arr), thisValue )
```

- The map() method accepts two parameters:

function(currentValue, index, arr):
This is a required parameter that runs on each element of array.
It contains three parameters: currentValue, index and arr.

thisValue: This parameter is optional. It holds the value of passed to the function.

- The function that is be passed is given arguments by the map method in the following order

```
function callbackfn (value: any, index: number, array: any[ ])
```

Example

```
let circles = [ 10, 30, 50 ];
```

1. Calculate the area of each circle and push the result into a new array.

```
let areas = []; // to store areas of circles
let area = 0;
for (let i = 0; i < circles.length; i++) {
    area = Math.floor( Math.PI * circles[ i ] * circles[ i ]);
    areas.push(area);
}
console.log(areas);
```

- Starting from ES5, JavaScript Array type provides the map() method that allows you to transform the array elements in a cleaner way.

```
function circleArea(radius) {
    return Math.floor(Math.PI * radius * radius);
}
let areas = circles.map(circleArea);
console.log(areas);
```

- Shorten the code using Anonymous Function

```
let areas = circles.map(function(radius){
    return Math.floor(Math.PI * radius * radius);
});
console.log(areas);
```

- Using arrow function in ES6

```
let areas = circles.map(radius => Math.floor(Math.PI * radius * radius));
console.log(areas);
```

Example

```
var arr = [10, 20, 30, 40, 50] ;

arr1 = arr . map(a => a * 2);
console.log(arr);
console.log("doubled array:",arr1);
```

Uses of Map( )

A. Generic use of map()

```
let map = Array.prototype.map
let a = map.call('Hello World', function(x) {
  return x.charCodeAt(0)
})
```

Example 1 : Mapping an array of numbers to an array of their square roots

```javascript
let numbers = [3, 25, 100]
let roots =
numbers.map(function(num) {
    return Math.sqrt(num)
})
```

Example 2: Mapping an array of numbers with a function containing an argument

```javascript
let numbers = [3, 25, 100]
let doubles =
numbers.map(function(num) {
  return num * 2
})
```

Example 3 : Converting Temperatures

```
const celsiusTemperatures = [0, 10, 20, 30, 40];
const fahrenheitTemperatures = celsiusTemperatures.map(
celsius => ( celsius * 9/5) + 32);
console.log(fahrenheitTemperatures);
```

Example 4: Capitalizing Names

```
const names = ['john', 'jane', 'alex', 'emily'];
const capitalizedNames = names.map(name =>
name.charAt(0).toUpperCase() + name.slice(1));
console.log(capitalizedNames);
```

Example 5 : Extracting Property Values

```
const books = [
  { title: 'The Great Gatsby', author: 'F. Scott Fitzgerald' },
  { title: 'To Kill a Mockingbird', author: 'Harper Lee' },
  { title: '1984', author: 'George Orwell' },
];

const bookTitles = books.map(book => book.title);

console.log(bookTitles);
```

Example 5 : Adding Unique IDs

```
const data = [
  { name: 'Alice', age: 25 },
  { name: 'Bob', age: 30 },
  { name: 'Charlie', age: 22 },
];

const newData = data.map((item, index) => ({ ...item, id: index + 1 }));

console.log(newData );
```

Cases where map() function should not be used

1. When You Don't Need a New Array

2. When You Need Conditional Transformations

3. When You Need Side Effects

forEach()

A method that is very similar to a for loop.
For every element found in the array, the forEach() method executes a callback function on it.

```
var arr = [6, 4, 5, 6, 7, 7];
arr.forEach(function(element) {
 console.log(element * 2);
})
```

reduce() Method

- ✓ The reduce method reduces the array to a single value from left to right.
- ✓ This method leaves the original array unchanged.
- ✓ The syntax of the method is as follows:

```
arr.reduce(<function>);
```

- ✓ The reduce method gives arguments to the passed function in the following order:

```
function callbackfn( prev : any, curr : any, index: number, array: number[ ])
```

- ✓ For each element, the **callbackfn** will be passed with the previous callbackfn function's return value as the first argument, and the value of the element as the second argument.

- ✓ If the array has only one value, that value is returned. For an empty array, an error is thrown.

