JAVASCRIPT QUESTIONS GITHUB

<https://github.com/ganqqwerty/123-Essential-JavaScript-Interview-Questions>

MOST IMP JAVASCRIPT INTERVIEW QUESTIONS

<https://javascriptcentric.medium.com/top-30-javascript-interview-questions-and-answers-for-2024-7f1e2d1d0638>

Programming languages are commonly classified into several categories based on their features, usage, and level of abstraction. Here are some of the primary classifications:

1. **Low-level vs. High-level Languages**:
   * **Low-level Languages**: Close to machine code, offering little abstraction from the hardware (e.g., Assembly language).
   * **High-level Languages**: Provide greater abstraction, making them easier to read and write (e.g., Python, Java).
2. **Compiled vs. Interpreted Languages**:
   * **Compiled Languages**: Converted into machine code by a compiler before execution (e.g., C, C++).
   * **Interpreted Languages**: Executed line-by-line by an interpreter at runtime (e.g., Python, JavaScript).
3. **Procedural vs. Object-Oriented vs. Functional Languages**:
   * **Procedural Languages**: Based on procedure calls, focusing on a sequence of commands (e.g., C, Pascal).
   * **Object-Oriented Languages**: Based on object and classes, encapsulating data and behavior (Java C++).
   * **Functional Languages**: Emphasize functions and immutability, avoiding state and mutable data (e.g., Haskell, Lisp).
4. **Scripting Languages**:

A scripting language is a type of programming language designed for integrating and communicating with other programming languages. It is typically used to automate tasks, control applications, and manipulate data within a runtime environment. Scripting languages are often interpreted rather than compiled, which means that their code is executed directly by an interpreter rather than being converted into machine code.

1. **Domain-Specific Languages (DSLs)**:
   * Designed for specific tasks or industries (e.g., SQL for databases, HTML for web development).
2. **Markup Languages**:
   * Used for annotating text, often in web development (e.g., HTML, XML).

**What is Javascript?**

JavaScript is a lightweight, interpreted scripting language primarily used for creating dynamic and interactive web pages. It enables developers to add functionality and responsiveness to websites, enhancing the user experience.

**A static website** consists of web pages with fixed content. Each page is coded in HTML and displays the same information to every visitor. The content does not change unless a developer manually updates the code. Characteristics of static websites include:

**A dynamic website** displays different content and can be interactive, often in response to user input or other variables. These websites are generated on the fly, often using server-side scripting languages like PHP, Python, or JavaScript (Node.js). Characteristics of dynamic websites include:

**Client-side execution** refers to code that is run on the user's device, typically in their web browser. This includes technologies like HTML, CSS, and JavaScript, which are used to build and render the user interface, handle user interactions, and perform tasks such as form validation or making asynchronous requests to the server (using AJAX, for example). Client-side code is visible to the user and can be easily viewed and manipulated.

**Server-side execution** involves code that runs on a server, handling tasks such as data processing, business logic, and database interactions. Technologies like Node.js, PHP, Python, Ruby, Java, and others are commonly used for server-side development. Server-side code is not directly visible to the user, as it typically generates responses (like HTML pages or JSON data) that are sent to the client. Server-side execution is crucial for tasks that require secure data handling, authentication, and dynamic content generation.

**What are features of Javascript?**

**Scripting Language**

JavaScript is a lightweight scripting language made for client-side execution on the browser. Since it is not designed as a general-purpose language and is specially engineered for web applications, the set of libraries is also geared primarily towards web applications.

**Interpreter Based**

JavaScript is an interpreted language instead of a compiled one. In that sense, it is closer to languages like Ruby and Python. The browser interprets JavaScript’s source code, line by line and runs it. In contrast, a compiled language needs to be compiled into a byte-code code executable. Java and C++ are examples of compiled languages.

**Event Handling**

An event is an action or an occurrence in a system that communicates about said occurrence so that you can respond to it somehow. For example, a user clicks on a button, and the system tells you to respond to the button click event with an action, say an information box.

JavaScript enables you to handle events and even generate custom events.

**Light Weight**

JavaScript isn’t a compiled language, so it doesn’t get converted to byte-code beforehand. However, it does follow a paradigm called Just-In-Time (JIT) Compilation. Meaning it gets converted to bytecode just as it’s about to run. This enables JS to be lightweight. Even less powerful devices are capable of running JavaScript.

**Input validation**

Javascript allows you to validate user input before sending it to server for backend operations.

**Control over browser**

Javascript gives more control over browser due to which you can change the background colour of this page as well as the text on the browser's status bar.

#### Async Processing

JavaScript supports asynchronous behaviour through the use of Promises and Async functions. In Promises, we can make a request and attach a .then() clause to it, which only executes on the completion of the promise. The other alternative is to use the async-await syntax functions; async functions don’t execute sequentially but parallelly, which positively affects pages’ processing time and responsiveness.

### **Prototype-based**

JavaScript is a prototype-based scripting language, which means it uses the prototypal inheritance model instead of the commonly known class inheritance. This means instead of creating classes and deriving objects from them, we define the Objects prototype and use this object prototype to create more Objects of the same type. This behaviour is similar to the Object factory method, and thus such a design pattern is commonly seen in JavaScript code.

**What are advantages of Javascript?**

**Faster speed :** JavaScript is fast because it run immediately within the client- side browser. Javascript is not dependant on network unless backend data is required to be processed. Need to compile Javascript on the client–side as it is interpreted directly by web browsers.

### **Interoperability :** JavaScript seamlessly integrates with other programming languages, so many developers favour using it to create various apps. Any webpag or script of another computer language can incorporate it.

**Rich interfaces :** Javascript has vast libraries like (charts, drag and drop, sliders etc.) which enables you to provide attractive look to your website.

**Reduction in server load :** Since Javascript is client-side scripting language it reduces load on website servers as many operations can be performed at client-side which reduces load on server and enables it serve to more users. What are advantages of Javascript?

**Browser Compatibility:** JavaScript supports all modern browsers, allowing developers to create cross-browser compatible applications.

**What are disadvantages of Javascript?**

**Client-side security :** JavaScript code executes in users computer hence in some cases it can be manipulated for malicious purpose.

**Browser support :** JavaScript is sometimes interpreted differently by different browsers.

**Slow Rendering:** JavaScript DOM (Document Object Model) is slow and can be a never-fast rendering with HTML. If an error occurs in JavaScript, it can stop rendering the whole website.

**Security:** JavaScript is a client-side language, which means that it can be vulnerable to security threats. Malicious code can be injected into the JavaScript code, allowing hackers to access sensitive information.

**Error Handling:** JavaScript is not as strict as other programming languages, which can lead to bad habits and errors. This can make it difficult to debug and maintain large JavaScript applications.

**What is meant by server and client side web applications?**

Server-side and client-side web applications refer to where the application logic is executed—either on the server or the client's browser. Both approaches have distinct characteristics, benefits, and use cases.

### **Server-Side Web Applications**

**Server-side** web applications are those where the majority of the logic, processing, and data handling occurs on the server. The server generates the HTML, CSS, and JavaScript and sends it to the client's browser to be rendered.

### **Client-Side Web Applications**

**Client-side** web applications execute the majority of the logic on the client's browser. The server mainly serves static assets like HTML, CSS, and JavaScript files. The browser then executes the JavaScript to render the page and handle user interactions dynamically.

**What is difference between Javascript and ECMAScript?**

JavaScript is a scripting language that has been formed by keeping ECMAScript specification at its core.

ECMAScript is nothing but a standard or specification defined in order to create different scripting languages and one of them is JavaScript.

Javascript, Jscript and ActionScript are few scripting languages that follow ECMAScript specifications.

**Who developed Javascript?**

JavaScript was created in 1995 by Brendan Eich during his time at Netscape Communications. It was inspired by Java, Scheme and Self.

**How to insert Javascript in Web page?**

You can use <script> tag in html. <script> tag has type attribute which defined which code is there inside the script tag.

You can use <script> element in web pages in following ways:

* In head element
* In body element
* As an external script file

To use Javascript as a scripting language for web pages in <head> or <body>

you can define type as “text/javascript”. e.g.

<script type=”text/javascript”>

</script>

Sometime you may need to use same Javascript code in several web pages, in such cases you can store Javascript code in external file and save file as

<filename>.js file.

Then this script can be made available to web page using src attribute of

<script> tag e.g.

<script src=”external file URL”>

</script>

**What are advantages of using external javascript?**

Placing JavaScript code in external js files has few advantages over inline scripts:

* Segregating HTML and JavaScript code helps to manage the code base better.
* To improve development output designers can work along with coders in parallel without code conflicts.
* This approach also works well with modern source code version control systems like GIT and SVN.
* Each of these files can maintain history.
* Segregating HTML and JavaScript makes code as well as HTML is easily readable.
* Segregated external JavaScript files are cached by browsers and can speed up page load times. These small js files can be minified to reduce the size and make it not readable by humans, using Google closure or YUI Compressor or other.
* Many popular JavaScript libraries are available as hosted on content delivery networks (cdn) and you can simply point to them using the URL in the src, this avoids copying the js file to local folder.
* Using external Js you can take benifits of advanced tools such as RequireJS or CommonJS to load these scripts logically and modularly.

**Data types in javascript**

**JavaScript has 8 Datatypes**

String

Number

Bigint

Boolean

Undefined

Null

Symbol

Object

**JavaScript data types are divided into primitive and non-primitive types.**

**Primitive Data Types:** They can hold a single simple value. String, Number, BigInt, Boolean, undefined, null, and Symbol are primitive data types

**Non-Primitive Data Types:** They can hold multiple values. Objects are non-primitive data types.

The object data type can contain both built-in objects, and user defined objects:

Built-in object types can be:

objects, arrays, dates, maps, sets, intarrays, floatarrays, promises, and more.

### **Primitive Types**

**Number**:

Represents both integer and floating-point numbers.

Example: 42, 3.14

**String**:

Represents a sequence of characters.

Example: "Hello, world!"

**Boolean**:

Represents a logical entity and can have two values: true or false.

Example: true, false

**Null**:

Represents the intentional absence of any object value.

Example: null

**Undefined**:

Represents a variable that has been declared but not assigned a value.

Example: undefined

**Symbol**:

Symbols are often used as keys for object properties. This is useful in scenarios where property names need to be unique or when integrating with other code, avoiding the risk of property name collisions.

**// Creating symbols**

**const symbol1 = Symbol();**

**const symbol2 = Symbol('description');**

**const symbol3 = Symbol('description');**

**// Symbols are unique**

**console.log(symbol1 === symbol2); // false**

**console.log(symbol2 === symbol3); // false, even with the same description**

**// Using symbols as object keys**

**const obj = {**

**[symbol1]: 'value1',**

**[symbol2]: 'value2'**

**};**

**console.log(obj[symbol1]); // 'value1'**

**console.log(obj[symbol2]); // 'value2'**

**// Symbols are not enumerable in for...in loops**

**for (let key in obj) {**

**console.log(key); // Nothing will be logged**

**}**

**// Symbols are ignored by JSON.stringify**

**console.log(JSON.stringify(obj)); // '{}'**

**// Accessing symbol properties using Object.getOwnPropertySymbols**

**const symbols = Object.getOwnPropertySymbols(obj);**

**console.log(symbols); // [ Symbol(), Symbol(description) ]**

**console.log(obj[symbols[0]]); // 'value1'**

**BigInt**:

Represents integers with arbitrary precision.

