**JAVASCRIPT**

1. **Defination :** JavaScript is a high-level, versatile, and widely-used programming language primarily used for creating interactive and dynamic content in web development.

It is executed in web browsers, enabling developers to manipulate and modify web page elements in real-time, respond to user interactions, and create engaging and interactive user interfaces.

* In HTML, JavaScript code is inserted between <script> and </script> tags.
* JavaScript was invented by **Brendan Eich** in **1995.**

1. **What is EcmaScript :**

* ECMAScript is a standard on which JavaScript is based!
* It was created to ensure that different documents on JavaScript are actually talking about the same language.
* JavaScript and ECMAScript can almost always be used interchangeably. JavaScript is very literal in what it allows.

1. **Comments :**

* Single Line Comment : \\
* Multi Line Comment : /\* code \*/

1. **Variables :** a variable is a symbolic name or identifier used to store data or values in computer memory for later reference or manipulation.

**OR** Variables are the container used to store the values.

**Types :**

|  |  |
| --- | --- |
| * Automatically * Using Var * Using Let * Using Const | **Examples :**   1. Var a = “Piyush” 2. Let b = “Shreyash” 3. Const c = “Herik” |

1. **Variable Scope :**

|  |  |
| --- | --- |
| * **Var :** var is globally scoped var can be **updated** & **re-declared** within its scope | **Example :**  var a = "Piyush";  {      console.log(a);  }  a = 355555;  console.log(a);  **Output :** Piyush  355555 |
| * **Let :** let cannot be Redeclared, let have Block Scope. | **Example :**  let b = “Piyush”  {      let b = 1000;      console.log(b);  }  console.log(b);  **Output :** 1000  Piyush |
| * **Const :** const cannot be Redeclared , const cannot be Reassigned , const have Block Scope. | **Example :**  {      const b = 1000;      console.log(b);      b = "Piyush"; 🡨 this will show error      console.log(b);  } |

1. **Operators :** There are different types of JavaScript operators:

|  |  |
| --- | --- |
| **Operators -** | **Examples -** |
| 1. **Arithmetic Operators** | **+ - \* % / ++ --** |
| 1. **Assignment Operators** | **= += -= \*= %= \*\*=** |
| 1. **Comparison Operators** | **== === != > < >= <= ?** |
| 1. **String Operators** | **+ (concatenate strings)** |
| 1. **Logical Operators** | **&& (and) || (or) ! (not)** |
| 1. **Bitwise Operators** | **& (and) | (or) ! (not)** |
| 1. **Ternary Operators** | **?** |
| 1. **Type Operators** |  |

1. **Data Types :**

**Typeof ( )** 🡪 The typeof operator in JavaScript is used to determine the data type of a value or expression.

|  |  |
| --- | --- |
| JavaScript has 8 Datatypes  1. String 2. Number 3. Bigint 4. Boolean 5. Undefined 6. Null 7. Symbol 8. Object { key : value } | The Object Datatype  The object data type can contain:  1. An object 2. An array 3. A date |

**TypeCasting :** Type casting in JavaScript refers to the process of converting a value from one data type to another.

|  |  |
| --- | --- |
|  | **Examples -** |
| * **parseInt( )** * **parseFloat( )** * **String( )** * **Number( )** * **Boolean( )** | a = 10.999999999  a = parseInt(a)  a = 10.999999999  b = parseFloat(a)  c = String(b)  d = Number(c)  e = Boolean(d)  console.log(a,b,typeof(c),typeof(d),typeof(e));  **Output :** 10.999999999 10.999999999 string number boolean |

**Object :** Object are key : value pair , Objects are variables too. But objects can contain many values.

|  |
| --- |
| const obj = {      "key":"value",  }  console.log(obj.key);  console.log(obj["key"]);  if key is not present it will return undefinied. |

1. **Conditional Expressions :** Conditional statements are used to perform different actions based on different conditions.

|  |  |
| --- | --- |
| * **If – else**   if (condition) {    //  block of code to be executed if the condition is true }  else {  // block of code to be executed if the condition is false  } | * **Switch**   const day = 2;  switch(day){      case 1: console.log("Monday"); break;      case 2: console.log("Tuesday"); break;  case 3: console.log("Wednesday"); break;      case 4: console.log("Thrusday"); break;      case 5: console.log("Friday"); break;      case 6: console.log("Saturday"); break;      case 7: console.log("Sunday"); break;      default : console.log("Invalid Input");  }  **o/p :** Tuesday |

1. **Loops :**

|  |  |
| --- | --- |
| **Syntax -** | **Example -** |
| * **For Loop :** Loop a block of code no of times. | for (var i = 0; i < 5; i++) {      console.log(i);  } |
| * **For in :** loops through the keys of an object. | const obj = {      "fname":"Piyush",      "lname":"Thaware",      "age":25  }  for (item in obj)  {      console.log(item);  }  **Output :** fname  Lname  age |
| * **For Of :** Loops through the values of the object. for...of loop is used to iterate over iterable objects such as arrays, strings, maps, sets, etc. | let str = "Piyush"  for (let b of str) {      console.log(b);  } |
| * **While Loop :** The while loop loops through a block of code as long as a specified condition is true. | While(condition)  {  // code  Increment the condition values like a = a + 1  } |
| * **Do-while Loop :** It must be executed atleast one time | do  {  // code  Increment the condition values like a = a + 1  } while( condition ) |

1. **Function :** A JavaScript function is a block of code designed to perform a particular task.They are reusable.