```
var arr = [10, 20, 30, 40, 50];
var val = arr.reduce((prev, curr) => prev + curr);
console.log("arr:",arr);
console.log("reduced val:", val);
```

uses of reduce :

Example -1 : Sum the values of an array :

```
let sum = [0, 1, 2, 3].reduce(function (accumulator,currentValue)
{
  return accumulator + currentValue
}, 0)
```

Example -2 : Flatten an array of arrays

```
let flattened = [[0, 1], [2, 3], [4, 5]].reduce( function(accumulator, currentValue)
{
   return accumulator.concat(currentValue)
}, [ ] )
```

uses of reduce :

Example -3 : Finding Maximum Value:

```
const numbers = [8, 2, 5, 10, 3];
const max = numbers.reduce((maxValue, currentValue) =>
Math.max(maxValue, currentValue), -Infinity);
console.log(max); // Output: 10
```

Example -4 : Flatten an array of arrays

```
let flattened = [[0, 1], [2, 3], [4, 5]].reduce( function(accumulator, currentValue)
{
    return accumulator.concat(currentValue)
}, [ ] )
```

Example-5 : Group objects by a property

```
let people = [
  { name: 'Matt', age: 25 },
  { name: ‘Asma ', age: 23 },
  { name: ‘ Cami ', age: 29 },
  { name: ‘ Alok', age: 23}
];

function groupBy ( objectArray , property) {
  return  objectArray.reduce(function (acc, obj) {
    let key = obj [property]
    if (!acc [key]) {
      acc [key] = []
    }
    acc[key].push(obj)
    return acc
  }, {})
}

let groupedPeople = groupBy(people, 'age')
```

Example-6 : String Concatenation

```
const words = ['Hello', ' ', 'World', '!'];

const concatenatedString = words.reduce((accumulator, currentValue) => accumulator + currentValue, '');

console.log( concatenatedString ) ; // Output: "Hello World!"
```

Example-7 : Counting Occurrences

```
const fruits = ['apple', 'banana', 'apple', 'orange', 'banana', 'apple'];
const fruitCount = fruits.reduce((count, fruit) => {
  count[fruit] = (count[fruit] || 0) + 1;
  return count;
}, {});
console.log( fruitCount);
```

Using reduce and map() together

we need to count array elements that satisfy a certain condition

1. Map the array into an array of zeros and ones.
2. Reduce the array of zeros and ones into the sum.

```
var arr = ['Hello', 1, true, NaN, 'Bye'];
var countArr = arr.map( ele => typeof ele === 'string' ? 1 : 0);
var sum = countArr.reduce((prev, curr)=> prev + curr);
console.log("arr:",arr);
console.log("array from map:", countArr);
console.log("number of Strings:",sum);
```

Assignment

Given an array of objects representing products with name, price, and quantity properties, calculate the total cost of all products that have a price greater than $20.

```
const products = [
  { name: 'Laptop', price: 1000, quantity: 2 },
  { name: 'Phone', price: 500, quantity: 3 },
  { name: 'Tablet', price: 300, quantity: 4 },
  { name: 'Headphones', price: 50, quantity: 5 },
];
```

reduceRight()

The reduceRight() method is just like reduce() except that it iterates from right to left instead of left to right.

```
var arr = [6, 4, 5, 6, 7, 7];
var reduced = arr.reduce(function(curr, next) {
 return curr + next;
}, 0);
var reducedRight = arr.reduceRight(function(curr, next) {
 return curr + next;
}, 0)
console.log(reduced);
console.log(reducedRight);
```

filter()

If the element passes the test, you push that element to a new array.

```
var arr = [6, 4, 5, 6, 7, 7];
var x = arr . filter(function(element) {
 return element%2==0;
})
```

every()

The every() array method checks to see if every single element in the array satisfies the condition you have passed it.

```
var arr = [6, 4, 5, 6, 7, 7];
arr.every(function(element) {
 return element % 2 === 0; //checks to see if even
}); // false
```

some()

The some() method is almost exactly like the every() method, with the exception that it checks to see if at least one element satisfies the condition you have set for it.

