Value vs Reference value

Javascript is a complex Language.

1. Value vs Reference Variable Assignment.

Javascript always assigns variables by value.

* When assigned value is one of the javascript's Five primitives types

Boolean, null, Undefined, String, Number

* When assigned valve is an array, functions or object, a reference to the object is assigned.

Closures

Closures are a fundmental concept in javascript that allow a function to access variables from its outer scope even after that scope has closed.

Closures are powerful tools that enable many advanced programming techniques in javascript, from function factories and private state management to asynchronous programming and callBacks.

Basic Closure Example

```
// Outer function
function outerFunction() {
  // A variable defined in the outer function's scope
  const outerVariable = 'I am from outer scope';

  // Inner function
  function innerFunction() {
    // The inner function has access to variables in the outer function's scope
    console.log(outerVariable);
  }

  // Returning the inner function creates a closure
  return innerFunction;
}

// Create a closure by calling the outer function and storing its return value
const closureFunction = outerFunction();

// Call the closure function, which has access to the outer function's variables
closureFunction(); // Output: I am from outer scope
```

Data Privacy Example

```
// Function to create a counter
function createCounter() {
  // Private variable, not accessible
from outside the function
  let count = 0;

  // Return an object with two methods
that form a closure
  return {
    // Method to increment the count
    increment() {
      count++;
      console.log(count);
    },
    // Method to decrement the count
    decrement() {
      count--;
      console.log(count);
    }
  };
}

// Create a counter instance
const counter = createCounter();

// Call the methods, which have access
to the private variable 'count'
counter.increment(); // 1
counter.increment(); // 2
counter.decrement(); // 1
```

- hiding Access to count variable.

- rather creating functions to modify the variable.

Function Factory

```
// Function to create a multiplier
function createMultiplier(multiplier)
{
  // Return a new function that forms
a closure with the 'multiplier'
variable
  return function (number) {
    // This function multiplies its
input by the 'multiplier' value
    return number * multiplier;
  };
}

// Create multiplier functions
const double = createMultiplier(2);
const triple = createMultiplier(3);

// Call the functions, which retain
access to their 'multiplier' value
console.log(double(5)); // 10
console.log(triple(5)); // 15
```

Event Handler

```
// Setup function to add an event
listener
function setup() {
  // A variable defined in the outer
function's scope
  const message = 'Event triggered!';

  // Add an event listener to the
button

document.getElementById('myButton').ad
dEventListener('click', function() {
    // The inner function (event
handler) has access to 'message' from
the outer function's scope
    alert(message);
  });
}

// Call the setup function to set up
the event listener
setup();
```

When user clicks on the button, alert pops up with message: "Event triggered!"

Example of Closures W/ get & post API

Forms a closure with the "apikey" variable.

```
function apiConnect (apikey)
{
    function get(route){
    return fetch (`${route}?key = ${apikey}`);
    }

    Function Post (route, params) {
    return Fetch [ route, {
        method: 'post',
        body : JSON.Stringify (params)
        header: {
            'Authorization': `Bearer ${apikey}`
        }
    })
}

    return {get, Post} // Returning an object with all
                       inner function
}
```

Destructuring

Destructuring is a syntax feature in javascript that allows you to unpack values from arrays or properties from objects into distinct variables. This feature makes it easier to work with complex data structures by providing a concise & readable way to extract data.

When Destructuring arrays, you use square Brackets [] to specify the elements you want to extract.

Example

```
Const numbers = [1,2,3,4,5];
// Destruct the first and second elements.
Const [First, second] = numbers;

console.log(first); // 1.
Console.log(second); // 2.
```

You can also skip elements by leaving spaces between the commas.

Example 2

```
Const [first, , third] = numbers;

console.log(first); // 1
console.log(third); // 3
```

Destructuring Objects

When destructuring objects, you use curly braces {} to specify the properties you want to extract.

Example 3

```
Const person = {
    name : 'John Doe',
    age : 30,
    Job : Developers
}
```

```
Const {name, age} = Person;

console.log(name); // John Doe
console.log(age); // age
```

You can also rename the variables while destructuring.

Example 4

```
Const { name : FullName, age : yearsOld } = Person;

Console.log ( FullName); // John Doe
console.log ( yearsold ); //30
```

Nested Destructuring

You can destructure nested objects and arrays as well.

```
const employee = {
    id: 1,
    details : {
        name : 'Jane Smith',
        position : 'Manager'
    }
}
```

```
Const {
    details : { name, position}
    = employee;

console.log( name); // Jane Smith
console.log ( position ); // Manager
```

Example with nested arrays

```
const nestedArray = [1, [2,3], 4];

const [First, [sceond, third], Fourth] = nestedArray;

console.log (first); //1
Console.log( sceond); //2
Console.log(third); //3
console. log(Fourth); //4
```

Default Values.

You can provide default values for variables in case the value unpacked is 'undefined'

```
const [a = 10, b = 20] = [undefined, 30];
Console.log (a); // 10
Console.log (b); // 20
```

Object Destructuring with Default values.

You can provide default values for variables in case the value unpacked is 'undefined'.

```
Const [a = 10, b=20] = [undefined, 30];
 console.log(a); // 10 (default value is used).
 console.log(b); // 30
```

Function parameter Destructuring

```
const User = {
    id: 42,
    isVerified: true
}

function displayUser({ id, isVerified})
{
    console.log(`User ID: ${id}`);
    console.log(`Is Verified: ${isVerified}`);
}

displayUser(user);
// User ID: 42
// Isverified: true
```

Spread syntax

The **Spread Syntax** in javascript, denoted by three dots **`...`**, allows you to expand elements of array or properties of an object. It's versatile feature that can be used in various context, such as copying arrays, merging arrays or objects, and spreading elements or properties into new arrays or objects.