Example: 1234567890123456789012345678901234567890n

### Objects

**Object**:

Represents a collection of key-value pairs.

Example: { name: "John", age: 30 }

**Array**:

Represents an ordered list of values.

Example: [1, 2, 3, 4]

**Function**:

Represents a block of code designed to perform a particular task.

Example:

function greet() {

console.log("Hello!");

}

**Date**:

Represents a single moment in time.

Example: new Date()

**RegExp**:

Represents a regular expression, used for pattern matching within strings.

Example: /ab+c/

### **Special Objects**

**Map**:

Represents a collection of key-value pairs where keys can be of any type.

Example:

let map = new Map();

map.set('key', 'value');

**Set**:

Represents a collection of unique values.

Example:

let set = new Set([1, 2, 3, 4]);

**WeakMap**:

Similar to Map but holds weak references to keys. Unlike a regular Map, where keys can be of any type, WeakMap only allows objects as keys. The key-value pairs in a WeakMap are weakly referenced. This means that if there are no other references to a key object, the key and its associated value can be garbage collected, freeing up memory.

Example:

let weakMap = new WeakMap();

let obj = {};

weakMap.set(obj, "Some value");

console.log(weakMap.get(obj)); // Outputs: "Some value"

obj = null; // The object is now eligible for garbage collection

// The entry in the WeakMap will be removed automatically

**WeakSet**:

WeakSet is similar to Set, but it only allows objects as values. Objects added to a WeakSet are weakly referenced. If no other references to an object exist, it can be garbage collected, removing it from the WeakSet.

Example:

let weakSet = new WeakSet();

let obj = {};

weakSet.add(obj);

console.log(weakSet.has(obj)); // Outputs: true

obj = null; // The object is now eligible for garbage collection

// The entry in the WeakSet will be removed automatically

**Difference between primitive and non-primitive data types.**

**Primitive Data Types**

**Definition**: Primitive data types are basic data types provided by JavaScript, which stores only one value.

**Storage:** Primitive values are stored directly in the variable’s memory location. Each time you assign a primitive value to a variable, a new space in memory is allocated for that value. This direct storage means that primitive values are compared by their actual content.

**Behavior**: Primitive values are compared by value. For example, two strings with the same content are considered equal.

**Non-Primitive Data Types**

**Definition**: Non-primitive data types are objects that can hold multiple values.

**Storage**: Non-primitive values are stored by reference. The variable stores a reference (or pointer) to the memory location where the object is stored, not the object itself. When you assign a non-primitive value to a variable, you're actually assigning the reference to the memory location. This reference-based storage means that non-primitive values are compared by their reference, not their content.

**Behavior**: Non-primitive values are compared by reference. Even if two objects have the same content, they are considered different if they are different instances.

**Explain Pass by reference and Pass by value in javascript.**

## **Pass By Value**

When a function is called, the value of the variable is directly passed as an argument. Therefore, any modifications made inside the function do not impact the original value.

The parameters passed as arguments generate their own copies. Consequently, any alterations made inside the function apply to the copied value, not the original v alue.

**function** Passbyvalue(a, b) {

**let** tmp;

tmp = b;

b = a;

a = tmp;

console.log(`Inside Pass by value

function -> a = **${**a**}** b = **${**b**}**`);

}

**let** a = 1;

**let** b = 2;

console.log(`Before calling Pass by value

Function -> a = **${**a**}** b = **${**b**}**`);

Passbyvalue(a, b);

console.log(`After calling Pass by value

Function -> a =**${**a**}** b = **${**b**}**`);

**Output**

Before calling Pass by value

Function -> a = 1 b = 2

Inside Pass by value

function -> a = 2 b = 1

After calling Pass by value

Function -> a =1 b = 2

## **Pass by Reference**

* In Pass by Reference, Function is called by directly passing the reference/address of the variable as an argument. So changing the value inside the function also change the original value. In JavaScript **array and Object**follows pass by reference property.
* In Pass by reference, parameters passed as an arguments does not create its own copy, it refers to the original value so changes made inside function affect the original value.

**function** PassbyReference(obj) {

**let** tmp = obj.a;

obj.a = obj.b;

obj.b = tmp;

console.log(`Inside Pass By Reference

Function -> a = **${**obj.a**}** b = **${**obj.b**}**`);

}

**let** obj = {

a: 10,

b: 20

}

console.log(`Before calling Pass By Reference

Function -> a = **${**obj.a**}** b = **${**obj.b**}**`);

PassbyReference(obj)

console.log(`After calling Pass By Reference

Function -> a = **${**obj.a**}** b = **${**obj.b**}**`);

**Output**

Before calling Pass By Reference

Function -> a = 10 b = 20

Inside Pass By Reference

Function -> a = 20 b = 10

After calling Pass By Reference

Function -> a = 20 b = 10

**Note:** In Pass by Reference, we are mutating the original value. when we pass an object as an arguments and update that object’s reference in the function’s context, that won’t affect the object value. But if we mutate the object internally, It will affect the object .

**Example 1:** Updating the object reference in the function.

**function** PassbyReference(obj) {

*// Changing the reference of the object*

obj = {

a: 10,

b: 20,

c: "GEEKSFORGEEKS"

}

console.log(`Inside Pass by

Reference Function -> obj `);

console.log(obj);

}

**let** obj = {

a: 10,

b: 20

}

console.log(`Updating the object reference -> `)

console.log(`Before calling Pass By

Reference Function -> obj`);

console.log(obj);

PassbyReference(obj)

console.log(`After calling Pass By

Reference Function -> obj`);

console.log(obj);

**Output**

Updating the object reference ->

Before calling Pass By

Reference Function -> obj

{ a: 10, b: 20 }

Inside Pass by

Reference Function -> obj

{ a: 10, b: 20, c: 'GEEKSFORGEEKS' }

After calling Pass By

        Reference Function -> obj

{ a: 10, b: 20 }

**Example 2:** Mutating the original Object.

**function** PassbyReference(obj) {

*// Mutating the original object*

obj.c = "GEEKSFORGEEKS";

console.log(`Inside Pass by

Reference Function -> obj `);

console.log(obj);

}

**let** obj = {

a: 10,

b: 20

}

console.log(`Mutating the original object -> `)

console.log(`Before calling Pass By

Reference Function -> obj`);

console.log(obj);

PassbyReference(obj)

console.log(`After calling Pass By

Reference Function -> obj`);

console.log(obj);

**Output**

Mutating the original object ->

Before calling Pass By

Reference Function -> obj

{ a: 10, b: 20 }

Inside Pass by

Reference Function -> obj

{ a: 10, b: 20, c: 'GEEKSFORGEEKS' }

After calling Pass By

        Reference Function -> obj

{ a: 10, b: 20, c: 'GEEKSFORGEEKS' }

**What is Javascript engine?**

JavaScript engine is a computer program used to execute Javascript code.

JS engines were developed by web browser vendors and every major browser has one.

Chrome V8 from google is most used engine, Google chrome use it. SpiderMonkey is developed by Mozilla for use in firefox. JavaScriptCore is Apples engine for its Safari browser.

**Explain the difference between var, let, and const keywords.**

**var:**

You can declare it again and change its value.

JavaScript makes it available from the start of the function but sets it to undefined if not initialized.

**let:**

You can't redeclare it in the same scope, but you can update it.

JavaScript makes it available only after its declaration.

**const:**

You can't declare it again or change its value.

You must give it a value when you declare it.

JavaScript makes it available only after its declaration.

**What is automatic type conversion?**

Automatic type conversion, also known as type coercion, is a feature in programming languages like JavaScript where the interpreter automatically converts values from one data type to another when performing operations involving different types. This process happens implicitly, without explicit instructions from the programmer.

**What is break statement?**

Break statement stops execution of loop entirely.

**What is continue statement?**

Continue statement stops execution of current iteration in a loop and continues with next iteration of loop.

**What is the difference between comparing variables using "==" and "===" operator?**

The ‘==’ operator tests for abstract equality i.e. it does the required type conversions before doing the equality comparison.

But the ‘===’ operator tests for strict equality i.e. it will not do the type conversion thus if the two values are not of the same type, when compared, it will return false.

**What is typeof operator?**

The typeof operator is used to get the data type of its operand. The operand can be either a literal or a data structure such as variable, function or an object.

e.g.

console.log(typeof somevar);

The typeof operator returns below values as string: object, Boolean, function, number, string and undefined.

**14. Explain the concept of hoisting in JavaScript.**

Hoisting in JavaScript is the **default behavior** where variable and function **declarations are moved to the top of their containing scope** during the **compilation phase, before the actual code execution**. This means that you can use a variable or call a function before it’s declared in your code.

When you declare a variable using var, the declaration is hoisted to the top of its containing function or block and **initialized with the default value of “undefined”.**

console.log(x); // Outputs: undefined  
var x = 5;

Variables declared with let and const are hoisted as well but have a "temporal dead zone" where they cannot be accessed before their declaration.

console.log(x); // Throws an error (ReferenceError)  
let x = 5;

Function declarations are also hoisted to the top of their containing scope. You can call a function before it’s declared in your code.

sayHello(); // Outputs: "Hello, world!"  
function sayHello() {  
 console.log("Hello, world!");  
}

**Hoisting is not happening with an arrow function, function expression, or variable initialization.**

**What is difference between undefined and null?**

The undefined means a variable has been declared but has no value has yet been assigned.

On the other hand, null is basically a value which has been assigned. Also, undefined is a type itself (undefined) while null is an object.

Null represents intentional absence of value;

Unassigned variables are initialized with a default value of undefined by

JavaScript or undefined can be assigned to variable through code. Whereas JavaScript never sets a value to null.

That must be done programmatically.

**What is output of null == undefined?**

null == undefined will return true .

However, null === undefined will return false .

**What are functions in javsascript?**

Functions in JavaScript are reusable blocks of code designed to perform a particular task. They allow you to encapsulate code for modularity, reusability, and organization.

**Types of functions?**

### **1. Function Declarations**

A standard way to declare a function using the function keyword. These functions are hoisted, meaning they can be called before they are defined in the code.

function greet(name) {

return "Hello, " + name + "!";

}

### **2. Function Expressions**

Functions can be defined as part of an expression and assigned to variables. Unlike function declarations, these are not hoisted.

const greet = function(name) {

return "Hello, " + name + "!";

};

### **3. Arrow Functions**

A concise way to write functions using the => syntax, introduced in ES6. Arrow functions do not have their own this binding, arguments object, super, or new.target bindings.

const greet = (name) => {

return "Hello, " + name + "!";

};

// For single-line functions, you can omit the braces and return keyword.

const greet = name => "Hello, " + name + "!";

### **4. Anonymous Functions**

Functions without a name. They are often used as arguments to other functions or assigned to variables.

setTimeout(function() {

console.log("This is an anonymous function.");

}, 1000);

### **5. Immediately Invoked Function Expressions (IIFE)**

Functions that are defined and executed immediately. Often used to create a new scope and avoid polluting the global scope.

