Date : / /
Page No. :

JAVA SCRIPT HANDWRITTEN NOTES

BY

@ codewith_sandom

Date : ..../..../....
Page No. : .....

TOPICS

1 → Introduction
2 → Location to Put JavaScript Code
3 → JavaScript Comment
4 → Variables
5 → Data Types
6 → Operators
7 → Control Statements
8 → Functions
9 → Scope
10 → Math
11 → JavaScript BOM
12 → JavaScript DOM
13 → Closures
14 → Prototype
15 → Encapsulation
16 → Inheritance
17 → Polymorphism
18 → Abstraction
19 → Events
20 → Event Handling
21 → Error Handling
22 → Higher-Order Function
23 → Modules
24 → Asynchronous Programming
25 → Design Pattern
26 → Functional Programming

Date: / /
Page No.:

Introduction

JavaScript developed by "Brendan Eich" at Netscape in the mind 1990s, It's first name is Live Script, but later renamed Javascript to capitalize on the popularity of Java.

JavaScript is a high-level programming language, which is mainly used for web development. It was orginally designed as a client-side scripting language to enhance web pages with interactivity and dynamic content. But Now it becomes a versatile language that's why it is also used for server-side development.

JavaScript can Interact with DOM → [DOCUMENT OBJECT MODEL] to modify the structure, styling, content of the webpages.

It is often used with Node.Js, a runtime environment which allows Java Script codes to be executed at the outside of the brouser. This enables serverside scripting and building full-stack Java, Script Applications.

JavaScript's Universality and Versatility have made it a fundamental skill for web developers.

Date : ...../...../.....
Page No. : .....

→ Locations to Put JavaScript Code

JavaScript code can be placed at three different places, Inline, Internal and External.
Let's understand these places in brief.

i) Inline JavaScript :

Inline JavaScript, JavaScript code can be placed directly within HTML elements using the "onclick", "onload" attributes or similar type event attributes.

example

```
<button onclick="myFunction()">Click Me
                                <|button>
```

•> < button > ---- < / button > is the 'HTML' element.

•> (On click = " my Function ( )") is the Java Script code.

ii) Internal Java Script

In Internal JavaScript, JavaScript code can be placed at two locations, Between the <head> tag of HTML and between the < Body> tag of HTML. We use < script> ----- <|Script > tag for place. Java Script code at these two places.

Date ..../..../....
Page No. : .....

Example

• html , file include JavaScript codes

```
<head>
    <Script>
        function myFunction () {
            alert ("Hello , Pawan");

        }
</Script>
</head>
    <body>
        //HTML CODES

    <Script>
        Function myFunction () {
            alert ("Hellow Pawan");

        }
    </Script>
</body>
```

JavaScript Codes

iii) External JavaScript

In External JavaScript , code can be placed at a different file, means external (.js) files and linked with an HTML document using the < script tag with the " Src" attribute.

Date ..../..../....
Page No. : .....

Example

• html , file include JavaScript codes

```
<head>
    <Script>
        function myFunction () {
            alert ("Hello , Pawan");

        }
</Script>
</head>
    <body>
        //HTML CODES

    <Script>
        Function myFunction () {
            alert ("Hellow Pawan");

        }
    </Script>
</body>
```

JavaScript Codes

iii) External JavaScript

In External JavaScript , code can be placed at a different file, means external (.js) files and linked with an HTML document using the < script tag with the " Src" attribute.

Date : / /
Page No. :

Example

- html file

```
<HTML>
    <Head>
        < Script  Src = " first . Js" > < /Script >
    < /Head >
    <body>
            //HTML  CODES
    < /body >
< /HTML >
```

- js file

```
Function my Function () {
        alert ("Hellow, Pawan");
}
```

→ JavaScript Comment

JavaScript comments are mainly used for add explanation about the codes, Because It is ignored by the brouser or interpreter.

In the JavaScript, two types of comment are present, single - line comments and Double or Multiline comments.

Single - line comments start with double slashes (//). And Multi - line Comments begin with /* and end with */.

Date : ..../..../....
Page No. : .....

Example

// This is Single - line Comment

/* This is
Multi - line
Comment
*/

→ Variables

In the JavaScript, Variables are used for store data values, It is like container in JavaScript which hold different - different types of information, such as strings, numbers, booleans, etc.

To declare a variable in JavaScript we use "var", "let", or "const" keyword, followed by the variable name.

Let's learn these three keywords in details.

i) "Var"

The "var" Keyword is the main way for declare variables in javascript. It is accessible within the entire function where it is declared, Because It has function scope, Variables declared within "var" can be redeclared and reassigned.

Example

```
//Declaring and assigning variable using
var
var age = 20;
Console.log(age);
```

•> A variable can be assigned a data value using the assignment operator equal sign (=).

↓

(we will study about this in coming section.)

```
//Reasigning the value of variable
age = 22;
Console.log(age);
```

•> A variables can be reasigned in JavaScript by assigning a new value to the variable without using the "var" keyword.

ii) "let"

The "var" keyword have same issue, that's why "let" keyword was introduced in ES6 (ECMA Script 2015) to address that issues "let" is only accessible within the nearest block where it is declared, because it has block scope. Variables declared with "let" can be

reassigned but not redeclared with the same scope.

Example

```
// Declaring and assigning a variable using let
let name = 'Pawan';
console.log (name);
```

{ // Output : Pawan }

```
// Block Scope with let
if (true)
{
    let x = 20;
    Console.log (x);
}
```

{ Output : 20 }

Date ...../...../.....
Page No. : .....

iii) "Const"

The "const" stands for "constant", It is also introduced in ES6. It has also block scope like "let", but in "const" declaration of variable cannot be reassigned after it has been assigned a value.

example

```
[// Declaring a "Const" variables

    Const PI = 3.14;
        Console . log (PI);
```

{Output = 3.14}

→ Data Types

Data types are used for hold and manipulated the values in the JavaScript. The common data types are numbers, strings, booleans, undefined, null, array, object, function and Symbols. Each data type has specific behaviour and properties which allow to the programmer to perform a different - different operation and Calculations.

Understanding Data Types is important for error - free code.

Here is the main data type with example.

i) Number

It is used for representing the numeric value in the Java Script. It includes both floating - value numbers and integers

example

```
let age = 10;
let temperature = 99 - 1
```

In this example,

- The variable "temperature" is assigned a floating - point value of "99.1".
- The variables "age" is assigned an integer value of "10".

→ We can perform arithmetic operation also on numbers value such as addition, subtraction, division and multiplication.

Date: ...../...../.....
Page No.: .....

ii) String

It is used for representing the sequence of character values in JavaScript. It is enclosed within single quotes (') or double quotes (" ").

example

```
let name = 'Pawan';
let message = "Hello, Pawan";
```

In this example,

- The variable "name" is assigned the string value 'Pawan',
- The variable "message" is assigned the string value "Hello, Pawan".

iii) Boolean

It is used for representing the logical value that can be either true or false. It is mainly used for control the flow of program and making decisions.

example

```
let has Permission = False;
let is hogged = True;
```

In this example,

- The variable "has Permission" is assigned the Boolean value 'false', show that the user does not have permission.
- The variable "is logged" is assigned the Boolean value 'true', show that the user is logged in.

iv) Undefined

It is used for representing the variable that has been declared but has not been assigned a value. It shows the absence of uninitated (Not - stored) variable.

example

```
let x ;
let y = undefined ;
```

In this example,

- The variable "x" is declared without assigning a value, so its value is undefined by default.
- The variable "y" is obviously assigned the value 'undefined'. It is after used to check if a variable has been assigned a value.

Date : / /
Page No :

v) Null

It is used for representing the intentional absence of any object value. It is mainly used to show that a variable or attribute has no value or is "null".

example

```
let data = null;
```

In this example,

- The variable 'data' is obviously assigned the value 'null', shows that it has no assigned value.

vi) Object

It is complex data type which representing the collection of properties and their value. It is used for organize and store related information for function together. In other words object provide a way to represent and manipulate complex data structure in Java Script.

Here is the example of object.

Date: ..../..../....
Page No.: .....