Spread Arrays

1. Copying Arrays

The spread syntax can create a shallow copy of an array.

```
const originalArray = [1,2,3];
const copiedArray = [...originalArray];

console.log(copiedArray); // [1,2,3]
console.log(copiedArray === originalArray); false (different references).
```

2. **Merging Arrays**

you can merge multiple arrays into one.

```
Const array1 = [1, 2, 3];
Const array2 = [4, 5, 6];
Const mergedArray = [...array1, ...array2];

console.log(mergedArray); // [1, 2, 3, 4, 5, 6]
```

3. **Spreading Elements**

```
Const numbers = [1, 2, 3];

Function sum(a, b, c) {
    return a + b + c;
}

console.log(sum(...numbers)); // 6
```

Spread in objects

1. Copying objects

The spread syntax can create a shallow copy of an object

```
Const originalObject = { a:1, b:2 };
const copiedObject = { ...originalObject };

console.log( copiedobject ); // { a:1, b:2 }
console.log( copied Object === originalobject ) // false (different ref)
```

2. Merging Objects

You can merge multiple objects into one. If there are properties with the same name, the latter Object's properties will overwrite the former's.

```
const object1 = { a:1, b:2 };
const object2 = { b:3, c:4 }
const mergedObject = { ...object1, ...object2, ...object3 };

console.log (mergedobject) ; // { a:1, b:3, c:4 }
```

3. ## Adding properties to objects

You can use the spread syntax to add properties to an existing object while copying its properties.

```
Const originalObject = {a:1, b:2};
Const newObject = {...originalobject, c:3};

console.log (newObject); // {a:1, b:2, c:3}
```

Using Spread with Function parameters

You can use the spread syntax to handle an arbitrary number of function arguments as an array.

```
function sum (...numbers) {
    return numbers.reduce ((acc + curr) => acc + curr, 0);
}

console.log (sum(1,2,3,4,)); // 10
```

Object property overwrite

```
const person = { name: 'Alice', age: 25 };
const updatedPerson = { ...person, age: 26 };

console.log(updatedPerson); // { name: Alice, age: 26 };
```

Destructuring with Spread

```
const [first, ...rest] = [1,2,3,4];

console.log(first); // 1
console.log(...rest); // [2,3,4]
```

Callback functions

A callback function in javascript is a function that is passed as an argument to another function and is executed after some kind of event or operation has completed. This allows you to control the order of execution and handle asynchronous operations more effectively.

Function declaration

```
function myFunc (test, callBack) {
    setTimeout (function () {
        callBack (Text);
    }, 200);
}
```

callBack function

Function Execution

```
myFunction ('Hello world', console.log );
```

- The console.log function is being passed as callBack to myFunc. It gets executed when setTimeout is executed.

Promises

Imagine you're a kid and you ask your parents for a new toy. They don't give you an answer right away. Instead, they promise that they will either get you the toy or explain why they can't. This promise means you can stop asking and just wait until they fulfill it. This is similar to how promises work in JS.

A **promise** in Javascript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to write asynchronous code in a more synchronous fashion, making it easier to manage.

Key Concepts

1. **pending :-** The initial state. The operation is ongoing.
2. **Fulfilled :-** The operation completed successfully, & the promise has a result.
3. **Rejected :-** The operation failed, and promise has a reason for failure.

You can create a promise using the Promise constructor, passing a function called the "executor", which takes two arguments : resolve and reject.

How To create a promise

```
const myPromise = new Promise({resolve, reject} => {

  // Asynchronous operation (e.g. fetching data)
  let success = true; // change this to false to see
                      // rejection
  if (success) {
    resolve("Toy received"); // operation succeeded.
  }
  else {
    reject("couldn't get the toy");
  } // operation failed.
});
```

Once a promise is created, you handle the result using `.then()` for fulfilled cases and `.catch()` for rejected cases. You can also use `.finally()` for code that should run regardless of the outcome.

Using promises

myPromise

```
.then( (result) => {
    console.log(result);
})
.catch( (error) => {
    console.error(error);
})
.finally( () => {
    console.log("promise has been settled.");
});
```

```
const promise = new Promise(function(res, rej){
    setTimeout(function(){
        if (Math.Random() < 0.9)
        {
            return res('Hooray');
        }
        return rej('oh no');
    }, 1000);
});
promise.then(function(data){
    console.log('success' + data);
})
.catch(function(err){
    console.log('Error' + err);
});
```

```
Const API = {
  methods: {
    GET: function (url) {
      return new promise (function (resolve, reject) {
        Vue.http.get (url).then (function (res) {
          resolve (res.body);
        }).catch (function () {
          reject (Error ("API Error"));
        });
      });
    post: function (url, req) {
      return new promise (function (resolve, reject) {
        vue.http.post (url, req).then (function (res) {
          resolve (res.body);
        }).catch (function () {
          reject (Error ("API Error"));
        });
      });
    }
}
```

Async/Await

async and await are syntactic features in javascript that simplify working with promises and asynchronous operations. They allow you to write asynchronous code that looks and behaves more like synchronous code, making it easier to read and maintain.

The await keyword

The await keyword can only be used inside an async function. It pauses the execution of the function until the promise is resolved or rejected. It can be used to wait for a promise to resolve and then get its result.

```
let result = await promise;
```

async and await provide a powerful and readable way to handle asynchronous generations in javascript. They make it easier to write and understand asynchronous code, avoiding the so-called callback hell or chaining multiple .then calls. Understanding and using async/await effectively is crucial for modern javascript development.

```
async function fetchData() {
    let promise = new promise((resolve, reject) => {
        setTimeout(() => resolve('Data fetched'), 100);
    });

let result = await promise; // waits for the promise to resolve.
console.log(result); // output Data Fetched.

fetchData();
```

Error Handling

Handling errors in async/await functions can be done using try and catch blocks, making it similar to synchronous error handling.

```
async function fetchData() {
    try {
        let promise = new Promise((resolve, reject) => {
            setTimeout(() => reject(new Error('Something went wrong')), 1000);
        });

        let result = await promise;
        console.log(result);
    } catch (error) {
        console.error(error); // output : Error: Something went wrong
    }
}

fetchData();
```

Using async / await with Fetch API

The Fetch API is a modern way to make network requests. Combining it with async/await makes for very readable and maintainable code.

```
async function getUserData() {
    try {
        let response = await fetch('https://api.example.com/user');
        IF (!response.ok) {
            throw new Error('Network response was not ok');
        }
        let data = await response.json();
        console.log(data);
    } catch (error) {
        console.error('fetch error', error);
    }
}

getUserData();
```

Running Multiple Promises Concurrently.