(function() {

console.log("This is an IIFE.");

})();

### **6. Higher-Order Functions**

Functions that take other functions as arguments or return functions as their result. They are fundamental to functional programming.

function higherOrder(fn) {

return function() {

return fn() + " from higher-order function";

};

}

function greet() {

return "Hello";

}

const newGreet = higherOrder(greet);

console.log(newGreet()); // "Hello from higher-order function"

### **7. Callback Functions**

Functions that are passed as arguments to other functions and executed later, often used in asynchronous programming.

function fetchData(callback) {

setTimeout(() => {

callback("Data received");

}, 1000);

}

function displayData(data) {

console.log(data);

}

fetchData(displayData); // "Data received" after 1 second

### **8. Generator Functions**

Functions that can be paused and resumed, allowing them to produce a sequence of values over time. Defined with an asterisk (\*) and using the yield keyword.

function\* generatorFunction() {

yield 1;

yield 2;

yield 3;

}

const generator = generatorFunction();

console.log(generator.next().value); // 1

console.log(generator.next().value); // 2

console.log(generator.next().value); // 3

### **9. Async Functions**

Functions that allow for asynchronous operations using the async keyword. They return a Promise and can use the await keyword to wait for Promises.

async function fetchData() {

let response = await fetch('https://api.example.com/data');

let data = await response.json();

console.log(data);

}

### **10. Constructor Functions**

Functions used to create objects, typically with the new keyword. They serve as templates for creating multiple objects with similar properties and methods.

function Person(name, age) {

this.name = name;

this.age = age;

}

const person1 = new Person("Alice", 30);

console.log(person1.name); // Alice

**Differences between Arrow and Regular function, i.e**

1. Syntax
2. No arguments **(arguments are array-like objects)**
3. No prototype object for the **Arrow function**
4. Cannot be invoked with a new keyword **(Not a constructor function)**
5. No own this **(call, apply & bind won’t work as expected)**
6. It cannot be used as a Generator function
7. Duplicate-named parameters are not allowed

**What is isNaN?**

It is a function which determine whether or not value is an illegal number. The isNan() method returns true if the passed value is NaN(Not a number) and is of type number, else it returns false.

e.g.

Input: ‘213’ Output: false

Input:’hello’ Output: true

**What is alert, confirm and prompt?**

In JavaScript, alert, confirm, and prompt are built-in methods used to interact with users through dialog boxes. They are part of the window object and are primarily used for displaying messages and getting input from the user. Here’s a brief overview of each:

**alert**

The alert method displays a simple message in a dialog box with an OK button. It is typically used to provide information to the user. The user must click the OK button to close the dialog and continue interacting with the webpage.

alert("This is an alert message.");

**confirm**

The confirm method displays a dialog box with a specified message, along with OK and Cancel buttons. It is used to ask the user to confirm or cancel an action. It returns true if the user clicks OK, and false if the user clicks Cancel.

const userConfirmed = confirm("Are you sure you want to proceed?");

if (userConfirmed) {

console.log("User confirmed the action.");

} else {

console.log("User canceled the action.");

}

**prompt**

The prompt method displays a dialog box that asks the user for input. It contains a specified message, a text input field, and OK and Cancel buttons. It returns the input value as a string if the user clicks OK. If the user clicks Cancel, it returns null.

const userInput = prompt("Please enter your name:");

if (userInput !== null) {

console.log("Hello, " + userInput + "!");

} else {

console.log("User canceled the prompt.");

}

**What are scopes?**

In JavaScript, scope determines where variables and functions are accessible. The three main types of scope are:

1. **Global Scope**: Variables declared outside any function or block are globally scoped and accessible throughout the code.
2. **Function Scope**: Variables declared inside a function using var, let, or const are only accessible within that function.
3. **Block Scope**: Variables declared inside a block (e.g., within {}) using let or const are only accessible within that block.

**Hoisting**: JavaScript moves variable and function declarations to the top of their scope before code execution. Variables declared with var are hoisted but not their initializations, while function declarations are fully hoisted.

**Lexical scoping**

Lexical scoping in JavaScript refers to the concept of determining the accessibility of variables and functions based on their position in the source code. It is also known as static scoping. In JavaScript, lexical scoping is used to resolve variable names when a function is created, not where it is invoked.

Here's a brief overview:

1. **Function Scope**: Variables declared within a function are only accessible inside that function, including any nested functions. This is the most common type of lexical scoping in JavaScript.

function outer() {

let outerVar = 'I am outside!';

function inner() {

console.log(outerVar); // Accesses outerVar due to lexical scoping

}

inner();

}

outer(); // Outputs: I am outside!

1. **Block Scope**: With the introduction of let and const in ES6, JavaScript also supports block scope. Variables declared with let or const are only accessible within the block they are defined in (e.g., within {} braces).

{

let blockVar = 'I am inside a block!';

console.log(blockVar); // Outputs: I am inside a block!

}

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

1. **Closure**: Lexical scoping allows functions to retain access to their outer scope even after the outer function has finished executing. This is known as a closure.

function createCounter() {

let count = 0;

return function() {

count++;

return count;

};

}

const counter = createCounter();

console.log(counter()); // Outputs: 1

console.log(counter()); // Outputs: 2

**47.**   **What is strict mode?**

* Strict mode prevents certain actions and throws more exceptions. The statement “use strict” orders browser to use the Strict mode, which is a reduced and safer feature set of JavaScript.
* **Disallows Duplicates**: It disallows duplicate parameter names in function definitions and object literals.
* **Eliminates Silent Errors**: It converts some silent errors into throw errors. For example, assigning values to undeclared variables will throw an error in strict mode.
* Strict mode resolves mistakes that make it difficult for JavaScript engines to perform optimizations hence strict mode code can sometimes run faster than identical code that’s not strict mode.
* Strict mode forbids some syntax likely to be defined in future versions of ECMAScript.
* It prevents, or throws errors, when unsafe actions are taken (such as gaining access to the global object).
* It disables features that are confusing or poorly thought out.
* Due to Strict mode it becomes easier to write secure JavaScript.
* Strict mode applies to individual functions or to entire scripts. It doesn't apply to block statements enclosed in {} braces; attempting to apply it to such contexts does nothing.
* To invoke strict mode for an entire script, put statement "use strict" before any other statements.
* To invoke strict mode for a function, put statement "use strict" in the function's body before any other statements.

**What is a closure ?**

**Closure** is a feature that allows the function to capture the environment (**or to retain access to variables from the scope** ) where it is defined, even after that scope has closed.

We can say the closure is a **combination of a function and lexical environment where that function is defined.**

In other words, a closure gives a function access to its own scope, the scope of its outer function, and the global scope, allowing it to “remember” and continue to access variables and parameters from these scopes.

**Key Characteristics of Closures**

1**. Access to Outer Function's** Scope: An inner function has access to the variables in its outer function’s scope.

2. **Preservation of Variables:** Variables in the outer function's scope are preserved even after the outer function has returned.

3. **Private Variables:** Closures can be used to create private variables, which are not accessible from the outside scope.

function outerFunction() {

let outerVariable = 'I am from the outer function';

return innerFunction() {

console.log(outerVariable); // Accessing outerVariable from the outer function's scope

}

}

let myFunction = outerFunction();

myFunction(); // Output: I am from the outer function

**Closure is created every time when a function is created at the time of function creation and when you define a function inside another function.**

*Execution context is an environment where JavaScript code is executed. For each function call a separate execution context is created and pushed into the execution stack. Once function execution completed it is popped off from stack.*

*Every execution context has a space in memory where its variables and function are stored, and once the function popped off from the execution stack a JavaScript Garbage collector clear all of these things.*

*In JavaScript, anything is garbage-collected only when there are no references to it.*

In the above example, the anonymous execution context still has a reference to the variables to the memory space of its outer environment. Even though the outerFunction() is finished. (It can access the **outerVariabl**e variable and use it inside console.log(outerVariable)).

**Practical Use of Closures**

Closures are often used in practical programming scenarios for tasks such as creating private variables, callbacks, and event handlers.

**Creating Private Variables**

Closures can be used to create private variables, making it possible to hide implementation details and expose only what is necessary.

function counter() {

let count = 0; // Private variable

return function() {

count += 1;

return count;

};

}

const increment = counter();

console.log(increment()); // 1

console.log(increment()); // 2

console.log(increment()); // 3

#### Maintaining State in Asynchronous Operations

Closures are commonly used in asynchronous operations to maintain state.

function asyncOperation(message, delay) {

setTimeout(function() {

console.log(message);

}, delay);

}

asyncOperation("Hello, after 1 second", 1000);

asyncOperation("Hello, after 2 seconds", 2000);

**Explain setInterval and setTimeout in Javascript**

setTimeout and setInterval are both JavaScript functions used for executing code after a specified delay. Here's a brief overview of each:

**setTimeout**

setTimeout is used to execute a function or piece of code once after a specified delay (in milliseconds).

**Syntax:**

let timeoutID = setTimeout(function, delay, arg1, arg2, ...);

**Example:**

setTimeout(() => {

console.log('Executed after 2 seconds');

}, 2000);

**Cancelling setTimeout:**

You can cancel a timeout using the clearTimeout function and passing the ID returned by setTimeout.

let timeoutID = setTimeout(() => {

console.log('This will not run');

}, 2000);

clearTimeout(timeoutID);

**setInterval**

setInterval is used to execute a function or piece of code repeatedly, with a fixed time delay between each call.

**Syntax:**

let intervalID = setInterval(function, delay, arg1, arg2, ...);

**Example:**

setInterval(() => {

console.log('Executed every 2 seconds');

}, 2000);

**Cancelling setInterval:**

You can cancel an interval using the clearInterval function and passing the ID returned by setInterval.

let intervalID = setInterval(() => {

console.log('This will not run repeatedly');

}, 2000);

clearInterval(intervalID);

**What is an event?**

**JavaScript Events** are**actions or occurrences**that happen in the browser. They can be triggered by various user interactions or by the browser itself.

**What are different events in Javascript?**

### Mouse Events

* **click**: Fired when a pointing device button is pressed and released on an element.
* **dblclick**: Fired when a pointing device button is clicked twice on an element.
* **mouseover**: Fired when a pointing device is moved onto an element.
* **mouseout**: Fired when a pointing device is moved off an element.
* **mousedown**: Fired when a pointing device button is pressed on an element.
* **mouseup**: Fired when a pointing device button is released over an element.
* **mousemove**: Fired when a pointing device is moved while over an element.
* **mouseenter**: Fired when a pointing device is moved onto an element, but does not bubble.
* **mouseleave**: Fired when a pointing device is moved off an element, but does not bubble.

### Keyboard Events

* **keydown**: Fired when a key is pressed down.
* **keyup**: Fired when a key is released.
* **keypress**: Fired when a key is pressed (deprecated, use keydown or keyup instead).