|  |  |  |
| --- | --- | --- |
| * **Function without argument** | * **Passing arguments** | * **Create Function** |
| function myfunc()  {      console.log("Hello");  }  myfunc(); | function myfunc(name)  {      console.log("Hello",name);  }  myfunc("Piyush"); | Function myFunc() { // code }  **OR : Using Arrow Keywords**  Const myFunc = ( ) => {  // code  } |

1. **Introduction To String :** String are the sequence of character.string are storing and manipulating text.
   * **Single Quotes String ‘ Piyush ‘**
   * **Double Quotes String “Piyush “**
   * **Template Literals ` Piyush `**

**String Methods :**

|  |  |
| --- | --- |
| 1. String.length |  |
| 1. String.toUpperCase | 1. String.toLowerCase |
| 1. String.startsWith(value) | 1. String.endsWith(value) |
| 1. String.replace(old Value,new Value) | 1. String.slice(start,ends) |
| 1. String.concat() | 1. String.charAt() |
| 1. String.split() | 1. String.trim() |

**String Search Methods :**

|  |  |
| --- | --- |
| **Method -** | **Description -** |
| 1. IndexOf | The indexOf() method returns the index (position) the first occurrence of a string in a string  **String.indexOf(value)** |
| 1. lastIndexOf | The lastIndexOf() method returns the index of the last occurrence of a specified text in a string.  **String.lastIndexOf(value)** |
| 1. match | The match() method returns an array containing the results of matching a string against a string (or a regular expression).  let name="Piyush"  console.log(name.match("us"));  **Output :** [ 'us', index: 3, input: 'Piyush', groups: undefined ] |
| 1. includes | The includes() method returns true if a string contains a specified value.  let name="Piyush"  subname = "yu";  console.log(name.includes(subname)); |

1. **Array :** An array is a collection of items of same data type stored at contiguous memory locations.

Example : let arr = [1,2,3,4,false,undefined,"Piyush"]

Arr[0] 🡪 1

**Array Methods :**

|  |  |
| --- | --- |
| **Syntax -** | **Example -** |
| * **Array to string** | let arr = [1,2,3,4,5]  console.log(arr.toString()); 🡪 1,2,3,4,5 |
| * **Length of array** | Arr.length |
| * **Joining items of an array** | Arr.join(symbol)  Console.log(arr.join(“-”)) 🡪 1-2-3-4-5 |
| * **pop items from an array** | arr.pop(); |
| * **add item to array** | arr.push(2000) |
| * **Remove first element from array** | arr.shift() |
| * **add or insert item to an array** | arr.unshift(100) |
| * **Array elements can be deleted using the delete operator.** | delete arr[value] |
| * **sort() method is used to sort an array alphabetically.** | Arr.sort() |
| * **Reverses the elements in the source array.** | Arr.reverse() |
| * **splice can be used to add new items to an array.** | const numbers = [1,2,3,4,5]  numbers.splice(2,1,23,24)  console.log(numbers);  //here, 2 is the position to add, 1 depicts the no. of elements to remove, and 23, 24 are the elements of be added.  **Output : [ 1, 2, 23, 24, 4, 5 ]** |
| * **slice() – slices out a piece from an array. It creates a new array.** | const num = [1,2,3,4]  console.log(num.slice(2));  console.log(num.slice(1,3));  [ 3, 4 ]  [ 2, 3 ] |

**Array Loops :**

|  |  |
| --- | --- |
| **Syntax -** | **Example -** |
| 1. **For Each Loop :** calls a function, once for each array element. | let num = [1,2,3,4]  num.forEach((ele)=>{      console.log(ele\*ele);  })  **Output :** 1 4 9 16 |
| 1. **Array.from** : It used to create an array from any other object. | let name = "Piyush"  let arr = Array.from(name)  console.log(arr);  **Output :** [ 'P', 'i', 'y', 'u', 's', 'h' ] |
| 1. **For Of :** for-of loop can be used to get the values from an array. | let num = ["Piyush","Herik","Json","Shreyash"]  for (let ele of num)  {      console.log(ele);  }  **Output :** Piyush  Herik  Json  Shreyash |
| 1. **For in :** for-in loop can be used to get the keys from an array. | let num = ["Piyush","Herik","Json","Shreyash"]  for (let ele in num)  {      console.log(ele);  }  **Output :** 0 1 2 3 |

1. **High Ordered Functions :**

|  |  |
| --- | --- |
| **Functions** | **Example :** |
| 1. **Map :**   The map function is used to create a new array by applying a specified function to each element in an existing array.  It doesn't modify the original array but returns a new one with the results of applying the provided function to each element. | let num = [1,2,3,4,5]  let mapping = **num.map(function(item){**  **return item\*item;**  **})**  console.log(mapping);  Output : [ 1, 4, 9, 16, 25 ] |
| 1. **Filter :**   The filter function is used to create a new array with all elements that pass a test specified by a provided function.  It returns a new array containing only the elements for which the provided function returns true. | let num = [1,2,3,4,5]  let filtering = **num.filter(function(item){**  **return item>3**  **})**  console.log(filtering);  **Output :** [ 4, 5 ] |
| 1. **Reduce :**   The reduce function is used to reduce an array to a single value by applying a specified function to each element of the array.  It accumulates the results of the callback function into an accumulator (also known as a reducer) and returns the final accumulated result. | let num = [1, 2, 3, 4, 5]  let reducing = **num.reduce(function (accumulator, currentValue) {**  **return (accumulator + currentValue)**  **}, 0)**  console.log(reducing);  **Output :** 15 |