Example

```
let person = {
    name : 'Pawan',
    age : 20
    is Student : true

};
```

In this example,

-> The variable "person" is an object that contains properties such as 'name', 'age', and 'isstudent'. These properties have assigned values such as 'Pawan', '20', and 'true', respectively.

vii) Array

It is used for representing an ordered list of values. It can store multiple values of any data type, such as strings, objects, numbers.

example

```
let numbers = [1, 2, 3, 4];

let vegetables = ['carrot', Ternip', Radish'];
```

Date: ...../...../.....
Page No.: .....

•> The 'id' is the optional string parameter which is a description for the symbol and it is useful for Indentifying symbol.

→ Operators

It is keywords which is used for perform an operation on values, such as calculation, comparisons and more.

In the JavaScript six types of operators are present.

1) Arithmetic Operators

• Addition (+)	It is used for add two values.
• Subtraction (-)	It is used for subtract one value from another.
• Multiplication (*)	It is used for multiplies two values.
• Division (/)	It is used for divide one value by another.
• Modulus (%)	It return the remainder of division operation.
• Exponentation (**)	It raises a number to the power of another.

Date : ...../...../.....
Page No. : .....

ii) Assignment Operators

• Assignment (=)	It is used for Assigns a value to a variable.
• Addition Assignment (+ =)	It is used for adds a value to the variable and assign result
• Subtract Assignment (- =)	It is used for subtract a value from the variable and assign the result.
• Multiply Assignment (* =)	It is used for Multiples the variable by a value and assign the result.
• Division Assignment (/=)	It is used for divide the variable by a value and assign the result.

iii) Comparison Operators

• Equal to (= =)	Checks, if two values are equal.
• Not Equal to (! =)	Check, if two values are not equal.
• Strict Equal to (= = =)	Checks, if two values are equal in both value and type.
• Strict Not Equal to (! = =)	Checks, if two values are not equal in either value or type.
• Greater than (>)	Checks, if one value is greater than another.

• Less than (<)	Checks, if one value is less than another.
• Greater than or Equal to (>=)	Checks, if one value is greater than or equal to another.
• Less than or Equal to (<=)	Checks, if one value is less than or equal to another.

iv) Bitwise Operators

• Bitwise AND (&)	It is used for perform a Bitwise AND operators.
• Bitwise OR (|)	It is used for perform a Bitwise OR operation.
• Bitwise XOR (^)	It is used for perform a XOR (exclusive OR) operation
• Bitwise Left shift (<<)	It is used for shifts the bits of a number to the left.
• Bitwise right shift (>>)	It is used for shifts the bits of a number to the right, preserving the sign.
• Bitwise zero-fill right shift (>>>)	It is used for shifts the bits of number to the right, filling with zeroes.

Date : ...../...../.....
Page No. : .....

v) Logical operators

• Logical AND (&&)	It return true if both operands are true.
• Logical OR (||)	It return true if either operand is true.
• Logical NOT (!)	It denies the truth value of an expression.

vi) Special operators

• Conditional operator (?:)	It provides a short Syntax for conditionaly assigning a value based on condition.
• Comma (,)	It allows multiple expression to be evaluated as single statement.
• delete	It is used for delete a property from an object.
• in	It is used for check the property is exist in a object.
• instance	It is used for check the object is an construct function or instance of specific class.
• type of	Used for return the type of value.

Date : ..../..../....
Page No. : .....

→ Control Statements

An order of execution of statement in a program is determined by the Control Statement, also called control flow statements, which are constructs in the programming language that determine the order in which statement are executed.

In order to control the flow of execution, you can use loops, Jumps and conditions to determine the flow of executions.

In the JavaScript Four types of Control Statements are present.

i) Conditional Statements
ii) Switch Statements
iii) Looping Statements
iv) Jump Statements

i) Conditional Statements

Programming structure called conditional statements, also known as conditional control statements, let you decide what to do and run distinct blocks of code depending on certain condition. They enable a program's execution to take one of several paths depending on whether a condition is true or false.

Date : / /
Page No. :

In the conditional statement we have three different types of statements are present.

- If - statement
- if - else statement
- else - if statement

If - Statement

"if - statement" in JavaScript is a conditional control that enables you to run a block of code if certain condition is true. It gives you a way to make choices and regulate, how your program is executed based on various circumstances.

Syntax

```
if (condition {
        // CODES
}
```

Date : / /
Page No. :

example

```
var num = 5;
if (num > 0) {
    console.log ("Bigger than zero");
}
```

In this example,

- It evaluates the statement "num > 0".
- The code block "console.log ("Bigger than zero");" is called if num is Bigger than zero,

var num = 5;

and it prints "Bigger than zero" to the console

- if - else statement

"if - else statement" in JavaScript enables you to run one block of code if a certain condition is true and a different block of code if the condition is false.

It offers a means of controlling the course of execution and coming to judgements based on various circumstances.

In other words

It provide a way to make decisions & control the flow of execution based on different conditions.

Syntax

```
if (Condition) {
        // CODES
    } else {
        // CODES
    }
```

example

```
var num = 5;

if (num > 0) {
    Console.log ("Bigger than zero");
} else {
    console.log ("Not Bigger than zero");
}
```

In this example,

- The value '5' has been set to the variable num.
- The condition "num > 0" is evaluated by the if-else-clause.

Date : ...../...../.....
Page No. : .....

- The code block inside the first set of curly bracis is run if the condition is true, and it prints " Bigger than zero" to the console.
- The code block inside the else statement which is indicated by the second set of curly bracis, is run if the condition is false means num is less than zero, then it prints "Not Bigger than Zero" to the console.

else - if statement

There is no direct "else - if" expression in Javascript. However by employing a series of "if - statement", you may {imitation}.

the functionality of a "else - if" statements.

you are able to examine various circumstan in succession using this method.

In other words

This approach allows you to evolute multiple conditions sequentially.

Date : / /
Page No. :

Syntax

```
if (condition) {
        // CODES

} else if (condition 2) {
        // CODES

} else {
        // CODES

}
```

example

```
var num = 5;

if (num >0) {
        console.log ("Bigger than Zero");
} else if (num <0) {
        Console = log ("Not Bigger than zero");
} else {
        Console.log ("Equal to zero");
}
```

In the "if-statement" the code block run if the condition is true, And In the "if-else statement" code block run if the condition is true or false

But in the "else - if statement" we set one more condition or statement

Let's understand this by explanation of example.

In this example,

- "if - statement" code block is run if the num is Bigger than zero.
- if num is not bigger than zero than "else - if" code block is run.

BUT, just think if the numbers is equal to the zero then what happened,

that's why we gave third condition or also called final destination output if upper all condition are wrong.

Important :- In the "else - if statement" we set unlimited condition, there is not limit of conditions,

(But the main thing is a small fault can change the way or working of code then just concentrate on it and done it correct for correct output.)

↳ Just knowledge purpose

Date : ...../...../.....
Page No. : .....

ii) Switch Statement

To run one code from many expressions, use the switch statement in Javascript. It is similar to another if statement that we learned in conditional statement, But it is more practical than if --- else -- if--

It can be used with integers, character and other type data.

Syntax

```
Switch (expression) {
    case value 1:
        //CODES
    break ;
    Case value 2 :
    break ;
    default :
        //CODES
    break ;
}
```

Date : ..../..../....
Page No. : .....

example

```
var day = 3;
var day Name ;

Switch (day) {
    case 1:
        day Name = "Monday";
    break;

    Case 2:
        day Name = "Tuesday";
    break;

    case 3:
        day Name = "Wednesday";
    break;

    Case 4:
        day Name = "Thursday";
    break;

    Case 5:
        day Name = "Friday";
    break;

    Case 6:
        day Name = "Saturday";
    break;

    Case 7:
        day Name = "Sunday";
    break;
```

```
default :
        dayName = "Invalid";
    break;

}

console.log("The day is:" + dayName);
```

In this example,

- The variable day is set to 3 which corresponds to wednesday.
- Day's value is accessed by the switch statement which then matches it with the proper case.
- The day being 3, the third case's code block is carried out. The variable dayName recieved the value "Wednesday" as result.

And the result or Output is,

"The day is: Wednesday"

- The default case's code block would be performed and dayName would be given the value "Invalid" if day didn't fit any of the cases.
- "break" we will learn this in "Jump Statements".

