JAVASCRIPT NOTES

❖ What is JavaScript ?

JavaScript and Java are completely different languages, both in concept and design.

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

ECMA-262 is the official name of the standard. ECMAScript is the official name of the language.

- Used to provide functionality.
- Can provide logic.
- Can create file by using .js extentions

You don't have to get or download JavaScript.

JavaScript is already running in your browser on your computer, on your tablet, and on your smart-phone.

JavaScript is free to use for everyone.

JavaScript using consol

Consol works on this concept-

Using REPL-read evaculate print loop

Consol used for practice purpose. It would not going to save it it is temporary

Ex:- open the consol and try basic calculation like 1+2

Wanted to clear consol :- ctrl+L for windows & ctrl+k for mac

Upword arrow used to see previous calculations

JavaScript Can Change HTML Content

One of many JavaScript HTML methods is getElementById().

The example below "finds" an HTML element (with id="demo"), and changes the element content (innerHTML) to "Hello JavaScript"

```
<h2>What Can JavaScript Do?</h2>

<p id="demo">JavaScript can change HTML content.</p>

<button type="button"
onclick='document.getElementById("demo").innerHTML = "Hello
JavaScript!"'>Click Me!</button>
```

- **The <script> Tag**

In HTML, JavaScript code is inserted between <script> and </script> tags

```
<script>
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
```

JavaScript in <head> or <body>

You can place any number of scripts in an HTML document.

Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both.

- **JavaScript in <head>**

In this example, a JavaScript function is placed in the <head> section of an HTML page.

The function is invoked (called) when a button is clicked:

```
<!DOCTYPE html>
<html>
<head>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
</head>
<body><h2>Demo JavaScript in Head</h2>

<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>

</body>
</html>
```

- **JavaScript in <body>**

In this example, a JavaScript function is placed in the <body> section of an HTML page.

The function is invoked (called) when a button is clicked:

```
<!DOCTYPE html>
<html>
<body>
<h2>Demo JavaScript in Body</h2>
<p id="demo">A Paragraph</p>
<button type="button" onclick="myFunction()">Try it</button>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>

</body>
</html
```

❖ External JavaScript

Scripts can also be placed in external files:

External scripts are practical when the same code is used in many different web pages.

JavaScript files have the file extension .js.

To use an external script, put the name of the script file in the src (source) attribute of a <script> tag:

```
<script src="myScript.js"></script>
```

External scripts cannot contain <script> tags.

External JavaScript Advantages

Placing scripts in external files has some advantages:

It separates HTML and code

It makes HTML and JavaScript easier to read and maintain

Cached JavaScript files can speed up page loads

❖ JavaScript Display Possibilities

JavaScript can "display" data in different ways:

- Writing into an HTML element, using innerHTML.
- Writing into the HTML output using document.write().
- Writing into an alert box, using window.alert().
- Writing into the browser console, using console.log().

• Using innerHTML

To access an HTML element, JavaScript can use the document.getElementById(id) method.

The id attribute defines the HTML element. The innerHTML property defines the HTML content:

```
<body>
<h1>My First Web Page</h1>
<p>My First Paragraph</p>
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML = 5 + 6;
</script>
</body>
```

You can skip the window keyword.

```
<script>
alert(5 + 6);
</script>
```

❖ What is variable?

A variable is simply name of storage location.

Ex. tea powder can store in tea powders container so container used to store the tea powder same goes with code it has to mainted or stored in container called variable

X=10;

Y=20;

Area of rectangle=x*y=10*20

Here x and y is variable .

Name= “vaishnavi”;

Age=“24”;

Address: “navi Mumbai”;

❖ JavaScript has 7 premitive Datatypes

String, Number, Boolean, Undefined ,Null, Bigint,,Symbol

Ex:- a=20;

Typeof a

name="vaishnavi“

'vaishnavi‘

typeof name

'string'

```
// Numbers:
let length = 16;
let weight = 7.5;

// Strings:
let color = "Yellow";
let lastName = "Johnson";

// Booleans
let x = true;
let y = false;

// Object:
const person = {firstName:"John", lastName:"Doe"};

// Array object:
const cars = ["Saab", "Volvo", "BMW"];
```

- **NUMBER**

Number can be positive or negative ex:- 5 or -5.

Numbaer can be integer ex (45,-50)

Floating numbers with decimal value :- ex 99.99

- **BIGINT**

```
<p id="demo"></p>

<p>You cannot perform math between a BigInt type and a Number type.</p>

<script>

let x = BigInt("123456789012345678901234567890");

document.getElementById("demo").innerHTML = x;

</script>
```

- **UNDEFINED**

In JavaScript, a variable without a value, has the value undefined. The type is also undefined.

A variable that has not been asiigned to any value

```
<p id="demo"></p>

<script>

let car;

document.getElementById("demo").innerHTML =

car + "<br>" + typeof car;

</script>
```

Empty Values

An empty value has nothing to do with undefined.

```
<p id="demo"></p>

<script>

let car = "";

document.getElementById("demo").innerHTML =

"The value is: " + car + "<br>" + "The type is: " + typeof car;

</script>
```

• JAVASCRIPT ARRAYS

JavaScript arrays are written with square brackets.