1. **Console Objects :** In JavaScript, the console object is a built-in global object that provides methods for interacting with the browser's console or the console of a JavaScript runtime environment.

|  |  |
| --- | --- |
| 1. **console.log():** This method is used to print messages or values to the console. | console.log("Hello, world!"); |
| 1. **console.error():** This method is used to log error messages to the console. | console.error("This is an error message."); |
| 1. **console.warn():** This method is used to log warning messages to the console. | console.warn("This is a warning message."); |
| 1. **console.info():** This method is used to log informational messages to the console. It is similar to console.log() but can be used to convey that the message contains additional information. | console.info("This is an informational message."); |
| 1. **console.debug():** This method is used to log debugging information to the console. | console.debug("Debugging information goes here."); |
| 1. **console.clear():** This method is used to clear the console, removing all previously logged messages. | console.clear(); |
| 1. **console.assert():** This method is used to test a condition and log an error message if the condition is false. It's a way to perform assertions during debugging. | console.assert(1 === 2, "This assertion failed!"); |
| 1. **console.table():** This method is used to display tabular data in a table format in the console. It is particularly useful for visualizing arrays or objects. | const data = [{ name: "Alice", age: 30 }, { name: "Bob", age: 25 }];  console.table(data); |

1. **Alert , Prompt & Confirm :**

|  |  |
| --- | --- |
| 1. **Alert() :** The alert() function displays a simple dialog box with a message and an "OK" button. | Alert(“Message you want to display”); |
| 1. **Prompt() :** The prompt() function displays a dialog box with a message, an input field where the user can enter text, and "OK" and "Cancel" buttons. It is used to gather input from the user. | Prompt(“Please Enter the text”); |
| 1. **Confirm()** : The confirm() function displays a dialog box with a message and "OK" and "Cancel" buttons. | confirm("Are you sure you want to delete this item?"); |

1. **DOM Manupulation :**

|  |  |
| --- | --- |
| 1. **Element Node :** | Element nodes represent the HTML or XML elements in the document.  Element nodes can have attributes and may contain other elements (nested elements) as well as text content.  Ex : div,a ,p ,h1 |
| 1. **Text Node :** | Text nodes represent the textual content within an element. They contain the actual text that is displayed on the web page.  Ex : text between tags like <p>I am Piyush</p>  I am piyush 🡪Is text node |
| 1. **Comment Node :** | Comment nodes represent HTML or XML comments in the document. Comments are not visible to the user and are used for documentation or adding notes within the markup. |

1. **Accessing Dom Parents, Childs & Siblings :**

Const Parent = document.getElementById(“ParentElement”);

|  |  |
| --- | --- |
| * **FirstChild :** The firstChild property returns the first child node of a specified node. | Const first = parent.firstChild; // return a text node  Const firstElement = parent.firstChildElement; // return element |
| * **LastChild :** The lastChild property returns the last child node of a specified node. | Const last = parent.lastChild; // return a text node  Const lastElement = parent.lastChildElement; // return element |
| * **nextSibling :** The nextSibling property returns the next sibling node of a specified node. | const nextSibling = element.nextSibling;  const nextElementSibling = element.nextElementSibling; |
| * **previousSibling :** The previousSibling property returns the previous sibling node of a specified node. | const previousSibling = element.previousSibling;  const previousElementSibling = element.previousElementSibling; |
| * **childNodes :** The childNodes property returns a live NodeList that contains all child nodes of a specified node. | const children = parent.childNodes; |
| * **children :** The children property returns a live HTMLCollection that contains only the child element nodes of a specified node. | const childElements = parent.children; |

1. **Searching the DOM :**

|  |  |
| --- | --- |
| * **getElementById():**This method retrieves an element with a specific id attribute. | const element = document.getElementById('elementId'); |
| * **getElementsByClassName():**This method retrieves a collection (HTMLCollection) of elements with a specific class name. | const elements = document.getElementsByClassName('className'); |
| * **getElementsByTagName():**This method retrieves a collection of elements with a specific HTML tag name. | const elements = document.getElementsByTagName('tagName'); |
| * **querySelector():**This method retrieves the first element that matches a specified CSS selector. | const element = document.querySelector('CSS selector'); |
| * **querySelectorAll():**This method retrieves a NodeList of all elements that match a specified CSS selector. | const elements = document.querySelectorAll('CSS selector'); |
| * **getElementByName() (for form elements):**This method retrieves a collection of form elements with a specific name attribute. | const elements = document.getElementsByName('inputName'); |
| * **querySelector() for searching within a specific element:**You can also use querySelector() and querySelectorAll() on a specific element to search for elements within that element's subtree. | const parentElement = document.getElementById('parentElement');  const childElement = parentElement.querySelector('.childClassName'); |



|  |  |
| --- | --- |
| 1. **match() Method :**   The match() method is used to check if an element matches a specific CSS selector. It returns true if the element matches the selector; otherwise, it returns false. | <div class="box" id="first"> First Child<span id="span1" class="temp">This is spane</span></div>  console.log("class present = ", f.matches(".box")); // True  console.log("class present = ", f.matches(".fun")); // False |
| 1. **closest() Method:**   The closest() method is used to find the closest ancestor (parent or any ancestor) of an element that matches a specified CSS selector. It returns the closest matching ancestor or null if no match is found. | const element = document.getElementById('myElement');  const closestParent = element.closest('.parentClass'); |
| 1. **contains() Method:**   The contains() method is used to check if an element contains another element as its descendant. It returns true if the specified element is a descendant of the calling element; otherwise, it returns false. | const parentElement = document.getElementById('parentElement');  const childElement = document.getElementById('childElement');  if (parentElement.contains(childElement)) {  // parentElement contains childElement  } else {  // parentElement does not contain childElement  } |

1. **InnerHTML 🡪** The innerHTML property allows to get the HTML inside the element as a string. (valid for element nodes only)
2. **OuterHTML** 🡪 The outerHTML property contains the full HTML, innerHTML + the element itself.