Date : ...../...../.....
Page No. : .....

iii) Looping Statement

Looping statement in JavaScript are used to run a piece of code continually until a certain condition is fulfilled. They offer a technique to cycle over a set of data and automate repetitive processes.

In the looping statement we have three different types of statements are present.

i) For loop
ii) While loop
iii) do-while loop

i) For Loop

Using the "for loop" control statement, you may a run a block of code continually for a predetermined number of times. It is mainly used at that time when you already know the numbers of iteration.

The For loop consists three parts: initalization, conditon and interation.

Syntax

```
for (initalization; condition, iteration) {

    // CODES

}
```

Let's breakdown the Syntax:-

-> Intialization

This section, which is carried out just before the loop begins, after initialises a loop variables. It is just run once at the beginning.

-> Condition

Each repetition of the section begins with an evaluation. The code block inside the loop is run if the condition is satisfied. And the loop ends if the condition is false.

-> Iteration

This section normally changes the loop variable and is carried out at the end of each iteration.

example:-

```
for (var i=0 ; i<6 ; i++) {
    console.log("Here is the Iteration no:
                +1);
}
```

Date : / /
Page No.

In this example,

- From $i = 0$ through $i < 6$, the "for loop" is programmed to iterate.
- The initial value of 'i' loop variable is '0'. Each iteration begins with an evaluation
- of condition $i < 6$.
- The code black inside the loop is performed as long as the condition is true.
- In each iteration the code;
  'console . log ("Here is the Iteration no. :" + i) is executed.
- Here is the result of example;

Output: Here is the Iteration no. : 0
Here is the Iteration no. : 1
Here is the Iteration no. : 2
Here is the Iteration no. : 3
Here is the Iteration no. : 4
Here is the Iteration no : 5

- The loop runs six time because the condition "$i < 6$" is satisfied for the value of "i" from '0' to '5'.

- After each iteration, the 'i' variable is incremented by '1' using the i++ notation.

Date: / /
Page No.:

ii) While Loop

Another sort of loop in JavaScript is the while loop, which continually runs a piece of code as long as predetermined condition is true, and it exists when the condition becomes false.

Syntax

```
while (condition) {
        // CODES
}
```

example

```
var i = 0;
while (i < 6) {
console.log ("Here is the Iteration No."+i);
        i++;
}
```

In this example,

- As long as the "i < 6" condition is true the while loop iterates.

Date: ...../...../.....
Page No.: .....

- The loop initializes the variable 'i' to '0' and uses, console. log "Here is the Iteration No. "+i);
  to repeat the value of 'i' to the console after each iteration. The value is printed and then "i++" is used to increase "i" by "1".

- The loop runs six times because the condition is "i < 6" is satisfied for the value of 'i' from '0' to '5'.

- Here is the result of the example:

Output: Here is the Iteration No. : 0
Here is the Iteration No. : 1
Here is the Iteration No. : 2
Here is the Iteration No. : 3
Here is the Iteration No. : 4
Here is the Iteration No. : 5

iii) do - while loop

As long as a certain condition is met, you may continually run a block of code using the "do - while" loop control statement. A "do - while" loop ensures that the code block will be performed at least once, even if the main distinction between it and other loop structures like the while or for loop.

Date : / /
Page No. :

Syntax

```
do {
        // CODES
    } while (condition);
```

example

```
var i = 0;

do {
    console.log ("Here is the Iteration
                No.: " + i);

    i++;

} while (i<5);
```

In this example,

- The variable "i" is initialised to "0" at the beginning of the "do-while" loop.
- using the code block
  console.log ("Here is the Iteration No." + i)

  the loop's code block writes the current value of 'i' to the console
- The "i" variable is increased by "1" once the code block has been using "i++".

Date: ..../..../....
Page No.: .....

iv) Jump Statements

Jump Statements provide the program's execution flow flexibility and control. They provide you the option to the end Functions easily, skip iterators, or exist loops easily.

In the Jump statements three different types of statement are present.

- break statement
- Continue statement
- return statement

⇒ break statement

The 'break' statement is used to end or quit easily from a loop or switch statements. When this conditions is met, the program instantly exists the final enclosing loop or switch statement and moves on to the statement that follows the loop or switch.

⇒ Continue Statement

The remaining code in a interation of a loop can be skipped by using the "Continue" command, which moves to the next iteration. The code that follows the "Continue" statement within the loop is ignored when it is faced, causing the program to instantly

go to the next iteration of the loop.

return Statement

A function can be ended and a value returned by using the 'return' command when it is met, it puts an end to the function's execution and return the supplied value. The return statement also causes any surrounding loop or switch statement to end.

Date: ..../..../....
Page No.: .....

→ Functions

Java Script Functions are effective tools for structuring and organising your code. They lead you to reuse code, encapsulate functionality and improve the modularity and maintainability of your programs. Functions are crucial for developing clear and effective Javascript code, whether you're creating a simple script or advanced online applications.

Here is the types of functions in Java Script.

- Named function
- Generator Function
- Immetiat Invoked Function
- Anonymous Function
- Higher - order Function

→ Named Function

Named Function is that you may write reusable code that is simply called by it name. They Support flexibility, readability and code organization.

You javascript code might utlize named ~~command~~ Function to make it simpler for you to manage and learn.

Date : ...../...../.....
Page No. : .....

Functions can be define by using the "Function" Keyboard and it is Followed by the name of Functions, arguments and the body of Functions.

example

Here is the example of named Function which is used For add two numbers.

```
var result = add Numbers (8,9);
console .log (result);
```

In this example,

- The name of the function is "add Number" and the argument are '8' and '9'.
- The sum of two numbers which is also called a returned value is stored in the variable 'result'.
- The Output of this example is

Output : 17

Date : ..../..../....
Page No. : .....

ii) -> Generator Function

Generator Function offer a potent technique for a building iterators. They provide a means to build sequences slowly and may be applied to the efficient processing of huge datasets or the more legible and synchronous like handling of asynchronous processes.

Generator Function are defined by using the 'Function' syntax.

example . 1

```
Function* numbersequence () {

        let num = 1;
        while (true) {
            yield num;

            num ++ ;
        }
}
```

In this example,

-> The generator Function is "numbersequence", which is used while loop to generator the number sequence.

-> "num" is first initalised to 1, and the "yield" Keyboard is used to continually yield "num" most recent value.

Date :
Page No. :

-> The value of "num" is increased by "1" after each iteration.

example . 2

In this example, we set the limit of number or yields.

```
Functions* number sequence () {

    let num = 1;
    while (num <= 10) {
        yield num;

        num ++;

    }
}
```

-> With a limit of numbers, the number '10' will cause the generator to stop the working.

→ Immediate Invoked Function

Code encapsulation, avoiding variable leaking and upholding a clear global name space are all made possible by Immediate Invoked Function. They are frequently employed in the building

of self - contained modules or intilization code in modular Javascript.

example

```
(Function () {
        var message = 'Hellow, Pawan';
        console.log (messaage);
    })();
```

In this example,

- The Anonymous function is immediately invoked.
- It creates a message named 'Hello, Pawan' and logs that value to the terminal or console.
- Then,
  Following its closing curly braces = brackets () are added to call the function.

Date : ...../...../.....
Page No. : .....

→ Anonymous Function

In situations where Functions are used as values or when reusability is not required anonymous Functions offer a meachanism to define Function on the fly.

Example

```
var result = Square (10);
console.log (result);
```

In this example,

- "square" is assigned by anonymous function, which is invoked with the argument '10'.
- The square of 10 which is also called return value. It is saved in the variable "result".
- The Output of this example is

< Output : 100 >

→ Higher Order Function

In Functional programming paradigms, higher order Function are frequently used in Java Script to support code reuse, structure and the implementation of strong structure. They provide you with a versatile approach to interacting with functions and enables you to

write code that is more creative and flexible.

example

Here is the example of higher - order function with a call back :-

```
Function greeting () {
    console .log ('Hello ,Pawan');

Function higher order function (call back) {
    console . log (Before the (call back');

        Call Back ();

    console . log ('After the call back');
}

        higher order Function (greetings);
```

In this example,

- The 'greetings' Function is takes as an argument by the 'higher order Function' and executes it within its body.
- Here is the output of the example :-

Output : Before the callback
Hello, Pawan
After the call Back

•> Higher Order Function (Imp)

The upper example, shows how the

↙ 'higher order Function' ↙

may accept and invoke a callback function enabling you to after the higher-order function's behaviour by offering several callback Functions.