### Form Events

* **submit**: Fired when a form is submitted.
* **change**: Fired when the value of an element has been changed.
* **input**: Fired when the value of an element is changed through user input.
* **focus**: Fired when an element gains focus.
* **blur**: Fired when an element loses focus.
* **reset**: Fired when a form is reset.
* **select**: Fired when some text is selected in an element.

### Window Events

* **load**: Fired when the whole page and all dependent resources have finished loading.
* **resize**: Fired when the document view (window) has been resized.
* **scroll**: Fired when the document view or an element is scrolled.
* **unload**: Fired when the document or a resource is being unloaded.
* **beforeunload**: Fired when the window, the document and its resources are about to be unloaded.
* **hashchange**: Fired when the fragment identifier of the URL has changed.

### Clipboard Events

* **copy**: Fired when the user initiates a copy action.
* **cut**: Fired when the user initiates a cut action.
* **paste**: Fired when the user initiates a paste action.

### Drag and Drop Events

* **drag**: Fired when an element is being dragged.
* **dragend**: Fired when a drag operation ends.
* **dragenter**: Fired when the dragged element enters a drop target.
* **dragleave**: Fired when the dragged element leaves a drop target.
* **dragover**: Fired when an element is being dragged over a drop target.
* **dragstart**: Fired when the user starts dragging an element.
* **drop**: Fired when the dragged element is dropped on a drop target.

### Media Events

* **play**: Fired when playback is started.
* **pause**: Fired when playback is paused.
* **ended**: Fired when playback has stopped because the end of the media was reached.
* **volumechange**: Fired when the volume has changed.
* **timeupdate**: Fired when the playback position changed.

### Touch Events (for touch-enabled devices)

* **touchstart**: Fired when a touch point is placed on the touch surface.
* **touchend**: Fired when a touch point is removed from the touch surface.
* **touchmove**: Fired when a touch point is moved along the touch surface.
* **touchcancel**: Fired when a touch point has been disrupted.

### Focus Events

* **focus**: Fired when an element gains focus.
* **blur**: Fired when an element loses focus.
* **focusin**: Fired when an element is about to gain focus, bubbles.
* **focusout**: Fired when an element is about to lose focus, bubbles.

### Document Events

* **DOMContentLoaded**: Fired when the initial HTML document has been completely loaded and parsed.
* **readystatechange**: Fired when the ready state of the document changes.

### Miscellaneous Events

* **contextmenu**: Fired when the right button of the mouse is clicked (before the context menu is displayed).
* **wheel**: Fired when a mouse wheel or similar device is rotated.
* **error**: Fired when an error occurs while loading an external file.

**What are event handlers?**

Event handlers are functions or pieces of code that are executed in response to specific events in JavaScript. They enable developers to define what should happen when a particular event occurs on an element or the document as a whole. Event handlers can be used to create interactive and dynamic web applications by responding to user actions such as clicks, key presses, form submissions, and more.

### **How Event Handlers Work**

When an event occurs, the browser checks if there are any event handlers registered for that event on the target element or its ancestors (in case of event bubbling). If it finds any, it calls them in the order they were registered.

### **Adding Event Handlers**

There are several ways to add event handlers in JavaScript:

1. **Inline Event Handlers:**
   * Defined directly in the HTML attributes of an element.

<button onclick="alert('Button clicked!')">Click me</button>

1. **DOM Level 0 Event Handlers:**
   * Assigning an event handler directly to a property of a DOM element.

const button = document.getElementById('myButton');

button.onclick = function() {

alert('Button clicked!');

};

1. **DOM Level 2 Event Listeners:**
   * Using addEventListener to attach an event handler.

const button = document.getElementById('myButton');

button.addEventListener('click', function() {

alert('Button clicked!');

});

### **Removing Event Handlers**

Event handlers added with addEventListener can be removed using removeEventListener:

const button = document.getElementById('myButton');

function handleClick() {

alert('Button clicked!');

}

button.addEventListener('click', handleClick);

button.removeEventListener('click', handleClick);

**What is addEventListener() method?**

The addEventListener() method attaches an event handler to specified element. You can add multiple event handlers to one element. It is possible to add event listener to any DOM object. e.g.

document.getElementById(“someUniqueDivId”).addEventListener(“click”, respondtoclick);

function respondtoclick() {

console.log(“Do some stuff!!!”);

}

**What is event bubbling and event capturing?**

### **Event Bubbling**

In event bubbling, the event starts at the target element (the one that triggered the event) and moves up the DOM tree to the root. This means the event is first handled by the innermost element and then by its parent elements, one by one.

**Example:** If you click on a button inside a div, the click event first triggers on the button, then on the div, and then on its parent elements, all the way up to the root.

### **Event Capturing**

In event capturing, the event starts at the root of the DOM tree and moves down to the target element. This means the event is first handled by the outermost element and then by its child elements, one by one, until it reaches the target element.

**Example:** If you click on a button inside a div, the click event first triggers on the root element, then on the div, and finally on the button.

### **Adding Event Listeners for Capturing and Bubbling**

By default, event listeners are registered for the bubbling phase. However, you can specify whether an event listener should be invoked during the capturing phase by setting the capture parameter to true in addEventListener.

// Event listener for the bubbling phase (default)

element.addEventListener('click', function() {

console.log('Bubbling phase');

}, false);

// Event listener for the capturing phase

element.addEventListener('click', function() {

console.log('Capturing phase');

}, true);

### Example to Demonstrate Event Bubbling and Capturing

html

Copy code

<!DOCTYPE html>

<html>

<head>

<title>Event Propagation Example</title>

</head>

<body>

<div id="outerDiv">

<button id="innerButton">Click me</button>

</div>

<script>

// Event listener for outerDiv during the capturing phase

document.getElementById('outerDiv').addEventListener('click', function() {

console.log('Capturing: Outer Div');

}, true);

// Event listener for outerDiv during the bubbling phase

document.getElementById('outerDiv').addEventListener('click', function() {

console.log('Bubbling: Outer Div');

}, false);

// Event listener for innerButton

document.getElementById('innerButton').addEventListener('click', function() {

console.log('Button Clicked');

}, false);

</script>

</body>

</html>

When you click the button:

1. The event starts at the root and moves downwards in the capturing phase.
2. Capturing: Outer Div is logged (since it's set to capture).
3. The event reaches the target element and Button Clicked is logged.
4. The event bubbles back up and Bubbling: Outer Div is logged.

**what is event delegation**

Event delegation is a technique in JavaScript that allows you to handle events at a higher level in the DOM rather than directly on the target elements. This is especially useful when you have multiple child elements that need similar event handling. Instead of adding an event listener to each child element, you add a single event listener to a parent element and use the event's target property to determine which child element triggered the event.

### **How Event Delegation Works**

1. **Event Propagation**: Event delegation relies on event bubbling, where an event propagates up from the target element to the parent elements.
2. **Single Event Listener**: Instead of adding multiple event listeners to individual child elements, you add one event listener to a common ancestor element.
3. **Event Targeting**: Use the event.target property to determine which child element triggered the event and execute the appropriate logic.

### **Benefits of Event Delegation**

* **Improved Performance**: Reduces the number of event listeners, which can improve performance, especially for a large number of child elements.
* **Dynamic Elements**: Easily handle events for dynamically added elements without needing to reattach event listeners.

<!DOCTYPE html>

<html>

<head>

<title>Event Delegation Example</title>

</head>

<body>

<ul id="parentList">

<li>Item 1</li>

<li>Item 2</li>

<li>Item 3</li>

</ul>

<script>

// Add event listener to the parent element

const parentList = document.getElementById('parentList');

parentList.addEventListener('click', function(event) {

// Check if the clicked element is an <li> (list item)

if (event.target.tagName === 'LI') {

alert('You clicked on ' + event.target.innerText);

}

});

</script>

</body>

</html>

In this example:

An event listener is added to the ul element (parentList).

When any li (list item) is clicked, the event bubbles up to the ul element.

The event handler checks if the clicked element (event.target) is an li and then executes the logic (displaying an alert with the text of the clicked item).

### Key Points

**Event delegation** simplifies the management of events for multiple child elements by using a single event listener on a parent element.

It is particularly useful for dynamic content where child elements may be added or removed dynamically.

Event delegation relies on event bubbling to propagate events from child elements to their parents.

**What are Objects?**

In JavaScript, an object is a collection of key-value pairs, where each key is a string and each value can be any data type, including a function. Objects are used to store and manipulate data in a structured way.

### **Creating Objects**

1. **Using Object Literals:**

const person = {

name: 'John',

age: 30,

isEmployed: true,

greet: function() {

console.log('Hello, my name is ' + this.name);

}

};

1. **Using the new Object() Syntax:**

const person = new Object();

person.name = 'John';

person.age = 30;

person.isEmployed = true;

person.greet = function() {

console.log('Hello, my name is ' + this.name);

};

1. **Using a Constructor Function:**

function Person(name, age, isEmployed) {

this.name = name;

this.age = age;

this.isEmployed = isEmployed;

this.greet = function() {

console.log('Hello, my name is ' + this.name);

};

}

const person = new Person('John', 30, true);

1. **Using ES6 Classes:**

class Person {

constructor(name, age, isEmployed) {

this.name = name;

this.age = age;

this.isEmployed = isEmployed;

}

greet() {

console.log('Hello, my name is ' + this.name);

}

}

const person = new Person('John', 30, true);

**5. Using Object.create():**

The Object.create() method creates a new object, using an existing object as the prototype of the newly created object.

const person = {

isHuman: false,

printIntroduction: function () {

console.log(`Am I human? ${this.isHuman}`);

}

};

const me = Object.create(person);

### **Accessing Object Properties**

1. **Dot Notation:**

console.log(person.name); // John

console.log(person.age); // 30

1. **Bracket Notation:**

console.log(person['name']); // John

console.log(person['age']); // 30

### **Modifying Object Properties**

1. **Using Dot Notation:**

person.age = 31;

person.isEmployed = false;

1. **Using Bracket Notation:**

person['age'] = 31;

person['isEmployed'] = false;

### **Adding New Properties**

1. **Using Dot Notation:**

person.address = '123 Main St';

1. **Using Bracket Notation:**

person['address'] = '123 Main St';

### **Deleting Properties**

delete person.age;

### **Looping Through Object Properties**

You can use a for...in loop to iterate over an object's properties:

for (let key in person) {

console.log(key + ': ' + person[key]);

}

### **Object Methods**

Objects can have methods, which are functions that are properties of the object.

const person = {

name: 'John',

greet: function() {

console.log('Hello, my name is ' + this.name);

}

};

person.greet(); // Hello, my name is John

### **Built-in Object Methods**

Javascript provides several built-in methods for working with objects, such as:

* Object.keys(obj): Returns an array of a given object's property names.
* Object.values(obj): Returns an array of a given object's property values.
* Object.entries(obj): Returns an array of a given object's own enumerable string-keyed property [key, value] pairs.

const person = {

name: 'John',

age: 30,

isEmployed: true

};

console.log(Object.keys(person)); // ['name', 'age', 'isEmployed']

console.log(Object.values(person)); // ['John', 30, true]

console.log(Object.entries(person)); // [['name', 'John'], ['age', 30], ['isEmployed', true]]

**Which are built-in or native objects?**

JavaScript provides Number, Boolean, String, Array, Date, Math, RegExp which are built in objects.

