#JavaScript Topics###

1. History

2. Introduction

3. Javascript Runtime Environment

4. Token

5. Scope

6. Global Execution Context

7. Var Let Const

8. Function

9. Practice Questions

10. Lexical Scope

11. Closure

12. Interview Questions

13. Array

14. Object

15. Json

16. Call Apply Bind

17. Constructor Function

18. This Keyword

19. Destructuring

20. Rest And Spread

21. Prototype

22. Dom

#JavaScript Notes###

#History

1. JavaScript was first created by Brendan Eich in just 10 days in May 1995 while he was working at Netscape Communications Corporation.
2. The initial release was called Mocha and later renamed to LiveScript, and finally JavaScript.
3. Initially, JavaScript was designed to be a lightweight scripting language for adding interactivity to web pages.
4. At the time, web pages were mostly static and lacked interactivity, and the only way to add dynamic content to a web page was to use a server-side programming language like PHP or Perl.
5. However, this approach had limitations and was not well-suited to creating responsive, interactive user interfaces.
6. The idea behind JavaScript was to create a scripting language that could be executed on the client-side (in the user's web browser) and could be used to add interactivity to web pages.
7. This allowed web developers to create more engaging and interactive websites, without having to rely on server-side programming languages.
8. Its popularity grew rapidly as it was one of the few languages that could be executed directly in web browsers, without the need for additional plugins or software.
9. In 1996, Microsoft released JScript as a competitor to JavaScript, which was their own implementation of the language for their Internet Explorer browser.
10. However, JScript was very similar to JavaScript, and the two languages were largely interchangeable.
11. Over time, JavaScript has evolved and grown to become a full-fledged programming language, capable of creating complex applications on both the client and server-side.
12. The development of JavaScript has been heavily influenced by a number of factors, including the emergence of new web technologies, changes in programming paradigms, and the rise of new development frameworks and libraries.
13. Today, JavaScript is one of the most widely used programming languages in the world, powering many of the most popular websites and web applications.
14. It continues to evolve and adapt to the changing needs of the web, with new features and capabilities being added on a regular basis.

#Introduction

1. Javascript is scripting and programming language.

2. It is purely object based language.This means that variables, functions, and even primitive data types like numbers and strings are object,everything is object in javascript.

3. It is dynamically typed language , it means type of value stored in memory block is checked at runtime because of this nature we can store any type of value in variable.

4. It is object oriented programming language , it means we can create our own object. (It is not purely object oriented programming language)

5. It is interepreted language

6. It is synchronous language , it has single threaded architecture. Instructions get executed line by line.

7. It is single call stack

8. Mainly introduced to instruct the browser

9. Js helps to provide behavior and functionality to webpage and helps to develop dynamic webpage

10. Every browser have js engine to run js code. Therefore browser become environment to run js code.

11. To run Js code outside browser we just need Javascript runtime environment (Node).

12. Js is used to add functionality to website.

#Javascript Runtime Environment

1. JavaScript Runtime Enviroment provides the enviroment where we can run our JavaScript code.

2. Two javascript runtime enviroments are :
I. Browser
II. Node.js

#Browser

1. A browser is a software application that is used to access and view information on the World Wide Web (WWW).

2. It allows users to interact with web pages, view multimedia content, and surf the internet.

3. The Browser acts as a JavaScript runtime environment because it includes a JavaScript engine that interprets and executes JavaScript code

#JavaScript Engine

1. A JavaScript engine is a computer program that executes JavaScript code.

2. It is a core component of web browsers, server-side JavaScript platforms, and other JavaScript-based environments.

#Some Popular JavaScript Engines Include:

1. **V8 (fastest Js engine)**: developed by Google, used in Google Chrome and Node.js

2. **SpiderMonkey** : developed by Mozilla, used in Firefox

3. **JavaScriptCore** : developed by Apple, used in Safari

4. **Chakra** : developed by Microsoft, used in Microsoft Edge and Internet Explorer (legacy)

#Node.Js

1. **The main reason of javascript popularity.**

2. Node.js is a software application that executes JavaScript code. It is not a framework or a library.

3. It allows developers to run JavaScript code outside of a web browser, such as on a server or command-line interface.

4. Node.js uses the V8 JavaScript engine, which is also used in Google Chrome.

5. It is built on top of an event-driven, non-blocking I/O model, which allows it to handle large numbers of simultaneous connections without blocking the execution of other code.

6. This makes it well-suited for building scalable, high-performance applications that can handle a large amount of traffic.

7. Because after introduction of Nodejs, we were able to run javascript anywhere like in web servers, as command-line tools, desktop applications, and even IoT (Internet of Things) devices.

Features of JavaScript

1. **Scripting Language**:

   - **What it means**: JavaScript is used to write small programs that automate tasks in web pages.

   - **Why we call it that**: It helps make web pages interactive. For example, it can change text when you click a button or show a pop-up message.

2. **High-Level Language**:

   - **What it means**: JavaScript is easy for humans to read and write.

   - **Why we call it that**: You don’t need to know the details of how the computer works to use JavaScript. It uses simple commands and is close to human language, making it accessible for beginners.

3. **Interpreted Language**:

   - **What it means**: JavaScript code runs directly in the web browser without needing to be converted into another form first. It has interpretor which runs the code line by line.

   - **Why we call it that**: You can see the results of your code immediately in the browser. This makes it easy to test and fix your code quickly.

4. **Synchronous Language**:

   - **What it means**: JavaScript runs commands one after another, in order.

   - **Why we call it that**: When you write JavaScript code, it does each task step by step, making it easier to understand what’s happening in your program.

5. **Object-Based Language**:

   - **What it means**: JavaScript uses objects to store and organize data and functions.

- **Why we call it that**: In javaScript most of the things are are internally objects. Objects help keep related code together. For example, you can have an object for a car that includes properties like color and methods like drive.

6. **Object-Oriented Language**:

- **What it means**: JavaScript supports creating complex structures using classes and objects.

- **Why we call it that**:We can create our own objects using classes in javascript.

7. **Loosely Typed Language**:

- **What it means**: In JavaScript, you don’t have to specify what type of data (like a number or text) a variable holds. Also we don’t need to follow the syntax very strictly eg. No need to write semicolons.

- **Why we call it that**: This makes writing code faster and easier because you don’t have to worry about declaring data types and don’t need to strictly follow the syntax. You can just start using variables right away.

8. **Dynamically Typed Language**:

- **What it means**: The type of data a variable holds can change as your program runs.

- **Why we call it that**: You can have a variable that starts as a number and then later hold a string (text) without any extra work. This flexibility makes JavaScript powerful for writing dynamic programs.

9. **Single-Threaded Language**:

- **What it means**: JavaScript can only do one thing at a time.

- **Why we call it that**: This simplicity makes it easier to write and understand JavaScript code. Even though it does one task at a time, JavaScript can handle many tasks quickly by using techniques like callbacks and promises.

#Token

1. It is the smallest unit of programming language.

2. We have 5 types of operators,punctuators,keywords ,identifiers , literals.

#Operators

In JavaScript, operators are used to perform operations on variables and values. Here are the main types of operators in JavaScript, along with examples and brief descriptions:

1. *Arithmetic Operators*

Arithmetic operators are used to perform arithmetic calculations.

- + (Addition): Adds two values.

```
javascript

let sum = 5 + 3; // sum is 8
```

- - (Subtraction): Subtracts the second value from the first.

```
javascript

let difference = 10 - 4; // difference is 6
```

- * (Multiplication): Multiplies two values.

```
javascript

let product = 6 * 7; // product is 42
```

- / (Division): Divides the first value by the second.

```
javascript

let quotient = 20 / 5; // quotient is 4
```

- % (Modulus): Returns the remainder of a division.

  javascript

  let remainder = 10 % 3; // remainder is 1

- ** (Exponentiation): Raises the first value to the power of the second.

  javascript

  let power = 2 ** 3; // power is 8

2. *Assignment Operators*

Assignment operators are used to assign values to variables.

- =: Assigns a value to a variable.

  javascript

  let x = 10;

- +=: Adds and assigns the result.

  javascript

  x += 5; // x is now 15

- -=: Subtracts and assigns the result.

  javascript

  x -= 3; // x is now 12

- *=: Multiplies and assigns the result.

  javascript

  x *= 2; // x is now 24

- /=: Divides and assigns the result.

  javascript

  x /= 4; // x is now 6

- %=: Takes modulus and assigns the result.

  javascript

  x %= 5; // x is now 1

- **=: Raises to power and assigns the result.

  javascript

  x **= 3; // x is now 1

3. *Comparison Operators*

Comparison operators are used to compare two values.

- == (Equal to): Checks if two values are equal.

  javascript

  5 == '5'; // true

- === (Strict equal to): Checks if two values are equal and of the same type.

  javascript

  5 === '5'; // false

- != (Not equal to): Checks if two values are not equal.

  javascript

  5 != '5'; // false

- !== (Strict not equal to): Checks if two values are not equal and/or not of the same type.

  javascript

  5 !== '5'; // true

- > (Greater than): Checks if the left value is greater than the right.

  javascript

  10 > 5; // true

- < (Less than): Checks if the left value is less than the right.

  javascript

  10 < 5; // false

- >= (Greater than or equal to): Checks if the left value is greater than or equal to the right.

  javascript

  10 >= 10; // true

- <= (Less than or equal to): Checks if the left value is less than or equal to the right.

  javascript

  10 <= 5; // false

4. *Logical Operators*

Logical operators are used to combine multiple boolean expressions.