→ Scope

The term "scope" describes the visibility and use of variables, Functions and objects inside a certain area of your code. It establishes which sections of your code are permited to access and use particular variables or Functions.

In JavaScript, two types of Scopes are Present

i) Local Scope
ii) Global Scope

Date: ...../...../.....
Page No.: .....

i) Local Scope

When variables a Functions are declared inside a certain block, such as Function, a loop, or an if statements, a scope known as local scope.

A local scope's variable are only used inside the specific scope and are hidden From other scopes.

example

```
Function local scope example () {

    var local var = "Local Variable";

    console.log (local var);
```

Local variable is defined in the Function of local scope

Access with the Function

```
local scope example ();
console.log (local var);
```

local variable is not define outside the function

Date: ...../...../.....
Page No.: .....

i) Local Scope

When variables a Functions are declared inside a certain block, such as Function, a loop, or an if statements, a scope known as local scope.

A local scope's variable are only used inside the specific scope and are hidden From other scopes.

example

```
Function local scope example () {

    var local var = "Local Variable";

    console.log (local var);
```

Local variable is defined in the Function of local scope

Access with the Function

```
local scope example ();
console.log (local var);
```

local variable is not define outside the function

Date: ...../...../.....
Page No.: .....

i) Local Scope

When variables a Functions are declared inside a certain block, such as Function, a loop, or an if statements, a scope known as local scope.

A local scope's variable are only used inside the specific scope and are hidden From other scopes.

example

```
Function local scope example () {

    var local var = "Local Variable";

    console.log (local var);
```

Local variable is defined in the Function of local scope

Access with the Function

```
local scope example ();
console.log (local var);
```

local variable is not define outside the function

In this example

→ "local var" is declared within the Function "local scope example". It is only access within the function, If we try this outside the function the error will occur because the variable is confined to the local scope of the function.

ii) Global Scope

When a variable or Function is defined outside of a function or block, it is said to be in the global scope of Javascript, Anywhere in your code can access the variables and functions.

example

```
Var global variable = "Global scope";

Function global scope example () {
    console.log (global variable);
}

global scope example ()
    console.log (global variable);
```

Access in Function

Access outside the Function

Date : ..../..../....
Page No. : .....

In this example,

- "global variable" is global variable since we define it outside of any Function or block.
- This indicates that it may be accessible From both inside the outside of the "global scope example ()" method.
- "global scope example ()" correctly logs, the value of "global variable" inside the function when it is called. Similar to this "global variable" is accessible and its value is seen when it is logged outside of the function.

→ Math

The "Math" objects is a built - in object that provide mathematical Functions. It includes properties and procedures for carrying out different - different operations Here is the Functions which is provided by "Math" object.

i) Math . abs (x)
ii) Math . Floor (x)
iii) Math . ceil (x)
iv) Math. round (x)
v) Math . Pow (x, y)
vi) Math. sin (x), Math . cos(x), Math , tan (x)
vii) Math . sqrt (x)

Date : / /
Page No. :

i) Math . abs (x)

It return the absolute value for the given number.

example

```
let x = - 6;
let absolute value = Math. abs (x);
console . log (absolute value);
```

In this example,

- "Math . abs (x) Function is used to find the absolute value of the x. Regardless of the intial of x, it return the positive value of x.

- A single parameter (x), which be a integer or a floating - point number is needed by the Math . abs () Function

- Output of the example :-

Output : 6

ii) Math . Floor (x)

It return the integer or number which is less than or equal to the 'x'.

Date : / /
Page No. :

example

```
let x = 4.7;
    let Floorvalue = Math.Floor (x);
        console.log (Floorvalue);
```

In this example,

- Find the nearest integer which is less than the value of 'x' is done by "Math.Floor (x)" Function.

- Output of the example,

Output : 4

iii) Math.ceil (x)

It return the integer or number which is greater than or equal to the x.

example

```
let x = - 8.9;
    ceilvalue = Math.ceil (x);
    console.log (ceilvalue);
```

In this example,

- Find the nearest integer or number which is ~~grater~~ greater than or equal to value of 'x' is done by,

→ "Math . ceil (x)" Function.

→ Output of this example,

Output : - 8

iv) Math. round (x)

It return the rounded and nearest integer or number of the value "x".

example

```
let x = 10.9 ;
let roundvalue = Math. round (x);
Console. log (round value);
```

In this example,
→ Find the nearest and rounded integer or number is done by,
"Math round (x)" Function

→ Output of this example:-

Output : 11

Date : ..../..../....
Page No. : .....

v) Math.pow (x,y)

It change the value of (x,y) into the x^y.

example

```
let x = 2 ;
let y = 3 ;
let result = Math.pow (x,y);
console.log (result);
```

In this example,

- Find the value "x" raised to the power of "y" is done by "Math.pow (x,y)" Function
- Output of the example:-

Output : 8

vi) Math.sin (x), Math.cos (x), Math.tan (x).

It return the trignometer since, consine and tangent of x.

example

```
let x = Math.PI/4 ;
```

↓

Angle in radian

```
let sinvalue = Math.sin(x);
console.log(sinvalue);
```
→ (For Sine value)

```
let console = Math.cos(x);
console.log = (cosvalue);
```
→ (For cosine value)

```
let tanvalue = Math.tan(x);
console.log = (tanvalue);
```
→ (For tangent value)

Math.PI represent π/4 radians, which is equally to 45 degree.

vii) Math.sqrt(x)

It return the square root of the given number.

example

```
let x = 25
let square root = Math.sqrt(x);
console.log(square root);
```

→ Output : 5

If we calculate the square root of negative no. or non-numeric values the output will be "NAN" (Not a Number).

Date: ...../...../.....
Page No.: .....

→ JavaScript BOM

JavaScript BOM stands for "JavaScript Browser object Model". It is a set of objects and APIS (Application Programming Interfaces) that are made available by web browser to let JavaScript programmers communicate and interact with the browser window and its element.

The "window", "document" are the main objects to which access is granted Developers may carry out operations including altering the DOM (Document Object Model), managing browser history, managing the browser window, and acquiring data about the browser window, and acquiring data about the browser and the user's system thanks to these objects, which stand for various elements of the browser environment.

Let's learn about the main objects of BOM in brief.

i) Window object

It acts as the BOM's entrance and resembles the browser window or tab. It offers way to manage child window, handle events, control the window its self, and more via its methods and attributes.

We have mainly five Methods of window object.

i) alert ()
ii) confirm ()
iii) prompt ()
iv) open ()
v) Set Time out ()

1) alert ()

It display a dialog box with a message and an OK Button. It is mainly used for provide simple notification or alert to the user.

example

```
<body>
    <button onclick="Show alert()">Click Me
    </button>

<Script>

        Function Show alert () {

            alert ("Button is clicket");
    }

</Script>

</body>
```

Date: ..../..../....
Page No.: .....

ii) Confirm ()

It display a model dialog box with a message and two button, "OK" and "Cancel". It is mainly used to get user confirmation.

example

```
<body>
    <button onclick="confirmaction()">
        Delte </button>

<Script>
        Function confirmation () {

  var result = confirm ("You want to Delete?");

      if (result) {
            alert ("Deleted");

      } else {
            alert ("No Deleted");

        }
  }

</Script>

</body>
```

iii) Prompt ()

It is dialog box to the user with a message, an input field, and button for "OK" and "Cancel".

It is mainly used for get the input from user.

example

```
<body>
    <button onclick="Show prompt ()">Click
                Me </button>

<Script>
        Function showprompt () {

  var name = prompt ("Enter Your Name");

    if (name != null && name != = " ") {
            alert ("Hello ," + name + "!");

    } else {
            alert ("you not Enter Name");

    }
}

</Script>

</body>
```

Date: ..../..../....
Page No.: .....

iv) Open ()

It is used to open a new browser window or tab programtically.

example

```
<body>
    <button onclick="open new window ()">
        Click Me

  </button>

<Script>
        function onenew window () {

        open ("https://www example.com");

    }

</Script>

</body>
```

v) Set Time out

It allows you to Schedule the execution of a function after a specified delay. It is mainly used for time delay.