**What is ‘this’ keyword?**

The this keyword refers to the object it belongs to. In an object method, this refers to the object to which method belongs.

When used alone, the owner is the Global object, so this refers to the Global object (Windows object).

In a function, this refers to the Global object (Windows object). In strict mode, when used in a function, this is undefined.

In HTML event handlers, this refers to the element in html that received the event.

**What is RegExp object ?**

The RegExp object in JavaScript is used for matching text with a pattern. Regular expressions (regex or regexp) are patterns used to match character combinations in strings. They can be used for searching, replacing, and validating string data.

### **Creating a RegExp Object**

There are two ways to create a RegExp object in JavaScript:

1. **Using a Regular Expression Literal:**

const regex = /pattern/flags;

1. **Using the RegExp Constructor:**

const regex = new RegExp('pattern', 'flags');

### **Flags**

Regular expressions can have optional flags that affect the search behavior. Some common flags are:

* g (global): Perform a global match (find all matches rather than stopping after the first match).
* i (ignore case): Perform case-insensitive matching.
* m (multiline): Treat beginning and end characters (^ and $) as working across multiple lines.
* u (unicode): Treat the pattern as a sequence of Unicode code points.
* s (dotAll): Allow the dot (.) to match newline characters.

### **Methods of RegExp**

1. **test()**: Tests for a match in a string. Returns true or false.

const regex = /hello/i;

console.log(regex.test('Hello World')); // true

1. **exec()**: Executes a search for a match in a string. Returns an array of information or null.

const regex = /hello/i;

const result = regex.exec('Hello World');

console.log(result); // ['Hello', index: 0, input: 'Hello World', groups: undefined]

### String Methods Supporting RegExp

Several string methods support regular expressions:

1. **match()**: Retrieves the result of matching a string against a regular expression.

const str = 'Hello World';

const regex = /hello/i;

console.log(str.match(regex)); // ['Hello', index: 0, input: 'Hello World', groups: undefined]

1. **replace()**: Replaces occurrences of a pattern with a replacement string.

const str = 'Hello World';

const regex = /world/i;

const newStr = str.replace(regex, 'JavaScript');

console.log(newStr); // 'Hello JavaScript'

1. **search()**: Searches for a match and returns the index of the first match, or -1 if not found.

const str = 'Hello World';

const regex = /world/i;

console.log(str.search(regex)); // 6

1. **split()**: Splits a string into an array of substrings.

const str = 'Hello World';

const regex = / /;

console.log(str.split(regex)); // ['Hello', 'World']

**What is namespace?**

A namespace is a container that holds a set of identifiers such as variables, functions, classes, and objects, and allows for the organization and management of these identifiers in a way that prevents name conflicts. Namespaces are particularly useful in large programs or when integrating multiple libraries, as they help ensure that identifiers from different parts of the program or from different libraries do not collide.

### **Key Concepts of Namespaces**

1. **Avoiding Name Conflicts:**
   * Namespaces help prevent naming conflicts by allowing the same name to be used for different purposes in different contexts. For example, two different libraries might define a function with the same name, but by placing them in different namespaces, they can coexist without conflict.
2. **Organizing Code:**
   * Namespaces provide a way to organize code into logical groups. This can make the code more readable and maintainable.

### **Namespaces in JavaScript**

JavaScript does not have built-in support for namespaces as seen in some other languages like C++ or C#. However, there are several patterns and techniques that can be used to create namespace-like structures.

#### Using Objects as Namespaces

One common way to implement namespaces in JavaScript is by using objects. An object can act as a namespace, grouping related functions, variables, and other objects together.

// Create a namespace

const MyNamespace = {

foo: function() {

console.log('foo function');

},

bar: function() {

console.log('bar function');

},

data: {

value1: 10,

value2: 20

}

};

// Accessing members of the namespace

MyNamespace.foo(); // foo function

MyNamespace.bar(); // bar function

console.log(MyNamespace.data.value1); // 10

#### Immediately Invoked Function Expressions (IIFE)

Another approach is to use an Immediately Invoked Function Expression (IIFE) to create a private scope and expose only the desired variables and functions.

const MyNamespace = (function() {

const privateVariable = 'I am private';

function privateFunction() {

console.log(privateVariable);

}

return {

publicFunction: function() {

privateFunction();

},

publicVariable: 'I am public'

};

})();

// Accessing members of the namespace

MyNamespace.publicFunction(); // I am private

console.log(MyNamespace.publicVariable); // I am public

#### ES6 Modules

With the introduction of ES6, JavaScript modules provide a built-in way to manage namespaces. Modules allow you to encapsulate code and export only the parts you want to be public.

// myModule.js

const privateVariable = 'I am private';

function privateFunction() {

console.log(privateVariable);

}

export function publicFunction() {

privateFunction();

}

export const publicVariable = 'I am public';

// main.js

import { publicFunction, publicVariable } from './myModule.js';

publicFunction(); // I am private

console.log(publicVariable); // I am public

### **Summary**

* **Namespace**: A container that holds identifiers and prevents name conflicts.
* **JavaScript Namespaces**: Implemented using objects, IIFEs, or ES6 modules.
* **Benefits**: Prevents name conflicts, organizes code, and enhances code maintainability and readability.

**What is prototype in javascript?**

In JavaScript, a prototype is an object that provides a mechanism for inheritance. Every JavaScript object has a prototype, which is also an object. Through prototypes, objects can inherit properties and methods from other objects.

### **Key Concepts of Prototype in JavaScript**

1. **Prototype Property:**
   * For functions (which can be used as constructors), the prototype property is an object that is shared among all instances created by the constructor.
   * For objects, the prototype is accessible via \_\_proto\_\_ (although not recommended to use directly) or Object.getPrototypeOf().
2. **Prototype Chain:**

When you try to access a property or method on an object, JavaScript first looks at the object itself. If it doesn't find it there, it looks at the object's prototype, and then at the prototype's prototype, and so on. This chain is known as the prototype chain.

1. **Inheritance:**
   * Prototypes allow objects to inherit properties and methods from other objects. This enables code reuse and the creation of hierarchical object structures.

### **Example: Using Prototype with Constructor Functions**

// Constructor function

function Person(name, age) {

this.name = name;

this.age = age;

}

// Adding a method to the prototype

Person.prototype.greet = function() {

console.log('Hello, my name is ' + this.name);

};

// Creating instances

const person1 = new Person('John', 30);

const person2 = new Person('Jane', 25);

// Both instances share the same greet method

person1.greet(); // Hello, my name is John

person2.greet(); // Hello, my name is Jane

### Example: Prototype Chain

// Constructor function

function Animal(name) {

this.name = name;

}

Animal.prototype.eat = function() {

console.log(this.name + ' is eating');

};

// Another constructor function

function Dog(name, breed) {

Animal.call(this, name); // Call the parent constructor

this.breed = breed;

}

// Inherit from Animal

Dog.prototype = Object.create(Animal.prototype);

Dog.prototype.constructor = Dog;

Dog.prototype.bark = function() {

console.log(this.name + ' is barking');

};

// Creating instances

const dog = new Dog('Rex', 'Golden Retriever');

dog.eat(); // Rex is eating (inherited from Animal)

dog.bark(); // Rex is barking (defined in Dog)

### **ES6 Classes and Prototype**

With the introduction of ES6, the class syntax provides a more straightforward and modern way to work with prototypes and inheritance, although it is syntactic sugar over the existing prototype-based inheritance.

class Person {

constructor(name, age) {

this.name = name;

this.age = age;

}

greet() {

console.log('Hello, my name is ' + this.name);

}

}

const person1 = new Person('John', 30);

person1.greet(); // Hello, my name is John

### **Checking the Prototype**

You can check the prototype of an object using Object.getPrototypeOf() or the \_\_proto\_\_ property:

const person1 = new Person('John', 30);

console.log(Object.getPrototypeOf(person1) === Person.prototype); // true

console.log(person1.\_\_proto\_\_ === Person.prototype); // true (not recommended to use \_\_proto\_\_)

### **Summary**

* **Prototype**: An object from which other objects inherit properties.
* **Prototype Chain**: The chain of references that JavaScript follows when looking for properties and methods.
* **Inheritance**: Allows objects to share properties and methods, enabling code reuse.
* **ES6 Classes**: Provides a modern syntax for working with prototypes and inheritance, making it easier to understand and use.

**Explain call() Method.**

The call() method in JavaScript is a built-in method available on functions that allows you to invoke a function with a specified this context and arguments provided individually. This method is particularly useful for borrowing methods from one object and using them on another object.

### Syntax

function.call(thisArg, arg1, arg2, ...)

* **thisArg**: The value to be used as the this context within the function. It can be any object or null.
* **arg1, arg2, ...**: Arguments to be passed to the function.

### How call() Works

1. **Invokes a Function with a Specific this Context**: The call() method allows you to specify what this should refer to inside the function. This is useful when you want to control the context in which a function executes.
2. **Passes Arguments Individually**: Unlike the apply() method, which takes an array of arguments, call() requires arguments to be passed individually.

### Example

function greet(greeting, punctuation) {

console.log(greeting + ', ' + this.name + punctuation);

}

const person1 = { name: 'Alice' };

const person2 = { name: 'Bob' };

// Using call() to set the `this` context to person1

greet.call(person1, 'Hello', '!'); // Hello, Alice!

// Using call() to set the `this` context to person2

greet.call(person2, 'Hi', '.'); // Hi, Bob.

### Borrowing Methods

You can use call() to borrow methods from one object and use them on another object.

const person1 = {

name: 'Alice',

sayName: function() {

console.log('My name is ' + this.name);

}

};

const person2 = {

name: 'Bob'

};

// Borrowing the sayName method from person1 and using it on person2

person1.sayName.call(person2); // My name is Bob

### Using call() for Inheritance

call() can also be used to simulate inheritance by calling a constructor function with a different this value.

function Animal(name) {

this.name = name;

}

function Dog(name, breed) {

Animal.call(this, name); // Call the Animal constructor with `this` set to the new Dog instance

this.breed = breed;

}

const myDog = new Dog('Rex', 'Golden Retriever');

console.log(myDog.name); // Rex

console.log(myDog.breed); // Golden Retriever

### Summary

* **call() Method**: Invokes a function with a specified this context and individual arguments.
* **thisArg**: Specifies what this should refer to inside the function.
* **Arguments**: Passed individually.
* **Common Uses**:
  + Invoking functions with a specific this context.
  + Borrowing methods from one object to use on another.
  + Simulating inheritance by calling a constructor function with a different this value.

**Explain apply() method.**

The apply() method in JavaScript is similar to the call() method. It is used to invoke a function with a specified this context, but the main difference is that apply() accepts an array (or an array-like object) of arguments instead of listing them individually.