InnerHTML is valid only for element nodes. For other node types we can use nodeValue or data.

1. **HTML Attributes :**

|  |
| --- |
| <div class="box" id="first">First Child</div>  let first = document.getElementById("first"); |

|  |  |
| --- | --- |
| 1. **hasAttribute() :** The hasAttribute method can be used to check if a specific attribute exists on an element. It returns true if the attribute is present and false if it's not. | first.hasAttribute("class")  **Output :** true |
| 1. **getAttribute() :** You can use the getAttribute method to both check if an attribute exists and retrieve its value. If the attribute doesn't exist, it will return null. | first.getAttribute("class")  **Output :** box (if present otherwise null) |
| 1. **setAttribute() :** you can use the setAttribute method to set or update attributes on an HTML element. | first.setAttribute("class","attribite setting") |
| 1. **removeAttribute() :** This method is used to remove attribute from the element | first.removeAttribute("class") |
| 1. **element.attributes() :** This method returns the collection of all the attributes. | first.attributes  **Output :** NamedNodeMap {0: id, id: id, length: 1} |
| 1. **toggleAttribute() :** This method check if the attribute present or not if it present then it will remove it otherwise it will add to the element |  |

1. **Insertion Methods :** 
   * **CreateElement :** In JavaScript, you can use the document.createElement() method to create new HTML elements dynamically. This method allows you to create an element in memory, which you can then modify and add to the DOM (Document Object Model).

**Example :** let div = document.createElement('div');

|  |  |
| --- | --- |
| **Usage -** | **Example -** |
| 1. **appendChild(element)** 🡪 AppendChild is a method that allows you to add an element as the last child of another element. | parentElement**.appendChild(newElement);** |
| 1. **prepend(element) 🡪** prepend is a method that allows you to add an element as the first child of another element. | parentElement**.prepend(newElement);** |
| 1. **after(element) 🡪** after is a method that allows you to insert an element immediately after another element in the same parent. | referenceElement**.after(newElement);** |
| 1. **replaceWith(new element) 🡪** replaceWith is a method that allows you to replace an element with another element in the DOM. | oldElement**.replaceWith(newElement);** |

1. **InsertAdjacentHTML :** insertAdjacentHTML allows you to insert new HTML as a string at a specified position relative to an element. It parses the provided HTML and inserts it into the DOM.

**Syntax** 🡪 element.insertAdjacentHTML(position, text);

|  |  |
| --- | --- |
| 1. **Beforebegin :** Inserts the HTML before the element itself (as a previous sibling). 2. **afterbegin:** Inserts the HTML as the first child of the element. 3. **beforeend:** Inserts the HTML as the last child of the element. 4. **afterend:** Inserts the HTML after the element itself (as a next sibling). | **Example :**  <div id="first">Hello I am first</div>  <script>          let ele = document.getElementById("first")          ele.insertAdjacentHTML("beforebegin","<h4>Before Begin</h4>")          ele.insertAdjacentHTML("beforeend","<h4>Before End</h4>")          ele.insertAdjacentHTML("afterbegin","<h4>After Begin</h4>")          ele.insertAdjacentHTML("afterend","<h4>After End</h4>")  </script>  **Output :**  Before Begin  After Begin  **Hello I am first**  Before End  After End |

1. **Changing HTML Classes using Javascript :** You can change HTML classes using JavaScript by manipulating the classList property of an HTML element.

|  |  |
| --- | --- |
| **ClassName :** Using classname you can add or remove classes | let f = document.getElementById('first');  f.className = "blue text-white"; |
| **ClassList :** It is a read-only property that returns a DOMTokenList, which is a collection of the element's class attributes as individual tokens. The classList property includes methods for adding, removing, toggling, and checking the presence of CSS classes on an element.  **Element.classList** | **classList.remove(className) :**  To remove a class from an element, you can use the classList.remove() method. |
| **classList.add(className) :**  To add a class to an element, you can use the classList.add() method. | **classList.toggle(className) :**  To toggle (add or remove) a class on an element, you can use the classList.toggle() method |

1. **SetInterval & SetTimeOut :**

|  |  |
| --- | --- |
| **SetInterval –**  setInterval is used to repeatedly execute a specified function or code block at a specified time interval (in milliseconds).  **Syntax :** **setInterval(function, interval);**  **Example :**  Const timer = setInterval(()=>{      console.log("SetInterval is called");  },2000)  **ClearInterval –** clearInterval function is called to stop it.  clearInterval(interval Id)  interval ID : timer | **SetTimeOut –**  setTimeout is used to execute a specified function or code block after a specified delay (in milliseconds).  **Syntax :**  **setTimeout(function, delay);**  **Example:**  Const timer = setTimeout(()=>{      console.log("SetTimeout is called");  },2000)  **ClearTimeOut –** It is used to cancel the execution of a function scheduled by setTimeout.  clearTimeout(timeout Id);  timeout ID : timer |

1. **Browser Events :** Browsing events, also known as DOM (Document Object Model) events or browser events, are a fundamental part of web development.