- && (Logical AND): Returns true if both operands are true.

  javascript

  true && false; // false

- || (Logical OR): Returns true if at least one operand is true.

  javascript

  true || false; // true

- ! (Logical NOT): Inverts the boolean value.

  javascript

  !true; // false

5. *Bitwise Operators*

Bitwise operators are used to perform bit-level operations.

- & (AND): Returns a 1 in each bit position for which the corresponding bits of both operands are 1.

 javascript

 5 & 1; // 1 (0101 & 0001)

- | (OR): Returns a 1 in each bit position for which the corresponding bits of either or both operands are 1.

 javascript

 5 | 1; // 5 (0101 | 0001)

- ^ (XOR): Returns a 1 in each bit position for which the corresponding bits of either but not both operands are 1.

 javascript

 5 ^ 1; // 4 (0101 ^ 0001)

- ~ (NOT): Inverts the bits of its operand.

 javascript

 ~5; // -6 (not 0101)

- << (Left shift): Shifts bits to the left by the specified number of positions.

 javascript

 5 << 1; // 10 (0101 << 1)

- >> (Sign-propagating right shift): Shifts bits to the right, preserving the sign.

 javascript

```
5 >> 1; // 2 (0101 >> 1)
```

- >>> (Zero-fill right shift): Shifts bits to the right, filling with zeros.

```
javascript

5 >>> 1; // 2 (0101 >>> 1)
```

6. *String Operators*

String operators are used to manipulate strings.

- + (Concatenation): Joins two or more strings.

```
javascript

let greeting = 'Hello' + ' ' + 'World'; // "Hello World"
```

- +=: Appends the right operand to the left operand.

```
javascript

let text = 'Hello';

text += ' World'; // "Hello World"
```

7. *Conditional (Ternary) Operator*

The ternary operator is a shorthand for an if-else statement.

- condition ? expr1 : expr2: Returns expr1 if the condition is true, otherwise returns expr2.

javascript

let result = (5 > 3) ? 'Greater' : 'Lesser'; // "Greater"

8. *Type Operators*

Type operators are used to determine the type of a variable or value.

- typeof: Returns the type of a variable or expression.

javascript

typeof 42; // "number"

- instanceof: Tests whether an object is an instance of a particular class or constructor.

javascript

let arr = [];

arr instanceof Array; // true

These operators are essential for performing various operations in JavaScript, from simple arithmetic to complex logical expressions.

#Punctuators

1. These are symbols used to group, separate, or punctuate code.

2. Examples include parentheses (), curly braces {}, square brackets [], commas ,, semicolons ;, and the period . (used to access object properties).

#Keywords

1. These are reserved words that have a special meaning in the language.

2. Examples like if, else, for, while, function, and return,etc.

#Identifiers

1. These are user given names to variables, functions, and other objects in the code.

2. Identifier name can not start with number.

3. Identifier name should not be a keyword

4. If Identifier is of multiple word, instead of using space, we have to use underscore.

5. identifier name should not have special character but can start with underscore(_) and dollar($).

#Literals

1. These are values used in our program like number(2),string('hello world') , etc.

#Types Of Literals / Datatypes

1. Primitive

2. Non-Primitive

#Primitive Literals

1. In JavaScript, a primitive data type is a data type that represents a single value.

2. JavaScript treats primitive values as immutable values, means that their value cannot be changed. Instead, when you perform an operation that appears to modify a primitive value, you are actually creating a new object with new value and assigning it to a variable. Here , variable will hold the reference of latest object with new value and the previous object with it's value will garbage collected.

3. We have 8 primitive types of literals -**number , bigint , boolean , nan , undefined , null , symbol , string.**.

#Primitive Datatypes

1. Number

1. This data type represents a numeric value. It can store both integers and floating-point values.

2. It's range is from $-2^{53}-1$ to $2^{53}-1$.

2. BigInt

1. It is used to represent integers that are larger than the Number data type

2. It's range is more than $-2^{53}-1$ and more than $2^{53}-1$.

3. To represent the given integer as bigint , we have to suffix 'n' after the integer.
Example : 10 is number type and 10n is bigint type.

3. Boolean

1. This datatype represents a logical entity and can only have two values: true or false.

4. Null

1. This datatype represents a null or empty value.

2. It is used to mark the memory block empty intentionally.

5. Undefined

1. This datatype represents an uninitialized value.

2. When memory block is unintialized , js engine implicitly initialize that memory block with 'undefined' in variable phase.

3. For variable declared with 'var' it will initialize it in variable phase

4. For variable declared with 'let' and 'const' it will not initialize it in variable phase.

6. NaN

1. It stands for 'not a number'.

2. It represents computational error.

3. When js engine is not able compute result it returns 'NaN'.

4. Example : "Hello" + 1 = Hello1 and "Hello" - 1 = NaN
In first case , js engine concatnated the string with number.
In second case , js engine is able to compute anything because we can not subtract 1 from "Hello" string therefore it returns NaN.

7. Symbol

1. It represents a unique identifier.

2. We have Symbol function which is used to generate unique idenitifiers in our program.

8. String

1. It represents collection of characters.

2. We have two types of strings :- single line and multi line string.

3. **Single line string** :
- It is enclosed by single quotes (' ') and double quotes (" ") .
- It doesnot allow line breaks and whitespaces.

4. **Multi line string** :
- It is enclosed by backticks (` `).
- It allow line breaks and whitespaces.
- It is also called as template string.
- Template strings allow us to insert variables and expressions directly in the string using ` ${ variable_name }` notation.

#Non-Primitive Literals

1. In JavaScript, a non primitive data type is a data type that represents multi value.

2. JavaScript treats non-primitive values as mutable values, means that their value can be changed. When we try to update a value , new object is not created . Here value is changed in the same memory block.

3. Non-primitive datatype : **object ,array , etc**

#Scope

1. Scope defines the visibility or accessibility of a variable.

#We Have Two Scopes

1. Global Scope

2. Local Scope

#Global Scope

1. The variable declared in global scope can be accessed anywhere in the program.

2. Global scope has the highest accessibility.

3. Variable declared with var goes in Global scope.

#Local Scope

1. Local/block scope/function scope

2. The variable declared in local scope can be accessed in that block only i.e. we can not access the variable from outside.

3. JS engine creates local scope for **functions and blocks**.

4.

Function's Local Scope

- Local scope created for function is refered as function scope.
- Variable's declared in function's scope can not be accessed from outside.

•

Block's Local Scope

- Local scope created for block is refered as block scope.
- Variable's declared in block scope can not be accessed from outside.
- But only variables declared with var are accessible from outside of block.

Note: Variables declared with let and const are also locally scoped.
Firefox represent it as - Block scope.
Chrome represent it as - Script scope.

#Global Execution Context

1. When we give JS code to the browser, JS Engine will allocate (create) a global memory block for the execution of JavaScript code, called Global Execution Context.

2. Here, we have a window variable which have reference of Global Execution Context.

#Window Variable

1. Window variable or window object -> everything is object in js.

2. Window is a global variable which store the reference of Global Execution Context

3. Window object is also known as Global Object because it is available anywhere in the program.

4. Window object have pre-defined state and behaviour.

5. Variable declared with var always goes to global scope and can be accessible by window object.

6. Any variable created in global scope will be addes in Window object implicitly by JS Engine.

#JavaScript Code Run In Two Phases

1. Variable phase

2. Execution phase

#Variable Phase

1. In variable phase, JS Engine will check the complete JS Code and it will search for variable declaration statement.

2. If variable is declared then JS Engine allocate (provide) memory for them.

3. **Variable declared with var will be initialized storing "undefined" at the time of memory block creation. Variable declared with let and const will remain uninitialized (empty) at the time of memory block creation.**

#Execution Phase

1. In Execution phase, JS Engine will execute the instruction line-by-line.

#Var

1. Variable declared with var goes to global scope.

2. We can redeclare variable with same name in same scope.

3. We can update the value of variable.

4. We can declare variable without initialization.

5. Variable declared with var , can be hoisted.

6. Variable declared inside block , will go to global scope.

7. Variable declared inside function , will not go to global scope. It will be accessible inside function only.

#Let

1. Variable declared with let is block scoped.

2. We cannot redeclare variable with same name in same scope.

3. We can update the value of variable.

4. We can declare variable using let without initialization. But js engine will keep that memory block uninitialized (empty) untill js engine reads declaration statement in execution phase.

5. Because let variable is uninitialized (empty) in variable phase , it belongs to Temporal Dead Zone.

6. The variable declared using let does not belongs to global scope , we cannot access them with the help of window variable.

7. The variable declared using let is hoisted and belongs to temporal deadzone. Therefore it cannot be used before initialization (because at that moment it is uninitialized - TDZ) .

8. Variable declared inside function will be accessible inside function only.

#Const

1. Variable declared with const is block scope.

2. We cannot redeclare variable with same name in same scope.

3. The value of variable can not be modified.

4. We can not declare const without initialization.

5. The variable declared using const is hoisted and belongs to temporal deadzone. Therefore it cannot be used before initialization (because at that moment it is uninitialized - TDZ) .