You can multiple promises concurrently using promise.all with async / await

```
async function FetchMultipleData () {
    let promise1 = fetch('https://api.example.com/user');
    let promise2 = fetch('https://api.example.com/orders');

    Let [user, orders] = await Promise.all ([promise1, promise2]);
    Let userData = await user.json();
    Let ordersData = await orders.json();

    console.log( userData , ordersData);
}

fetchMultipleData();
```

Await in Loops

Await in Loops

Be careful when using await inside loops, as it can cause the loop to run sequentially, which might not be the desired effect if you want to run them in parallel.

Sequential (not recommended for independent tasks)

```
async function processArray(array) {
    for (let item of array) {
        await someAsyncFunction(item);
    }
}
```

parallel (recommended for independent tasks)

```
async function processArray(array) {
    let promises = array.map(item => someAsyncFunction(item));
    await Promise.all(promises);
}
```

Identity operator vs Equality Operator

Identity operator (===) vs. Equality operator (==).

- The `==` operator will do type conversion prior to comparing values whereas `===` operator will not do any type conversion before comparing.

```
console.log (0 == '0') ; // true;
console.log (0 === '0') ; // false.
```

Object comparison

* Do Not directly compare two objects. Variables are pointing to a referenee to the objects in memory, not the objects themselves.

→ A Safer way to compare objects is to pull in a library that is specializes in deep object comparison. (e.g. Lodash _.isEqual)

Array

In Javascript, an array is a special type of object that is used to store a collection of elements. These elements can be of any type, including numbers, strings, objects, or even other arrays. Arrays are ordered collections, which means that each element has a numbered position, or index, starting from 0.

1. Using square brackets

```
Let fruits = ['Apple', 'Banana', 'Cherry'];
```

2. Using the Array constructor :

```
let fruits = new Array('Apple', 'Banana', 'Cherry');
```

3. Creating an empty array :

```
let emptyArray = [];
let anotherEmptyArray = new array();
```

Accessing Elements

You can access elements in an array by their index:

```
let fruits = ['Apple', 'Banana', 'cherry'];
Console.log(Fruits[0]); // Apple
Console.log(Fruits[1]); // Banana
console.log(Fruits[2]); // cherry
```

Modifying Elements

You can change the value of an element by assigning a new value to a specific index:

```
Fruits[1] = 'Blueberry';
console.log(fruits); // output: ['Apple', 'Blueberry', 'cherry'];
```

Array properties and Methods

Length : The length property returns the number of elements in an array

```
let fruits = ['Apple', 'Banana', 'cherry'];
Console.log(Fruits.length); // output: 3.
```

- **push :** Adds one or more elements to the end of an array and returns the new length of the array.

```
fruits.push('Date');
console.log (fruits); // output : ['Apple', 'Banana', 'cherry', 'Date']
```

- **POP :-** Removes the last element from an array & returns that element.

```
let lastFruit = fruits.POP();
Console.log(lastFruit); // output : Date.
console.log (fruits); // [Banana , cherry]
```

- **Shift :** Removes the first element from an array & returns that element.

```
let firstFruit = fruits.shift();
Console.log(firstFruit); // output : Apple

Console.log(fruits); // output : ['Banana', 'Cherry'];
```

- **Unshift:** Adds one or more elements to the beginning of an array and returns the new length of the array.

```
fruits.unshift('Apricot');
Console.log(fruits); // output : ['Apricot', 'Banana', 'Cherry'];
```

- **Splice :** Changes the contents of an array by removing, replacing, or adding elements.

```
fruits.splice(1, 1, 'Blueberry'); // Removes 1 element at
Index 1 and adds Blueberry
Console.log(fruits); // outputs : ['Apricot', 'Blueberry', 'Cherry']
```

- **Slice :-** Returns a shallow copy of a portion of an array into a new array object.

```
let citrus = fruits.slice(1,2);
Console.log(citrus); // output : [BlueBerry]
```

- **forEach :** Executes a provided function once for each array element

```
fruits.forEach(function(fruit) {
        Console.log(fruit);
});
// output : Apricot, Blueberry, Cherry
```

- **map :** creates a new array with the results of calling a provided function on every element in the calling array.

```
let upperFruits = Fruits.map(function(fruit){
        return fruit.toUpperCase();
});
console.log(upperFruits) // [APRICOT, BLUEBERRY, CHERRY]
```

The Array Constructor is used to create arrays. It can be used in two primary ways.

1. Creating an Array with specified Elements:

```
Const arr = new Array(1, 2, 3);
Console.log(arr); // [1, 2, 3];
```

2. Creating an empty Array with a specified length:

```
Const arr = new Array(5);
Console.log(arr); // [<5 empty items>]
```

Array.From

method creates a new, shallow-copied 'Array' instance from an array-like or iterable object. This method is especially useful when you need to convert something that looks like an array into an actual array.