|  |  |
| --- | --- |
| * + Onmouseenter(task)   + onMouseLeaves(task)   + onClick(task or function)   + onChange(task or function) | * + onSubmit(task)   + ondrag(task) |

1. **Handling Browser Events :**

|  |  |
| --- | --- |
| 1. element.addEventListener(eventType, eventListenerFunction); 2. element.removeEventListener(eventType, eventListenerFunction); | **Element.addEventListener(“click”,function)** |

1. **Callback Function :** 
   * In JavaScript, a callback function is a function that is passed as an argument to another function and is typically executed after the completion of that function or at a specified time or event.
   * Callbacks are a fundamental concept in JavaScript and are often used in asynchronous programming to handle tasks that may take some time to complete, such as reading a file, making an HTTP request, or responding to user interactions.

|  |  |
| --- | --- |
| **function onLoading(callback) {**  **console.log("Loading Scripts...");**  **setTimeout(() => {**  **fetch("https://fakestoreapi.com/products")**  **.then((response) => { return response.json() })**  **.then((data) => console.log("Fetched Data : ", data[0]));**  **// Callback**  **callback();**  **}, 3000);**  **}**  function onSuccess() {      console.log("Data fetch successfully...");  }  **onLoading(onSuccess);** | **Output :**  Loading Scripts...  Data fetch successfully...  Fetched Data : {  id: 1,  title: 'Fjallraven - Foldsack No. 1 Backpack  price: 109.95,  description: 'Your perfect pack for everyday use and  image: 'https://fakestoreapi.com/img/81fPKd-2AYL.\_AC\_SL1500\_.jpg',  rating: { rate: 3.9, count: 120 }  } |

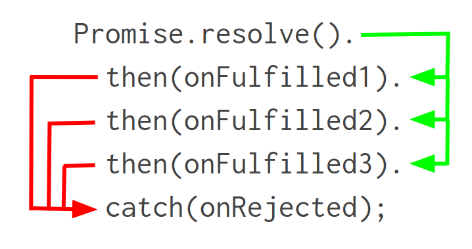
1. **Synchronous Function :** A synchronous function in JavaScript is a function that executes sequentially, one step at a time, and blocks the execution of subsequent code until it has completed its task. In other words, in a synchronous function, each line of code is executed one after the other in a predictable and linear fashion. **OR it is a straightforward execution of code.**
2. **Asynchronous Function :** An asynchronous function in JavaScript doesn't make your entire program wait for a task to finish. It allows your program to keep doing other things while waiting for a potentially time-consuming task, like fetching data from the internet or reading a file, to complete. Once the task is done, the function lets your program know, and you can then handle the result or continue with other tasks.
3. **Callback Hell & Pyramid Of Dom :** Callback Hell is essentially nested callbacks stacked below one another forming a pyramid structure. Every callback depends/waits for the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code.



1. **Promises :**

|  |  |
| --- | --- |
| * In JavaScript, a Promise is an object that represents a future value or the eventual completion (or failure) of an asynchronous operation. * Promises are a way to handle asynchronous code in a more structured and manageable manner. * They were introduced to help mitigate the complexities and callback hell often associated with asynchronous programming. * Promises have 3 states :  1. Pending 2. Fulfilled 3. Rejected | **Example –**  let myPromise = new Promise((resolve, reject) => {      setTimeout(() => {          const success = true;          if (success) {              resolve("Data fetch successfully");          }          else {              reject("Failed to fetch the data ....")          }      })  })  myPromise.then((result) => {      console.log("success", result);  }).catch((error) => {      console.log("Error", error);  }) |

**Promise Chaining :**



**Fetch Data Using Promises : Handling Multiple Promises**

|  |  |
| --- | --- |
| let myPromise = new Promise((resolve, reject) => {      console.log("Loading Scripts...");      setTimeout(() => {          const start = false;          if (start) {              fetch("https://fakestoreapi.com/products").then((response) => response.json()).then((json) => {                  console.log("Fetched Data : ", json[0]);                  resolve("Data is Fetched successfully...")              })          }          else {              reject("Failed to load the script & data");          }      }, 3000)  })  myPromise**.then((result) => {**  **console.log("Result : ", result);**  **}).then(()=>{**  **console.log("Done");**  **}).catch((error) => {**  **console.log("Error : ", error);**  **})** | **Output –**  Loading Scripts...  Fetched Data : {  id: 1,  title: 'Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops',  price: 109.95,  description: 'Your perfect pack for everyday use and walks in the forest. Stash your laptop (up to 15 inches) in the padded sleeve, your everyday',  category: "men's clothing",  image: 'https://fakestoreapi.com/img/81fPKd-2AYL.\_AC\_SL1500\_.jpg',  rating: { rate: 3.9, count: 120 }  }  Result : Data is Fetched successfully... |

1. **Async & Await :** 
   * **Async 🡪** You declare a function as async to indicate that it contains asynchronous code.

An async function always returns a Promise, even if you don't explicitly return one.