6. The variable declared using const inside block ,does not belongs to global scope we cannot use them with the help of window.

7. Variable declared inside function will be accessible inside function only.

#Practice Questions

1.

```
console.log("start");
let a = 10;
var b = 20;
const c = 30;
{
  let a = 100;
  var b = 200;
  const c = 300;
  console.log(a);
  console.log(b);
  console.log(c);
}
console.log(a);
console.log(b);
console.log(c);
console.log("end");
```

2.

```
console.log("start");
let a = 10;
console.log(b);
{
  var b = 200;
}
console.log(a);
console.log(b);
console.log("end");
```

3.

```
console.log("start");
let a = 10;
{
  console.log(a);
  let a = 10;
```

```
}
console.log(a);
console.log(b);
console.log("end");
```

4.

```
console.log("start");
var b = 20;
const c = 30;
{
  let a = 100;
  console.log(a);
  console.log(b);
  console.log(c);
}
console.log(a);
console.log(b);
console.log("end");
```

5.

```
console.log("start");
let a = 10;
var b = 20;
const c = 30;
{
  let a = 10;
  console.log(a);
  const c = 300;
  console.log(b);
  b = 200;
  c = 30;
  console.log(b);
}
console.log(a);
console.log(b);
console.log("end");
```

#Functions

1. Function is object.

2. Function is a block of instruction which is used to perform a specific task.

3. A function get executed only when it is called.

4. The main advantage of function is we can achieve code reusability.

5. To call a function we need its reference and ().

6. Name of function is variable which holds the reference of function object.

7. Creating a function using function keyword supports function hoisting.

8. Therefore we can also call a function before function declaration.

9. When we try to log function name the entire function defination is printed.

10. The scope within function block is known as local scope.

11. Any member with local scope cannot be used outside the function block.

12. A parameter of function will have local scope.

13. Variable written inside function even using var have local scope.

14. Inside a function we can use the members of global scope.

15. In javascript 'this' is a property of every function.(every function will have 'this' Keyword except arrow function)

#Parameter

1. The variables declared in the function defination is known as parameters.

2. The parameters have local scope (can be used only inside function body).

3. Parameters are used to hold the values passed by caller (or calling statement).

#Arguments

1. The values passed in the method call statement is known as arguments.

2. Note : An argument can be a literal, variable or an expression which gives a results.

#Return Keyword

1. It is a keyword used as control transfer statement in a function.

2. Return will stop the execution of the function and transfer control along with data to the caller.

#Ways To Create Functions

1. Function declaration statement : Create using function keyword

1.

Syntax :

```
function func_variable(parameters) {
  //statements
}
func_variable()
```

2.

Example : Create a function 'greet' which should print a message "Good Morning" when it is called.

```
function func_variable(parameters) {
  //statements
}
func_variable()
```

3.

Function can be Hoisted.

```
//Here, we are accessing function before it's declaration statement.

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

output : Good Morning

4. Function does not belongs to temporal dead zone.

2. Function as expression / expression function

1. Function which is passed to an variable as a value is called as first class function.

2. Function can not be Hoisted because it is object is created in execution phase.

3. Function does not belongs to temporal dead zone

#Functional Programming

1. Functional Programming is a programming technique where we pass a function along with a value to another function.

2. In this approach, we generate Generic Function. Here function task is not predefined. It perform multiple task not only single task

3. The Function which accept another function as a parameter or return a function is known as 'Higher Order Function'.

4. The Function which is passed to another function or the function which is returned by another function is known as 'Callback Function'.

#Types Of Functions

1. Function decalaration statement : Using function keyword

2. Function as expression / expression function

3. Immediate Invoke Function (IIF)

1. when a function is called as soon as it's object is created is known as Immediate Invoke Function.

2. We have to write the function inside the paranthesis to group it. [using Group operator -> (function code)].

3. The function is not visible(available) outside the scope.

4. After grouping it, we have to use paranthesis to call this function.

5. Immediate Invoke Function execute only once.

4. Arrow Function

1. The main function of arrow function is to reduce the function syntax.

2. Arrow Function is introduced in ES6.

3. If we have only single parameter, it is not necessary to use paranthesis for paramenter.

4. If function have single statement, then block (curly braces) is optional.

5. It does not have its own 'this' property.

6. **Implicit return :-** If there is only one statement and If block is not created then JS Engine will return that statement automatically.

7. **Explicit return :-** If block is created and function is not returning any value, JS Engine will return undefined. To return a value Explicitly from block, we have to use return keyword.If block is created then we have to use return keyword to return value otherwise JS Engine will return undefined.

5. Higher Order Function

1. The Function which accept another function as a parameter or return a function is known as 'Higher Order Function'.

6. Callback Function

1. The Function which is passed to another function or the function which is returned by another function is known as 'Callback Function'.

#Nested Function

1. The function inside another function is called as nested function.

2.

Example :

```
function outer(){
    function inner(){
    }
    return inner
}
```

3. The outer function is known as parent and the inner function is known as child.

4. The inner function is local to outer function, it cannot be accessed from outside.

5.

To use inner function outside, the outer function must return the reference of inner function.

```
function outer(){
    function inner(){
    }
    return inner
}
```

We can now call inner function from outside as follows:

1st Way:

```
let fun=outer();
fun();   // ——> inner() is called
```

2nd Way:

```
outer()();   // -——> inner() is called
```

Hoisting

Hoisting in JavaScript means that variable and function declarations are moved to the top of their scope before the code runs. This allows you to use them before they are actually declared in your code.

Variable Hoisting

1. **var**

   - Variables declared with `var` are moved to the top of their function scope.

   - They start as `undefined` until the line of code where they are assigned a value.

   ```javascript
   console.log(x); // undefined
   var x = 5;
   console.log(x); // 5
   ```

2. **let** and **const**

   - Variables declared with `let` and `const` are also moved to the top of their block scope.

   - They are not initialized until the code reaches their declaration.

   - Using them before they are declared gives a `ReferenceError`.

   ```javascript
   console.log(y); // ReferenceError: Cannot access 'y' before initialization
   let y = 10;
   console.log(y); // 10

   console.log(z); // ReferenceError: Cannot access 'z' before initialization
   const z = 15;
   console.log(z); // 15
   ```

Function Hoisting

1. **Function Declarations**

   - Function declarations are moved to the top of their scope.

   - You can call these functions before they are declared in the code.

   ```javascript

   console.log(sum(2, 3)); // 5

   function sum(a, b) {

     return a + b;

   }

   ```

2. **Function Expressions**

   - Function expressions (functions assigned to variables) are not fully hoisted.

   - Only the variable declaration is hoisted, not the function itself.

   ```javascript

   console.log(multiply); // undefined

   var multiply = function (a, b) {

     return a * b;

   };

   ```

   ```javascript
   ```

```
  console.log(divide); // ReferenceError: Cannot access 'divide' before initialization
  let divide = function (a, b) {
    return a / b;
  };
  ```


Key Points

- **`var` variables**: Moved to the top of the function scope, start as `undefined`.
- **`let` and `const` variables**: Moved to the top of the block scope, not initialized until declared.
- **Function declarations**: Fully moved to the top, can be used before they appear in the code.
- **Function expressions**: Only the variable part is moved, not the function assignment.

Temporal Dead Zone

The Temporal Dead Zone (TDZ) is the time span between variable declaration and its initialization. During this time, the variable declared with let and const cannot be used.

Example of TDZ

```javascript
```

```
console.log(a); // Error: Cannot access 'a' before initialization

let a = 10;

console.log(a); // 10
```

Here, `a` cannot be used before the line `let a = 10;`.

Key Points

1. **Variables with `let` and `const`**:
   - These variables are in the TDZ from the start of the block until they are declared.
   - Trying to use them before the declaration gives an error.

2. **Purpose of TDZ**:
   - The TDZ helps catch mistakes by not allowing the use of variables before they are properly declared.

Simple Example

```javascript
function example() {
  console.log(b); // Error: Cannot access 'b' before initialization
  let b = 20;
  console.log(b); // 20
}
```

example();

```

In this function, `b` is in the TDZ until `let b = 20;` is executed.

#### Comparison with `var`

```javascript
function exampleVar() {
  console.log(c); // undefined (no TDZ for `var`)
  var c = 30;
  console.log(c); // 30
}

exampleVar();
```

For variables declared with `var`, there is no TDZ. They are hoisted to the top and initialized as `undefined`.

Summary

- **TDZ**: Time when `let` or `const` variables can't be used.
- **Error**: Using these variables before they are declared gives an error.
- **Why**: This helps find mistakes in the code.
```

JavaScript Closures Explained Simply

Closure

A closure is a feature in JavaScript where a function remembers and can access variables from outside its own scope, even after the outer function has finished executing.

Example of a Closure

```
function outerFunction() {
  let outerVariable = 'I am outside!';

  function innerFunction() {
    console.log(outerVariable); // This is a closure
  }

  return innerFunction;
}

const closureFunction = outerFunction();
closureFunction(); // Logs: 'I am outside!'
```

Here, `innerFunction` remembers `outerVariable` from `outerFunction` even after `outerFunction` has finished running. This is a closure.

Key Points

1. Function Inside a Function :

- A closure is created when a function is defined inside another function, and the inner function accesses variables from the outer function.

2. Remembering Variables:

- The inner function "remembers" the variables from the outer function's scope even after the outer function has finished running.

3. Practical Use :

- Closures are useful for creating private variables and functions.

Simple Example

```
function createCounter() {
  let count = 0;

  return function() {
    count += 1;
    console.log(count);
  };
}

const counter = createCounter();
counter(); // Logs: 1
counter(); // Logs: 2
```

counter(); // Logs: 3

```

In this example, the inner function increments and logs the `count` variable each time it is called. The `count` variable is remembered between calls because of the closure.

