**JavaScript Interview Questions**

**What are the data types present in JS?**

There are two types of data types in JavaScript.

* Primitive data type
* Non-primitive (reference) data type

**What is Primitive data type in JS?**

Primitive data type has 7 Datatypes

**1. Number, 2. String, 3. BigInt, 4. Boolean, 5. Undefined, 6. Null, 7. Symbol**

1: Number: The number type represents both integer and floating-point numbers

let num = 2; let float\_num = 2.5;

2: String: Any value enclosed within double quotes (“”) or single quotes (‘ ’) represent string type variable.

let name = “Rohan”;

3: BigInt : The BigInt type is a numeric primitive in JavaScript that can represent integers with arbitrary magnitude. With BigInts, you can safely store and operate on large integers even beyond the safe integer limit (Number.MAX\_SAFE\_INTEGER) for Numbers.

A BigInt is created by appending n to the end of an integer or by calling the BigInt() function.

*// BigInt*

const x = BigInt(Number.MAX\_SAFE\_INTEGER); *// 9007199254740991n*

x + 1n === x + 2n; *// false because 9007199254740992n and 9007199254740993n are unequal*

*// Number*

Number.MAX\_SAFE\_INTEGER + 1 === Number.MAX\_SAFE\_INTEGER + 2; *// true because both are 9007199254740992*

4: Boolean: The boolean type has only two values: true and false.

5: Undefined: The special value undefined also stands apart. It makes a type of its own, just like null. The meaning of undefined is “value is not assigned”. If a variable is declared, but not assigned, then its value is undefined:

6: Null: The special null value does not belong to any of the types described above.

7: Symbol: The JavaScript ES6 introduced a new primitive data type called Symbol. Symbols are immutable (cannot be changed) and are unique. For example,

// two symbols with the same description

const value1 = Symbol('hello');

const value2 = Symbol('hello');

console.log(value1 === value2); // false

**What are the non-primitive data types?**

Non-primitive data types: The data types that are derived from primitive data types of the JavaScript language are known as non-primitive data types. It is also known as derived data types or reference data types.

Below is a list of Non-primitive data types.

(1). Object (2). Array

1. Object: An object in JavaScript is an entity having properties and methods. Everything is an object in JavaScript.

How to create an object in JavaScript:

Using Constructor Function to define an object:

*// Create an empty generic object*

var obj = new Object();

*// Create a user defined object*

var mycar = new Car()

Using Literal notations to define an object:

*// An empty object*

var square = {};

*// Here a and b are keys and*

*// 20 and 30 are values*

var circle = {a: 20, b: 30};

**Array:** JavaScript Array is a single variable that is used to store elements of different data types. JavaScript arrays are zero-indexed. The JavaScript Arrays are not associative in nature.

Ways to declare a single-dimensional array:

Using an array literal is the easiest way to create a JavaScript Array.

Syntax:

*const items = [item1, item2, ...];*

Using new keyword :

*// Call it with no arguments*

var a = new Array();

*// Call it with single numeric argument*

var b = new Array(10);

*// Explicitly specify two or*

*// more array elements*

var d = new Array(1, 2, 3, "Hello");

**2. What is difference between primitive and non-primitive data types?**

| **Primitive** | **Non-Primitive** |
| --- | --- |
| Primitive Data types are predefined. | Non-Primitive data types are created by the programmer |
| Primitive Data types will have certain values. | Non-Primitive data types can be NULL. |
| Size depends on the type of data structure. | Size is not fixed |
| Examples are numbers and strings. | Examples are Array and Linked List. |
| It can start with a lowercase. | It can start with uppercase. |

**3. What is difference between null and undefined data types?**

Null is basically an assignment value given to a variable. The variable which has been assigned as null contains no value.

When we assign null as a value to any variable, it means that it is empty or blank. It is to show that the variable has no value. Also, null is an object in JavaScript. When it gets assigned to a variable, it represents no value. The setting of the value must be done manually by the user as JavaScript never sets the value as null. An object can be emptied by setting it to null.

**Syntax of Null and Undefined in JavaScript:**

var x = null;

**Undefined:**

As the name suggests, undefined means "not defined". So we declare a variable but do not assign a value to it, the variable becomes undefined.