**Await inside an async function to pause the execution of the function until a Promise is resolved.**

* + **Await** 🡪 Inside an async function, you can use the await keyword before a Promise to pause the function's execution until that Promise is resolved. This allows you to write asynchronous code in a more sequential manner.

|  |  |
| --- | --- |
| async function fetchProduct() {      const url = "https://fakestoreapi.com/products";      const request = await fetch(url);      const response = await request.json();      console.log("Fetch Data : ", response[0]);  }  fetchProduct(); | **Output :**  Fetch Data : {  id: 1,  title: 'Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops',  price: 109.95,  description: 'Your perfect pack for everyday use and walks in the forest. Stash your laptop (up to 15 inches) in the padded sleeve, your everyday',  category: "men's clothing",  image: 'https://fakestoreapi.com/img/81fPKd-2AYL.\_AC\_SL1500\_.jpg',  rating: { rate: 3.9, count: 120 }  } |

1. **Fetch() :** 
   * In JavaScript, the fetch() function is used to make network requests to fetch resources (such as data or files) from a specified URL.
   * The fetch() function returns a Promise that resolves to the Response object representing the response to the request.

|  |  |
| --- | --- |
| const url = "https://fakestoreapi.com/products";  const fetching = fetch(url)  .then((response) => { return response.json() })  .then((json) => console.log(json[0])) | **Output :**  {  id: 1,  title: 'Fjallraven - Foldsack No. 1 Backpack, Fits 15 Laptops',  price: 109.95,  description: 'Your perfect pack for everyday use and walks in the forest. Stash your laptop (up to 15 inches) in the padded sleeve, your everyday',  category: "men's clothing",  image: 'https://fakestoreapi.com/img/81fPKd-2AYL.\_AC\_SL1500\_.jpg',  rating: { rate: 3.9, count: 120 }  } |

1. **Exception Handling :** In JavaScript, try and catch are used together to handle exceptions and errors in your code.
   * **Try** 🡪 When an error occurs within the try block, it does not immediately stop the execution of your program. Instead, it transfers control to the associated catch block if an error is thrown.
   * **Catch 🡪** The catch block is used to define what should happen when an error or exception occurs inside the try block.

|  |  |
| --- | --- |
| **Syntax –**  try {  // Code that might throw an error  } catch (error) {  // Code to handle the error  } | **Example –**  try{      console.log(b);  }  catch(error)  {      console.log("Error : ",error);  }  Output : Error : ReferenceError: b is not defined |

1. **Error Object :** When something doesn’t work as plan then the error occurs.

|  |  |
| --- | --- |
| console.log(error.name); // "Error"  console.log(error.message); // "This is a custom error message"  console.log(error.stack); // Stack trace (varies by environment)  **Creating Custom Error :** new Error(“Message”);  **Throw :** throw keyword is used to throw new errors . | const age = 14;  try {      if (age < 18) {          throw new Error("Age should be greater than 18 year old");      }      else{          console.log("Age is valid");      }  }  catch (error) {      console.log("Error : ", error);  } |

* + **Finally :** The finally block is often used to ensure that cleanup or finalization tasks.

It will run weather if the exception is handle or not.

**Example :** try{

**// code to be check**

}

Catch(error){

// handle error code

}

Finally{

// code or console.log(“finally block”);

}

1. **Cookies :** Cookies are small pieces of data that can be stored on a user's device (typically in the browser) and are sent back to the server with subsequent requests.

sometime when we login to amazon in web and add some product to cart after closing that app after reopen that app you will see those item are still in JS

storing data to browser is alson known as cookies.

|  |
| --- |
| 1. **Setting Cookies**   document.cookie = “first\_name = Piyush”   1. **Cookie attributes**  * **Adding expire date to cookie**   Document.cookie = “name=Piyush;expires=Thu,24 Mar 2024 12:00:00 UTC”   * **Updating Cookies**   Document.cookie = “**First name = Piyush**”  |  Document.cookie = “**first\_name = Shyreash**”   * **Deleting Cookies**   Document.cookie = “first\_name = ;  expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;“ |

1. **Storage :**
2. **Local Storage 🡪** Data stored in localStorage store even after the browser is closed and can be retrieved on subsequent visits to the same website. It remains stored until explicitly removed by the web application or the user.

|  |
| --- |
| * **Storing data in localStorage :**   localStorage.setItem( “key”,”Value” )  Example 🡪 localStorage.setItem("Name","Piyush");   * **Retrieving data from localStorage :**   localStorage.getItem(“key”)  Example 🡪 console.log(localstorage.getItem(“Name”)); 🡪 if present then it will show name otherwise it will show null.   * **Updating data in localStorage :**   localStorage.setItem(“Existing key”,”new Value”);   * **Remove data from sessionStorage :**   localStorage.removeItem(“key”); |

1. **Session Storage 🡪** Data stored in sessionStorage is only available for the duration of the page session. It is cleared when the page session ends, typically when the user closes the browser tab or navigates away from the page.

|  |
| --- |
| * **Storing data in sessionStorage :**   sessionStorage.setItem(“key”,”value”)  Example : sessionStorage.setItem(“name”,”Piyush”)   * **Retrieving data from sessionStorage :**   sessionStorage.getItem(“key”)  Example 🡪 console.log(sessionStorage.getItem(“Name”)); 🡪 if present then it will show name otherwise it will show null.   * **Updating data in sessionStorage :**   sessionStorage.setItem(“Existing key”,”new Value”);   * **Remove data from sessionStorage :**   sessionStorage.removeItem(“key”)   * **Clear All Key Value Pair :** The sessionStorage.clear() method in JavaScript is used to remove all key-value pairs (data) stored in the session storage for the current page's session   sessionStorage.clear() |

**Length of storage : localStorage.length OR sessionStorage.length**

**Index access of localStorage : localStorage.key(0) Output 🡪 Name**

1. **Prototype 🡪** JavaScript objects have a special property called prototype that is either null or references another object.