Date : ...../...../.....
Page No. : .....

example

```
<body>
    <h1 id = "message"> Hello </h1>
    <button onclick = "delay message()"> click
                                        Me
    </button>

<Script>
        Function show message () {

var message = document.get Element + By Id
                ("message");
    message.inner HTML = " After 2 seconds";
}

    Function delay message () {

        Set Time out (Show message, 2000)

    }

</Script>
</body>
```

Delay the Show message Function by 2000millise and (2 seconds

Date: ...../...../.....
Page No.: .....

ii) document object OR JavaScript DOM

A HTML document's structures and content are represented as a tree-like structure using the DOM programing interface, where each documents element is represented as a node.

JavaScript can communicate with and after the elements, attributes and text contained within a page thanks to the DOM,

The DOM offers number of methods and a attributes that enables programmer to access and dynamically change the organization, content and design of webpages. you may add or remove event handers, edit an elements properties, create, delete or modify an elements text content using JavaScript.

we can access the content of document by its methods, DOM have Four main methods for access documents.

Here is the methods.

i) get Elements By Tag Name ()
ii) get Elements By Id ()
iii) get Elements By Class Name ()
iv) get Elements By Name ()

Date : ...../...../.....
Page No. : .....

i) get Elements By Tag Name

The Document Object Model (DOM) in JavaScript provides the "get Elements By Tag Name" Function, which enables you to obtain a group of HTML elements according to their tag name.

example

```
<body>
    <h2> I am Heading </h2>
    <button onclick" change color()">
        Change </ button >

< Script >
        Function changecolor() {

  var heading = document . get Elements By
                Tag Name ("h2");

For (var i=0, i< heading . length; i++) {

    headings [i] . style . color = "red";

  }
}

</ script >

</ body>
```

Here we can target our <h2> , heading

Date: ..../..../....
Page No.: .....

ii) get Elements By Id ()

"get Elements By Id()" Function, enables you to obtain an HTML element from the document by providing its unique identifier, also known as the "id" attribute.

example

```
<body>
    <h1 id = "heading" > Hello, Pawan <h1>
    <button onclick = "changetext()" > Change Text
    </button>

<Script>
        Function change text () {

  var heading Element = document.get Element
                        by Id ("heading");
    heading Element. text content = "Text
                        Changed";
  }

                        Here the target is
                        completed or done
</Script>                   OR
                        Here we target the
</body>                     id - heading
```

In this example,

> HTML heading element "<h>" with an id of "heading" and Also there is button which is used for as a trigger

For change the color of text when clicked

- The element with the id "heading" is located using "getElementsBy Id()" by the JavaScript method "Changetext()", which then assign it to the variable heading element

- The "heading" elements' text content is then changed by setting the new value. "Text changed" to its text content attribute.

[4] get Elements By Class Name ()

Like "get Element By Id()", "get Element By Class Name()" is also enables you to obtain an HTML element from the document by providing it unique identifier, also known as "class" attribute.

example

```
<body>
    <h1 class = "heading"> Hello, Pawan <h1>
    <button on click = " change Text()">
                        Text Change
    </button>
```

Date : ...../...../.....
Page No. : .....

```
< Script >
        Function change text () {

Varheading Element = document . get Elements By
                        Class Name ("heading");

  heading . Element . text content = " Text changed";

  }

</ Script >

</ body >
```

The description of this example is also same as "get Elements By Id ()", just a small change here "id" replace with "class".

iv) get Elements By Name ()

The Document object Model (DOM) in JavaScript also has the "get Elements By Name ()" Function. Instead of using a collection of HTML elements, class or id you may access them based on the "name" property.

Here is the example, →

Date :
Page No. :

example

```
<body>

    <input type="text" name="username">
    <input type="text" name="email" >
    <input type="text" name="phone" >

  <button on click="display values()" > Display
    </button >

<Script >
        Function display values () {

var username = document.get Elements By Name
            ("username") [0];
var email = document. get Elements By Name
            ("email") [0];
var phone = document. get Elements By Name
            ("phone") [0];

  Here is the console's For output

console. log ("username:" + username. value);
console. log ("email:" + email. value);
console. log ("phone:" + phone. value);
}

</Script>

</body>
```

Date: / /
Page No.:

In this example,

- we have three different types element with different "name" attributes: "username", "email", "phone".

- A collection of items is returned by the "getElementsByName()" Function, thus we retrive specific elements by index. We use [0] to retrive the first member of each collection in this because we believe there is only one element with each name.

→ Closures

Closures are a sharp concept in JavaScript that describes the union of a function and the semantic context in which ~~text~~ that Function was defined. To put it another way, a closure enables a function to remember and access its own scope in addition to the variables and parameter of its parent Function even after the parent Function has been done running.

example

In this example we create a counter that increments each time is when we clicked button.

Date : ..../..../....
Page No. : .....

```
< body>
    <button id ="btn"> Increment </button>
    <p id ="counter" > 0 </p>

< Script >
        Function create counter () {
            var count = 0;

        Function increment () {
            count ++;
        document . get Element By Id("counter")

                    . innerText = count ;

        }

            return increment ;
    }

    var counter = create counter ();
    document . get Elements by Id ("btn")

        . add . EventListener ("Click , counter);

  </ script >
</ body>
```

This example shows how closures may be used to preserve information and encapsulate logic, offering a useful method for including interactive features in web applications.

Date : / /
Page No. :

explanation about example,

In this example,

- The counter mechanism is included within the "creat counter" Function, which is defined in the JavaScript code. Both the variable "count" and nested Function "increment". are created.

- The "increment" Function raises the "count" variable by one and modifies the "<p>" elements content to reflects the new count.

- The "increment" Function is returned by the "create counter" method, creating a clause. Even after the execution of "create counter" is complete, the returning Function continues to have access to the variable "count".

- "create counter" is executed in the script section, and the function that is returned is then assigned to the variable "counter". The "counter" Function is then invoked each time the button is Pressed by adding an event listener to the button element.

- The "count" is increased by one when the button is clicked using the "increment" Function.

Date: / /
Page No.:

→ Prototype

Every JavaScript object has a prototype attributes that points to its prototype object. The prototype object is just another object like any other, with the potential to have unique attribute and Functions.

Javascript initally looks for a property and Function in the object itself which you attempt to access one. If it can't discover it, then searches the prototype objects prototype chain for the property or method.

example

Constructor Function

```
<script>

Function Person (name) {
    this.name = name;
}

Person.prototype.sayHello = Function() {
    console.log("Hello, am" + this.name);
}
```

Adding Method to the prototype of Person

```
var person 1 = new Person ('Pawan');
var person 2 = new Person ( Preet);
```

↕

Creating objects using the Person constructor

```
person 1. say Hello ();
person 2. say Hello ();
```

</script>

Calling the method, Define in the Prototype

Prototype allow you to accomplish inheritance and let numerous objects share similar attribute and functions, which saves memory and encourages Javascript code reuse.

→ Encapsulation

Encapsulation in JavaScript refers to the practise of combinning data and methods into single entity known as an object. By doing so, you can structure and organisms your code and manage who has access to the attributes and methods of an objects.

example

```
<body>

  <button onclick = "say hello.( )"< click </button>

<script>
        var Person = {
```

Define object using object literal syntax

```
        name: "Pawan"
```

Private data property

Public Method

```
        greet : Function ( ) {
                alert ("Hello, am"+ this.name);
        }
    };

    function say hello ( ) {
        person.greet ( );
    }
```

Access the Public Method

```
</script>
</body>
```

Better Code organization and security are provided through encapsulation, which enable us to hide the implementation specifics and expose only the essential public interface.

Date : ..../..../....
Page No. : .....