1. Converting a string to an array

```
Const  Str = 'hello';
const  arr = Array.From(Str);
console.log(arr); //  ['h', 'e', 'l', 'l', 'o'];
```

2. Converting a set to an array

```
const  set  =  new Set([1, 2, 3]);
const  arr  =  Array.from(set);
Console.log(arr); [1, 2, 3];
```

3. Using a Mapping function

```
Const arr = Array.from([1,2,3], x => x * 2);
Console.log(arr); // [2, 4, 6];
```

4. Creating an Array from an Array-like objects.

```
function argsArray() {
    return Array.from(arguments);
}

Const arr = argsArray(1, 2, 3);
console.log(arr); // [1, 2, 3];
```

Array(): primarily used for creating arrays with a specified length or with specified elements.

```
const arr1 = new Array(5); // [<5 empty items>]
const arr2 = new Array(1,2,3); // [1, 2, 3]
```

Array.from(): used for creating arrays from iterable objects or array-like objects, and can apply a mapping function.

```
const arr1 = Array.from([1,2,3], x => x * 2); // [2,4,6]
```

Array.of()

Method creates a new Array instance with a variable number of elements, regardless of number or type of the arguments.

```
Const arr = Array.of(1, 2, 3);
Console.log(arr); // [1, 2, 3]

Const singleElementArray = Array.of(7);
console.log(singleElementArray); // [7]
```

Array.prototype.flat

The Flat() Method creates a new array with all sub-array elements concatenated into it recursively up to the Specified depth

```
Const arr = [1, 2, [3, 4, [5, 6]]];
Const Flattened = arr.flat(2);
console.log(flattened); // [1, 2, 3, 4, 5, 6]
```

Prototype

In javascript, prototypes are a fundmental mechanism by which object inherit features from one another. Every object in javascript has a prototype, and through this prototype, objects can share and inherit properties and methods.

The Concepts of Prototype

At its core, a prototype is simply another object from which an object inherits properties. When you try to access a property or method on a object, the javascript engine will first look for that property on the object itself. If it doesn't find it, it will look at the object's prototype, and then the prototype's prototype, and so on, forming a chain called the prototype chain.

How Prototypes work

1. prototype property (__proto__)

Every javascript object has an internal property called __proto__ (often referred to as the object's prototype). This property points to the prototype object from which object inherits properties and methods.

```
let obj = {};
console.log(obj.__proto__); // output : {}
```

2. Prototype Chain

The prototype chain is a series of linked objects that javascript follows when searching for a property or method. If an object doesn't have a property or method, javascript will look up the chain until it finds it or reaches the end (the object.prototype)

```
let obj = {};
console.log(obj.toString()); // output : [object object]
```

The prototype chain is a series of linked objects that javascript follows when searching for a property or method. if an object doesn't have a property or method, javascript will look up the chain until it finds it or reaches the end (the Object.proto

Creating and Using Prototypes

1. Object Prototypes

The most common way to create an object prototype is by using constructor function or ES6 classes.

Using Constructor functions

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

person.prototype.greet = function() {
  console.log('Hello, my name is ${this.name} and I am
  ${this.age} years old');
};

let john = new person('John', 30);
John.greet(); // Hello, my name is John and I am 30 years old.
```

Using ES6 Classes Example

```
class person {
  constructor (name, age) {
    this.name = name;
    this.age = age;
  }

  greet () {
    console.log(`Hello my name is ${this.name} and I am
    ${this.age} years old`);
  }
}

let John = new Person ('John', 30);
John.greet (); // output : Hello, my name is J
```

Using Object.Create()

2. Another way to create objects with a specific prototype is by using object.create()

```
let personPrototype = {
    greet() {
        console.log(`Hello, my name is ${this.name}.`);
    }
};

let john = Object.create(personPrototype);
john.name = 'John';
John.greet(); // output: Hello, my name is john.
```

Prototype chaining is used to build new types of objects based on existing ones. It is similar to inheritance in a class based language.
The prototype on object instance is available through Object.getPrototypeOf(object) or __proto__ property whereas prototype on constructors function is available through Object.prototype.

Prototype Inheritance

Javascript allows you to create complex inheritance chains using prototypes. This can be achieved by setting one object's prototype to another object.

```
function Animal(name) {
    this.name = name;
}

Animal.prototype.speak = function() {
    console.log(`${this.name} makes sound`);
}

function Dog(name) {
    Animal.call(this, name); // call the parent constructor
}

Dog.prototype = Object.create(Animal.prototype);
    // set the prototype chain.
Dog.prototype.constructor = Dog;
Dog.prototype.speak = function() {
    console.log(`${this.name} barks.`);
}
```

```
let Dog = new Dog('Rex');
dog.speak(); // output : Rex barks
```

The Prototype Property vs __proto__

prototype :- This property is only present on functions, including constructor functions. It is used when creating new instances of function to set the __proto__ property of the new object.

```
function Person(name){
    this.name = name;
}

console.log(Person.prototype); // output :- Person {}
```

__proto__: This property exists on all objects and points to the prototype of the object.

```
let obj = {};
Console.log( obj.__proto__ === Object.Prototype ); // output : true.
```

The Object.prototype

At the top of the prototype chain is `object.prototype`. All objects in Javascript ultimately inherit from object.prototype, which means they have access to its methods unless they are overriden.

Checking Properties

hasOwnProperty: This method checks if a property exists directly on the object itself, not on its prototype chain.

```
let obj = { name: 'John' };
console.log (obj.hasOwnProperty('name')); output : true
console.log (obj.hasOwnProperty('toString')); output : false
```

IIFE

Immediately Invoked Functions Expression

Is a javascript function that runs as soon as it is defined.

```
(function()
    { // logic goes here }
)();
```

An immediately invoked function Expression (IIFE) is a javascript function that runs as soon as it is defined. It's a common design pattern used to create a new scope and to avoid polluting the global namespace.

Syntax

An IIFE is a function expression that is immediately executed. The syntax involves wrapping the function in parentheses to treat it as an expression and then invoking it with another set of parentheses.

Basic Syntax

```
(function () {
  // code that runs immediately
})();
```

Basic Example

```
(function () {
  console.log('This function runs immediately');
})();
```

Advantages of IIFE

1. Avoid Global Namespace Pollution: Variables declared inside an IIFE are not accessible outside, preventing conflicts with other code.

2. Encapsulation: It provides a way to create private variables and functions

3. Initializations: It allows you to run initialization code once without leaving any global variables behind.