Unlike null, the value of an undefined variable is set by JavaScript as undefined. The variable gets created at the run-time. When we do not pass an argument for a function parameter, the default value is taken as undefined. Besides, when a function doesn't return a value, it returns undefined.

var x;

console.log(x) // undefined

**What is type Coercion?**

Type Coercion refers to the process of automatic or implicit conversion of values from one data type to another. This includes conversion from Number to String, String to Number, Boolean to Number etc. when different types of operators are applied to the values.

In case the behaviour of the implicit conversion is not sure, the constructors of a data type can be used to convert any value to that datatype, like the Number(), String() or Boolean() constructor.

**There are two types of type coercion:**

Implicit and Explicit.

**Implicit** type coercion happens automatically when JavaScript converts one value to another without the programmer explicitly instructing it to do so.

**Explicit** type coercion, on the other hand, occurs when the programmer intentionally converts a value from one data type to another using functions like parseInt() or Number().

**What is a NaN property in JS?**

In JavaScript, NaN stands for Not a Number. It represents a value that is not a valid number. It can be used to check whether a number entered is a valid number or not a number. To assign a variable to NaN value, we can use one of the two following ways.

Syntax:

let a = NaN

// OR

let a = Number.NaN

**What is Hoisting?**

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

1.Creation

2.Execution

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

**Variable Hoisting:**

Variable hoisting means the JavaScript engine moves the variable declarations to the top of the script. For example, the following example declares the counter variable and initialize its value to 1:

console.log(counter); *//  undefined*

var counter = 1;

**Function Hoisting:**

The JavaScript engine also hoists the function declarations. This means that the JavaScript engine also moves the function declarations to the top of the script. For example:

let x = 20, y = 10;

let result = add(x, y);

console.log(result); *//  30*

function add(a, b) {

  return a + b;

}

**What is Temporal Dead Zone?**

A temporal dead zone (TDZ) is the area of a block where a variable is inaccessible until the moment the computer completely initializes it with a value. A block is a pair of braces ({...}) used to group multiple statements. Initialization occurs when you assign an initial value to a variable.

Suppose you attempt to access a variable before its complete initialization. In such a case, JavaScript will throw a ReferenceError.

{

*// bestFood’s TDZ starts here (at the beginning of this block’s local scope)*

*// bestFood’s TDZ continues here*

*// bestFood’s TDZ continues here*

*// bestFood’s TDZ continues here*

    console.log(bestFood); *// returns ReferenceError because bestFood’s TDZ continues here*

*// bestFood’s TDZ continues here*

*// bestFood’s TDZ continues here*

    let bestFood = "Vegetable Fried Rice"; *// bestFood’s TDZ ends here*

*// bestFood’s TDZ does not exist here*

*// bestFood’s TDZ does not exist here*

*// bestFood’s TDZ does not exist here*

  }

In the snippet above, the block’s TDZ starts from the opening curly bracket ({) and ends once the computer initializes bestFood with the string value "Vegetable Fried Rice".

When you run the snippet, you will see that the console.log() statement will return a ReferenceError.

JavaScript will return a ReferenceError because we used the console.log() code to access

bestFood before its complete initialization. In other words, we invoked bestFood within the

temporal dead zone.

**What is difference between let, var and const?**

**Difference between var and let**

|  | **var** | **let** |
| --- | --- | --- |
| **1.** | The var is a keyword that is used to declare a variable | The let is also a keyword that is used to declare a variable. |
| **2.** | Syntax -:  **var name = value;** | Syntax -:  **let name = value;** |
| **3.** | The variables that are defined with **var**statement have function scope. | The variables that are defined with **let**statement have block scope. |
| **4.** | We can declare a variable again even if it has been defined previously in the same scope. | We cannot declare a variable more than once if we defined that previously in the same scope. |
| **5.** | Hoisting is allowed with **var**. | Hoisting is not allowed with **let**. |
| **6.** | Example -:  **var animal\_object = “ Lion”;** | Example -:  **let x = 69;** |
| **7.** | var is an ECMAScript1 feature. | let is a feature of ES6. |
| **8.** | Its supported browsers are: Chrome, Internet Explorer, Microsoft Edge, Firefox, safari, opera | Its supported browsers are -: Chrome49, Microsoft Edge12, firefox44 , safari11, opera36 |

**Why do we use debugger word in JS?**

Using debugger keyword: The debugger keyword is used in the code to force stop the execution of the code at a breaking point and calls the debugging function. The debugger function is executed if any debugging is needed at all else no action is performed.

let a = 6;

let b = 9;

let c = a \* b;

*// stops the execution*

debugger;

console.log(c);

**What is function?**

A function is a reusable block of code that performs a specific task or set of tasks. Functions are used to organize code into logical units, making it more modular and easier to understand.

