JAVASCRIPT BASICS

* Javascript on client side:
* Js works along with HTML and CSS, HTML adds structure to web page, CSS styles it and Javascript brings it to life by allowing users to interact with elements on the page, such as actions on clicking buttons, filling out forms and showing animations.
* Js on the client side is directly executed in the user’s browser.
* Js primarily used for web page interactivity.
* Applications of javascript:
* Js is used in a wide range of applications, from enhancing websites to building complex applications. Here are some example:
* Web Development: JS adds interactivity and dynamic behavior to static websites, with popular frameworks like **AngularJs** enhancing development.
* Web Applications: JS powers robust web applications, leveraging APIs,React and Electron to create dynamic user experiences like google maps.
* Server Applications**: NodeJs** brings Javascript to the server side, enabling powerful server applications and full-stack development.
* Game Development: LJS combined with HTML5 and libraries like **Ease Js**, enables the creation of interactive games for the web.
* Smartwatches: **Pebble JS** allows javascript to run on smartwatches supporting apps that require internet connectivity.

1. Variables and Datatypes in Javascript:

* Variables and data types are foundational concepts in programming, serving as the building blocks for storing and manipulating information within a program. In JavaScript, getting a good grasp of these concepts is important for writing code that works well and is easy to understand

1. Variables:

* A variable is like a container that holds data that can be reused or updated later in the program. In JavaScript, variables are declared using the keywords[var](https://www.geeksforgeeks.org/javascript-var/), [let](https://www.geeksforgeeks.org/javascript-let/), or [const](https://www.geeksforgeeks.org/javascript-const/).

1. Var keyword:- This is the older way to declare variables. Variables declared with var are function-scoped or globally scoped, meaning their accessibility can be broad, potentially leading to issues in larger codebases.
2. Let keyword:- Introduced in ES6, let provides block-scoped variables. This means a variable declared with let is only accessible within the block of code (e.g., inside an if statement or a for loop) where it's defined. let variables can be reassigned.
3. Const keyword:- Also introduced in ES6, const declares block-scoped constants. Once a value is assigned to a const variable, it cannot be reassigned. This is ideal for values that should not change during the program's execution.
4. Datatypes:

* JavaScript supports various datatypes, which can be broadly categorized into primitive and non-primitive types.

1. Primitive Datatypes - Primitive datatypes represent single values and are immutable.
2. Number- For numerical values, including integers and floating-point numbers (e.g., 10, 3.14).
3. String- For textual data, enclosed in single or double quotes (e.g., "Hello World", 'JavaScript')
4. Boolean- Represents a logical entity with two possible values: true or false
5. Undefined- Represents a variable that has been declared but has not yet been assigned a value.
6. Null- Represents the intentional absence of any object value. It's a primitive value.
7. Symbol- Introduced in ES6, symbols are unique and immutable primitive values, often used for unique object property keys.
8. BigInt- For integer numbers of arbitrary length, exceeding the limit of the number type.
9. Non- Primitive Datatypes - Non-primitive types are objects and can store collections of data or more complex entities.
10. Object
11. Array
12. Function

* Console.log - used for debugging and displaying information in the browser's developer console.
* document.getElementById("myParagraph").innerHTML targets an HTML element with the ID "myParagraph" and changes its content.

1. Numbers:
   * JavaScript has a number type that can represent both integers and floating-point numbers.
2. String:
   * A **char** is a single character (For example: 1, 6, %, b, p, ., T, etc.)
   * The **string** type is a special type that consists of multiple **char**s.
   * To initialize a string value in a variable, enclose it within single or double quotation marks:
   * String in js must be surrounded by quotes.
3. Let str = “hello”
4. Let str2 = ‘single quotes are ok too’
5. Let phrase = ` can embed another ${str}`

* There are three types of quotes.

1. Double quotes: “Hello”.
2. Single quotes: ‘Hello’.
3. Backticks: `Hello`.­­­

Boolean:

* A **bool** (Boolean) type has only 2 possible values: true or false.
* Booleans are building blocks for logic in the programs we write.