```
let myModule = (function () {
    // private variable
    let privateVar = 'I am private';
    // Private function
    function privateFunction () {
        console.log(privateVar);
    }
    return {
        // Public Fonction
        publicFunction : function () {
            privateFunction ();
        }
    };
})();
```

```
myModule.publicFunction(); // output: I am private
Console.log(myModule.privateVar); // Undefined
console.log(myModule.privateFunction); // undefined.
```

1. **Private Members** Variables and functions inside the IIFE are private to the function and can't be accessed from outside.

2. **Public Members** The returned object exposes only the members that you want to make public.

IIFE with Parameters

You can also pass parameters to an IIFE.

```
(function (message) {
    console.log(message);
})('Hello world');
```

Using IIFE with Modern JavaScript

With the advent of ES6 modules, the need for IIFEs to avoid global namespace pollution has diminished, as ES6 modules inherently provide a module scope. However, IIFEs can still be useful in certain contexts, such as initializing configuration settings or setting up event listeners.

```
import { someFunction } from './someFunction.js';

(function() {
    let config = {
        setting1: true,
        setting2: false
    };

    someFunction(config);
})();
```

Understanding the scope

Scope in Javascript refers to the context in which variables, functions, and objects are accessible. It determines the visibility and lifetime of these entities within different parts of your code. There are different types of scopes in Javascript

Types of Scope

1. Global scope
2. Function scope
3. Block scope
4. Lexical Scope

Global Scope

Variables declared outside of any function or block have global scope. They are accessible from anywhere in the code.

```
let globalVar = 'I am global';
function showGlobal () { lobal';
    console.log(globalVar); // Accessible here
}

showGlobal();
console.log(globalVar); // Accessible here to
```

Scope

2. Function Scope

Variables declared within a function are local to that function and cannot be accessed from outside.

```
function showMessage() {
  let message = 'Hello world';
  console.log(message); // Accessible here.
}

showMessage();
console.log(message); // Error: message is not defined
```

3. Block Scope

Variables declared with 'let' and 'const' inside a block (i.e. within curly braces {}) are blocked scoped. They are only accessible within that block.

Block scope

```
if (true) {
    let blockVar = 'I am block scoped';
    console.log(blockVar); // Accessible here
}

console.log(blockVar); // Error; blockVar is not defined.
```

The var keyword, on other hand, does not have block scope and is instead function-scoped or globally-scoped.

```
if (true) {
    var functionScopedVar = 'I am function scoped';
    console.log(functionScopedVar); // Accessible here
}

console.log(functionScopedVar); // Accessible here to
```

Lexical Scope

Lexical Scope : Lexical scope (or static scope) refers to the fact that the scope of a variable is determined by its location within source code. Nested functions have access to variables declared in outer scope

```
function outerFunction() {
    let outerVar = 'I am from outer function';

    function innerFunction() {
        console.log(outerVar); // Accessible here
    }

    innerFunction();
}

outerFunction();
```

Scope Chain

Scope chain : when a variable is referenced, javascript starts searching for that variable in the current scope. If it doesn't find it, it moves up the scope chain to the outer scope, and so on, until it reaches the global scope.

```
let globalVar = 'I am global';

function outerFunction() {
    let outerVar = 'I am from outer function';
    function innerFunction() {
        let innerVar = 'I am from inner function';
        console.log(innerVar); // found
        console.log(outerVar); // found
        console.log(globalVar); // found
    }
    innerFunction();
}

outerFunction();
```

Closures

Closures : a function that retains access to its outer scope, even after the outer function has finished executing. This possible because of lexical scope.

```
function outerFunction() {
    let outerVar = 'I am from outer function';

    return function innerFunction() {
        Console.log(outerVar); // Still accessible here.
    };
}

let closure = outerFunction();
Closure(); // output: I am from outer function.
```

LET VS VAR VS CONST

Let - available only in function scope or Block Scope.

Var - A variable defined with var in a function scope can't be accessed outside the scope, while variable defined with var in a block scope is available outside of that Block.

const - is exactly like Let, but defines a constant reference for a variable. We can't change the value of a constant reference. If we put a primitive value in a constant, then the value will be protected from getting changed.

```
const speed = 42;
Speed = 43; // error
```

We can still change the properties of the object, but we can't change the object itself.

Constants : are popularly used when importing things from other libraries so, that they don't get changed accidentally.

var	let	const
The scope of a var variable is functional scope.	The scope of a let variable is block scope.	The scope of a let variable is block scope.
It can be updated and redeclared into the scope.	It can be updated but cannot be re-declared into the scope.	It cannot be updated or redeclared into the scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
Hoisting done, with initializing as ‘default’ value	Hoisting is done, but not initialized (this is the reason for the error when we access the let variable before declaration/initialization	Hoisting is done, but not initialized (this is the reason for the error when we access the const variable before declaration/ initialization

Module

In Javascript, modules are used to encapsulate code into reusable units. They allow developers to organize their code by splitting it into separate files and components, making it easier to manage, maintain, and debug. Modules also help in avoiding namespace pollution by keeping different parts of the code isolated from each other.

Key Concepts of Module

1. **Encapsulation:** Modules can contain functions, variables, and objects that are private to the module. They expose only what is neccessary using exports.

2. **Reusability:** Modules can be reused across different parts of an application or even in different applications.

3. **Maintainability:** By Breaking down code into smaller, manageable pieces, it becomes easier to maintain and update.

Types of Modules

1. CommonJS Modules
2. ES6 Modules (ESM)
3. AMD Modules
4. UMD Modules

- ### CommonJs Modules

CommonJS is a module format used primarily in NodeJS. Each file is treated as a separate module. You can use require to import and module.export or export to export.

math.js