### Syntax

function.apply(thisArg, [argsArray])

* **thisArg**: The value to be used as the this context within the function. It can be any object or null.
* **argsArray**: An array or array-like object containing the arguments to be passed to the function.

### How apply() Works

1. **Invokes a Function with a Specific this Context**: The apply() method allows you to specify what this should refer to inside the function, just like call().
2. **Passes Arguments as an Array**: Instead of passing arguments individually, you pass them as an array or array-like object. This is particularly useful when you need to pass a variable number of arguments.

### Example

function greet(greeting, punctuation) {

console.log(greeting + ', ' + this.name + punctuation);

}

const person1 = { name: 'Alice' };

const person2 = { name: 'Bob' };

// Using apply() to set the `this` context to person1

greet.apply(person1, ['Hello', '!']); // Hello, Alice!

// Using apply() to set the `this` context to person2

greet.apply(person2, ['Hi', '.']); // Hi, Bob.

### Borrowing Methods

You can use apply() to borrow methods from one object and use them on another object, similar to call().

const person1 = {

name: 'Alice',

sayName: function() {

console.log('My name is ' + this.name);

}

};

const person2 = {

name: 'Bob'

};

// Borrowing the sayName method from person1 and using it on person2

person1.sayName.apply(person2); // My name is Bob

### Using apply() for Variadic Functions

The apply() method is especially useful when dealing with functions that take a variable number of arguments.

function sum() {

return Array.prototype.reduce.call(arguments, (a, b) => a + b, 0);

}

const numbers = [1, 2, 3, 4, 5];

// Using apply() to pass the numbers array as individual arguments to the sum function

const result = sum.apply(null, numbers);

console.log(result); // 15

### Using apply() for Inheritance

apply() can also be used to simulate inheritance by calling a constructor function with a different this value and passing arguments as an array.

function Animal(name) {

this.name = name;

}

function Dog(name, breed) {

Animal.apply(this, [name]); // Call the Animal constructor with `this` set to the new Dog instance

this.breed = breed;

}

const myDog = new Dog('Rex', 'Golden Retriever');

console.log(myDog.name); // Rex

console.log(myDog.breed); // Golden Retriever

### Summary

* **apply() Method**: Invokes a function with a specified this context and arguments passed as an array.
* **thisArg**: Specifies what this should refer to inside the function.
* **argsArray**: An array or array-like object containing the arguments to be passed to the function.
* **Common Uses**:
  + Invoking functions with a specific this context.
  + Borrowing methods from one object to use on another.
  + Handling functions that take a variable number of arguments.
  + Simulating inheritance by calling a constructor function with a different this value and passing arguments as an array.

**Explain bind() method.**

The bind() method in JavaScript is used to create a new function that, when called, has its this keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called. This method allows you to set the this context for a function and also to partially apply function arguments.

### Syntax

function.bind(thisArg, arg1, arg2, ...)

* **thisArg**: The value to be used as the this context within the new function.
* **arg1, arg2, ...**: Arguments to prepend to arguments provided when the bound function is called.

### Key Features

1. **Creates a New Function**: The bind() method does not execute the function immediately. Instead, it returns a new function with the specified this context and arguments.
2. **Partial Application**: Arguments provided to bind() are prepended to the arguments provided when the new function is invoked. This allows for partial application of functions.
3. **Cannot Be Overridden**: A bound function's this context cannot be overridden, even with call() or apply().

### Examples

#### Setting this Context

const person = {

name: 'Alice',

greet: function(greeting) {

console.log(greeting + ', ' + this.name);

}

};

const greet = person.greet.bind(person);

greet('Hello'); // Hello, Alice

#### Partial Application

function multiply(a, b) {

return a \* b;

}

const double = multiply.bind(null, 2); // Pre-set `a` to 2

console.log(double(5)); // 10

console.log(double(10)); // 20

#### Borrowing Methods with this Context

const person1 = {

name: 'Alice',

sayName: function() {

console.log('My name is ' + this.name);

}

};

const person2 = { name: 'Bob' };

const sayNameForPerson2 = person1.sayName.bind(person2);

sayNameForPerson2(); // My name is Bob

#### Using bind() for Event Handlers

const button = document.getElementById('myButton');

const handler = {

message: 'Button clicked!',

handleClick: function() {

console.log(this.message);

}

};

button.addEventListener('click', handler.handleClick.bind(handler));

In this example, bind() ensures that the handleClick method runs with the correct this context (the handler object) when the button is clicked.

### Differences from call() and apply()

* **Immediate Execution**:
  + call() and apply() immediately invoke the function with the specified this context and arguments.
  + bind() creates a new function that can be invoked later.
* **Argument Passing**:
  + call() passes arguments individually.
  + apply(), passes arguments as an array.
  + bind() allows partial application of arguments, pre-setting some arguments for later calls.

### Summary

* **bind() Method**: Creates a new function with a specific this context and optionally, partially applied arguments.
* **thisArg**: Specifies what this should refer to inside the new function.
* **Partial Application**: Pre-sets some arguments, allowing for partial application of functions.
* **Common Uses**:
  + Setting this context for functions.
  + Partially applying function arguments.
  + Ensuring the correct this context for methods used as event handlers or callbacks.

**What is Promise?**

A Promise in JavaScript is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. Promises are used to handle asynchronous operations more effectively, avoiding callback hell and making the code more readable and maintainable.

### **Key Concepts of Promises**

1. **Pending**: The initial state. The operation is not yet complete.
2. **Fulfilled**: The operation completed successfully, and the promise has a resulting value.
3. **Rejected**: The operation failed, and the promise has a reason for the failure.

### **Creating a Promise**

A Promise is created using the Promise constructor, which takes a function (the executor) as an argument. The executor function has two parameters: resolve and reject. These are functions used to settle the promise.

const myPromise = new Promise((resolve, reject) => {

// Asynchronous operation here

let success = true;

if (success) {

resolve('Operation successful!');

} else {

reject('Operation failed.');

}

});

### **Using Promises**

To handle the result of a promise, you use the .then(), .catch(), and .finally() methods.

#### .then()

Used to handle a fulfilled promise and execute a callback with the resolved value.

myPromise.then(result => {

console.log(result); // 'Operation successful!'

});

#### .catch()

Used to handle a rejected promise and execute a callback with the rejection reason.

myPromise.catch(error => {

console.error(error); // 'Operation failed.'

});

#### .finally()

Used to execute a callback once the promise is settled, regardless of its outcome.

myPromise.finally(() => {

console.log('Promise settled (fulfilled or rejected).');

});

### **Chaining Promises**

Promises can be chained to perform a series of asynchronous operations sequentially.

const fetchData = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Data fetched');

}, 1000);

});

};

fetchData()

.then(data => {

console.log(data); // 'Data fetched'

return 'Processing data';

})

.then(processedData => {

console.log(processedData); // 'Processing data'

return 'Displaying data';

})

.then(displayData => {

console.log(displayData); // 'Displaying data'

})

.catch(error => {

console.error('Error:', error);

});

### **Handling Multiple Promises**

#### Promise.all()

Waits for all promises to be fulfilled or for any to be rejected.

const promise1 = Promise.resolve(3);

const promise2 = 42;

const promise3 = new Promise((resolve, reject) => {

setTimeout(resolve, 100, 'foo');

});

Promise.all([promise1, promise2, promise3]).then(values => {

console.log(values); // [3, 42, 'foo']

});

#### Promise.race()

Waits for the first promise to be settled (fulfilled or rejected).

const promise1 = new Promise((resolve, reject) => {

setTimeout(resolve, 500, 'one');

});

const promise2 = new Promise((resolve, reject) => {

setTimeout(resolve, 100, 'two');

});

Promise.race([promise1, promise2]).then(value => {

console.log(value); // 'two'

});

#### Promise.allSettled()

Waits for all promises to be settled (fulfilled or rejected) and returns an array of objects describing each promise's outcome.

const promise1 = Promise.resolve('resolved');

const promise2 = Promise.reject('rejected');

Promise.allSettled([promise1, promise2]).then(results => {

console.log(results);

// [{status: 'fulfilled', value: 'resolved'}, {status: 'rejected', reason: 'rejected'}]

});

#### Promise.any()

Waits for the first promise to be fulfilled and ignores rejections.

const promise1 = Promise.reject('rejected');

const promise2 = Promise.resolve('resolved');

Promise.any([promise1, promise2]).then(value => {

console.log(value); // 'resolved'

});

### Summary

* **Promise**: An object representing the eventual completion or failure of an asynchronous operation.
* **States**: Pending, Fulfilled, Rejected.
* **Methods**: .then(), .catch(), .finally() for handling outcomes.
* **Chaining**: Allows sequential asynchronous operations.
* **Multiple Promises**: Promise.all(), Promise.race(), Promise.allSettled(), Promise.any() for handling multiple promises.

**What is Browser Object Model?**

The browser object model (BOM) is a hierarchy of browser objects that are used to manipulate methods and properties associated with the Web browser itself.

The default object of browser is window means you can invoke all the functions of window by specifying window or directly

Objects that make up the BOM include the window object, navigator object, screen object, location object, history, and the document object.

**What is Window Object?**

In a tabbed browser, each tab is represented as Window object.

Every object, variable, and function defined in a web page uses of the window as its Global object.

Window object provides methods like alert(), blur(), close(), confirm(), print(), prompt(), open().

**How to redirect other webpage?**

It is possible to redirect to other webpage in javascript by directly assigning value to window.location or by using location.assign(),location.replace() and location.reload() methods.

**What is Navigator Object?**

The navigator object is used to get browser information like name, version, type, language. It has methods like javaEnabled() and taintEnabled().

**How to identify operating system of client device?**

“Navigator.appVersion” is used to find operating system of client device.

**What is History Object?**

The History object consist of array of URLs which are visited by a user in browser.

History object provides method like back(), forward() and go().

**How to load previous page in browser programmatically?**

history.back() can be used to load previous page in browser through code.

**How to load next page in browser programmatically?**

history.forward() can be used to load next page in browser through code.

**What is go method of history object?**

The go() method loads a specific URL from the history list. history.go(number|URL)

number|URL parameter can either be a number which goes to the URL within the specific position (1 goes forward one page, -1 goes back one page), or a string. The string has to be a partial or full URL, and the function will go to the first URL that matches the string.

**What is Screen Object?**

The Screen object consist of information about display screen like height, width and colour bits of screen.

**What is Location Object?**

The Location object consist of information about current URL of window object.

Location object provides methods like assign(), reload and replace().

**How to print a web page?**

The window.print() will print the current web page when invoked

**What is Document Object?**

The Document object represents HTML document that is displayed in window. Document object has properties which allows access and modification of document content.

The way document is accessed and modified is called Document Object

Model or DOM.

Document object provides methods like open(), close(), write(), getElementById(), getElementByName(), getElementByTagName().

**What is DOM?**

It represents the hierarchical structure of a document as a tree of objects, where each node corresponds to a part of the document, such as elements, attributes, and text. The DOM allows scripts to access and manipulate the content, structure, and style of web pages dynamically.