The syntax for creating a function in JavaScript is quite simple. Functions can take input in the form of parameters and can return a value or output.

Functions help you organize and structure your code. They also allow for code reuse and make it easier to understand and maintain large codebases

**Declare a function:**

To declare a function, you use the function keyword, followed by the function name, a list of parameters, and the function body as follows:

function functionName(parameters) {

*// function body*

*// ...*

}

**Calling a function:**

To use a function, you need to call it. Calling a function is also known as invoking a function. To call a function, you use its name followed by arguments enclosing in parentheses like this:

functionName(arguments);

**What is IIFE?**

IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined. The signature of it would be as below,

(function () {

*// logic here*

})();

The primary reason to use an IIFE is to obtain data privacy because any variables declared within the IIFE cannot be accessed by the outside world. i.e, If you try to access variables with IIFE then it throws an error as below,

(function () {

var message = "IIFE";

console.log(message);

})();

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

**15. What is HOF?**

Higher-order function is a function that accepts another function as an argument or returns a function as a return value or both.

const firstOrderFunc = () =>

  console.log("Hello, I am a First order function");

const higherOrder = (ReturnFirstOrderFunc) => ReturnFirstOrderFunc();

higherOrder(firstOrderFunc);

**16. Explain map, filter and reduce?**

Map :

* The map() method calls a callback function on every element of an array and returns a new array that contains the results.
* The map() method takes two named arguments, the first one is required whereas the second one is optional.

Syntax:

arrayObject.map(callback[,contextObject]);

Similar to the other iterative method such as every(), some(), filter(), forEach() and sort(), the callback() function has the following form:

The callback() function takes three arguments:

function callback(currentElement,index,array){

*// ...*

  }

The currentElement is the current element of the array that is being processed.

* The index is the index of the currentElement.
* The array is the array object being traversed.
* The currentElement is required while the index and array arguments are optional.

If you pass the contextObject to the map() method, you can reference the contextObject inside the callback() function using the this keyword.

**It’s important to note that the map() method does not change the original array, it creates a new array of all elements that have been transformed by the callback function.**

For example:

let numbers = [16, 25, 36];

let results = numbers.map(Math.sqrt);

console.log(results);

[4,5,6] // output

**Filter :**

The filter() method creates a new array with all the elements that pass the test implemented by the callback() function.

Syntax :

arrayObject.filter(callback, contextObject);

Internally, the filter() method iterates over each element of the array and passes each element to the callback function. If the callback function returns true, it includes the element in the return array.

The filter() method accepts two named arguments: a callback function and an optional object.

Like other iterative methods of the Array object such as every(), some(), map() and forEach(), the callback function has the following form:

function callback(currentElement, index, array){

*// ...*

 }

The callback function takes three arguments:

* The currentElement argument is the current element in the array that is being processed by the callback function.
* The index of the currentElement that is being processed by the callback function.
* The array object being traversed.
* The index and array arguments are optional.

The contexObject argument of the filter() method is optional. If you pass the this value, you can reference it by using this keyword inside the callback function.

It is important to note that the filter() method does not change the original array.

For example:

let arr = [20,30,50,35,10]

let geeaterThan = arr.filter(ele => ele>20)

console.log(geeaterThan)

[30,50,35]*//output*

**Reduce:**

The reduce() method is an iterative method. It runs a "reducer" callback function over all elements in the array, in ascending-index order, and accumulates them into a single value. Every time, the return value of callbackFunction is passed into callbackFunction again on next invocation as accumulator. The final value of accumulator (which is the value returned from callbackFunction on the final iteration of the array) becomes the return value of reduce().

**Or**

The reduce() method reduces an array of values down to just one value. To get the outputvalue, it runs a reducer function on each element of the array.

Syntax :

arr.reduce(callback[, initialValue])

callbackFunction is invoked only for array indexes which have assigned values. It is not invoked for empty slots in sparse arrays.

Unlike other iterative methods, reduce() does not accept a thisArg argument. callbackFunction is always called with undefined as this, which gets substituted with globalThis if callbackFunction is non-strict.

reduce() does not mutate the array on which it is called, but the function provided as callbackFunction can.

Note, however, that the length of the array is saved before the first invocation of callbackFunction.