```
var arr = [6, 4, 5, 6, 7, 7];

arr.some(function(element) {
 return element % 2 === 0; //checks to see if even
}); //true
```

indexOf() and lastIndexOf()

- ✓ If you need to search for a particular element in an array, you can do that with indexOf() and lastIndexOf().
- ✓ indexOf() returns the first index of the search parameter if it’s found, otherwise it returns a -1.
- ✓ In lastIndexOf(), it gives us the last index of the search element in the array. Again, if not found, it’ll return -1.

isArray()

- ✓ This method checks to see if the object passed to it is an array or not. Returns a boolean.

var, let and const

Earlier before 2015, the only keyword available to declare variables was the var keyword.

Variables in JavaScript:

Variable allows us to store, retrieve and change the stored information.

Variable Scopes in JavaScript:

The scope of a variable is nothing but the area or the region of the program in which it is defined.

var : Function in which the variable is declared
Let : Block in which the variable is declared
const: Block in which the variable is declared

Global Scope

Private Scope

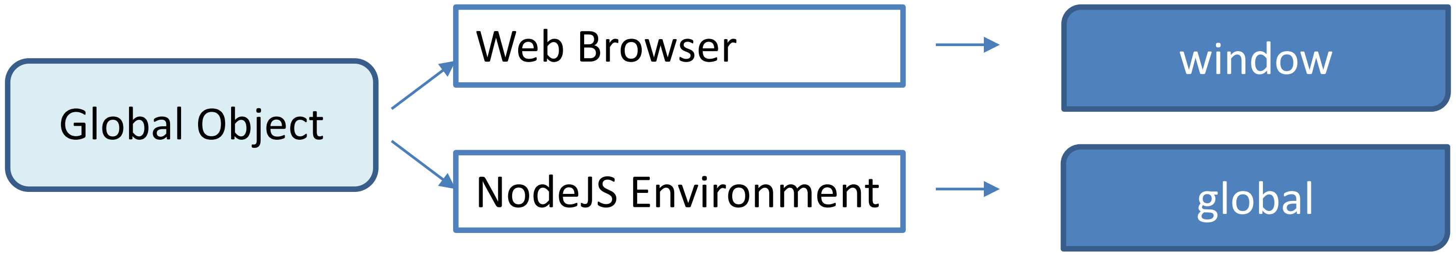
The var variables belong to the global scope when you define them outside a function.

When you declare a variable inside a function using the var keyword, the scope of the variable is local.