Summary

- Closure : A function that remembers and can use variables from outside its own scope.

- How : Defined inside another function, accessing the outer function's variables.

- Use : Useful for maintaining state or creating private variables and functions.

## Javascript Date() object and its methods

### 1. `getDay()`

- **Description:** Returns the day of the week (0 for Sunday, 1 for Monday, ..., 6 for Saturday).

```javascript

const now = new Date();

const dayOfWeek = now.getDay();


switch (dayOfWeek) {

  case 0:
```
```

```
    console.log("Sunday");
    break;
  case 1:
    console.log("Monday");
    break;
  case 2:
    console.log("Tuesday");
    break;
  case 3:
    console.log("Wednesday");
    break;
  case 4:
    console.log("Thursday");
    break;
  case 5:
    console.log("Friday");
    break;
  case 6:
    console.log("Saturday");
    break;
}
```

2. `getDate()`

- **Description:** Returns the day of the month (1-31).

```javascript
const now = new Date();
const dayOfMonth = now.getDate();
console.log(`Day of the month: ${dayOfMonth}`);
```

3. `getHours()`

- **Description:** Returns the hour (0-23).

```javascript
const now = new Date();
const hour = now.getHours();
console.log(`Current hour: ${hour}`);
```

4. `getMinutes()`

- **Description:** Returns the minutes (0-59).

```javascript
const now = new Date();
const minutes = now.getMinutes();
```

```
console.log(`Current minutes: ${minutes}`);
```

5. `getSeconds()`

- **Description:** Returns the seconds (0-59).

```javascript
const now = new Date();
const seconds = now.getSeconds();
console.log(`Current seconds: ${seconds}`);
```

6. `getMilliseconds()`

- **Description:** Returns the milliseconds (0-999).

```javascript
const now = new Date();
const milliseconds = now.getMilliseconds();
console.log(`Current milliseconds: ${milliseconds}`);
```

7. `getMonth()`

- **Description:** Returns the month (0 for January, 1 for February, ..., 11 for December).

```javascript
const now = new Date();
const month = now.getMonth();

switch (month) {
  case 0:
    console.log("January");
    break;
  case 1:
    console.log("February");
    break;
  case 2:
    console.log("March");
    break;
  case 3:
    console.log("April");
    break;
  case 4:
    console.log("May");
    break;
  case 5:
    console.log("June");
    break;
```

```
case 6:
  console.log("July");
  break;
case 7:
  console.log("August");
  break;
case 8:
  console.log("September");
  break;
case 9:
  console.log("October");
  break;
case 10:
  console.log("November");
  break;
case 11:
  console.log("December");
  break;
}
```

8. `getFullYear()`

- **Description:** Returns the full year (e.g., 2024).

```javascript
const now = new Date();
const year = now.getFullYear();
console.log(`Current year: ${year}`);
```

Math methods:

1. `Math.cbrt()`

- **Description:** Returns the cube root of a number.

```javascript
const number = 27;
const cubeRoot = Math.cbrt(number);
console.log(`Cube root of ${number} is ${cubeRoot}`);
```

```

### 2. `Math.floor()`

- **Description:** Returns the largest integer less than or equal to a given number.

```javascript
const number = 5.8;
const flooredNumber = Math.floor(number);
console.log(`Floor of ${number} is ${flooredNumber}`);
```

3. `Math.min()`

- **Description:** Returns the smallest of zero or more numbers.

```javascript
const num1 = 10;
const num2 = 5;
const minNumber = Math.min(num1, num2);
console.log(`Minimum of ${num1} and ${num2} is ${minNumber}`);
```

4. `Math.max()`
```

- **Description:** Returns the largest of zero or more numbers.

```javascript
const num1 = 10;
const num2 = 5;
const maxNumber = Math.max(num1, num2);
console.log(`Maximum of ${num1} and ${num2} is ${maxNumber}`);
```

5. `Math.pow()`

- **Description:** Returns the base to the exponent power, that is, base^exponent.

```javascript
const base = 2;
const exponent = 3;
const result = Math.pow(base, exponent);
console.log(`${base} raised to the power of ${exponent} is ${result}`);
```

6. `Math.random()`

- **Description:** Returns a pseudo-random number between 0 (inclusive) and 1 (exclusive).

```javascript
```

```
const randomNumber = Math.random();

console.log(`Random number between 0 and 1: ${randomNumber}`);
```

7. `Math.sqrt()`

- **Description:** Returns the square root of a number.

```javascript
const number = 16;

const squareRoot = Math.sqrt(number);

console.log(`Square root of ${number} is ${squareRoot}`);
```

#Practice Questions

1. Write a program to find square and cube of a given number.

2. Write a program to check if a given year is a leap year or not.

3. Write a program to calculate the sum of the first 100 natural numbers.

4. Write a program to check if a given number is even or odd.

5. Write a program to print the sum of all even numbers from 1 to any given number.

6. Write a program to print the sum of all odd numbers from 1 to any given number.

7. Write a program to count the number of digits in a given number.

8. Write a program to calculate the sum of the digits of a given number.

9. Write a program to print the multiplication table of a given number.

10. Write a program to reverse a given number. For example, if the input is 12345, the output should be 54321.

11. Write a program that prints the numbers from 1 to 100. But for multiples of three, print 'Fizz' instead of the number, and for multiples of five, print 'Buzz.' For numbers that are multiples of both three and five, print 'FizzBuzz.'

12. Write a program to calculate the power of a number without using the Math.pow() function.

13. Write a program to check if a given number is prime or not.

14. Write a program to find and print all the prime numbers within 1-100.

15. Write a program to calculate the factorial of a given number.

16. Write a program to print the Fibonacci series up to a given number.

17. Write a program to calculate the sum of the first 20 Fibonacci numbers.

18. Write a program to check if a given number is a perfect number.

19. Write a program to check if a given number is an Armstrong number.

20. Write a program to check if a given number is a strong number.

21. Write a program to check a number whose last digit is 7.

22. Print the number which ends with 7 or is divided by 7.

23. Print numbers up to 500 that are divided by 7 and end with 7.

24. Write a program to print the factors of a number and also print the number of factors of that number.

#Lexical Scope/Scope Chain

1. The ability of js engine to search for a variable in the outer scope when variable is not available in local scope is known as

lexical scope or scope chain.

2. It is ability of child to access variable from outside if its not present in local scope
3. **Lexical scope : A function and global object.**

```
let a = 10;
function test() {
   a++;
   console.log( a );
}
test();
```

Output : 11

When test function is executed js engine looks for ' a ' in local scope. Since it will not available it will look for a in outer scope that is global window object .

4. **Lexical scope : The child function and parent function with a help of closure.**

```
function outer() {
  let a = 10;
     function inner() {
        console.log(a);
     }
  return inner;
}

let res = outer();
res();
```

Output : 10

When the function inner is executed and console.log a is encountered, js engine looks for a in the local scope of function inner.

Since, a is not present and function inner is child of function outer js engine will search for a in the parent function outer scope with the help of closure.

#Closure

1. A closure is created when a function is defined within another function and inner function need to access variables in the outer function's scope.

2. Closure helps to achieve lexical scope from child function to parent function.

3. Closure preserves the state of parent function even after the execution of parent function is completed.

4. A child function will have reference to the closure.

5. Every time a parent function is called the new closure is created.

6. **Disadvantage** : High memory consumption.

#Interview Questions

1. What is JavaScript? (Write 6 points)

2. What is JRE? Name two JRE?

3. Write the names of JavaScript engines for Chrome, Firefox, Edge, and Safari.

4. What is a JS engine?

5. What are the differences between Var, let, and const?

6. What is hoisting?

7. What is the temporal dead zone?

8. What is a function? (Write 6 points)

9. Write the types of functions with syntax.

10. What is a higher-order function and a callback function?

11. What is explicit and implicit return in arrow functions? Provide an example.

12. What is closure? When is a closure created? Explain with an example.

#Array

1. Array is object in javascript.

2. It is non-primitive type of literal.

3. It is a block of memory which is used to store multiple type of value (any type of literal) in same memory block.

4. Array size is dynamic (size is not fixed like JAVA) , it means we can store 'N' number of elements and JS engine will handle memory usage automatically.

5. Values stored inside array are refered as **array elements**.

6. Array elements are arranged in a sequence that is represented by integer number called as **index**. Array index starts from zero to array size - 1 (suppose array has 5 elements it's first index will be - 0 and last index will be 4).

7. We can access the array element with the help of array object reference , square brackets and index **(array_object_ref[index])**.

8. If we try to access the index that it greater than the array length we will get undefined.

9. Array elements should be separated by comma(,)

#Ways To Create Array

1.

By using square brackets [] and literals.

```
let arr = [];
```

```
// empty array

let arr = [10,20,30];
// array with literals
```

2.

By using new keyword and Array() constructor.