→ Inheritance

A key idea in object oriented programming (OOP) is inheritance, which enables objects to take an characteristics and functions from other objects. The attribute and methods of an object can be inherited in JavaScript from prototype object.

example

Parent class Constructor

```
< script >
    function Fruit (name, color) {
        this.name = name;
        this.color = color;
    }

Fruit.prototype.introduce = function () {
    console.log ("Am" + this.color + " " + this.
                name + " ");
};
```

Parent class Method

```
function Apple (color) {
    Fruitcall (this, "Apple", color);
}
```

Child class constructor 1

Date : ..../..../....
Page No. : ....

```
Apple.prototype.object.create(Fruit.prototype);
Apple.prototype.constructor = Apple;
```

Inherit From the Parent class

```
Apple.prototype.taste = Function () {
    console.log ("crunchy and sweet");
};
```

Child class Method 1

```
Function Banana (color) {
    Fruit.call (this, "Banana", color);
}
```

Child class constructor 2.

```
Banana.prototype = object.create (Fruit.prototype);
Banana.prototype constructor = Banana;
```

Here we Inheritance From the Parent class

```
Banana.prototype.taste = Function () {
    console.log ("Soft and creamy");
};
```

Child Class Method 2.

```
var redApple = new Apple ("red");
var yellowBanana = new Banana ("yellow");
```

Create instance of the child classes

```
redApple.introduce ();
yellowBanana.introduce ();
```

Call Inherited Method

```
redApple.taste ();
yellowBanana.taste ();
</script>
```

Call Child Class Method

JavaScript's inheritance Feature is a helpful tool for writing modular, reusable, code that is easy to maintain. It encourages code - organization, reuse and polymorphism, all of which help to promote more effective and scalable development.

Date : ...../...../.....
Page No. : .....

→ Polymorphism

In JavaScript the term "Polymorphism" refers to an object or Function's Capacity to assume several kinds or Forms. Through method overloading and over-riding, you may design objects or Functions that can act differently based on the situations in which they are utilized.

It is allows you to use a single interface to represent different types of objects.

example < Script >

```
( Function shape () {
  }
)  ←  [ Base Class ]

( Shape.prototype.calculate area = Function () {
        return 0;
  }
)
```

[Method to calculate the area
(It will be over-riding by subclass)]

Date : ...../...../.....
Page No. : .....

```
Function Rectangle (width , height) {
        this. width = width;
        this. height = height;
}
```

Rectangle Sub-class

```
Rectangle . prototype = object . create (Shape. prototype);
Rectangle . prototype . constructor = Rectangle;
```

Inherit From the shape class

```
Rectangle . prototype . calculatorea = Function () {
        return this . width * this. height;
}
```

Overriding the calculate area method for Rectangle

```
Function circle (radius) {
        this. radius = radius;
}
```

Circle Subclass

```
circle.prototype = object.create(shape.prototype);
circle.prototype.constructor = circle;
```

Inherit From the shape class

```
circle.prototype.calculateArea = function() {
    return Math.PI * this.radius * this.radius;
```

Over-riding the calculate method of circle

```
var rectangle = new Rectangle(15);
var circle = new circle(9);
```

Creating instance of Rectangle and circle

```
console.log(rectangle.calculate area());
console.log(circle.calculate area());
```

Calling the Calculate Area method on different shapes

→ Abstraction

Abstraction is the technique of keeping consumer from seeing anything. But the most important features or function while concealing complicated implementation details. By offering a streamlined interface for interfacing with complicated systems, it enables developers to produce reusable and modular code.

- Let's imagine a situation where we need to use HTML codes to abstract a slideshow component in JavaScript. The objective is to provide a straight forward interface for managing and creating slideshows while concealing the implementations inner workings.

example

```
<body>

  <div id="slideshow"> </div>
  <button on click="createslideshow()"> slideshow</
                                        button>
  <button onclick="addslide()"> Add slide </button>

<script>
      const slideshow = {
slideshow Element: document get Element By Id
                    ('slide show') slides: [ ],
```

Abstractions layer for slide show Functionality

```
create : Function () {
  this slideshow Element . inner HTML
                    = 'slideshow created';
}
```

create slideshow logic

```
addslide : Function () {

const slide = document . create Element ('div');
slide . innerText = 'New slide';
this . slides . push (slide);
this . slide show element . append child (slide);
}

};
```

Add slide logic

```
Function create slideshow () {
    slide show . create ();
}

Function addslide () {
    slideshow . addslide ();
}
```

This is simple example of abstraction. Abstraction may include more complicated Functions or interaction.

Date: ..../..../....
Page No.: .....

→ Events

User interaction or browser events can cause JavaScript events, which are actions or occurance. They make it possible for JavaScript code to react to keyboard of input, form submission, mouse clicks, and more. Event listener are inline event characterstics can be added to HTML elements to handle events like click, submit and key down.

A callback function is performed when an event happen, enabling programmer to carry out particular tasks or dynamically change the website. Websites becomes more interesting and interactive thanks to events, which enables JavaScript to react to user interaction.

We have four different types of Events

i) Mouse Event
ii) Keyboard Event
iii) Form Events
iv) Window Events

Let's learn about these events in brief.

Date : / /
Page No. :

i) Mouse Events

Mouse Events are that type of event which is triggered by user interaction with the mouse. Because of these occasions, JavaScript code may react to operation like - clicking, hovering and moving the mouse cursor across HTML component.

Here the some commonly used types of Mouse Events.

- click
- mouse over and mouse out
- mouse down
- mouse up

Click

It occurs when a mouse button is clicked.

example

```
<body>
    <button on click ="click()"> Click Me </button>
<script>
    Function click () {
        alert ("Button clicked");
    }
```

```
</script>
</body>
```

- when the button is clicked, the "Click ()" Function is triggered, and an alert dialog box pops up displaying the message.

→ mouse over and mouse out

"mouse over" – It is triggered when the mouse pointer moves over an element.

"mouse out" – It is Fixed when the mouse pointer leave an element.

example

small use of CSS, just for explain

```
<head>
    <style>
        .highlight {
            background-color : yellow;
        }
    </style
</head
<body>
    <h1 id = "my heading" = Heading </h1>
```

Get the heading element

```
< script >

var heading = document.getElementById
                ("my heading");

heading.addEventListener ("mouseover",
        Function () {
    heading.classlist.add ("Highlight");

  });
```

Add Event Listener For the mouse over Event

```
heading.addEventListener ("mouseout", Function() {
    heading.classlist.remove ("highlish");

  });
</ script >
</ body >
```

Add Event Listener For mouse out Event

In this example

- The "mouse over" event is trigged whenever the mouse cursor passes over a heading, and the callback Function when applies the CSS class "highlish to the heading element, giving it a background color of yellow.
- The "mouse out" event is trigged when the

mouse pointer leaves the heading, and the callback method takes away the "highlight" class to bring the original style.

→ mouse down

It occurs when a mouse button is pressed down.

example

```
<body>
    <button onmousedown="mousedown()">click me
    </button>

<script>
        Function mousedown () {
            alert ("Button clicked");
    }
</script>
</body>
```

In this example,

→ when the user clicks the button the "mousedown ()" Function will be executed displaying an alert with the message.

Date : ..../..../....
Page No. : .....

→ mouse up

It is triggered when a mouse button is released.

example

```
<body>
  <button id="button">Click Me</button>
<script>
    var button = document.getElement by Id
                ("button");

  Function mouse up (event) {
    console.log ("Button released");
  }

    button.add Event Listener ("mouseup",
                mouseup);
<|script>

<|body>
```

Get the button element

Define the Event Handler Function

Attach the Event Listener

Date : ...../...../.....
Page No. : .....

ii) Keyboard Events

Keyboard Event are actions taken by the user on the Keyboard. These events give programmers the ability to recognise and react to Keyboard input, including Key presses, Key releases, and other related activites.

Here are same mainly used of Keyboard events.

- Keyboard
- Key up
- Key press

- "Key down" → This event is Fired when a Key is intially pressed down.

- "Key up" → This event is fired when a Key is released.

- "Key press" → This event is Fired when a Key is Pressed down and generates a character value.