```
// Define a function to add two numbers.
function add(a, b) {
    return a + b;
}

// Export the add function
module.exports = {
    add
};
```

app.JS

```
// Import the math module.
const math = require('./math');

// Use the add function from the math module.
console.log(math.add(2, 3)); // output : 5.
```

ES6 Modules (ESM)

ES6 introduced a standardized module system that works both in browsers and Node.js. you use import to bring in modules and export to expose them.

math.JS

```
// Define a function to add two numbers.
export function add(a, b) {
    return a + b;
}
```

app.JS

```
// Import the add function from the math module.
import {add} from './math.JS';

// Use the add function
console.log( add(2, 3)); //output : 5
```

Dynamic Import

ES6 also allows dynamic importing of modules using the `import()` function, which returns a promise.

```
async function loadMathModule() {
    Const Math = await import('./math.JS');
    console.log( math.add(2,3));//output : 5
}
loadMathModule();
```

Asynchronous Module Definition (AMD)

Asynchronous Module Definition (AMD) is used in browsers. It allows for asynchronous loading of modules, which can improve performance.

math.js

```
define([], function() {
    return {
        add: function(a,b) {
            return a+b;
        }
    };
});
```

app.js

```
require(['math'], function(math) {
    console.log(math.add(2,3)); // output : 5
});
```

```
var myModule = (function () {
    'use strict';

    var _privateProperty = 'Hello World';

    function _privateMethod() {
        console.log(_privateProperty);
    }

    return {
        PublicMethod: function() {
            _privateMethod();
        }
    };

})();
```

these modules can have exported to the other JS files using export keyword

```
// myModule.js file

export default myModule
```

some names.

Modules can import to another JS File

```
// second.js file
import myModule from './myModule';
```

Hoisting

Hoisting :- is a javascript mechanism where variables & Functions declarations are moved to the top of their Scope before execution.

javascript only hoists declarations, not initializations

Hoisting is the default behaviour of javascript where all the variable and function declarations are moved on top.

This means that irrespective of where the variables and functions are declared, they are moved on top of the scope. The scope can be both local and global.

```
exports.i = 'am exports'
console.log (this) // { i: 'am exports' }

Let util = {
    f1: function () {  // regular scope
        console.log (this);
    },
    f2 () => {
        console.log( this ) ; // arrow scope.
    }
}

util.f1() // { f1: [Function : F1] , f2: [function : F2] }
util.f2() // { i: 'am exports' } Parent scope.
```

when we call f1, the this keyword in f1 will be the caller, which is the util object, while the this keyword in f2 will be Parent scope, which is the exports object.

- this is a huge benefit when working with closures.

```
// Const Component = require ('react').Component
// Const { component } = require ('react');
```

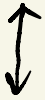
Some thing.

```
Const PI = 3.14159;
const sum = (a,b) => a + b;
Const square = a => a * a;

Const X = {
    PI: PI,
    sum: sum,
    square: square
};
```

When we define an object like X here & we have other objects, functions, or modules that we want to export as properties on the X object

we can omit the values when the property name & values are the same



Something

```
Const x = {
    PI,
    sum,
    square
};
```

When we want to read properties from X & assign them as variables with same name, we can use the destructive syntax instead of typing the name twice.

```
// const square = X.square;
const {square} = X;
```

Something

```
Const {square, PI, sum} = X;
```

Class

In javascript, classes provide a more Intuitive and modern syntax for creating objects & handling inheritance, building on the prototype-based architecture inheritance model. Introduced in ECMAScript 6 (ES6), classes allows developers to define object templates in way that is more similar to class-based languages like java or c++.

Defining a class

A class in javascript is defined using the class keyword followed by the class name. The body of the class is enclosed in the curly braces `{}`.

```
Class Person {
    constructor (name, age) {
        this.name = name;
        this.age = age;
}

greet() {
    console.log(`Hello, my name is ${this.name} and I am
    ${this.age} years old.`);
}
```

Constructor Method

The constructor method is a special method for creating and initializing objects created with the class. Each class can only have one constructor method. If a class does not explicitly define a constructor, default constructor is used.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

Creating an Instance.

You create an instance of a class using new keyword.

```
let john = new Person('john', 30);
console.log(john.name); // output : john
console.log(john.age); // output : 30
john.greet(); // output : Hello, my name is john and I am 30 years old.
```

Methods

- Classes can contain methods, which are functions defined within the class. These methods can be called on instances of the class.

```
Class Person {
  Constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log(`Hello, my name is ${this.name} and I am
      ${this.age} years old`);
  }

  haveBirthday() {
    this.age += 1;
    Console.log(`Happy Birthday! I am now ${this.age} years old`)
  }
}
```

Static Methods

Static Methods are defined on the class itself, not on instances of the class. They are called directly on the class.

```
Class MathHelper {
    Static add(a, b) {
        return a + b;
    }
}

Console.log(MathHelper.add(5, 7));
```

Inheritance

Classes in javascript can extend other classes, allowing for inheritance. The extends keyword is used to create a subclass, and the super keyword is used to call the constructor of the parent class.

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name); // Call the parent class constructor.
        this.breed = breed;
    }

    speak() {
        console.log(`${this.name} barks`);
    }
}

let rex = new Dog('Rex', 'German shepherd');
rex.speak(); // Rex barks.
```

Getters And Setters

Classes can also have getter and setter methods to control how properties are accessed and mutated.

```
class Person {
  constructor(name) {
    this._name = name;
  }

  get name() {
    return this._name;
  }

  set name(newName) {
    this._name = newName;
  }

  let john = new Person('John');
  console.log(john.name); // output : John
  john.name = 'Johnny';
  console.log(john.name); // output : Johnny
```