```
let arr = new Array();
//empty array

let arr = new Array(10,20,30)
//array with literals -> [10,20,30]
```

NOTE : Here , 'arr' is a variable which holds the reference of array object. To access array element at index -> 1Syntax : array_object_ref[index]Example : console.log(arr[1]); // 20

#Array Methods

1. push(value) method

1. It is used to insert element at last of array.

2. It returns the length of array.

3.

Example:

```
let arr=[10,20,30,40,50];
arr.push(100);
output: [10,20,30,40,50,100]
```

2. pop() method

1. It is used to delete element from last index of array.

2. It returns deleted element.

3.

Example:

```
let arr=[10,20,30,40,50];
arr.pop();
output: [10,20,30,40]
```

3. unshift(value)

1. It is used to insert element at first index of array.

2. It returns array length

3.

Example:

```
let arr=[10,20,30,40,50];
arr.unshift(200);
output: [200,10,20,30,40,50]
```

4. shift() method

1. It is used to delete element from first index of array.

2. It returns deleted element.

3.

Example:

```
let arr=[10,20,30,40,50];
arr.shift();
output: [20,30,40,50]
```

5. splice() method

1. It is used to perform insertion , deletion and updation in array.

2. It will modify the original array.

3. It returns array of deleted elements.

4.

Example:

arr_ref.splice(a,b,c)
a - starting index
b - number of elements to be deleted
c - elements to be inserted

5.

Example : Delete three elements from index 1.

```
let arr=[10,20,30,40,50];
arr.splice(1,3); // deleted: [20,30,40]
console.log(arr);
```

Output : [10,50]

6.

Example : Update value at index 3 to 500.

```
let arr=[10,20,30,40,50];
arr.splice(3,1,500);
console.log(arr);
```

Output : [10,20,30,500,50]

7.

Example : Insert 100,200,and 300 from index 2.

```
let arr=[10,20,30,40,50];
arr.splice(2,0,100,200,300);
console.log(arr);
```

Output : [10,20,100,200,300,30,40,50]

6. slice() method

1. It is used to copy array elements.

2. It will not modify the original array

3. It returns array of copied elements.

4.

Syntax :

arr.slice(a,b);
a - starting index
b - last index

Here, last index is excluded -> last index -1

5.

Example : Copy array from index 0 to 2.

```
let arr=[10,20,30,40,50];
let copy_elements = arr.slice(0,3);
console.log(copy_elements);
```

Output: [10,20,30]

7. indexOf() methods

1. It used to get the index of array element.

2. If element is available -> it returns element's index.

3. If element is not available -> it returns -1.

4.

Syntax :

arr.indexOf(a,b)
a - value to be searched
b - search starting index

If we does not pass last argument , it will 0 by default.

5.

Example : Check given array has element 30 or not and search from index 0 and 3 , if present print index.

```
let arr=[10,20,30,40,50];
console.log(arr.indexOf(30)); // 2
console.log(arr.indexOf(30,3)); // -1
```

8. includes() method

1. It is used to check element is available or not.

2. If element is available -> returns true.

3. If element is not available -> returns false.

4.

Syntax :

arr_ref.includes(a,b);
a - value to be searched
b - search starting index

If we does not pass last argument , it will 0 by default.

5.

Example : Check given array has element 30 or not and search from index 0 and 3 , if present print true.

```
let arr=[10,20,30,40,50];
console.log(arr.includes(30)); // true
console.log(arr.includes(30,3)); // false
```

9. reverse() method

1. It is used to reverse the array.

2. It will modify the original array.

3.

Example:

```
let arr=[10,20,30,40,50];
console.log(arr.reverse());
```

Output: [50,40,30,20,10]

10. sort(callback) method

1. It will modify the original array

2. If callback returns -ve value -> it will sort in ascending order

3. If callback returns +ve value -> it will sort in decending order.

4. If callback returns 0 value -> it will not sort.

5.

Example : Sort array in ascending order.

```
let arr = [100, 2000, 380, 940, 50, 0, 2];
console.log(arr.sort((a, b) => a - b));
```

Output: [0, 2, 50, 100, 380, 940, 2000]

6.

Example : Sort array in descending order.

```
let arr = [100, 2000, 380, 940, 50, 0, 2];
console.log(arr.sort((a, b) => b - a));
```

Output: [2000, 940, 380, 100, 50, 2, 0]

11. foreach(callback)

1. It is a higher order function.

2. It is used to iterate over array elements and index.

3. It doesnot return anything , so js engine implicitly returns undefined.

4.

Syntax :

```
arr_ref.foreach((value,index,array)=>{
      // statements
})
```

5.

Example : Print Even numbers from given array.

```
const arr = [1, 2, 3, 4, 5];
arr.forEach((val)=> {
  if(val % 2 === 0)
  {
    console.log(val+" "+"is even number;");
  }
});
```

12. map(callback)

1. It is a higher order function.

2. It is used to iterate over array.

3. It will not modify original array.

4. It returns a new array.

5. **The value returned by callback function will be inserted in new array , if it doesnot return anything 'undefined' will be stored**.

6.

Syntax :

```
arr_ref.map((value,index,array)=>{
    // statements
})
```

7.

Example: Create new array where each element of given array is multiple of 8.

```
let arr=[10,20,30,40,50];
let new_arr = arr.map(value => value * 8 );
console.log(new_arr);

Output: [80,160,240,320,400]
```

13. filter(callback)

1. It is a higher order function.

2. It is used to iterate over array.

3. It will not modify original array.

4. It returns a new array.

5. **Here, element will be inserted in new array only when callback function returns true.**

6.

Syntax :

```
arr_ref.filter((value,index,array)=>{
    // statements
})
```

7.

Example: Create new array where elements are greater than 40.

```
let arr=[10,20,30,40,50,60,70];
let new_arr = arr.filter(value => {
  if(value > 30)
    return true;
});
console.log(new_arr);

Output: [40,50,60,70]
```

14. reduce(callback,initial_value)

1. It is a higher order function.

2. It is used to iterate and conclude result to a single value.

3. It will not modify original array.

4. **It returns a single value.**

5. **Here , single value is returned after complete iteration of array. Value is stored in a variable which is used to result , we refer it as accumulator.**

6.

Syntax :

```
arr_ref.reduce((accumulator,value,index,array)=>{
    // statements
},initial_value_of_accumulator)
```

If we does not pass initial value of accumulator first element of array will be stored automatically.

7.

Example : Find the sum of all elements of array.

```
let arr=[10,20,30,40,50,60,70];
let result = arr.reduce((acc,value) => {
  acc = acc + value;
  return acc;
},0);
console.log("Sum of all elements : ",result);

Output: Sum of all elements :  280
```

15. Array.isArray(literal)

1. It is used to check given literal is array or not.

2. If it is array -> it will return true.

3. if it is not array -> it will return false.

5.

Example: Check given literal is array or not.

```
console.log(Array.isArray({}));  //false
console.log(Array.isArray(10));  //false
console.log(Array.isArray([10,20,30]));  //true
```

16. Array.from(literal)

1. It is used to convert iterable literals (like object or string) to array.

2. If literal is iterable -> it returns new array of elements.

3. If literal is not iterable -> it returns empty array.

4.

Example : Convert string to Array.

```
const str = "hello";
const arr = Array.from(str);
console.log(arr);

Output: ["h", "e", "l", "l", "o"]
```

#Object

1. An Object is a block of memory which has state(variable) , behaviour(methods) and where we can store heterogenous data.

2. An object is a collection of key-value pairs that can contain various data types, such as numbers, strings, arrays, functions, and other objects.

3. In one object we can have multiple key value pair and it should be separated by ',' comma.

4. We can access value of object using (.) Operator or square bracket [] , object reference and key_name.

#Object Key (Property)

1. Object key (property) will be automatically converted into string by js engine.

2. If keys name are in Number , js engine will convert them into string and arrange them in ascending order.

3. To write space separated key names , we have to enclose key name with double quotes.

4. If we want to give computed or user defined property then we have to use square brackets and variable name.

5.

If key-name is same as variable name which hold the value , instead of writing two times we can write varaiable name only once.

```
let phone = 8800425635;
let obj = {
  phone,
  // phone:phone
};
```

#Ways To Create Object

1.

By using curly braces { } and literals.

```
let obj = {}
// empty object

let obj = { name:"chombi",age:16}
// object with literals
```

2.

By using new keyword and Constructor.

```
let obj = new Object();
// {} empty object

let obj=new Object({ name:"chombi"});
// { name:"chombi"} object with literals
```

3. By using new keyword and Constructor function

4. By using class

#Access Object Value

1.

By using dot operator (.) and key name.

```
let obj = { name:"chombi",age:16}

console.log(obj.name)// chombi
console.log(obj.age)// 16
```

2.

By using square brackets ([]) and key name.

```
let obj = { name:"chombi",age:16}

console.log(obj["name"])// chombi
console.log(obj["age"])// 16
```

3. If we try to access property which is not available in object we will get undefined.

#Object Methods

1. In JavaScript, object methods are functions that are attached to the object, and can be called on that object reference.

2. To call a function , we use square brackets instead dot operator.

3.

Here, speak is a variable which holds the function reference.