Date :
Page No. :

example

```
<body>
    < input type ="text" id = "input" >
< script >
    var input =document . getElement By
                Id ("input");

    Function handle keydown (event) {
        console. log ("Key down. "event key);
    }

    Function handle keyup (event) {
        Console . log ("Key up:" event . Key);
    }

    Function handle Keypress (event) {
        console. log ("Key press :"+ event. key);
    }

input. add Even Listener ("Keydown", handle Keydown);
input. add Even Listener ("Key down", handle keyup);
input. add Even Listener ("Key down", handle Keypress);
< /script >
< /body >
```

Get the input element

Define the event handler Function

Attach the Event Listener

Date : ...../...../.....
Page No. : .....

iii) Form Events

When a user interacts with HTML Form components, Form events in JavaScript takes place. These events give programmer the ability to recognise and react to the variety of activites, like submitting a form, altering input values, concentrating on or removing haze from input areas and more.

Form events are very helpful for regulating Form behaviours, verifying user input and carrying out actions on user interactions.

Here are some mainly used Form events.

"Submit" - When a Form is submitted, either by pressing the submit button or Enter inside by input field, this event is started.

"input" - When the value of an input element changes, this event is triggered. It is frequently employed for updating dependent fields or performing real-time validation.

Focus - When an input element receives Focus, usually as a result of a click or keyboard navigation, this event takes place.

"blur" – Because an input element lose's focus typically because a user clicks outside the field or switches tabs – this event is launched

example

```
<body>
    <Form id = "Form" >

<input type ="text" id="name" placeholder ="Enter Name" >
<input type ="email" id="email" placeholder ="Enter Email">
<input type ="submit" > submit </button >

    </Form>

</script >
```

Get the Form and input Elements

```
var Form= document.get Element By Id ("Form");
var name = document.get Element By Id ("name");
var email = document.get Element By Id ("email");
```

Define the event handler

Function

```
Function handlesubmit (event) {
    event.prevent Default ();
    console.log ("Form submitted");
}
```

Perform Form validation and Submit Data

```
Function handle Input (event) {
  console.log ("Input value changed" + event.taget.
                                        value);
}

Function handle Focus (event) {
  console.log ("Input Focused" + event.taget.id);
}

Function handle Blur (event) {
  console.log ("Input blurred" + event.taget.id);
}
```

Attach the Event Listener

```
Form.add Event Listener ("submit", handle submit);
name.add Event Listener ("input", handle Submit);
email.add Event Listener ("Focus", handle Focus);
email.add Event Listener ("blur", handle Blur);
```

```
</script>
</body>
```

Date : ...../...../.....
Page No. : .....

iv) Window Events

Window Events in JavaScript are those relating to the page as a whole or the browser window. These actions at the window level, such as page loading, window resizing and window scrolling, cause these events to be fired.

Here are some common window events.

"load" – when the web page and all of its resources have full loaded, this event is triggered.

"resize" – when the user changes the size of the browser window, the event takes place.

"scroll" – when the user scroll inside the window, this event is triggered.

example

```
<Script>

    Function handleload (event) {
        console.log ("Page loaded");
}
```

```
Function handleResize (event) {
    console.log ("window Resize");
}
    Function handleScroll (event) {
    console.log ("window scrolled");
}

window.addEventListener ("load", handleLoad);
window.addEventListener ("resize", handleResize);
window.addEventListener ("scroll", handleScroll);

</Script>
```

→ Event Handling

The process of establishing and managing events that take place on a webpage is known as handling events. An event is an activity or occurrence that occurs, such as when a user presses a key, moves the mouse, or clicks a button. Event handling enables you to react to these occurances and run certain codes or procedures.

We can handle events with two steps.

i) Event Registration
ii) Event Handling Function

Date : ...../...../.....
Page No. : .....

i) Event Registration

In the first step, you need to register an event listener or handle function to an HTML element. This is a done by using the "add Event Listener ()" method.

example

```
<body>
  <button id = "button"> Click Me </button>

<script>

    const button = document.get Element By Id
                                ("button")
```

Get the Button Element

Event Registration

```
    Button.add Event Listener (Click, Function
                                () {
        alert ("Button Clicked");
    });
```

Date : / /
Page No. :

ii) Event Handling Function

In second step, you give a function that will run when the event happens. The event handler or callback Function is the frequent name for this Function. It has the code that should be executed when the event is triggered.

example — Event Handling Function

```
<script

button.addEventListener('Click', Function(){
        alert('Button Clicked');

});
```

→ Error Handling

Error handling is the act of responding to unusal or unexpected events that may arise while a programme is being executed. Errors can occur For a number of reasons, including bad user input, network problems, or code errors, By identifying and resolving these Problems, proper error handling raises the dependability and stability of JavaScript programming.

Date : ...../...../.....
Page No. : .....

Here is the main error handling teachnique

i) try - catch statement
ii) throw statement

i) try - catch statement

The "try - catch" statement is basic component of JavaScript's error handling system. It enables you to manage possible mistakes or exceptions that can happen while a block of code is being executed,

The code that could throw an error in the "try" block and the "catch" block is used to capture and handle the problem if it happens.

example

Code that may throw an error

```
<script>
    try {
        var result = some Function ();
        document.write ("Result :" + result);
    } catch (error) {
        document.write ("An Error occured :
                        "error.message);
    }
</script>
```

Error Handling Code

11) throw statement

In JavaScript, an execution or Error can be manually thrown with the "throw" command. It enables you to expressly state that your code has encountered an uncommon circumstances.

The execution of a "throw" statement interupts the usual flow of the program and jumps to the next enclosing "try — catch" block or the global error handling mechanism.

example

```
<body>
<input type="text" id="input field" place
                    holder="Enter value" >
<script>
        Function validatinput (input) {
            if (!input) {
                throw new Error ['Input is
                                  Not Empty'];
}
try {
    const userinput = document.getElementById
        ('input Field').value;

    Validate input (useinput)
    console.log ('Input is validated');
```

Here we throw an error

```
catch (error) {
    console.log ('Error Occurred; error.message);
}

</script>

</body>
```

→ Higher Order Function

A JavaScript Function that may take other functions as input or return other Function as results in known as a higher - order Function.

It treats Function like values, enabling more modular and flexible programming. Code may be made more reusable and simpler by abstracting common behaviour into higher - order Function.

example

```
<body>

<button id ="button"> Click Me </button>

<script>
```

Date : ...../...../.....
Page No. : .....

```
Function add button click Listener (button Id, call back){
const button = document. get Element by Id ('button');
    button . add Event Listener ('click , call back);
}
```

Higher order Function that add click event listener to button.

```
Function handle click () {
    console. log ('Button Clicked');
}
```

Here is the call back Function while is executed when the button is clicked.

```
add Button Click Listener ('button, 'handleclick)

</Script>
```

Higher order Function which is used for click event listener

→ Modules

Modules are independent pieces of code that contain a certain type of Functionality. They support, reusability, encourage, organization and structure in the code, and improve co-operations and code maintenance. Files group of files, or package that contain related code are all examples of modules

By dividing issues and presenting a straight Forward interfaces for dealing with the functionality of the Module, they let developers design modular, modular - applications.

example

In this example, you learn that how we use modules in Javascript.

```
<head>
    < script  src = "main module.js" > </script>
    < script  src = "greeting module.js" > </ script>
    < script  Src = "display module.js" > </script>
</head>
```

Here we attach these Modules or js Files with HTML

Date : ...../...../.....
Page No. : .....

→ greeting module .js : Module 1.

```
const greeting = 'Hello';

Function greet (name) {
    return greeting + ' ' + name + '!';
}

expert { greet };
```

Here is the First module "greeting module.js" which expert a "greet" Function that generates or greeting message.

→ display module.js : Module 2.

```
Function display(message) {
const output Element = document.get Element by Id ('message');

Output Element.text content = message;
}
```

Here is the Second Module "display module.js" which expert a "display message" Function that displays a message on the webpage.

Date : ...../...../.....
Page No. : .....

→ main.js : Module 3.

```
import { greet } from './greeting module.js';
import { display message } from './display module.js'


const name = 'Pawan';
const message = greet (name);
display message - (message);
```

Here is our third or Final module "main.js" which import the "greet" and "display" message" Functions from "greetingmodule.js" and "displaymodule.js respectively.

"main.js" uses those functions to generate a greeting message.

Date : ...../...../.....
Page No. : .....