Private Fields

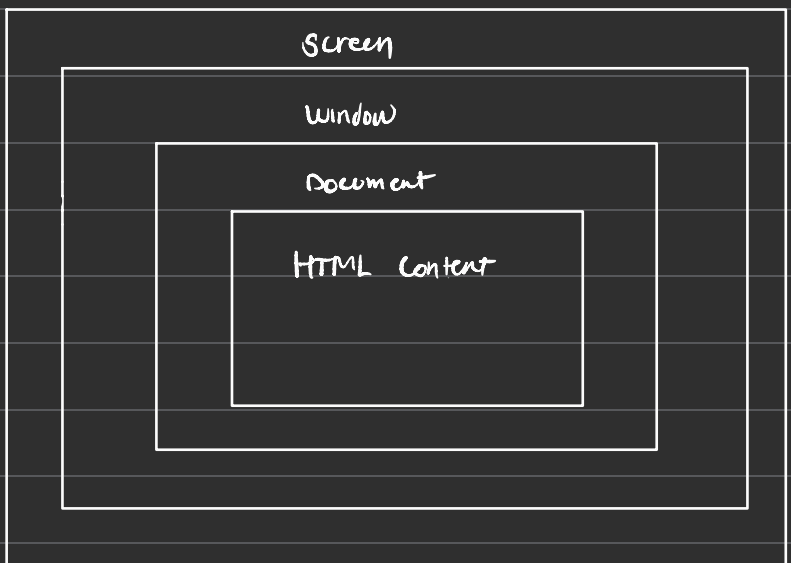
Javascript also supports private fields, which are not accessible outside of the class body. private fields are prefixed with a #.

```
Class Person {
  #ssn;
  Constructor(name, ssn) {
    this.name = name;
    this.#ssn = ssn;
  }

getSSN() {
  return this.#ssn;
}

let john = new Person('john', '123-45-6789');
Console.log(john.name); // output : John
Console.log(john.getSSN()); // output : 123-45-6789.
```

Screen vs Window vs Document vs HTML Content



Screen: Represents the entire screen or monitor.

Window: Represents the browser window or a frame within a window.

Document: Represents the HTML content loaded in the window.

This is the web page you see and interact with.

Screen Properties

- **availHeight :** The height of the screen excluding the operating system taskbar.
- **availWidth :** The width of the screen excluding the operating system taskbar.
- **colorDepth :** The color depth of the screen in bits per pixel.
- **height :** The height of the screen in pixels.
- **width :** The width of the screen in pixels.
- **orientation :** The current orientation of the screen (portrait or landscape).
- **pixelDepth :** The number of bits used to display one color.

Window Properties

1. **innerHeight :** The height of window's content area.
2. **innerWidth :** The width of window's content area.
3. **outerHeight :** The height of the entire browser window including toolbars & scrollbars.
4. **outerWidth :** The width of the entire browser window including toolbars and scrollbars.
5. **location :** The location object containing information about the current URL
6. **history :** The history object containing the browser's session history.
7. **navigator :** The Navigator object containing information about the browser.
8. **screen :** The screen object representing the screen where the window is displayed.
9. **document :** The Document object representing the loaded HTML document.
10. **localStorage :** The localStorage object for storing data that persist across sessions.
11. **sessionStorage :** The sessionStorage object for storing data that persists only for the duration of the page session.

Document Properties

Document Properties

1. URL : The full URL of the current of document.
2. title : The title of the document.
3. body : The body element of the document.
4. head : The head element of the document.
5. cookie : The cookies associated with the document.
6. referrer : The URL of the referring document.
7. domain : The domain of the document.
8. lastModified : The date and time the document was last modified.
9. readyState : The loading state of the document (loading, interactive, complete).
10. anchors : A collection of all anchor (<a>) elements in the document with a name attribute.
11. forms : A collection of all form elements in the document
12. images : A collection of all image elements in the document.
13. links : A collection of all anchor elements with href attributes.
14. script : A collection of all script elements in the document.

Window Properties

In javascript, window.parent, window.top, and window.self (which can be used interchangeably with window) are properties that allow you to navigate between different window contexts, especially useful when dealing with frames or iframes.

window.parent : Refers to the parent window of the current window or frame. If the current window is the top-level window, window.parent is the same as window.

window.top : Refers to the topmost window in the hierarchy of the window objects. If the current window is the top-level window, window.top is the same as window.

Export vs Export Default

In javascript, export and export default are used to make functionalities (variables, functions, or classes) available for use in other files or modules.

1. **export :** This keyword is used to export one or more values (variables, functions, or classes) from a module.

```
export const someVariable = 42;
export function someFunction() {...}
export class someClass {...}
```

2. **export default :** This syntax is used to export a single value as the default export from a module. There can only be one default export per module.

```
const defaultExport = 42;
export default defaultExport;
```

Multiplicity :- export allows multiple named exports per module, while export default is used for a single default export per module.

Import syntax : Named exports (export) are imported w/ curly braces & must match the exported name.

Default exports (export default) are imported without curly braces & can be assigned any name during import.

Call-Apply-Bind

In javaScript, call, apply, and bind are methods used to control the this context of a function. They allow you to Invoke a function with a specific this value and arguments.

Call

The call method call a function with a given this Value and arguments provided individually.

Syntax

```
func.call (thisArg, arg1, arg2, ...)
```

```
Function greet (greeting, punctuation) {
    console.log( greeting + ', ' + this.name + punctuation );
}

let person = { name : 'John' };
greet.call (person, 'Hello', '!');
// output : Hello, John !
```

Apply

- The apply method is similar to call, but it takes an array of arguments instead of individual arguments.