Array items are separated by commas.

The following code declares (creates) an array called cars, containing three items (car names):

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

```
<h2>JavaScript Arrays</h2>

<p>Array indexes are zero-based, which means the first item is [0].</p>

<p id="demo"></p>

<script>

const cars = ["Saab","Volvo","BMW"];

document.getElementById("demo").innerHTML = cars[0];

</script>
```

• JAVASCRIPT OBJECTS

JavaScript objects are written with curly braces {}.

Object properties are written as name:value pairs, separated by commas.

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

```
<h2>JavaScript Objects</h2>

<p id="demo"></p>

<script>

const person = {

  firstName : "John",
  lastName  : "Doe",
  age     : 50,
  eyeColor  : "blue"
};

document.getElementById("demo").innerHTML =
person.firstName + " is " + person.age + " years old.";
</script>
```

- **JAVASCRIPT OPERATORS**

JavaScript operators are used to perform different types of mathematical and logical computations.

Examples:

The **Assignment Operator =** assigns values

The **Addition Operator +** adds values

The **Multiplication Operator *** multiplies values

The **Comparison Operator >** compares values

- **JAVASCRIPT ASSIGNMENT**

The Assignment Operator (=) assigns a value to a variable:

```
// Assign the value 5 to x
let x = 5;
// Assign the value 2 to y
let y = 2;
// Assign the value x + y to z:
let z = x + y;
```

- **JavaScript Addition**

The **Addition Operator** (+) adds numbers:

```
let x = 5;
let y = 2;
let z = x + y;
```

- **JavaScript Multiplication**

The Multiplication Operator (*) multiplies numbers

```
let x = 5;
let y = 2;
let z = x * y;
```

❖ Types of JavaScript Operators

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- String Operators
- Logical Operators
- Bitwise Operators
- Ternary Operators
- Type Operators

- **JavaScript Arithmetic Operators**

Arithmetic Operators are used to perform arithmetic on numbers:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)
++	Increment
--	Decrement

- **JavaScript Assignment Operators**

Assignment operators assign values to JavaScript variables.

The Addition Assignment Operator (+=) adds a value to a variable.

Operator	Example	Same As
=	x = y	x = y
+=	x += y	x = x + y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y

- **JavaScript Comparison Operators**

Operator	Description
==	equal to
===	equal value and equal type
!=	not equal
!==	not equal value or not equal type
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
?	ternary operator

- **JavaScript String Comparison**

All the comparison operators above can also be used on strings:

```
let text1 = "A";
let text2 = "B";
let result = text1 < text2;
```

```
<h1>JavaScript String Operators</h1>

<p>All conditional operators can be used on both numbers and strings.</p>

<p id="demo"></p>

<script>

let text1 = "A";

let text2 = "B";

let result = text1 < text2;

document.getElementById("demo").innerHTML = "Is A less than B? " +
result;
```

```
</script>
```

Explanation of the Comparison

In this example, "A" and "B" are compared using the < operator.

The Unicode value of "A" is 65.

The Unicode value of "B" is 66.

Since 65 is less than 66, the result of the comparison text1 < text2 is true.

- **JavaScript String Addition**

The + can also be used to add (concatenate) strings:

```
let text1 = "John";
let text2 = "Doe";
let text3 = text1 + " " + text2;
```

The += assignment operator can also be used to add (concatenate) strings:

```
<p id="demo"></p>
<script>
let text1 = "What a very ";
text1 += "nice day";
document.getElementById("demo").innerHTML = text1;
</script>
```

- **Adding Strings and Numbers**

Adding two numbers, will return the sum, but adding a number and a string will return a string:

```
<p id="demo"></p>
<script>
let x = 5 + 5;
let y = "5" + 5;
let z = "Hello" + 5;
document.getElementById("demo").innerHTML =
x + "<br>" + y + "<br>" + z;
</script>
```

❖ JavaScript Variables

Variables are Containers for Storing Data

JavaScript Variables can be declared in 4 ways:

- Automatically
- Using var
- Using let
- Using const

It is considered good programming practice to always declare variables before use.

- The var keyword was used in all JavaScript code from 1995 to 2015.
- The let and const keywords were added to JavaScript in 2015.
- The var keyword should only be used in code written for older browsers.

Example of var:

```
<p id="demo"></p>
<script>
var x = 5;
var y = 6;
var z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script>
```

Example of let:

```
<p id="demo"></p>
<script>
let x = 5;
let y = 6;
let z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script>
```

Example of const:

```
<p id="demo"></p>
<script>
const x = 5;
const y = 6;
const z = x + y;
document.getElementById("demo").innerHTML =
"The value of z is: " + z;
</script>
```

The two variables price1 and price2 are declared with the const keyword.

These are constant values and cannot be changed.

The variable total is declared with the let keyword.

The value total can be changed.