1. Naming conventions:

* In JavaScript, there are certain rules and conventions for naming variables:
* Variable names are case-sensitive, it means MyVariable & Myvariable are considered different.
* Variable names can only contain letters(a-z, A-Z), numbers(0-9), underscores(\_), and dollar signs($).
* Variable name must starts with a letters, underscore or dollar sign, they can’t begin with a number.
* Certain words are reserved by javascript and can’t be used as variable names such as **let, const, function, if, else etc**..
* It’s a common practice to use camelCase for variable names, where each word except the 1st starts with a capital letter(e.g., **m**y**V**ariable**N**ame).

1. Empty Variables:

* In JavaScript, it's possible to declare variables without assigning them a specific value. This can be useful when you know you'll use a variable later in your code, but you don't yet know its initial value.
* To declare an empty variable in JavaScript, you can use the let keyword followed by the variable name, without assigning any value
* creates a variable named myVariable but doesn't assign any value to it. The variable is said to be **uninitialized.**

1. Constants:

* In JavaScript, sometimes you need variables that *never change*. This is where const comes in! It’s used to create constants—values that stay the same after being set.
* Use the const keyword, just like let, but once you set its value, you **cannot change it** later.

1. Concept of truthy & falsy value in Javascript and how it relates to javascript Booleans:

* Truthy values:
  + A value is considered "truthy" if it evaluates to true when coerced to a Boolean. Most values in JavaScript are truthy, including:
    - Non-zero numbers (e.g. `1`, `-1`, `3.14`).
    - Non-empty strings (e.g.`”hello”`, ””`).
    - Objects (e.g. `{ }`, `[ ]`).
    - Functions.
* Falsy values:
* A value is considered "falsy" if it evaluates to false when coerced to a Boolean. The following values are falsy in JavaScript:
  + `false`.
  + `0` (zero).
  + `” ”`(empty string).
  + `null`.
  + `undefined`.
  + `NaN` (Not-a-Number).

1. Operators:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Operation** | **Example** |
| **+** | Addition | 3 + 2 = 5 |
| **-** | Subtraction | 3 – 2 =1 |
| **\*** | Multiplication | 3 \* 2 = 6 |
| **/** | Division | 4 / 2 = 2 |

1. part-1

* The nullish coalescing operator (??):
* provides a concise way to handle cases where a variable might be null or undefinedArithmetic operators: ~it performs mathematical calculations.
* Provides a default value when the left-hand side is null undefined.
* Arithmetic shortcut Assignment operators:
* Javascript created a shortcut for self-arithmetic operations.
* This operation is valid for all arithmetic operations:

|  |  |
| --- | --- |
| **Operator** | **Shortcut** |
| **+** | **+=** |
| **-** | **-=** |
| **\*** | **\*=** |
| **/** | **/=** |
| **%** | **%=** |

* Modulo Operator:
* The Modulo operator % tells you what’s left over after dividing one number by another. Result = dividend % divisor.
* **Dividend**: The number being divided.
* **Divisor**: The number that divides the dividend.
* **Result**: The reminder of the division

1. Comparison operator:

* Comparison operators are used to compare two operands.
* Sometimes we need to check whether an operand is bigger/smaller/... than another operand. The following table shows possible operators for comparison:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| **==** | Equal | 1 == 2 returns **false** |
| **!=** | Not Equal | 1 != 2 returns **true** |
| **>** | Greater Than | 1 > 2 returns **false** |
| **<** | Lower Than | 1 < 2 returns **true** |
| **>=** | Greater or Equal | 1 >= 2 returns **false** |
| **<=** | Lower or Equal | 1 <= 2 returns **true** |

* Strict vs Loose Equality:
* In js there are two types of equality operators:

1. Loose Equality (==):

* Loose Equality (==) check if two values are equal after performing type coercion.
* This means if the operands have different types, Javascript will try to convert them to a common type before making the comparison.

1. Strict Equality (===):

* Strict Equality (===), on the other hand, checks if two values are equal without performing any type conversion.
* If operands have different types, they are considered not Equal.
* it's recommended to use strict equality (===) in most cases to avoid unexpected results due to type coercion.
* Similarly, there are also loose inequality (!=) and strict inequality (!==) operators that work in a similar way, but check
* for inequality instead of equality.
* Loose inequality (!=) checks if two values are not equal after performing type coercion, while strict inequality (!==) checks if two values are not equal without performing type conversion.