```
let obj1 = { name: "chombi",
age: 16,
speak: function () {
console.log('i can speak');
} }
console.log(obj1["speak"]());
//i can speak
```

4.

Access object property inside function - function declared with function keyword.

```
let obj1 = { name: "chombi",
age: 16,
speak: function () {
console.log('My name is' + this.name+',age' + this.age+' and i can speak');
} }
console.log(obj1["speak"]());
//My name is chombi , age 16 and i can speak
```

Here, we can access object property, by using 'this' keyword.

5.

Access object property inside function - Arrow function.

```
let obj1 = {
  name: "chombi",
  age: 16,
  speak: () => {
    console.log(
      "My name is" + obj1.name + " , age " + obj1.age + " and i can speak"
    );
  },
};
console.log(obj1["speak"]());
//My name is chombi , age 16 and i can speak
// Here, we can access object property, by using object reference.
```

Here , we can can access object property , by using object reference because arrow function is not having 'this' property.

#Add Key Value In Object

1. To add key-value pair we can using dot operator and square brackets

2.

By using dot operator (.) and key name

```
let obj = { name:"chombi",age:16}

obj.country = "india";
//new key-value added in object
// {
name:"chombi",
age:16,
country:"india",
}
```

NOTE : If property is already available with same name it will updated with new value.
Example:

```
let obj = { name:"chombi",age:16 }
obj.age = 18;
//age property value is updated
// {
name:"chombi",
age:18,
}
```

#Check Property Is Available In Object Or Not

1.

We can check using "in" operator.

Syntax : "property name" in object_name

```
let obj = { name:"chombi",age:16}
```

```
obj.country = "india";
//new key-value added in object
// {
name:"chombi",
age:16,
country:"india",
}
```

We can check using "in" operator.

```
let obj = { name:"chombi",age:16}

console.log("name" in obj )// true
console.log("age" in obj )// true
console.log("city" in obj )// false
```

#Copy Of Object

1. We can create copy of two types :

1. Shallow copy

2. Deep Copy

2. Shallow copy

1. The copy of object that is directly connected with original object is called as shallow object.

2. Here, we store reference of original object in a new varaiable , now new variable starts pointing to same memory block.

3. So if we make any changes in copy , it will be reflected to original object because both variables are pointing to same memory block.

4.

```
let obj = { name:"chombi",age:16}

let obj_cpy = obj;
```

```
//reference of obj is copied in obj_cpy

obj_copy.age=20;
console.log(obj_copy);
//{ name:"chombi",age:20 }
console.log(obj);
//{ name:"chombi",age:20 }
```

3. Deep copy

1. The copy in which original object is not connected with it's copy , is called as Deep copy.

2. Here , we create separate empty object and after that we copy key-value pair of original object into new empty object.

3. Now , if we make any changes in copy , it will not be reflected to original object because we have create separate memory blocks.

4.

Create copy using for loop.

```
let obj1 = {name:"chombi",age:16}

let obj2 = { }
// new empty object


for (prop in obj1) {
obj2[prop] = obj1[prop]
console.log(obj2)
}
//copy key-values into new object

obj2.age=20;
console.log(obj2);
//{ name:"chombi",age:20 }
console.log(obj1);
//{ name:"chombi",age:16 }
```

#Object In-Built Methods

1. Object.keys(obj_ref)

1. Returns an array of given object's property names.

2.

```
const obj = { a: 1, b: 2, c: 3 };
console.log(Object.keys(obj));
// Output: ["a", "b", "c"]
```

2. Object.values(obj_ref)

1. Returns an array of given object's values.

2.

```
const obj = { a: 1, b: 2, c: 3 };
console.log(Object.values(obj));
// Output: [1,2,3]
```

3. Object.entries(obj_ref)

1. Returns an array of key-value pairs in an array.

2.

```
const obj = { a: 1, b: 2, c: 3 };
console.log(Object.entries(obj));
// Output: [[a,1],[b,2],[c,3]]
```

4. Object.assign(target_obj,src1,...,srcn)

1. Copies key-value pair from one or more source objects to a target object.

2.

```
const target = { a: 1, b: 2 };
const source = { c: 4, d: 5 };
const result = Object.assign(target, source);
console.log(result);
// Output: { a: 1, b: 2, c: 4, d: 5 }
```

#What Is JSON?

1. JSON stands for javascript object notation.

2. It is data only format to represent values and objects.

3. It is used to transfer data between applications through apis.

4. JSON keys must be strings enclosed in double quotes.

5. It supports six data types: object, array, string, number, boolean, and null.

6. It supports nested structures, allowing objects and arrays to be nested within each other.

#JSON Methods

1. **JSON.stringify(value)**

- JSON.stringify() is a method that converts a JavaScript object or value into a JSON string.

- Returns JSON.

- It does not support : function properties,symbolic keys and values and Properties that store undefined.

2. **JSON.parse(value)**

- JSON.parse() is a method that converts JSON string into JavaScript object or value.

- Returns plain javascript object.

#Introduction

1. Call , Apply and Bind methods are used to store the object reference in 'this' keyword of function.

2. When function's 'this' have reference of object, then we can access states and behaviours of that object.

3.

For practice we will use these objects as reference.

```
let human1 = {
  name: "Chombi",
  age: 20,
};

let human2 = {
  name: "Dinga",
  age: 19,
};

let human3 = {
  name: "Nimbi",
  age: 18,
};
```

Below function we will use to access object's properties by using call, apply and bind methods.

```
function detailsAll(a, b, c) {
  console.log("Name : " + this.name);
  console.log("Age : " + this.age);
  console.log("value of a : " + a);
  console.log("value of b : " + b);
  console.log("value of c : " + c);
}
```

#Call

1. Call method accepts object reference as first argument And accepts 'n' number of arguments.

2. Here, arguments are passed to the function's parameter list.

3. It will call the function Immediately.

4.

Example : Print name , age of object human1 and print function arguments.

detailsAll.call(human1, 10,20,30);

Output -
Name : Chombi
Age : 20
value of a : 10
value of b : 20
value of c : 30

#Apply

1. Apply method accepts of 2 arguments where object reference is first argument and 2nd argument is the array of arguments.

2. Here arguments are passed to the function's parameters list.

3. It will call the function immediately

4.

Example : Print name , age of object human2 and print function arguments.

detailsAll.apply(human2,[11,22,33]);

Output -
Name : Dinga
Age : 19
value of a : 11
value of b : 22
value of c : 33

#Bind

1. Bind method accepts object reference as 1st argument and excepts 'n' number of arguments.

2. Here 'n' number of arguments are passed to the function's parameter list.

3. It will not call the function immediately.

4. It returns a new function in which 'this' Keyword is pointing to the object reference we have passed.

5. To execute the function we need function reference and parenthesis

6.

Example : Print name , age of object human3 and print function arguments.

```
let func = detailsAll.bind(human3, 77,88,99);
func();

Output -
Name : Nimbi
Age : 18
value of a : 77
value of b : 88
value of c : 99
```

#Constructor Function

1. A function which is used to create an object is known as constructor function.

2. A constructor function behaves like blueprint or template for object , and there is no need to write code again and again

3. It helps us to create multiple objects of same type.

4. Syntax : function identifier (parameter,...){ }

5. If the function is designed to use as a constructor than name of function should be upper camel case.

6. The list of parameter provided to the function will be treated as keys/properties of the object.

7. The argument pass when function is called will be value of object.

8. We can copy the values into the keys of the object from parameter using this keyword.

9. We can create a object using the constructor function with the help of new keyword.

10. To create constructor function we will not use arrow function because they does not have 'this' keyword .

11. Synatx : let variable = new function_name(arguments)

12.

Example :

```
function Car(model,color,engine) {
this.model = model;
this.color = color;
this.engine = engine;
}

let car1 = new Car(1021,"red","V8");
console.log(car1);
// { model:1021,color:"red",engine:"V8" }
```

#This Keyword

1. It is a keyword.

2. It is a variable , which holds the reference.

3. In GEC it holds the address of window object.

4. It is a local variable of every function in js, and holds the address of window object. Except in Arrow function (for arrow function is stores undefined).

5. Inside object methods, 'this' holds the reference of current object(not in arrow function).

This keyword notes

Global Context

In javascript this keyword refers to the object that executes the current function.

In the global execution context (outside of any function), `this` refers to the global object. In a web browser, this is usually the `window` object.

```javascript
console.log(this); // In a browser, this will log the window object
```

Function Context

Inside a regular function, `this` refers to the window object.

```javascript
function showThis() {
    console.log(this); // window (global object)
}

showThis();
```

Object Method

When a function is called as a method of an object, `this` refers to the object the method is called on.

```javascript
const person = {
    name: 'Prasad',
    greet: function() {
        console.log(this.name); // Logs 'Prasad'
    }
};

person.greet();
```

Constructor Function

When a function is used as a constructor with the `new` keyword, `this` refers to the newly created object.

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

const bob = new Person('Bob');
console.log(bob.name); // Logs 'Bob'
```

Arrow Functions

Arrow functions do not have their own `this`. Instead, they inherit `this` from the surrounding lexical context.

```javascript
const person = {
    name: 'Alice',
    greet: () => {
        console.log(this.name);
    }
};

person.greet(); // Logs undefined because `this` is inherited from the global context
```

Event Handlers

In an event handler, `this` refers to the element that received the event.

```javascript
document.getElementById('myButton').addEventListener('click', function() {
    console.log(this); // Logs the button element
});
```

`call`, `apply`, and `bind` Methods

You can explicitly set `this` using `call`, `apply`, and `bind`.