**The callback argument is a function that will be called once for every item in the array. This function takes four arguments, but often only the first two are used.**

Syntax:

function callbackFn( accumulator, currentValue, currentIndex, array) {

*/\* Some code\*/*

}

**accumulator - the returned value of the previous iteration**

**currentValue - the current item in the array**

**index - the index of the current item**

**array - the original array on which reduce was called**

The initialValue argument is optional. If provided, it will be used as the initial accumulator value in the first call to the callback function.

For example:

const numbers = [1, 2, 3, 4];

const sum = numbers.reduce((accumulator, currentvalue) => {

  return accumulator + currentvalue;

}, 0);

console.log(sum); *// 10*

**17. Explain this keyword in JavaScript.**

**“this”** keyword refers to an object that is executing the current piece of code. It references the object that is executing the current function. If the function being referenced is a regular function, “this” references the global object.

The “this” keyword refers to different objects depending on how it is used:

* In an object method, this refers to the object.
* Alone, this refers to the global object.
* In a function, this refers to the global object.
* In a function, in strict mode, this is undefined.
* In an event, this refers to the element that received the event.
* Methods like call(),apply(),and bind() can refer this to any object.

**7. Explain pass by value and pass by reference in JavaScript.**

**8. What do you mean by strict mode in JavaScript?**

**Explain window keyword in JavaScript.**

The window object is global

The global object of JavaScript in the web browser is the window object. It means that all variables and functions declared globally with the var keyword become the properties and methods of the window object. For example:

var counter = 1;

var showCounter = () => console.log(counter);

console.log(window.counter);

window.showCounter();

Because the counter variable and the showCounter() function are declared globally with the var keyword, they are automatically added to the window object.

The window object provides methods for manipulating a window such as open(), resize(), resizeBy(), moveTo(), moveBy(), and close().

**19. Explain call, apply and bind in JavaScript.**

In JavaScript, call, apply, and bind are methods associated with functions, and they are used to manipulate the “this” value in a function and also to pass arguments to a function.

**call:**

The call method is used to invoke a function with a specified “this” value and a list of arguments as arguments. Then, it returns the value returned by the function, which is called using the call method.

function test(arg1, arg2){

  console.log(this.num, arg1, arg2); *// 100, 10, 20*

}

test.call({num: 100}, 10, 20);

In the code above, we have called the test function using the call method. The call method will invoke the test function with the “this” value as the object passed to the call method.

**apply:**

Similar to “call”, the “apply” method is used to invoke a function with a specified “this” value, but it takes an array-like object as the second argument for passing multiple arguments and it returns the value returned by the function, which is called using the apply method.

Function test(...arguments){

  console.log(this.num, arguments);*//100, [1,2,3]*

}

test.apply({num: 100}, [1,2,3]);

In the code above, we have called the test function using the apply method and with the “this” value as the object passed to the apply method.

“call()” and “apply()” are identical in functionality, the only difference is that “call()” accepts a list of arguments whereas, “apply()” accepts a single array of arguments.

**bind:**

The bind method is used to create a new function with a specified “this” value, and optionally, some initial arguments.

Unlike call and apply, bind does not immediately invoke the function; instead, it returns a new function that can be invoked later.

function test(arg){

  console.log(this.number, arg);

 }

 let bindedFn = test.bind({number: 99}, "argument");

 bindedFn(); *// 99, "argument"*

In the above code, we bind the “this” value for the test function and invoke the returned function from the bind method.

**Summary**

call: binds the “this” value, invokes the function, and allows you to pass a list of arguments.

apply: binds the “this” value, invokes the function, and allows you to pass arguments as an array.

bind: binds the “this” value, returns a new function, and allows you to pass in a list of arguments.

**21. What is currying in javascript?**

Currying is the process of taking a function with multiple arguments and turning it into a sequence of functions each with only a single argument. Currying is named after a mathematician Haskell Curry. By applying currying, a n-array function turns it into a unary function.

*// For example :*

const multiArgFunction = (a, b, c) => a + b + c;

console.log(multiArgFunction(1, 2, 3)); *// 6*

const curryUnaryFunction = (a) => (b) => (c) => a + b + c;

curryUnaryFunction(1); *// returns a function: b => c =>  1 + b + c*

curryUnaryFunction(1)(2); *// returns a function: c => 3 + c*

curryUnaryFunction(1)(2)(3); *// returns the number 6*

**22.** Explain scope and scope chaining in JavaScript.

The scope is the space or environment in the code where a particular variable has been declared. In contrast, the scope of a variable is the region in the code where a particular variable can be accessed.