You can access using the Object.getPrototypeOf() method or simply as \_\_proto\_\_.

|  |
| --- |
| const myObject = {};  const prototypeOfMyObject = Object.getPrototypeOf(myObject);  console.log(prototypeOfMyObject);  **Output -** [Object: null prototype] {} |

1. **Classes & Objects 🡪**
   * **Classes :** A class is a blueprint for declaring and creating objects.
   * **Objects :** A Object is the instance of class.

|  |
| --- |
| **// Creating Class**  class ExamForm{      // code  }  **// creating ExamForm Object for student-1**  let student1 = new ExamForm();  console.log(student1);  **Output :** ExamForm {}  ============================================================================================  You can access class attribute using **this.attributeName**  **Example –**  class ExamForm {      salary = 10000;      company = "TCS";      showDetails(employee\_name) {          console.log(`Employee Name : ${employee\_name}\t Salary : **${this.salary}\**t Company : **${this.company}**`);      }  }  let student1 = new ExamForm();  **// Printing Class Attributes**  console.log(student1.salary); **// OUTPUT 🡪 10000**  console.log(student1.company); **// OUTPUT 🡪 TCS**  **// Calling Methods of class**  student1.showDetails("Piyush"); |
| * **Modifying Class Attribute :** If class attribute and instance attribute have same name then instance attribute have more priority then class attribute   class ExamForm {      salary = 10000;      showDetails(employee\_name) {          console.log(`Employee Name : ${employee\_name}\t Salary : ${this.salary}`);      }  }  let student1 = new ExamForm();  console.log(student1.salary); **// OUTPUT : 10000**  **// Modifying Class Attributes**  **student1.salary = 400000;**  student1.showDetails("Piyush");  **Output :** Employee Name : Piyush Salary : 400000 |

1. **Constructor :** It run as soon as the object is created

In JavaScript, constructors are special functions that are used to create and initialize objects.

When you create a constructor function, you can use it to create multiple objects with the same structure and behavior.

|  |
| --- |
| class ExamForm {  **// Initializing Constructor**      constructor(first\_name, last\_name, email\_address) {  **this.fname** = first\_name;          this.lname = last\_name;          this.email = email\_address;      }      fillform() {          console.log(`Fill Details : First Name -> **${this.fname}** \t|\t Last Name : ${this.lname} \t|\t Email : ${this.email}`);      }  }  student1 = new ExamForm("Piyush", "Thaware", "Piyush@gmail.com");  student1.fillform();  **Output :** Fill Details : First Name -> Piyush | Last Name : Thaware | Email : Piyush@gmail.com |

1. **Inheritance :** Inheritance is a fundamental concept in object-oriented programming (OOP) that allows you to create new classes (or types) based on existing classes.
   * **Extends :** The extends keyword is used to create a child class of another class (parent).

|  |
| --- |
| class Parent{      age = 35;      company = "TCS";  }  **// Creating Child Class from Parent Class**  **class Child extends Parent** {      age = 15;      currently = "studing";  }  **accessing class attibutes of Child Class**  let c1 = new Child();  console.log(c1.age); **// OUTPUT : 15**  console.log(c1.currently**); // OUTPUT : studing**  **// since we have extended Parent class to child class so the child class can also access Parent Class Attributes**  **console.log(c1.company); // 🡪 Output : TCS**  **accessing class attibutes of Parent Class**  let p1 = new Parent();  console.log(p1.age); **// Output : 35** |

1. **Super :** The super keyword to access and call methods or constructors of a superclass.

**Method Overriding 🡪** Method overriding occurs when a subclass (child class) has the same method as the parent class.

|  |  |
| --- | --- |
| **Super With Methods :** | **Super With Constructor** |
| class Parent{      alpha(x){          console.log("The Value of x is",x);      }  }  class Child extends Parent {      method(number){  **super.alpha(number)**          console.log("The Enter number is : ",number);      }  }  let c1 = new Child();  c1.method(34)  **OUTPUT :**  The Value of x is 34  The Enter number is : 34 | class Parent {  **constructor(fname) {**  **this.fname = fname;**  **console.log("Parent Class Constructor",this.fname);**  **}**  }  class Child extends Parent {      constructor(fname, lname) {  **super(fname)**          this.fname = fname;          this.lname = lname;          console.log("Child Class Constructor",this.fname,this.lname);      }  }  let c1 = new Child("Piyush", "Thaware");  **OUTPUT :**  Parent Class Constructor Piyush  Child Class Constructor Piyush Thaware |

1. **Static Method :** staticMethod is a function that is associated with the MyClass constructor or class itself, rather than with instances of MyClass. You can call it without creating an instance of MyClass.

|  |
| --- |
| class MyClass{  **static** name = "Piyush";  **static method()**  **{**  **console.log("I am the method");**  **}**  }  console.log(MyClass.name);  MyClass.method();  **OUTPUT :**  Piyush  I am the method |

1. **Getter & Setter :** In JavaScript, you can use getter and setter methods to control access to an object's properties. These methods allow you to get and set the values of object properties.
   * **Getter 🡪** Getter methods are used to retrieve the value of a property,
   * **Setter 🡪** Setter methods are used to set or modify the value of a property.

|  |  |
| --- | --- |
| class MyClass {  **get Name() { // Getter**  **return this.name;**  **}**  **set Name(newName) { // Setter**  **this.name = newName;**  **}**      showDetails() {          console.log("My Name is : ", this.name);      }  }  let c = new MyClass();  **c.Name = "Piyush" // Setting the value**  c.showDetails()  **OUTPUT :** My Name is : Piyush | const obj = {      language : "Python",  **get codingLanguage(){**  **return this.language;**  **},**  **set codingLanguage(newLanguage){**  **this.language = newLanguage;**  **}**  }  console.log(obj);  **// Setting New Language**  obj.language = "C++";  console.log(obj);  **OUTPUT :**  { language: 'Python', codingLanguage: [Getter/Setter] }  { language: 'C++', codingLanguage: [Getter/Setter] } |