Syntax

```
func.apply(thisArg, [argsArray])
```

```
function greet(greeting, punctuation)
{
    console.log(greeting + ', ' + this.name + punctuation);
}

let person = { name: 'John' };
greet.apply(person, ['Hello', '!']);
// Output: Hello, John!
```

Bind

The bind method creates a new function that, when called, has its this value set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

```
let boundFunc = func.bind(thisArg, arg1, arg2, ...)
```

```
function greet(greet, punctuation)
{
    console.log(greeting + ', ' + this.name + punctuation);
}

let person = { name: 'John' };
let greetJohn = greet.bind(person, 'Hello');

greetJohn('!'); // output: Hello, John!
```

New Keyword in JavaScript

The **new Keyword** in javascript can only be used with functions that are intended to be used as **constructors**. **These functions are usually called constructor functions.** When you use the new keyword with a function, it creates a new object and sets the function's 'this' value to that new object and sets the function's this value to that new object. Additionally, it sets the object's prototype to the function's prototype property.

```
function Person (firstName, lastName) {
    this.FIRSTNAME = firstName;
    this.LastName = lastName;
}

Const member = new person ("Lydia", 'Hallie');
Console.log (member);
// person : { FirstName : Lydia , LastName : Hallie }
```

(capitol)

Let's go through each step that happens when you use the new keyword with a function, and illustrate it

Step - by - Step

1. A new empty object is created.
2. The new object's __proto__ is set to the constructor function's prototype property.
3. The this keyword within the constructor function is set to the new object.
4. The function code executes, typically adding properties to this.
5. The new object is returned, unless the function returns its own object.

```
function Person (firstName, lastName) {
    // Step 3. 'this' is set to the new object.
    this.firstName = firstName;
    this.lastName = lastName;

    // Uncomment the following line to see steps in action
    // return { firstName: 'Returned', lastName: 'object' };
}

// Step 1: A new empty object is created.
// Step 2: The new object's __proto__ is set to Person.prototype.
const member = new Person("Lydia", "Hallie");
console.log(member);
//output: Person { firstName: Lydia, lastName: Hallie };
```

1. A new empty object is created.

when you call 'new person(Lydia, Hallie), javascript creates a new empty object.

```
Const obj = {};
```

2. The new object's __proto__ is set to function's prototype property.

The new object's __proto__ is set to Person.prototype.

```
obj.__proto__ = Person.prototype.
```

3. The this keyword within the constructor function is set to the new object.

Inside the person function, this refers to the newly created object.

```
function Person(firstName, LastName) {
    this.firstName = firstName;
    this.LastName = LastName;
}
```

4. The function code executes, typically adding properties to this.

The code inside the constructor function executes, adding properties to the new object (this).

```
this.firstName = firstName;
this.lastName = lastName;
```

Now, object looks like this.

```
obj = {
  firstName: 'Lydia',
  lastName: 'Halie'
};
```

5. The new object is returned, unless the function returns its own object

If the constructor function does not explicitly return an object, the new object (obj) is returned by default.

```
return obj;
```

Example : Using new with a Regular Function (Not intended as constructor)

If you try to use new with a regular function that is not designed to be constructor, it can lead to unexpected behavior or errors.

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

const greeting = new greet();
console.log(greeting);
```

In this case, greet does not set any properties on this, so, the resulting object is empty. While it technically, works, it's not meaningful to use new with a function like this.

Functions that Should Not Be used with new.

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

Const sum = new add(2,3); // This doesn't make sense.
console.log(sum); // output: add {}
```

Event Bubbling

Event bubbling and Capturing are two ways of event propagation in the HTML DOM API, when an event occurs in an element inside another element, and both elements have registered a handle for that event. The event propagation mode determines in which order the elements receive the event.

With Bubbling the event is first captured and handled by the innermost element and then propagated to outer elements.

With Capturing the event is first captured by the outermost element and propagated to the inner elements.

```
<div>
    <ul>
        <li></li>
    </ul>
</div>
```

In the structure above, assume that click event occured in the Li element.

In Capturing model the event will be handled by the div first (click event handlers in the div will fire first), then in the ul, then at the last in target element, li.

In Bubbling model the opposite will happen: the event will be first handled by the Li, then by the Ul, and at last by the div element.

From Github.com/lydiahallie/javascript-questions

Functions are objects! (Everything besides primitive types are objects)

A Function is a special type of object.

```
function Person(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
}

const member = new Person('Lydia', 'Hallie');
Person.getFullName = function() {
    return `${this.firstname} ${this.lastname}`;
};

console.log(member.getFullName());
```

What is the output?

A. TypeError, B. SyntaxError, C. Lydia Hallie, D. undefined undefined

In javaScript, functions are objects, and therefore, the method getFullName gets added to the constructor function object itself. For that reason, we can call Person.getFullName(), but member.getFullName throws a typeError.

If you want a method to be available to all object instances, you have to add it to the prototype property:

```
Person.prototype.getFullName = function() {
    return `${this.firstname} ${this.lastname}`;
}
```

```
function Person(firstname, lastname)
{
    this.firstname = firstname;
    this.lastname = lastname;
}

const lydia = new person('Lydia', 'Hallie');
const sarah = person('Sarah', 'Smith');

console.log(lydia);
console.log(sarah);
```

Answer: Person {firstname: 'Lydia', lastname: 'Hallie'} and undefined

For Sarah, we didn't use the new keyword. When using new, this refers to the new empty object we create. However, if you don't add new, this refers to the global object

→ We said that this.firstname equals "Sarah" & this.lastname equals "smith", we actually did, is defining global.firstname = 'sarah' and global.lastname = 'smith'. Sarah itself is left undefined, since we don't return a value from the person function.

→ All objects have prototype? True or False

⇒ All objects have prototypes, except for the base object. The base object is the object created by the user, or, an object that is created using the new keyboard. the base object has access to some methods & properties, such as .toString. This is the reason why you can use built-in javascript methods. All of such methods are available on the prototype. Although javascript can't find it directly on your object, it goes down the prototype chain & finds it there, which makes it accessible for you!