There are three types of scopes —

Global Scope

Function Scope

Block Scope

**Global Scope**

The scope of variables declared outside any function or block is Global i.e. they can be accessed everywhere in the code.

*// Global Scoped Variable*

var global\_variable = "GeeksforGeeks";

*// First function...*

function first\_function() {

  return global\_variable;

}

*// Second function...*

function second\_function() {

  return first\_function();

}

console.log(second\_function());

**Function Scope**

All the variables declared within a function have a function scope i.e. they cannot be accessed anywhere outside the function. All variables defined in a function whether they are declared using var, const or let have function scope.

function main\_function() {

var a = 2; *// Variable with Local Scope...*

*// Nested Function having Function Scope*

var multiply = function () {

    console.log(a \* 5); *// It can be accessed and altered as well*

  }

  multiply(); *// Will be called out when main\_function gets called*

}

console.log(main\_function()); *// Display's the result...*

*// Throws a reference error since it is a locally scoped variable*

console.log(a);

*// Throws a reference error since it is a locally scoped function*

multiplyBy2();

In the above code snippet, the variable a and nested function multiply have function scope i.e. they can only be accessed within the function.

**Block Scope**

The variables if declared using let or const within curly braces { } (apart from function) have block scope, i.e. these variables cannot be accessed outside the block. The variables declared with var within curly braces cannot be block scoped, i.e. they can be accessed outside the block too.

{

  let x = 13;

}

*// Throws a reference error*

*// since x is declared inside a block which*

*// cannot be used outside the block*

console.log(x);

**Scope Chain**

In JavaScript, every scope has access to the variables and functions of all the ancestral/ parent scopes. During execution, if the JS engine come across a variable, to access the variable and its value it first searches in the local scope, if it doesn’t find the variable in the local scope, it searches for the variable in its parent’s scope.

The search goes on from one scope to its parent scope to its parent's parent scope until it finds the variable. This whole concept of locating a variable from the inner scope to the outer scopes is known as the scope chain. The last and outermost scope is the global scope, which is the ultimate parent scope.

If a variable is not found in this whole scope chain, the JS engine throws a reference error and declares the variable not defined.

var global\_variable = 20;

function main\_function() {

*// Local Variable*

  var local\_variable = 30;

  var nested\_function = function () {

*// Display the value inside the local variable*

    console.log(local\_variable);

  }

  var another\_nested\_function = function () {

*// Displays the value inside the global variable*

    console.log(global\_variable);

  }

  nested\_function();

  another\_nested\_function();

}

main\_function();

**23. Explain closure in JavaScript.**

A closure is the combination of a function and the lexical environment within which that function was declared. i.e, It is an inner function that has access to the outer or enclosing function’s variables. The closure has three scope chains.

* Own scope where variables defined between its curly brackets
* Outer function’s variables
* Global variables

function Welcome(name) {

    var greetingInfo = function (message) {

      console.log(message + " " + name);

    };

    return greetingInfo;

  }

  var myFunction = Welcome("John");

  myFunction("Welcome "); *//Output: Welcome John*

  myFunction("Hello Mr."); *//output: Hello Mr.John*

As per the above code, the inner function (i.e, greetingInfo) has access to the variables in the outer function scope (i.e, Welcome) even after the outer function has returned.

**Explain Debouncing in JavaScript.**

Debouncing is a programming pattern or a technique to restrict the calling of a time-consuming function frequently, by delaying the execution of the function until a specified time to avoid unnecessary CPU cycles, and API calls and improve performance.

For example:

function debounce(func, timeout = 300){

    let timer;

    return (...args) => {

      clearTimeout(timer);

      timer = setTimeout(() => { func.apply(this, args); }, timeout);

    };

  }

  function saveInput(){

    console.log('Saving data');

  }

  const processChange = debounce(() => saveInput());

**It can be used on an input:**

<input *type*="text" *onkeyup*="processChange()" />

**Or a button:**

<button *onclick*="processChange()">Click me</button>

**Or a window event:**

window.addEventListener("scroll", processChange);

**What is callback function in JavaScript?**

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.

function mainFunction(callback) {

console.log("Performing operation...");

*// Use setTimeout to simulate an asynchronous operation*

  setTimeout(function() {

    callback("Operation complete");

  }, 1000);

}

*// Define the callback function*

function callbackFunction(result) {

  console.log("Result: " + result);

}

*// Call the main function with the callback function*

mainFunction(callbackFunction);

JavaScript functions are executed in the sequence they are called. Not in the sequence they are defined.