1. Operators: part-2
2. Logical operators part-1

* Logical operators are used to check combinations of comparisons that return true or false.
* For example the following statement contains two comparisons:
* Is 5 greater than 3 **and** less than 6?

|  |  |  |
| --- | --- | --- |
| **Operator** | **Meaning** | **Example** |
| **&&** | And- true if all operands are true | a && b |
| **||** | Or- true if any operands are true | a || b |
| **!** | Not- true if operands is false | !a |

1. Logical operators part-2

* Logical operators have a special table called “Truth Table” that shows what the combination of logical operators returns.
* Truth table for the **&&** operator:

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a && b** |
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

* The only way to get a **true** for the **&&** operator is if both **a** and **b** are true.
* Truth table for the **||** operator:

|  |  |  |
| --- | --- | --- |
| **a** | **b** | **a || b** |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

* In this case, to get a **true** result, either **a** or **b** should be **true.**
* Truth table for the **!** operator:

|  |  |
| --- | --- |
| **a** | **!a** |
| False | True |
| True | False |

* Here the value of **a** is reversed. If a is **false** then **!a** is **true**.

1. Logical Operators part-3:

* When working with logical expressions, sometimes we need to simplify or rearrange them.
* ! (not) in front of two conditions joined by && (and), you can split it into two separate parts . The &&(and) becomes || (or), and each parts gets its own !(not):
* !(A && B) is the same as (!A) || (!B).

1. Type Coercion:

* Type of coercion in js is the automatic conversion of values from one daa type to another. This can happen implicity in certain operations or explicity when you use functions like String(), Number(), or Boolean().
* Implicit type coercion often occurs when you use the loose equality operator == or when you perform operations between different types, like adding a number to a string.

1. Decision Making:[1-if, 2-else, 3-else if, 4-switch.]
2. **IF:**
3. **Condition If statement:**

* If statement allows us to execute code with conditions.
* To use an if statement we need to add parenthesis () that will determine the condition, and everything that is inside the if is in curly braces {}
* An "if" statement starts with the keyword "if" followed by a condition inside parentheses. This condition is an expression that can be evaluated as either true or false.

1. **Code Block (if true):**

* If the condition is true, the code block immediately following the "if" statement is executed.

1. **Code Block (if false):**

* If the condition is false, the code block within the "if" statement is skipped, and the program continues with the next statement after the "if" block.

1. **IF-ELSE:**

* In JavaScript, the if...else statement is a fundamental control flow structure that allows for the execution of different blocks of code based on whether a specified condition evaluates to **true** or **false**.

1. **If-Else Nested:**

* We can nest if-else if-else statements within each other. This allows us to create hierarchical decision-making structures.

**4)Switch Statement:**

* The switch statement is another way to control the flow of your program based on different conditions. It's similar to using if-else if-else, but it can be more concise and readable in certain situations**.**
* The expression is evaluated once.
* The value of the expression is compared with the values of each case.
* If there is a match, the associated block of code is executed.
* The break statement is used to exit the switch statement. Without it, execution would continue to the next case.
* The default case is optional. It's executed if no other case matches.
* Output:
* In programming, we often need to show information to the user. This is called output.
* In JavaScript, the simplest way to output something is to use console.log()
* Output with Variables:
* We use backticks `` instead of quotation marks "" and inside the string wherever we put curly braces {} with a dollar sign $ before it, it will insert the value of what is written inside it.
* Type Conversion - Part 1:
* To convert between different data types in JavaScript, we need to use **type conversion**. To convert between different data types in JavaScript, we need to use type conversion.
* JavaScript supports both implicit and explicit type conversion. Let's focus on the most common explicit conversion methods

### Part 1: parseInt and parseFloat part

* **Part 2: Boolean conversion, String conversion.**

1. Loops:
2. For loop:

* Sometimes when programming it’s necessary to perform same or almost the same operation a couple of times.
* To prevent writing the same thing over and over again we can use **loops .**
* **Syntax of for loop - for ( let i = start; I < end ; i++ {code;}**
* The let i = start determines the initial value of i, i < end determines the condition for the loop to continue, and i++ increments i after each iteration. The i will receive all values from start to end (not including end) sequentially. For example:

1. While loop:

* A while loop is different from the for loop. A for loop allows us to iterate over a specific range, whereas a while loop allows us to keep iterating as long as a certain **condition** is met.

1. Break:

* The break statement stops the loop instantly when it's encountered.

1. Continue:

* The continue statement stops the current iteration and continues to the next iteration.