→ Asynchronous Programming

Javascript's Asynchronous Programing paradign enables you to run multiple tasks at once without blocking the main program from running. It is especially helpful for managing time - consuming process like network requests, database, queries and input | output files.

JavaScript's typically uses a synchronous execution paradigm, which means that statement is carried out one at a time while stopping the execution until the current action is finished.

On the other hand Asynchronus code, can pause delay operations and make the user interface unresponsive or unusable in situations when you must carry out lengthy activities.

For settle this situations, in Java Script, Asynchronus Programming provide "Callback" and "promises".

→ "Callback"

When a certain event or task is finished, a Callback function is triggered. It is supplied as an argument to another function.

Date : ...../...../.....
Page No. : .....

In Asynchronous setting, callbak's are great method to make sure that certain function run when they should.

example

```
Function processresult (result, callback) {
  console.log ('Processing result;' + result);
```

Example of function that's support CallBack

```
Settime out (Function () {
var process Result = result.to upper case();
  Call back (processed Result);
} 3000;
}
```

Time out Based Asynchronous operation based

```
Function handle process Result (result) {
  console.log (Handled Result:' + result);
}
```

An Argument passing callback Function

process result ('Hello, Pawan', handleprocessed result);

using the call back to call the function

Why use callback ???

The handling of asynchronous actions, flexibility error handling, event - driven programming. Flow management and the promotion of modular code are all mode possible via Callbacks in Javascript.

[1] "Promises"

Promises are object that show how an asynchronous action turned out. They enable a more understandable and controlable code flow by giving an organized approach to handling asynchronous operations.

There are three possible states for promises is, fullfilled, pending or rejected.

In order to manage successful completion or error handling, you may attach callback to them. By separting concerns and fastering modular programming promises make it easier to link together several asynchronous activites and offer an alternative to convention all

Callback based coding, which improves code organisation and readability.

Function that simulate an asyncheanus activity by returing promise

example

```
Function Fetch Data () {
return new Promise (Function (resolve, reject) {
    Set Timeout (Function () {
        Var data = "Fetched Data";
        }, 4000;
    });
}
```

The Promise Resolve with the data

Using the time log to mimic an asynchronous

Simulate a 4sec delay

```
Fetch Data ()
  .then (Function (response) {
      console.log ('Data Added', response);
  })
  Catch (Function (error) {
      console.log ('Error', error);
  });
```

Date : ...../...../.....
Page No. : .....

→ Design Pattern

Reusable solution to basic software design - issues are known as design pattern. They offer standardized methods for organizing and structuring code in order to raise its quality, maintainability and scalability.

Examples include the 'Singleton pattern' for producing a single instance of a class, the "observer pattern" for establishing object - dependences, the "module pattern" for encapsulation.

Let's learn about these pattern in brief.

i) Singleton Pattern

A Singleton instance of a class is provided by the singleton pattern, which also offer a single point of access to it globally.

It is useful when you wish to limit the number of objects that may be created by an instance of a class, such as when managing shared resources or keeping track global data.

Date : ...../...../.....
Page No. : .....

example

```
var singleton = (Function () {
    var instance ;
```

Reference of the Singleton instance

```
Function init () {
    Function private methods () {
        Console . log ('Private Method');
}
```

Here is the Private Method

```
var private variable = 'Private variable'
```

Here is the Private Variable

```
return {
    public method: Function () {
        Console . log ('Public Method');
}
    public variable : 'Public variable'
};
}
```

Here is the Public variable

Here is the Public Method

```
return {
        get instance : Function () {
            if (!instance) {
                instance = init ();
        }
        return instance;
    }
};

})();
```

Here we get the singleton instance
OR
if it doesn't exist then create it

```
var singleton Instance 1 = singleton.get Instance()
var singleton Instance 2 = singleton.get Instance();
```

Here is the usage

```
console.log (Singleton Instance 1 = singleton Instance 2);
Singleton Instance 1. public Method ();
console.log (Singleton Instance 1. public variable);
```

Here is the console and public method for outputs.

Date : ...../...../.....
Page No. : .....

11] Observer Pattern

The Publish pattern, Subscribe pattern are known as observer Pattern, creates a one-to-many dependence between object. All of an object's dependent objects (the observer) are related and updated automatically if the status of one item (the Subject) changes.

The subject and observer relationship can be flexible when using the observer pattern. The subject is to need to be aware of the specifics of the observer and the subject and other observer are unaffected by the addition or removal of new observer. It encourages modularity and makes the code base's expansions and maintenance easier.

example

```
Function Subject () {
    this.observer = [];
```

```
Main opening Function for all
observer
```

```
this.add observer = function (observer) {
    this.observer.push (observer);
```

Function for add observer

```
this.remove observer = function (observer) {
    const index = this.observer.index of (observer);
    if (index !==-1) {
        this.observer.splice (index, 1);
```

Function for remove observe

```
this.notify observe = function (data) {
    this.observer.For Each (function (observer) {
        Observer.update (data);
    });
};

}
```

Function For notify observer

usage

```
const subject = new Subject ();
const observer1 = new Subject ();
const observer2 = new Subject ();
```

Date : ...../...../.....
Page No. : .....

```
Subject.add observer (observer 1);
Subject.add observer (observer 2);
```

```
Subject.notify observer ('Hello observer');
```

used for notify "Hello observer" to observer

```
Subject.notify observer ('~~Good Bye~~' observer);
```

used for ~~giving notify after~~ remove the observer

```
Subject.notify observer ('Good Bye');
```

used for giving notify after remove the observer

III) Module Pattern

The module Pattern is creates a single Module out of a group of connected variables and functions, keeping them private but giving a public interface for ~~accese~~ accessing them.

we may design self-contained modules that encourage encapsulation and information hiding using the module pattern. It offers a systematic method of organizing code that helps in preventing global namespace damage.

example

```
var module =(function () {

var private variable = "Private variable";

function private function () {
    console.log ('Private function');
}
    return {
        public variable : 'Public variable';

Public function : function () {
    console.log ('Public function');
    console.log ('private variable');
    private function ();
}

};

})();
```

Here we Access the private variable and function.

Date : ...../...../.....
Page No. : .....

Usage

```
console.log (Module.public variable);
Module.public Function ();
```

→ Functional Programming

Pure Functions, Functions, composition, immutability are the main points of the Programming paradigm known as Functional Programming. In order to support higher-order functions and function chaining, it points out treating functions as First class citizens. Functional programing guides clear of changeable states and side effect and encourages the use of pure functions and immutable data structures that provide predictable outcomes purely based on their inputs.

The goal of functional programming is to use these ideas to produce code that is simpler to understood, test and reason about when promoting modularity and reusability.

Functional Programing promotes a more clear functional style of programing.

that places a focus on function purity and composition.

Here is the main types of Functional Programming.

i) Purse Function
ii) Map Function
iii) Filter Function
iv) Function Composition

i) Pure Function

Pure Functions enable code that is simpler to learn and maintain, It is more predictable and simpler to test.

It is used for produce the same output from the same inputs and It has no side effects.

A Pure Function does not alter the system in any way, it just depends on the input parameter and delivers a result.

example

```
Function add (a, b) {
        return a+b;
}
```

Date : ...../...../.....
Page No. : .....

ii] Map Function

Map Function is mainly used for transform each element of any array into a new array, without any changing in the original array. It accepts a callback functions are an input and applies it to every element before producing a new array with the values that have been changed.

example

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

const double numbers = numbers.map((num) => num2);
```

Output: [2, 4, 6, 8, 10]

iii] Filter Function

Filtering Function is mainly used for create a new array that contains only that elements which passes from a certain condition which is specified by Callback Function.

Date : ...../...../.....
Page No. : .....

example

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

const even numbers = numbers . Filter ((num) =>
                                num % 2 = = = 0);
```

iv) Function Composition

The Process of mixing different Functions to produce new function is known as Function composition. It involves using a series of Function calls by sending the output of one Function as the input to another.

example

```
const add = (a, b) = a+b;
const multiply By 3 = (x) = x* 3;
const Subtract = (a, b) => a-b;

Const composed Function = compose (Subtract,
                                   multiply by 3, add);

Const result = composed Function (6, 2)
```

→ The input after mixing the function is

$\{(b+2)^{\star} 3 - 3 = 21\}$