**Explain the concept of Memoization in JavaScript.**

Memoization is a programming technique which attempts to increase a function’s performance by caching its previously computed results. Each time a memoized function is called, its parameters are used to index the cache. If the data is present, then it can be returned, without executing the entire function. Otherwise the function is executed and then the result is added to the cache.

JavaScript Memoization mainly depends on two concepts:

1. Closure
2. High-order function

*// Recursive Technique*

function fib(n) {

  if (n < 2){

      return n;

  }else{

      return fibo(n-2) + fibo(n-1);

  }

}

*// Recursive Memoization Technique*

const fib = (n, memo) => {

  memo = memo || {}

  if (memo[n]) return memo[n]

  if (n <= 1) return n

  return memo[n] = fib(n-1, memo) + fib(n-2, memo)

}

*//*

**What is DOM?**

DOM stands for Document Object Model. It is a programming interface that allows us to create, change, or remove elements from the document. We can also add events to these elements to make our page more dynamic.

The DOM views an HTML document as a tree of nodes. A node represents an HTML element.

<!DOCTYPE html>

<html *lang*="en">

  <head>

    <meta *charset*="UTF-8">

    <meta *name*="viewport" *content*="width=device-width, initial-scale=1.0">

    <meta *http-equiv*="X-UA-Compatible" *content*="ie=edge">

    <title>DOM tree structure</title>

  </head>

  <body>

    <h1>DOM tree structure</h1>

  <h2>Learn about the DOM</h2>

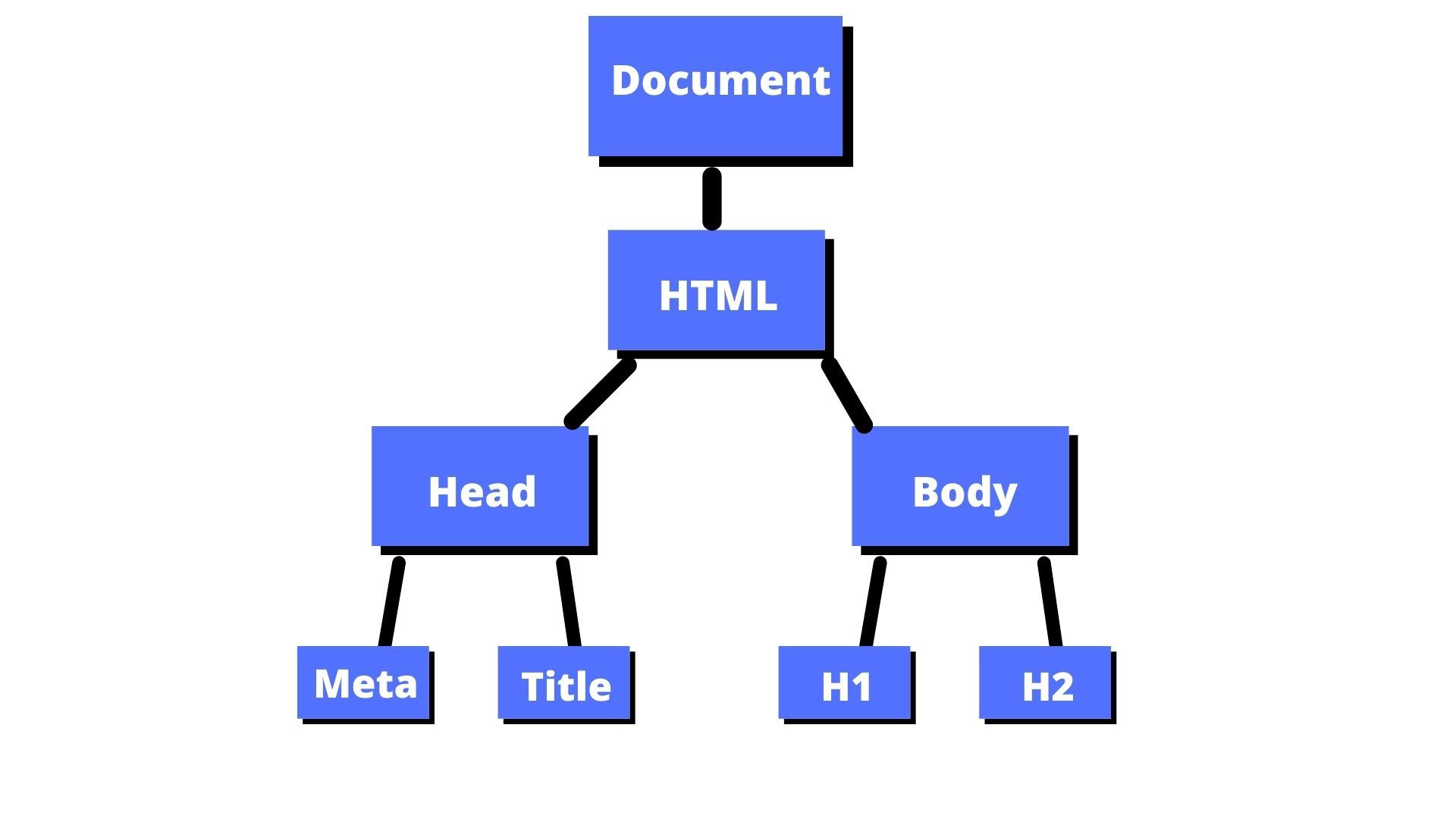
  </body>

</html>

Our document is called the root node and contains one child node which is the <html> element. The <html> element contains two children which are the <head> and <body> elements.

Both the <head> and <body> elements have children of their own.

Here is another way to visualize this tree of nodes.



There are a few different methods for selecting an element in the HTML document.

getElementById()

querySelector()

querySelectorAll()

**27. What is difference between DOM and BOM?**

**Difference between DOM & BOM:**

| Document Object Model (DOM) | Browser Object Model (BOM) |
| --- | --- |
| It mainly focuses on the structure of the displayed document. | It mainly focuses on browser-specific functionality. |
| It facilitates a standardized interface to access and modify the elements and content of an HTML or XML document. | It allows JavaScript to interact with browser features beyond the scope of manipulating the document structure. |