1. **InstanceOf** **:** The instanceof Operator allows to check whether an object belongs to a certain class. It returns true if obj belongs to the Class or any other class inheriting from it.

|  |
| --- |
| class Parent{  // code  }  class Child extends Parent{  // code  }  let c1 = new Child();  console.log(**c1 instanceof Parent**);  console.log(**c1 instanceof Child**);  let p1 = new Parent();  console.log(**p1 instanceof Parent**);  console.log(**p1 instanceof Child**);  **Output :**  True  True  True  False |

1. **IIFE ( Immediately Invoked Function Expressions ) :** Define and execute a function immediately after its creation.

|  |  |
| --- | --- |
| (      function(){          console.log("Hello My Name is Piyush");      }  )() | **Syntax –**  (function() {  // Your code here  })(); |

1. **Destructuring :**   
   Destructuring is a feature in JavaScript that allows you to extract values from objects or arrays and assign them to variables in a more concise and convenient way.

It makes it easier to work with complex data structures by simplifying the process of accessing their elements.

|  |  |
| --- | --- |
| **Example 1 : Array Destructuring**  let array = [1, 2, 3, 4]  console.log(array[0], array[3]);  **// Instead of above code use destructuring**  **let [ a,b,c,d ] = array;**  console.log(a,b,c,d);  **Output :**  1 4  1 2 3 4 | **Example 2 : Object Destructuring**  const obj = {      firstName: "Piyush",      lastName: "Thaware"  }  console.log(obj.firstName);  **const { firstName: fname, lastName: lname } = obj;**  console.log(fname, lname);  **Output :**  Piyush Thaware |
| **Example 3 : Passing Default Value**  const obj = {      firstName: "Piyush",  }  const { firstName, lname="Thaware" } = obj;  console.log(firstName,lname);  **Output :**  Piyush Thaware | **Example 4 : Ignoring Element**  const colors = ["red","blue","green","yellow","orange"];  **// Ignoring second color from above colors array.**  const [a , , c] = colors;  console.log(a , c);  **Output :** red green |
| **Example 5 :** The ...rest syntax, also known as the rest parameter.  const colors = ["red","blue","green","pink","orange"];  const [a , b **, ...remainigColors** ] = colors;  console.log(a,b,remainigColors);  **Output :** red blue [ 'green', 'yellow', 'orange' ] |

1. **Spread :** The **spread operator (...)** is a feature introduced in JavaScript (ES6) that allows you to spread elements of an iterable (e.g., an array or an object) into another array, object, or function call.

|  |  |  |
| --- | --- | --- |
| **Example 1 : Copying Array**  const array = [1,2,3,4,5]  const newArray = ["Piyush","Herik", **...array**];  console.log(newArray);  **Output :** [ 'Piyush', 'Herik', 1, 2, 3, 4, 5 ] | **Example 2 : Merge Multiple Array’s**  const array1 = [1,2,3,4]  const array2 = ["Piyush","Sahil","Herik"]  **const result = [ ...array1, ...array2 ]**  console.log(result);  **Output :** [ 1, 2, 3, 4, 'Piyush', 'Sahil', 'Herik' ] | **Example 3 : Copying Object**  const originalObject = { name: ‘John’, age: 30 };  **const copiedObject = { …originalObject };**  console.log(copiedObject);  **// Output:** { name: ‘John’, age: 30 } |
| **Example 4 : Spread With Function Call**  function sum(a,b,c){      return a+b+c;  }  const numbers = [1,2,3];  **const result = sum(...numbers);**  console.log(result);  **console.log(...numbers);**  **Output :**  **6**   1. **2 3** |

1. **Scopes :**

|  |
| --- |
| * **Global Scope :** Variables and functions declared outside of any function or block have global scope. They are accessible from anywhere in your JavaScript code, including within functions.   **Example 🡪** let name = "Piyush";  function myFunc(){      console.log("I am inside function",name);  }  console.log(name);  myFunc();  **Output 🡪** Piyush  I am inside function Piyush |
| * **Function Scope :** Variables declared inside a function have function scope. They are only accessible within that function.   **Example 🡪** function myFunc(){  let name = "Piyush";  console.log("I am inside function",name);  }  console.log(name); **🡨 ERROR**  myFunc(); |
| * **Block Scope (ES6 and later) :** Variables declared with let and const within a block (e.g., within loops and conditional statements) have block scope. They are only accessible within that block.   **Example** 🡪  {      let name = "Piyush"  }  console.log(name); **🡨 ERROR** |
| * **Closures :** Closures are a powerful feature of JavaScript that allows functions to maintain access to their enclosing scope's variables even after the outer function has finished executing.   **Example** 🡪  function outerFunction() {      var outerVar = 'I am outer!';  **function innerFunction() {**  **console.log(outerVar); // Accessible via closure**  **};**  **return innerFunction; // Return the inner function**  }  var closureFunc = outerFunction();  closureFunc(); // Outputs: "I am outer!" |

1. **Arrow Function :** Arrow functions are a effective way to write functions in JavaScript.

**Syntax** 🡪 (parameters) => expression

**Example 🡪** const add = **(a, b) => a + b;** console.log(**add(2, 3));** **// Output: 5**