```
for (var i = 0; i < 5; i++)
{
   console.log("Inside the loop:", i);
}

console.log("Outside the loop:", i);
```

The global var variables are added to the global object as properties.



However, the let variables are not added to the global object

The var keyword allows you to redeclare a variable without any issue , However, if you redeclare a variable with the let keyword, you will get an error.

The let variables have temporal dead zones while the var variables don’t.

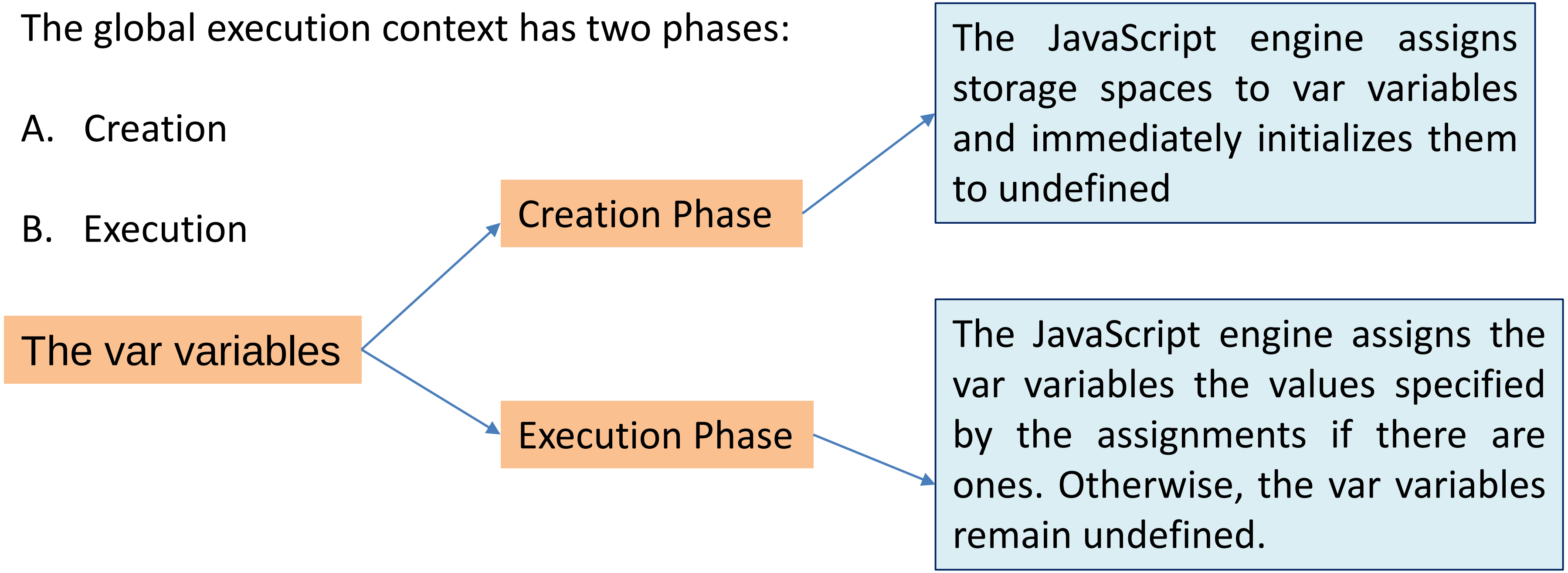
temporal dead zone

When the JavaScript engine executes the JavaScript code, it creates the global execution context.

The global execution context has two phases:

A. Creation

B. Execution



The JavaScript engine assigns storage spaces to var variables and immediately initializes them to undefined

The JavaScript engine assigns the var variables the values specified by the assignments if there are ones. Otherwise, the var variables remain undefined.

temporal dead zone

The let variables

Creation Phase

the JavaScript engine assigns storage spaces to the let variables but does not initialize the variables.

Referencing uninitialized variables will cause a ReferenceError.

Execution Phase

The JavaScript engine assigns the let variables the values specified by the assignments if there are ones. Otherwise, the let variables remain undefined.

The temporal dead zone starts from the block until the let variable declaration is processed. In other words, it is the location where you cannot access the let variables before they are defined.

Lexical Scoping in JavaScript:

Lexical Scope means depending on the position where it is declared, its scope is defined. It does not look whether it is declared inside the function body or outside the function body.

Lexical has two scopes i.e. **Global** and **Local** Scopes.

use strict

Rules of Thumb:

- ✓ Don’t use var, because let and const is more specific
- ✓ Default to const, because it cannot be re-assigned or re-declared
- ✓ Use let when you want to re-assign the variable in future
- ✓ Always prefer using let over var and const over let

What is Javascript hoisting?

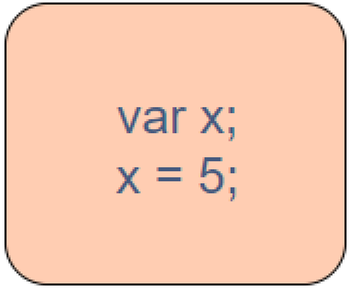
When the JavaScript engine executes the JavaScript code, it creates the global execution context. The global execution context has two phases: **Creation,Execution**

During the creation phase, the JavaScript engine moves the variable and function declarations to the top of your code. This is known as hoisting in JavaScript.

Hoisting variables

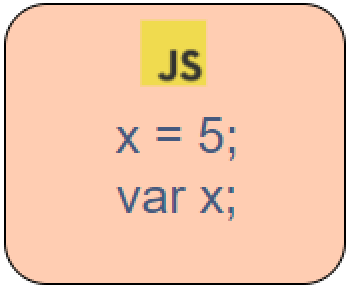
JS hoists the declaration of a variable up and allows it to be used. This way, a variable can be declared after it has been used.

A typical programming case



```
var x;
x = 5;
```

Variable hoisting in JS



```
x = 5;
var x;
```

Hoisting functions

```
let x = 20,
  y = 10;

let result = add(x, y);
console.log(result);

function add(a, b) {
  return a + b;
}
```

```
let x = 20,
  y = 10;

let result = add(x, y);
console.log(result);

let add = function(x, y) {
    return x + y;
}
```

Const Keyword

```
const CONSTANT_NAME = value;
```

ES6 provides a new way of declaring a constant by using the const keyword.

The const keyword creates a read-only reference to a value.

By convention, the constant identifiers are in uppercase.

Like the let keyword, the const keyword declares blocked-scope variables. However, the block-scoped variables declared by the const keyword can’t be reassigned.

you need to initialize the value to the variable declared by the const keyword.

Thank You