- **`call`**: Calls a function with a given `this` value and arguments.

```javascript
function greet() {
    console.log(this.name);
}

const person = { name: 'Alice' };
greet.call(person); // Logs 'Alice'
```

- **`apply`**: Calls a function with a given `this` value and an array of arguments.

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

const person = { name: 'Alice' };
greet.apply(person, ['Hello']); // Logs 'Hello, Alice'
```

- **`bind`**: Returns a new function, with a given `this` value.

```javascript
function greet() {
    console.log(this.name);
}

const person = { name: 'Alice' };
const boundGreet = greet.bind(person);
boundGreet(); // Logs 'Alice'
```

Summary

- **Global context**: `this` is the global object (`window` in browsers).
- **Function context**: `this` is the global object in non-strict mode, `undefined` in strict mode.

- **Object method**: `this` is the object the method is called on.

- **Constructor function**: `this` is the new object being created.

- **Arrow functions**: `this` is inherited from the surrounding lexical context.

- **Event handlers**: `this` is the element that received the event.

- **`call`, `apply`, `bind`**: Methods to explicitly set `this`.

#Destructuring

1. The process of extracting the values from the array or object into the variables is known as destructuring.

2. The two most used data structures in JavaScript are Object and Array, both allows us to unpack individual values into variables.

#Object Destructuring

1. The process of extracting the values from the object into the variables is known as object destructuring.

2. All the key names provided on LHS are consider as variable and these variables should be declared and written inside curly braces.

3. The variable name should same as object key name

4. Js engine will search for the key inside the object.

5. If the key is present , the value is extracted and copy into variable.

6. If the key is not present , undefined is store in the variable.

7. After destructuring , we can directly access variable names , without using object reference.

8.

Example :

```
let obj = {
name:"chombi" ,
age:16 ,
```

```
}

let {name,age,country} = obj;
console.log(name);
// chombi
console.log(age);
// 16
console.log(country );
// undefined
```

Here , we are trying extract name , age and country from obj. name and age is present in obj but country is not , so inside country js engine stored undefined and for name , and age we have respective values.

#Array Destructuring

1. The process of extracting the values from the array into the variables is known as array destructuring.

2. All the key names provided on LHS are consider as variable and should bewritten inside square brackets.

3. Js engine will extract the array values and stored them variables in the same order as they are present inside array.

4. if we try to access value which is not present inside array , js engine will store undefined inside that variable.

5.

Example :

```
let arr = [10,20,30,40,50]

let [a,b,c,d,e,f] = arr;
console.log(a);
// 10
console.log(b);
// 20
console.log(c);
// 30
console.log(d);
// 40
```

```
console.log(e);
// 50
console.log(f);
// undefined
```

Here , we are trying to extract value from array into variables a,b,c,d,e,f. As we are already learnt values will be extracted and stored into variable in the same sequence they are available inside array , so we have value inside a,b,c,d,e but not inside f because at sixth position no value is present and js engine stored undefined in it.

#Destructuring In Function

1. We can destructure array or object in function parameter so that we can access value directly.

2.

Destructuring object in function parameter

At the time of object destructuring , we have to make sure variable name is same as object key name and write within curly braces.

```
function details({name,age}) {
console.log(name);
// chombi
console.log(age);
// 16

}

let obj = {
name:"chombi" ,
age:16 ,
}

details(obj) // function call
```

Here , we have passed object as an argument to details function , and we have destructured values in parameter only.

3.

Destructuring array in function parameter

At the time of array destructuring , we have to keep variables between square brackets.
Values will be destructured in the same order , they are available in array.

```
function details([a,b,c,d,e]) {
console.log(a);
// 10
console.log(b);
// 20
console.log(c);
// 30
console.log(d);
// 40
console.log(e);
// 50

}

let arr = [10,20,30,40,50];
details(arr) // function call
```

Here , we have passed array as an argument to details function , and we have destructured values in parameter only.

#Rest And Spread

1. Rest parameter

1. Rest parameter is used to accept multiple values , stored them in an array and array's reference will stored the variable that we have used for rest.

2. Rest can accept n number of values and stored them in an array.

3. To make a variable rest , we have to put '...' before variable name.

4. Syntax : let ...variable_name;

5. We can use rest in function when we don't know the exact number arguments.

6. **In function , there can be only one rest parameter and it should be the last parameter.**

7.

```
function details(a,b,...z) { console.log(a);
//10
console.log(b);
//20
console.log(z[0]);
//30
console.log(z[1]);
//40
console.log(z[2]);
//50
console.log(z[3]);
//60
console.log(z[4]);
//70

}
details(10,20,30,40,50,60,70);
//function call
```

Here , if we pass two values it will be stored in a,b respectively and further value will stored by rest in an array. We can pass n number of values.

8.

Uses of REST parameter.

1. REST parameter can be used in function definition parameter list to accept multiple values.
2. It can also be used in array and object destructuring.

9.

If we pass literals to three dots (...) , it will accept all literals and behave as a rest parameter.

2. Spread operator

1. It is used to unpack elements from iterables (like array or object).

2.

Use cases :

- The unpack data can be sent to the function as an argument by using spread operator in the function call statement.

```
let arr = [10, 2, 3, 40, 500, 6];

function sum(...data) {
  let acc = 0;
  for (let val of data) {
    acc = acc + val;
  }
  return acc;
}
let result = sum(...arr);
console.log(result);
```

Output: 561

```
// ...data - rest parameter
// ...arr - spread operator
```

- We can ask the spread operator to store the unpack element in array object by using spread operator inside [ ] brackets.

```
let new_arr = [...arr];
console.log(new_arr);

Output: [10, 2, 3, 40, 500, 6];
```

- We can ask the spread operator to store the unpack element in object by using spread operator inside { } brackets.

```
let human1 = {
  name: "Chombu",
  age: 21,
};

let human2 = { ...human1 };

console.log(human2);

Output: { name: "Chombu", age: 21, };
```

3.

If we do not pass literals to three dots (...) , it will unpack all literals and behave as a spread parameter.

#Introduction to Prototype

1. In JavaScript, every function is associated with an object called as prototype.

2. It serves as a blueprint or a template from which other objects can inherit properties and methods.

3. **Prototypes are used to achieve inheritance in JavaScript.**

4. When you access a property or method on an object, JavaScript first looks for that property or method directly on the object itself. If it doesn't find it there, then it looks at the object's prototype, and

continues up the prototype chain until it either finds the property/method or reaches the end of the chain (where the prototype is null).

5. This allows us to define common properties and methods in a prototype, and all objects that inherit from that prototype will have access to those properties and methods.

#_ _proto_ _

1. The reference of prototype object is stored in _ _proto_ _

2. When an object is created a prototype object is not created instead the object will have reference of the prototype object (from which properties are to be inherited) referred using _ _proto_ _.

#Prototypal Inheritance

1. Prototypal inheritance is a fundamental concept in JavaScript's object-oriented programming model.

2. It allows objects to inherit properties and methods from other objects, forming a prototype chain.

3. In JavaScript inheritance is achieved using prototype hence it is known as Prototypal Inheritance.

#Prototype Chain

1. The prototype chain is a mechanism in JavaScript that allows objects to inherit properties and methods from other objects.

2. The prototype chain forms a hierarchy of objects, where each object's prototype is linked to its parent object's ptototype, creating a chain of inheritance.

3. By following this chain, objects can inherit properties and methods from their prototype objects.

#Dom

1. The Document Object Model (DOM) is a programming interface for web documents that represents the HTML or XML document as a tree structure, where each node represents an element, attribute, or piece of text in the document.

2. When a web page is loaded, the browser creates a DOM tree that represents the document's structure and content.

3. Each node in the tree is represented as js object , which we can access and manipulate using the DOM API.

4. Here , Html elements , comments , text , content , etc are refered as nodes of DOM tree.

#Dom Api

1. The DOM API (Application Programming Interface) is a set of programming interfaces and methods that allow developers to interact with the DOM tree and manipulate the content and structure of web documents.

2. The DOM API provides a standardized way to create, modify, and delete elements and attributes, change styles and classes, handle events, and more.

#Html Structure

1.

Reference Html structure

```
<body>
  <h1>Falling In Love With Javascript</h1>
  <div class="container">
    <div class="item item1" id="itemone">1</div>
    <div class="item item2">2</div>
    <div class="item item3">3</div>
    <div id="itemfour" class="item item4">4</div>
    <div class="item item5">5</div>
  </div>
  <p>hello i'm paragraph</p>
</body>
```

#Target Elements

1. getElementById('id_name')

1. It returns reference of single element object where id_name matches

2.

Example:

```
let divone = document.getElementById("itemone");
console.log(divone);
```

2. getElementsByClassName('class_name')

1. It returns htmlcollection all elements matches with class name.

2.

Example: Apply backgroundColor,margin,fontsize and padding on each div.

```
let div_child = document.getElementsByClassName("item");
console.log(div_child);
```

3. getElementsByTagName('tag_name')

1. It returns htmlcollection all elements matches with tag name.

2.

Example: Display parent div as flexbox.