The Document Object Model (DOM) represents HTML or XML page in such a way that programs can change document structure, content and style.

**What are DOM nodes?**

The DOM is structured as a tree of nodes. There are different types of nodes, including:

* **Element Nodes**: Represent HTML or XML elements (e.g., <div>, <p>, <img>, etc.).
* **Text Nodes**: Represent the text content inside elements.
* **Attribute Nodes**: Represent attributes of elements (e.g., class, id, src).
* **Comment Nodes**: Represent comments in the document.

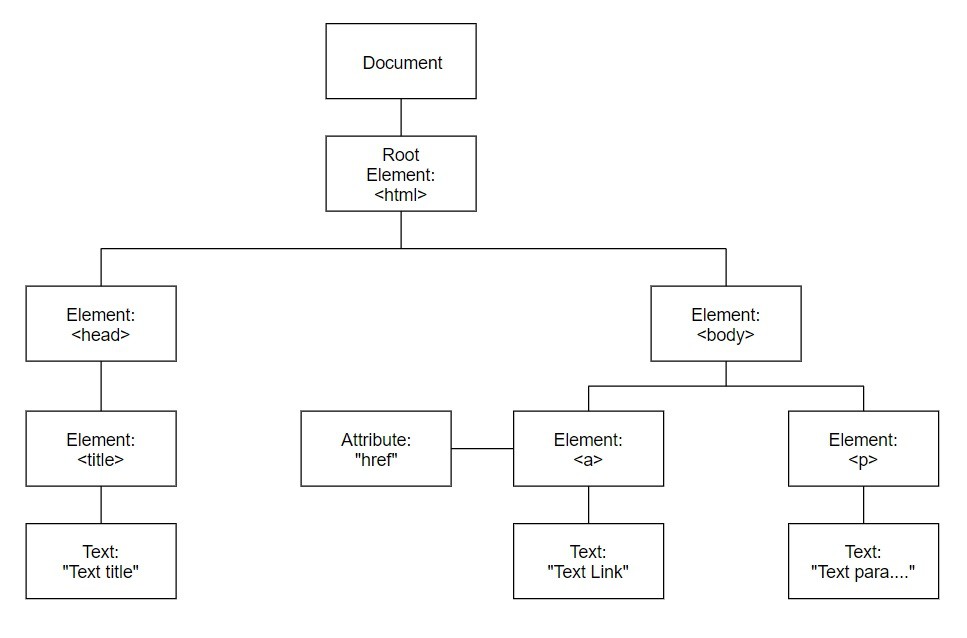
Document node represents whole document.

Element node represents every HTML element such as HTML, HEAD, BODY, A, H1 etc.

Text node represents text content inside the element. Attribute node represents every HTML attribute.

Node has node properties that contain information about the node

The nodeName property specifies name of the node. The nodeValue property specifies value of the node. The nodeType property specifies type of the node.



**How to get element with id in DOM?**

The getElementById() method of document object can be used to get element using id.

e.g. document.getElementById(“myUniqueId”);

**How to get element using class in DOM?**

The getElementByClassName () method of document object can be used to get element using class.

e.g. document.getElementByClassName(“myClass”);

**How to get content of any element?**

The innerHTML property is useful for getting or replacing the content of

HTML elements.

**What are DOM levels?**

The W3C DOM specifications are divided into different levels where each level contain some required and optional modules.

Level 0: Provide low-level set of interfaces.

Level1: DOM level 1 can be described in two parts: CORE and HTML.

CORE provides a low level interfaces that can be used to represent any structured document.

HTML provides high-level interfaces that can be used to represent HTML

document.

Level2: Consist of six specifications:

CORE2, VIEWS, EVENTS, STYLE, TRAVERSAL and RANGE. CORE2: extends functionality of CORE specified by DOM level 1.

VIEWS: views allow programs to dynamically access and manipulate content

of document.

EVENTS: events are scripts that are executed when user reacts to web page. STYLES: allow programs to dynamically access and manipulate content of

style sheets.

TRAVERSAL: allows programs to dynamically traverse the document. RANGE: allows programs to dynamically identify range of content in

document.

Level3: consists of five different specifications: CORE3, LOAD, SAVE, VALIDATIONS, EVENTS and XPATH.

CORE3: extents functionality of CORE specified by DOM level 2.

LOAD and SAVE: allows program to dynamically load the content of XML document into DOM document and save DOM document into XML document by serialization.

VALIDATION: allows program to dynamically update the content and structure of document while ensuring the document is valid.

EVENTS: extents functionality of Events specified by DOM level 2. XPATH: XPATH is a path language that can be used to access DOM tree.

**What are deferred scripts?**

By default, Javascript files will interrupt parsing of HTML document in order for them to be fetched and executed.

The defer attribute tells browser to only execute the script file once the

HTML document has been fully parsed.

<script defer src=”myscript.js”>

This reduces loading time of web page and web page is displayed faster.

**What are asynchronous scripts?**

By default, Javascript files will interrupt parsing of HTML document in order for them to be fetched and executed.

The async attribute is used to indicate browser that script file can be executed asynchronously.

<script async src=”somescript.js”>

The HTML parser does not have pause at the point it reaches the script tag to fetch and execute, the execution can occur whenever the script becomes ready after being fetched in parallel with document parsing.

**What is difference between attribute and property?**

**Attributes:** Provide more details on an element like id, type, value etc.

**Property:** Value assigned to the property like type=”text”, value=’Name’ etc.

**107.**   **What is the difference between innerHTML & innerText?**

**HTML vs. Plain Text**: innerHTML handles HTML content, including tags, while innerText deals with plain text, stripping out any HTML tags.

**Content Rendering**: innerHTML can modify the document's structure by adding new HTML elements, whereas innerText only changes the text content of existing elements.

**Security**: innerText is typically safer for user-generated content, as it does not execute HTML, reducing the risk of XSS vulnerabilities.

**108.**   **What is the difference between textContent & innerText?**

**Visibility Consideration**: textContent includes all text, even hidden or non-visible text, while innerText only includes visible text.

**CSS and Layout Impact**: innerText respects CSS styles and layout, making it sensitive to visual formatting, while textContent ignores CSS and layout.

**Performance**: textContent is generally faster and does not trigger reflows, whereas innerText can cause reflows and has more performance overhead due to its consideration of layout and styles.

**109.**   **What is HTMLCollection?**

The HTMLCollection interface represents a generic collection of elements (in document order) and offers methods & properties for selecting from the list.

HTMLCollection has length property which returns the number of items in the collection.

It is not possible to iterate over HTMLCollection list using forEach by default.

**110.**   **What is NodeList?**

NodeList objects are collections of nodes which are usually returned by properties such as Node.childNodes and functions such as document.querySelectorAll().

NodeList has length property which returns the number of nodes in nodelist. for...of loops will loop over NodeList objects accurately.

**111.**   **What are frames?**

Frame divides page into section and in each section different page can be displayed.

**112.**   **What is cookie?**

The Cookies are small items of data that consists of name and value pair. Cookies are stored on your computer so that it can be accessed by your web browser.

A web browser and server communicate through HTTP which is stateless protocol. Stateless protocol treats each request independently, so server does not keep data after sending it to browser. With cookies such data can be fetched directly from stored cookie file instead of communicating with server.

For example when user visits web page, user name can be stored in cookie. Now when next time user visits the page cookie belonging to the page is added to the request. This way server gets necessary data “remembered” by cookie.

**113.**   **How cookie helps client server HTTP communication?**

When a user sends a request to the server, then each of that request is treated as a new request sent by the different user.

When receiving an HTTP request, a server can possibly send a Set- Cookie header with the response.

Now, whenever a user sends a request to the server, the cookie is added with that request automatically. Due to the cookie, the server recognizes the users

**Where are cookies stored?**

During browsing session browser stored cookies in memory, at the time of quitting they go to file called as cookies.txt.

Different browser store cookies file in a different location on disk.

For instance, on windows chrome stores the cookies in below location, C:\Users\<YourUser>\AppData\Local\Google\Chrome\User Data\Default\

As cookie expires it is no longer saved on hard drive.

**115.**   **Where are parameters of cookie?**

There are six parameters of cookie: name, value, expires, path, domain and security.

The name and value are required whereas all other parameters are optional. document.cookie=”name=VALUE;expires=DATE;path=PATH;domain=DO

**Name and Value** : The first part of cookie must have name and value. The entire name/value must be a single string with no commas, semicolons or whitespace charactors.

**Expires** : The cookie will disappear when user exits the browser, to give more life to the cookies you must set an expiration date in the following format.

DD-Mon-YY HH:MM:SS GMT

**Path** : Usually the path is set to root level directory (‘/’), which means the cookie is available for all the pages of your site. If you want the cookie to be readable in specific directory <directoryname>, path should be specified as path=/< directoryname>.

**Domain** : Some websites have lots of domains. The purpose of the ‘domain’

is to allow cookies to other subdomains. In case, if website is http://www.

<domain>.com with subdomains http://www.<subdomainone>.

<domain>.com and http://www.<subdomaintwo>.<domain>.com. If web page on subdomainone set a cookie pages on subdomaintwo cannot read that cookie. But if you add domain=<domain> then all subdomains ending with

<domain> can read the cookie.

**Secure** : The last parameter of cookie is secure which is a Boolean value. Its default value is false. If cookie is marked as secure then cookie will be sent to web server and try to retrieve it using secure communication channel.

**116.**   **Can user disable cookies?**

Yes, user can disable cookies from browser.

In chrome, you can go to settings-->Advanced Settings-->Privacy and security-->Content setting-->Cookies and disable cookies.

**117.**   **How to create cookie?**

When visitor visits web page for first time he enters his or her name. This name will be stored in cookies as below,

function createCookie(username, value) {

document.cookie=username + ”=” + value;

}

**118.**   **How to read cookie?**

Javascript cookies can be read like this, var x = document.cookie;

**119.**   **How to delete cookie?**

While deleting cookie you don’t have to specify value.

Javascript cookies can be deleted by specifying expires parameter to a past date.

document.cookie=”username=; expires=Thu, 01 Jan 1970 00:00:00 UTC;

path= /; ”;

Some browsers will not let you delete cookie if you don’t specify the path.

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

**Local Storage** : For every HTTP request, the data is not sent back to the

server (HTML, images, JavaScript, CSS, etc) reducing the total traffic between client and server. Data will stay until it is manually cleared using settings or through program.

**Session Storage** : It is similar to local storage; the only difference is data stored in local storage has no expiration time whereas data stored in session storage gets cleared when the page session ends. Session Storage will cleared when the browser is closed.

**121.**   **What is form validation?**

Form validation verifies whether all fields in form are filled according to required format.

If data entered by user is not according to format then appropriate error message is displayed to the user.

Forms can be validated using server-side as well as client-side validations. Server-side validation is more secure and required server connection to

validate, whereas client-side validation is quicker and doesn’t require server connection but it is less secure.

Javascript is used for client side validation.

Advantages of client side validation is that, it saves time, reduces load on server and can validate form element even before form is submitted.