1. Recap- Factorial:

* Factorial is a mathematical operation.
* Factorial of n is the product of all positive integers less than or equal to n.
* Examples:
* Factorial of 3 is 6, which can be calculated as 1 \* 2 \* 3.
* Factorial of 6 is 720, which can be calculated as 1 \* 2 \* 3 \* 4 \* 5 \* 6
* Factorial of 2 is 2, which can be calculated as 1 \* 2

1. Do-While Loop:

* The do...while loop is similar to the while loop, but with one key difference: the code block is executed at least once before the condition is checked
* This means that the loop will always run the code once, and then it will check if it should continue.
* Here's the syntax of a do...while loop: do { code;

} while (condition);

* The code inside the do block is executed first. Then, the condition inside the while is evaluated.
* If the condition is true, the loop continues to the next iteration. If the condition is false, the loop stops.

1. Nested Loop:

* A nested loop is simply a loop inside another loop
* The inner loop will complete all its iterations for each single iteration of the outer loop.
* A good analogy for this is a clock: for each hour (outer loop), the minute hand (inner loop) must complete its full 60-minute cycle.

1. Functions:
2. Declare a Function :

* A function is a sequence of code that has a name.
* The purpose of a function is to reuse a piece of code multiple times.
* Important! The function code must come before it's call/execution

1. Arguments:

* An argument in a function is a value that you pass into the function when you call it.
* To add arguments to a function, we write them inside the parenthesis ():

1. Return:

* The return statement in a function is used to specify the value or values that the function should produce as its output.

1. Function Expression:

* a function expression is a way to define a function as part of an expression.
* Unlike function declarations, function expressions are not hoisted, which means they cannot be called before they are defined in the code.
* The function keyword is followed by an optional name for the function.
* If a name is provided, it becomes a named function expression; otherwise, it's an anonymous function expression.
* The parameters are specified in parentheses, and the function body is enclosed in curly braces.

1. Default Parameters:

* it's common to have functions where some parameters have default values.
* These are values that the parameter will take if no argument is provided during the function call.
* Default parameters make functions more flexible and easier to use.
* To define a function with default parameters in JavaScript, you assign a value to the parameter in the function definition using the = operator.
* Here's the basic syntax: function functionName(param1, param2 = defaultValue2, param3 = defaultValue3) { // Function body // ... }

1. Arrow Functions:

* Arrow functions provide a shorter way to write functions in JavaScript.
* They are especially handy for short, simple functions.
* An arrow function expression has a shorter syntax compared to function expressions and does not have its own bindings to the this, arguments, super, or new.target keywords.
* Arrow functions are always anonymous.

1. Arrays Part-1:
   1. Declaring an Array:

* An Array is a collection of items, and it can contain values of fdifferent types, such as numbers, strings or even other arrays.
* Arrays are created using square brackets[ ], and the itrms inside the array are separated with commas.
* Here is an example of how to create an Array:- let myArray = [ 1, 2, “three”, true ];
* To check the length of an array, we can use the length property:- let length = myArray.length;
  1. Accessing Array Elements:
* In JavaScript, we use arrays to store multiple values in a single variable.
* Each value in an array is called an element, and each element has an index.
* The indices start from 0 to the length of the array minus one.
  1. Modifying Arrays:
* To modify a specific element in an array, you can assign a new value to it using its index.
  1. Array Processor:
* Declaring arrays: Use square brackets [] to create arrays, separating elements with commas.
* Accessing elements: Use the index in square brackets to access specific elements, starting from 0.
* Modifying arrays: Assign new values to elements using their index.
* Array methods: Utilize built-in methods like push(), pop(), shift(), unshift(), sort(), reverse(), indexOf(), lastIndexOf(), and includes() to manipulate arrays.
  + Array Methods Part 1-
* Arrays are packed with many methods (functionalities). To access a method, write: someArray.method()
* push(element) - adds an element to the end of the array
* pop() - removes an element from the end of the array and returns it
* shift() - removes an element from the beginning of the array and returns it
* unshift(element) - adds an element to the beginning of the array
* sort() - sorts the elements of an array
* reverse() - reverses the order of elements in an array.
* indexOf(element) - returns the index of the first occurrence of an element in the array, or -1 if it's not found.
* lastIndexOf(element) - returns the index of the last occurrence of an element in the array, or -1 if it's not found.
* includes(element) - returns true if the array contains the specified element, otherwise returns false.
  + Array Methods part 2-
    - * + Iteration means going through elements one by one in a sequence.
        + With arrays, we can access each element systematically using different methods.
        + The most common way to iterate through an array is using a for loop together with the .length property.
* The forEach Method:
  + - * + The forEach method allows you to loop through a sequence (like an array) while keeping track of each item.
* for...of Loop:
  + - * + The for...of loop provides a simple way to iterate over the values of iterable objects, such as arrays, strings, maps, sets, and more. It combines the conciseness of forEach with the ability to break and continue.
* element: On each iteration, the next value from the iterable object is assigned to this variable.
* iterable: The object whose elements are being iterated over (e.g., an array or a string).
* Array Slicing-- Slicing allows us to extract portions of an array using the following syntax: arr.slice(start, stop). For example, consider this array:
* Array Methods Part 3-
  + - 1. concat(): Combines two or more arrays and returns a new array, without modifying the original arrays.
      2. join(separator): Joins all elements of an array into a string, separated by the specified separator (optional).
      3. slice(start, end): Extracts a section of an array and returns a new array. The start index is inclusive, and the end index is exclusive.
      4. splice(start, deleteCount, item1, item2, ...): Changes the contents of an array by removing, replacing, or adding elements. It modifies the original array and returns an array containing the deleted elements (if any).
* Array Methods Part 4-