```
let divs = document.getElementsByTagName("div");
console.log(divs)



divs[0].style.backgroundColor = "yellow";
divs[0].style.padding = "10px";
divs[0].style.display = "flex";
divs[0].style.gap = "10px";
divs[0].style.justifyContent = "space-between";

for (let i = 1; i < divs.length; i++) {
  divs[i].style.backgroundColor = "blue";

  divs[i].style.fontSize = "32px";
  divs[i].style.padding = "10px";
  divs[i].style.color = "white";
}
```

4. querySelector('css_selector')

1. It returns reference of the first element that matches a specified CSS selector.

2.

Example: Change fontSize of first div with 'item' class.

```
let ele = document.querySelector(".item");
ele.style.fontSize = "52px";
console.log(ele);
```

5. querySelectorAll('css_selector')

1. It returns Nodelist of all elements that matches a specified CSS selector.

2.

Example: Change fontWeight of all div with 'item' class.

```
let eles = document.querySelectorAll(".item");
for (ref of eles) {
  ref.style.fontWeight = "bold";
}
console.log(ele);
```

#Create And Insert Element

1. createElement('tag_name')

1. It is used to create a new html element of the specified type and returns a reference to it as a javascript object.

2.

Example: Create section tag.

```
let sec = document.createElement("section");
console.log(sec)
```

2. appendChild(element)

1. it is used to insert the element as last child.

2.

Example: Insert the section tag inside div tag having class 'container'.

```
let sec = document.createElement("section");
let pdiv = document.getElementsByClassName("container")[0];
pdiv.appendChild(sec);
```

.

3. insertAdjacentElement('posiiton',element)

1. It is used to insert an element as a child or sibling.

2. Positions : beforebegin ,afterbegin , beforeend , afterend.

3.

Example: Show how to display element as child and sibling of div having class 'container'.

```
pdiv.insertAdjacentElement("beforebegin", sec);
pdiv.insertAdjacentElement("afterend", sec);
pdiv.insertAdjacentElement("afterbegin", sec);
pdiv.insertAdjacentElement("beforeend", sec);
```

#Insert Text And Elements

1. textContent

1. It is used to insert text inside element.

2.

Example: Insert "Hello" text inside p tag.

```
let p = document.getElementsByTagName("p")[0];
p.textContent="Hello";
```

Example: Insert "Hello" text inside p tag and preserve previous text also.

```
let p = document.getElementsByTagName("p")[0];
p.textContent +="Hello";
```

2. innerHTML

1. It is used to insert text and html tag inside element.

2.

Example: Insert "Hello" inside p tag .

```
let p = document.getElementsByTagName("p")[0];
p.innerHTML="<strong>Hello</strong>";
```

Example: Insert "Hello" inside p tag and preserve previous text and element.

```
let p = document.getElementsByTagName("p")[0];
p.innerHTML+="<strong>Hello</strong>";
```

#Insert And Remove Attribute

1. setAttribute('attribut_name','value')

1. It is used to insert the attribute to an element.

2.

Example: Insert id="chombi" to third div in the container.

```
let divs = document.getElementsByClassName("item");
divs[2].setAttribute("id", "chombi");
```

2. removeAttribute('attribut_name')

1. It is used to remove attribute from an element.

2.

Example: Remove id attribute from third div of container.

```
let divs = document.getElementsByClassName("item");

divs[2].removeAttribute("id");
```

#Traverse Html Nodes

1. parentElement

1. It returns the reference of parent html element.

2.

Example: Print parent element of div whose class is "item".

```
let div_child = document.getElementsByClassName("item");
console.log(div_child[1].parentElement);
```

2. nextElementSibling

1. It returns the reference of next html node sibling.

2.

Example: Print next sibling element of third div whose class is "item".

```
let div_child = document.getElementsByClassName("item");
console.log(div_child[1].parentElement);
```

3. previousElementSibling

1. It returns the reference of next html node sibling.

2.

Example: Print previous sibling element of third div whose class is "item".

```
let div_child = document.getElementsByClassName("item");
console.log(div_child[2].previousElementSibling);
```

4. children

1. It returns htmlcollection of html all childs element.

2.

Example: Print childrens elements of div whose class is "container".

```
let pdiv = document.getElementsByClassName("container")[0];
console.log(pdiv.children);
```

5. childNodes

1. It returns Nodelist of all types of nodes like string , text , comment , etc.

2.

Example: Print all child nodes of div whose class is "container".

```
let pdiv = document.getElementsByClassName("container")[0];
console.log(pdiv.childNodes);
```

#Remove Html Element

1. remove()

1. It is used to delete html element.

2.

Example: Remove p element from DOM.

```
let p = document.getElementsByTagName("p")[0];
p.remove();
```

#What Are Events In Javascript?

1. Actions performed by user on browser are refered as events.

2. Whenever event occurs browser creats an object which contains all information about the event and object on which event occured.

3. Example: Like if user click on <h1> tag , browser automatically creates an object which have information about h1 tag and type of event occured (here , type is 'click').

#What Is Event Object?

1. Event object is a object created by the browser when user perform some action , which holds all information about type of event occured and the element on which the event occurred.

2. Event object is passed to respective event handler , every time event occured.

3. So, we can access event object in callback function.

#What Are Event Listeners?

1. Event listeners are functions that wait for a specific event to occur and then execute js code (callbacks) assigned to it.

2. By writting logic in callback , we can control what to do when event occurs like change text color, hide or show, etc.

#How To Attach Event Listeners?

1. We can attach event listeners by three ways :

1. As an HTML attribute

- In this approach , we attach event listener as a attribute in opening tag.

- Syntax : <tag onevent_name='function_reference()'>

- We have to prefix 'on' before the event name.

- When we pass a function , we have to function_reference and parenthesis.

- Example : When user clicks on div , it's color should change to red.

```
<div onclick="handleButtonClick()">Hello Honney Bunny 🐰</div>

<script>

   function handleButtonClick() {
     let div = document.querySelector('div');
     div.style.backgroundColor='red';
   }

</script>
```

2. As a JS Property

- In this approach , first we need the reference of element then we attach listener to it (the way we add property to an object).

- Syntax : element.onevent_name = function_reference;

- We have to prefix 'on' before the event name.

- At the time of attaching listener to element , we just need to pass function reference to it.(listener will automatically call that function).

- Example : When user clicks on div , it's color should change to red.

```
<div>Hello Honney Bunny 🐰</div>

<script>

   let div = document.querySelector('div');
   div.onclick = handleButtonClick;

   function handleButtonClick() {
     div.style.backgroundColor='red';
   }

</script>
```

3. Using addEventListener Method

- In this approach , first we need the reference of element then we attach listener to it (the way we add property to an object).

- Syntax : element.addEventListener(event_name,function_reference)

- First argument will be the event name (no need prefix 'on') and pass function reference.

- Example : When user clicks on div , it's color should change to red.

```
<div>Hello Honney Bunny 🐰</div>

<script>

   let div = document.querySelector('div');
```

```
    div.addEventListener('click',handleButtonClick)

    function handleButtonClick() {
      div.style.backgroundColor='red';
    }

</script>
```

#Type Of Events

1.

Keyboard Events

Event Name	Info
keydown	Triggered when a key is pressed down.
keyup	Triggered when a key is released.
keypress	Triggered when a key is pressed and released.

2.

Mouse Events

Event Name	Info
click	Triggered when the mouse is clicked.
dblclick	Triggered when the mouse is double-clicked.
mousedown	Triggered when a mouse button is pressed down.
mouseup	Triggered when a mouse button is released.
mousemove	Triggered when the mouse pointer moves.

3.

Form Events

Event Name	Info
submit	Triggered when a form is submitted.
reset	Triggered when a form is reset.
change	Triggered when the value of a form element changes.

#Exception

1. Exception is an unwanted or unexpected problem, which occurs during the execution of a program.

2. If unexpected problem occurs at runtime , program execution will be disrupted.

3. Example: file input/output errors, or network communication errors.

#Exception Handling

1. The process of handling unwanted or unexpected problem at runtime without affecting program execution is called as

Exception handling

.

2. We can handle these exceptions using try, catch and finally.

#Try , Catch And Finally

1. try

1. The try block contains the code that might throw an exception.

2. If an exception occurs within the try block, control is immediately transferred to the corresponding catch block.

2. catch

1. The catch block is used to handle exceptions that are thrown within the corresponding try block.

2. It contains code that will be executed if an exception is thrown, and is typically used to handle the exception in some way (e.g., logging an error message, displaying a user-friendly error message, or taking some other appropriate action).

3. finally

1. The finally block is used to specify code that should be executed regardless of whether or not an exception is thrown.

2. This block is typically used to clean up resources (e.g., closing files or closing network connections) that were allocated within the try block.

4. Example

1.

```
try {
  // Code that might throw an exception

  const num = Number(prompt("Enter a number"));
  if (isNaN(num)) {
    throw new Error("Invalid number");
  }
  console.log("You entered the number " + num);
} catch (err) {
  // Code to handle the exception

  console.error("An error occurred: "+err.message);
} finally {
  // Code to be executed regardless of whether an exception was thrown or not

  console.log("Execution complete");
}
```

#Throw

1. **throw** is a keyword used to manually trigger an exception.

2. When throw is used, it causes the JavaScript interpreter to stop executing the current block of code and transfer control to the nearest catch block that can handle the exception.