| When an HTML document gets loaded in the browser, then it becomes a document object. | In this case, the window object will be created automatically by the browser. |
| It facilitates access and manipulation, along with dynamically updating the structure, content, and styling of the web document. | It facilitates the different functionality for governing the browser window, handling the navigation, managing history, and accessing browser-related information. |
| It provides direct access control to the content of the web document, along with permitting the traversal and modification of its elements and attributes. | It doesn’t have any access to the content of the web document directly. |

**28. What is difference between Client side and serve side JavaScript?**

**29. What is an Arrow function? Explain the difference between normal function and arrow function.**

An arrow function expression is a compact alternative to a traditional function expression, with some semantic differences and deliberate limitations in usage:

* Arrow functions don't have their own bindings to this, arguments, or super, and should not be used as methods.
* Arrow functions cannot be used as constructors. Calling them with new throws a TypeError. They also don't have access to the new.target keyword.
* Arrow functions cannot use yield within their body and cannot be created as generator functions.

Arrow Function ( => ) Syntax:

1. **() => expression**
2. **param => expression**
3. **(param) => expression**
4. **(param1, paramN) => expression**
5. **() => {**

**Statements**

**}**

1. **param => {**

**Statements**

**}**

1. **(param1, paramN) => {**

**statements**

**}**

**30. What is difference between rest and spread operators?**

**31. What is promise in JavaScript?**

A promise is an object that may produce a single value sometime in the future with either a resolved value or a reason that it’s not resolved (for example, network error). It will be in one of the 3 possible states: fulfilled, rejected, or pending.

Syntax:

const promise = new Promise(function (resolve, reject) {

*// promise description*

    });

For example:

const promise = new Promise(

    (resolve) => {

      setTimeout(() => {

        resolve("I'm a Promise!");

      }, 5000);

    },

    (reject) => {}

  );