1. map(callback): Creates a new array populated with the results of calling a provided function on every element in the calling array.
2. filter(callback): Creates a new array with all elements that pass the test implemented by the provided function.
3. reduce(callback, initialValue): Executes a reducer function on each element of the array, resulting in a single output value.

* Membership Testing:
* can check whether an element is in an array or not using the indexOf() method:
* If the element is not found, indexOf() returns -1.
* can also use the includes() method.

1. Strings in Depth:

* In js, strings are immutable. This means that once a string is created, it’s value cannot be changed.
* Any operation that appears to modify a string actually creates a new string.
* However, remember that unlike arrays, strings are immutable, so methods that modify arrays (like push() or pop()) don't exist for strings.

1. Iterate Over Strings:

* In JavaScript, strings are iterable objects, meaning you can loop through each character in a string.
* There are many ways to iterate over a string.

1. Template Literals:

* Template literals use backticks `` to create strings and allow you to insert expressions using ${}.
* For instance, when creating greetings, you can dynamically embed names or other data directly into the template literal.
* Template literals also support multi-line strings without needing the\n escape character.
* You can simply press Enter for a new line, and the line break will be preserved in the output. Both approaches work:

1. String Methods:

* Here are some commonly used string methods:
* toUpperCase() and toLowerCase(): Convert a string to all uppercase or lowercase.
* replace(): Replaces a specified value with another value in a string. It only replaces the first occurrence by default.
* trim(): Removes whitespace from both ends of a string.
* charAt(): Returns the character at a specified index in a string.
* Multi-dimensional Arrays:
* 2D Arrays Basics
* A 2D array is an array containing sub-arrays. Each sub-array can be seen as a row in a grid.
* This array represents a 2x2 grid (2 items in each row, and there are 2 rows)
* Accessing 2D Array Elements
* Accessing elements in a 2D array is similar to accessing elements in a 1D array, but instead of using a single index, you use two indices: one for the row and one for the column.
* Nested Loops with 2D Arrays
* When working with 2D arrays, we often use nested loops to iterate over each row and each column, allowing us to access or modify every element in the array.
* Matrix Addition & Substraction
* To add or subtract two matrices, they must have the same dimensions (i.e., the same number of rows and columns). The operation is performed element-wise, meaning you add or subtract corresponding elements in each matrix.
* Jagged Arrays
* A jagged array is a 2D array-like structure where each row can have a different length.
* 3D Arrays And Beyond
* A 3D array can be visualized as an array of 2D arrays.
* For example, think of multiple buildings, each with multiple floors, each floor having multiple rooms.
* Access an element using three indices: the first for the building, the second for the floor, and the third for the room.

https://javascript.info/types

<https://www.codechef.com/practice/html>

<https://webflow.com/careers>

https://gsap.com/docs/v3/Plugins/ScrollTrigger/static.observe()