**122.**   **What is required attribute?**

The required attribute in HTML element prevents that element being submitted as blank.

e.g.

<input type=”text” name=”employeename” required>

In above case, as long as text field employeename is blank form submission will be prevented.

**123.**   **What is pattern attribute?**

The pattern attribute specifies a regular expression against which elements value is checked.

If element value does not match the regex pattern form submission will be prevented.

e.g

<input type=”text” name=”employeename” pattern=”[A-Za-z]”>

In above case, if employee name contains value which is not alphabet then form submission will be prevented.

**How to validate form using Javascript function?**

Below example shows validation of form using Javascript.

Here, we have created form having name input and then when while saving save validateName function will be called and it will validate if name is blank.

<script>

function validateName() {

var name = document.nameform.name.value;

if(name==undefined || name==””) {

alert(“Kindly enter the name!!!”);

return false;

}

Return true;

}

</script>

<form name=”nameform” method=”post” onsubmit=”return validateform()”>

Name: <input type=”text” name=”name”>

<input type=”submit” name=”save”>

</form>

**125.**   **How to validate email in the form?**

Below function can be used to validate email in the form. function validateEmail(emailField){

var reg = /^([A-Za-z0-9\_\-\.])+\@([A-Za-z0-9\_\-\.])+\.([A-Za-z]

{2,4})$/;

if (reg.test(emailField.value) == false)

{

alert('Invalid Email Address');

return false;

}

return true;

}

**126.**   **How to validate field without submitting form?**

To validate field without submitting form you can use validation function in onblur event of input field.

<input type="text" onblur="validateEmail(this);" />

Here, input field will be available to validateEmail function.

**127.**   **What is .test method?**

The .test() API runs a search for a match between a regex and a string.

The .test() API returns a Boolean(true/false), returns true if test passes and false if it doesn’t.

Using .test() returns no data, so don’t expect any.

**128.**   **What is .match method?**

Using .match() is best when you are expecting data back in test result,

.match() returns an array with matches or simply null if there are none.

With match you won’t just be testing for presence of data, you will also see if data pattern exist and return that data.

**129.**   **How to validate Date?**

function validateDate(dateField) {

var reg = /^([0-9]{2})\/([0-9]{2})\/([0-9]{4})$/

if (reg.test(dateField.value) == false) {

alert(“Invalid Date!!!”);

return false;

}

return true;

}

**130.**   **How to allow number only in input field?**

function validateNumber(numField) {

var reg = /^[0-9]+$/

if (reg.test(numField.value) == false) {

alert(“Invalid Number!!!”);

return false;

}

return true;

}

**131.**   **What is error object?**

When an exception occurs, an object representing the error is constructed and thrown.

The Error constructor creates an error object.

When runtime errors occur, instances of Error objects are thrown.

The Error object can be used as a base object for user-defined exceptions. var error = new Error("error message");

“Error” objects contain two properties, “name” and “message”. The “name” property specifies the type of exception The “message” property provides a more detailed description of the exception.

The “message” gets its value from the string passed to the constructor of exception.

**132.**   **What are different error types in Javascript?**

Below are primary error types in javascript:

**SyntaxError** : Raised when syntax error occurs while parsing the Javascript code.

**RangeError** : Raised when numeric value exceeds allowed range. **EvalError** : Raised when the eval() function is used in an incorrect manner. **ReferenceError** : Raised when an invalid reference is used

**TypeError** : Raised when type of variable is not as expected.

**URIError** : Raised when the encodeURI() or decodeURI() functions are used in an inaccurate manner.

**InternalError** : Raised when internal error in the javascript engine is thrown.

**133.**   **How to handle exceptions in JavaScript?**

JavaScript uses the try...catch...finally statement as well as the throw operator to handle exceptions. You can catch user-defined and runtime exceptions, but you cannot catch JavaScript syntax errors.

**134.**   **Explain try…catch…finally.**

**Try** : wraps suspicious code that may throw an error in try block.

**Catch** : Write code to do something when error occurs in catch block. The catch block can have parameters which will give you error information. Usually, catch block is used to log an error or display specific messages to the user.

**Finally** : code in finally block will always be executed regardless of the occurrence of an error. The finally block is usually used to complete the remaining task or reset variables that might have changed before error occurred in try block.

**135.**   **How to throw exceptions programmatically?**

It is possible to throw exceptions programmatically using “throw” statement. There is no restriction on data type that can be thrown as an exception.

throw “error occurred!!”;

throw new SyntaxError(“syntax error occurred!!”);

**What is debugging?**

Debugging is the process of detecting and fixing existing and potential errors in software code that can cause it to behave unexpectedly.

To debug a program, programmer has to start with problem, identify source of the problem and then fix it.

Sometimes it takes more time debugging a program than coding it.

**137.**   **What is debugger keyword?**

Debugger statement stops the execution of Javascript. If debugging functionality is not available, this statement has no effect.

Debugger keyword is like breakpoint in script source code. e.g.

function someErroraniousFunction() {

debugger;

code;

}

**138.**   **What is console object?**

Console object provides access to browsers debugging console.

If browser supports debugging you can use console.log() method to display required text in debugging window.

Console object provides methods like debug(), log(), error(), info(), trace(), warn() which are useful for code debugging.

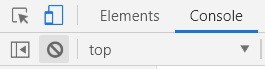
**139.**   **How to activate debugging in browser?**

You can activate debugging in browser by pressing F12 and then select console in debugger menu.

**How to get mobile devices view of webpage in desktop browser?**

In browser press F12,

Then click on toggle device toolbar,



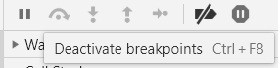
Then, select device for which you want webpage view.



**141.**   **How to deactivate breakpoint in browser?**

This can be done by clicking on “deactivate breakpoints” icon in “Sources”

tab of browser developer tool.



**142.**   **How to pause script execution?**

This can be done by clicking on “pause script execution” icon in “Sources”

tab of browser developer tool.



**143.**   **How to execute function line by line while debugging?**

This can be done by clicking on “Step into next function call” icon in

“Sources” tab of browser developer tool.



**144.**   **How to execute function without stepping into it while debugging?**

This can be done by clicking on “Step over next function call” icon in

“Sources” tab of browser developer tool.



**145.**   **What is code smell?**

In computer programming, a code smell is any characteristic in the source code of a program which possibly indicates a deeper problem.

Determining what is a code smell and what is not a code smell is subjective, and varies by language, developer, and development methodology.

The two main know open source tools used for JavaScript code analysis are JSLint and JSHint, the second being a fork of the first one. There are however many different tools that try to achieve the same goal and you might find something more suited to your own needs

**146.**   **What is AJAX?**

AJAX stands for Asynchronous Javascript and XML. It is collection of related technologies like Javascript, XML, JSON, HTML and XMLHttpRequest etc.

AJAX allows you to send and receive data asynchronously without reloading web page and hence makes web pages more fast and interactive.

**147.**   **What is difference between GET and POST?**

| **Feature** | **GET** | **POST** |
| --- | --- | --- |
| **Data Parameters** | Sends data in the URL (query string). | Sends data in the request body. |
| **Visibility** | Data is visible in the URL (query string). | Data is not visible in the URL. |
| **Security** | Less secure as data is visible in URL. | More secure as data is not visible in URL. |
| **Data Length** | Limited by browser/server (URL length). | Can send larger amounts of data. |
| **Caching** | Can be cached. | Not cached (except by explicit directive). |
| **Idempotent** | Yes (retrieving data should be idempotent). | No (usually used for operations that change data). |
| **Back/Forward Button Safety** | Safe (retrieving data). | Unsafe (may re-submit data). |
| **Bookmarked** | Can be bookmarked and cached. | Should not be bookmarked. |
| **Example** | Fetching search results. | Submitting a form (login, registration). |

* **Data Parameters**: GET sends data as part of the URL query string (?key=value&key2=value2), visible in the browser's address bar. POST sends data in the request body, which is not visible in the URL.
* **Visibility**: GET parameters are visible and can be bookmarked, while POST parameters are not visible and should not be bookmarked.
* **Security**: POST is generally more secure for sensitive data as it doesn't expose it in the URL.
* **Data Length**: GET is limited by browser and server restrictions on URL length, while POST can handle larger data payloads.
* **Caching**: GET requests can be cached by browsers and intermediaries (unless specified otherwise), while POST requests are not cached by default.
* **Idempotent**: GET requests are typically idempotent (retrieving data multiple times has the same effect), whereas POST requests are often used for actions that modify data.
* **Back/Forward Button Safety**: GET requests are generally safe for navigation as they don't modify data, whereas POST may cause re-submission issues if the user navigates back and resubmits the form data.
* **Bookmarking**: GET requests can be bookmarked since they are idempotent and retrieve data, whereas POST requests should not be bookmarked because they might re-submit data.

**148.**   **What is XMLHttpRequest object?**

XMLHttpRequest object is used for asynchronous communication between client and server.

It provides methods like open(), send(), setRequestHeader() for exchanging data between client and server.

**How to make HTTP POST call using AJAX?**

To make HTTP call in AJAX, you first need to initialize a new XMLHttpRequest() object.

Specify URL endpoint, HTTP method (GET) to open() method of XMLHttpRequest() object.

Then call send() method to hit the request.

Receive the response using XMLHttpRequest.onreadystatechange property.

const xmlHttpRequest = new XMLHttpRequest(); xmlHttpRequest.open(“GET”,”http://some.domain.com/method”); xmlHttpRequest.send(); xmlHttpRequest.onreadystatechange=(e)=>{

console.log(xmlHttpRequest.responceText);

}

**150.**   **How to make HTTP POST call using AJAX?**

To make HTTP call in AJAX, you first need to initialize a new

XMLHttpRequest() object.

Specify URL endpoint, HTTP method (POST) to open() method of

XMLHttpRequest() object.

Then call send() and pass data to send() method to hit the request.

Receive the response using XMLHttpRequest.onreadystatechange property. e.g.

const xmlHttpRequest = new XMLHttpRequest();

xmlHttpRequest.open(“POST”,”http://some.domain.com/method”); xmlHttpRequest.send(“fname=Pratik&lname=Bandal”); xmlHttpRequest.onreadystatechange=(e)=>{

console.log(xmlHttpRequest.responceText);

}

**What are HTTP status code?**

HTTP status code are the standard response code given by web site servers. These codes help identify the cause of problem when web page or other resource does not load properly.

**4xx Client Error:**

This category of HTTP status code includes those where request for a web page or other resource contains bad syntax or cannot be filled for some other reason, presumably due to fault of client.

Some common client error HTTP status codes are 404 (Not Found), 403 (Forbidden) and 400 (Bad request).

**5xx Client Error:**

This category of HTTP status code include those where the request for a web page or other resource is understood by the websites server but is incapable of filling it for some reason.

Some common server error HTTP status codes are 500 (Internal server error),

503 (Service Unavailable) and 502 (Bad Gateway).

There are also 1xx, 2xx and 3xx code that are informational, confirm success or dictate redirection which are not errors, so you shouldn’t be alerted about them.