**32. What is call stack?**

**33. What is difference between local storage and session storage?**

**34. Explain the working of setTimeOut and setInterval.**

**35. What is asynchronous JavaScript?.**

**36. Explain the execution of a javascript code.**

**37. Explain destructuring.**

**38. Explain prototype in javascript.**

**39. What is OOJS?**

**40: What is ES6 and what were the new improvements in it?**

**41. What is babel? What is the need of it in javascript?**

**42. Explain the class keyword on ES6.**

**43. What is class constructor?**

**44. What is difference between object constructor and function constructor?**

**45. What are the features of JavaScript?**

**46. What are the different ways to create an object?**

**47. What are the conventions of naming a variable in javascript?**

**48. What are imports and exports in javascript?**

**49. What is difference between document and window in javascript?**

**50. What do you mean by statically typed and dynamically typed language?**

**51. What is difference between exec() and test() methods?**

**52. What are the advantages of using the external javascript?**

**53. What are the types of errors in javascript?**

**54. What are generator functions?**

**55. What is a weakSet and weakMap?**

**56. What is difference between prototypal and classical inheritance?**

**57. What is difference between event capturing and event bubbling?**

**58. What is pure and impure function?**

**59. What is difference between nodelist and html collection?**

**Introduction to JavaScript:**

JavaScript is a light-weight object-oriented programming language which is used by several websites for scripting the webpages. It is an interpreted, full-fledged programming language that enables dynamic interactivity on websites when applied to an HTML document.

**Features of JavaScript**

There are following features of JavaScript:

1. All popular web browsers support JavaScript as they provide built-in execution environments.
2. JavaScript follows the syntax and structure of the C programming language. Thus, it is a structured programming language.
3. JavaScript is a weakly typed language, where certain types are implicitly cast (depending on the operation).
4. JavaScript is an object-oriented programming language that uses prototypes rather than using classes for inheritance.
5. It is a light-weighted and interpreted language.
6. It is a case-sensitive language.
7. JavaScript is supportable in several operating systems including, Windows, macOS, etc.
8. It provides good control to the users over the web browsers.

**Variables & Data Types**

**Variable :** Variables are containers for storing data (storing data values).

e.g: let message;

In JavaScript variable names are declared with var, let and const keword.

e.g : var animal = “Dog”;

let college = “Delhi University”;

const piValue = 22/7;

Rules for Variable name declaration:

1.The name must contain only letters, digits, or the symbols $ and \_.

2.The first character must not be a digit.

3. There is a list of reserved words, which cannot be used as variable names because they are used by the language itself.

e.g : let, class, return, and function are reserved.

# Difference between var, let and const.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **var** | **let** | **const** |
| **Scope** | Variables declared with var are in the function scope.  function f1()  {      var a=10;  }  console.log(a) | Variables declared as let are in the block scope.  for (let i = 0; i < 3; i++) {    console.log(i);  }  console.log(i); | Variables declared as const are in the block scope.  {    const x = 2;    console.log(x);  }  console.log(x); |
| **Reassign** | Variable with “var” can be reassign | Variable with “let” can be reassign | Variable with “const” can’t be reassigned |
| **Hoisting** | Variables declared using var are hoisted to the top of their scope and initialized with a value of undefined (special type). | Variables declared using let are hoisted to the top of their scope but are not initialized with any value. | Variables declared using const are hoisted to the top of their scope but are not initialized with any value. |
| **Redeclaration** | The redeclaration of variable with var keyword are allowed , that means we can declare the variable again.  var v1 = 1;  var v1 = 30;  console.log(v1); | The redeclaration of variable with let keyword are not allowed.  let v1 = 1;  let v1 = 30;  console.log(v1); | The redeclaration of variable with let keyword are not allowed.  const v1 = 1;  const v1 = 30;  console.log(v1) |

What is Class in JavaScript ?

Classes are a template or blueprint for creating objects. They encapsulate data and function with code to work on that data. Classes in JS are built on prototypes but also have some syntax and semantics that are unique to classes.

Classes are in fact "special functions", and just as you can define function expressions and function declarations, a class can be defined in two ways: a class expression or a class declaration.

// Declaration

class Rectangle {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

// Expression; the class is anonymous but assigned to a variable

const Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

// Expression; the class has its own name

const Rectangle = class Rectangle2 {

constructor(height, width) {

this.height = height;

this.width = width;

}

};

**Constructor:**

The constructor() method is a special method for creating and initializing objects created within a class.

The constructor() method is called automatically when a class is initiated, and it has to have the exact name "constructor", in fact, if you do not have a constructor method, JavaScript will add an invisible and empty constructor method.

Note: A class cannot have more than one constructor() method. This will throw a SyntaxError.

class Car {

constructor(brand) {

this.carname = brand;

}

present() {

return 'I have a ' + this.carname;

}

}

class Model extends Car {

constructor(brand, mod) {

super(brand);

this.model = mod;

}

show() {

return this.present() + ', it is a ' + this.model;

}

}

mycar = new Model("Ford", "Mustang");

console.log(mycar.show());