1.Always declare variables

2. Always use const if the value should not be changed

3. Always use const if the type should not be changed (Arrays and Objects)

4. Only use let if you can't use const

5. Only use var if you MUST support old browsers.

❖ JavaScript Identifiers

All JavaScript variables must be identified with unique names.

These unique names are called identifiers.

Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

The general rules for constructing names for variables (unique identifiers) are:

Names can contain letters, digits, underscores, and dollar signs.

Names must begin with a letter.

Names can also begin with $ and _ (but we will not use it in this tutorial).

Names are case sensitive (y and Y are different variables).

Reserved words (like JavaScript keywords) cannot be used as names.

Valid identifers

Short names:

```
var x = 10;
 var y = 20;
```

Letters and digits:

```
var user1 = "Alice";
 var score99 = 99;
```

Dollar signs:

```
var $price = 9.99;
var $discount = 5;
```

Starting with $:

```
var $element =
document.getElementById(
"example");
```

Underscores:

```
var first_name = "John";
var last_name = "Doe";
```

Case sensitivity:

```
var fruit = "apple";
 var Fruit = "banana"; // Different
variable
```

Starting with a letter:

```
var name = "Bob";
```

Descriptive names:

```
var age = 25;
var sum = 50;
 var totalVolume = 100;
```

Starting with _:

```
Var  _privateVariable =
"secret";
```

Invalid (using reserved word):

```
var for = "loop";                    // SyntaxError: Unexpected token for
```

Starting with a digit:

```
var 1variable = "invalid";              // SyntaxError: Invalid or unexpected token
```

•Containing spaces:

```
var my variable = "invalid";              // SyntaxError: Unexpected identifier
```

Containing special characters other than $ and _:

```
var my-variable = "invalid"; // SyntaxError: Unexpected token
 var @name = "invalid"; // SyntaxError: Unexpected token @
```

Starting with a special character other than $ or _:

```
var #id = "invalid"; // SyntaxError: Unexpected token #
```

Declaring a JavaScript Variable

Creating a variable in JavaScript is called "declaring" a variable.

You declare a JavaScript variable with the var or the let keyword:

ex:=

```
var carName; or let carName;
```

After the declaration, the variable has no value (technically it is undefined).

To assign a value to the variable, use the equal sign:

Ex:- carName = "Volvo";

You can also assign a value to the variable when you declare it:

```
let carName = "Volvo";
```

One Statement, Many Variables

You can declare many variables in one statement.

Start the statement with let and separate the variables by comma:

let person = "JohnDoe ", carName = "Volvo", price = 200;

- ## JavaScript Let

  The let keyword was introduced in ES6 (2015)

  Variables declared with let have **Block Scope**

  Variables declared with let must be **Declared** before use

  Variables declared with let cannot be **Redeclared** in the same scope

- ## Block Scope

  Before ES6 (2015), JavaScript did not have **Block Scope**.

  JavaScript had **Global Scope** and **Function Scope**.

  ES6 introduced the two new JavaScript keywords: let and const.

  These two keywords provided **Block Scope** in JavaScript:

Block-Scope	**Var**-Function-scoped	**Let**-Block-scoped	**Const**-Block-scoped

```
<script>
// var example
 if (true) {
     var x = 10;
}
 console.log(x); // 10 (function-scoped, accessible outside the block)

// let example
if (true) {
    let y = 20;
 }
//  console.log(y); // ReferenceError: y is not defined (block-scoped, not accessible outside the
block)
   Feature                var                    let                      const
// const example
if (true) {
    const z = 30;
    console.log(z);
}
// console.log(z); // ReferenceError: z is not defined (block-scoped, not accessible outside the
block)
</script>
```

Feature	var	let	const
Re-declaration	Allowed	Not allowed	Not allowed

```
  // var example
var a = 1;
var a = 2; // Allowed
console.log(a); // 2

// let example
let b = 1;
// let b = 2; // SyntaxError: Identifier 'b' has already been declared

// const example
const c = 1;
// const c = 2; // SyntaxError: Identifier 'c' has already been declared
```

Feature	var	let	const
Re-assignment	Allowed	Allowed	Not-Allowed

```
var d = 1;
d = 2; // Allowed
console.log(d); // 2

// let example
let e = 1;
e = 2; // Allowed
console.log(e); // 2

// const example
const f = 1;
// f = 2; // TypeError: Assignment to constant variable
console.log(f); // 1
```

Variables declared inside a { } block cannot be accessed from outside the block:

```
{
  let x = 2;
}
// x can NOT be used here
```

With let you **can not** do this:

let num = 5;

Value of num is 5
 num = 10;

Value of num is 10
//redecleared or update

Variables declared with the var always have **Global Scope**.

Variables declared with the var keyword can NOT have block scope:

```
{
  var x = 2;
}
// x CAN be used here
```

• JavaScript Const

The const keyword was introduced in ES6 (2015)

Variables defined with const cannot be Redeclared

Variables defined with const cannot be Reassigned

Variables defined with const have Block Scope

Cannot be Reassigned

A variable defined with the const keyword cannot be reassigned:

```
const PI = 3.141592653589793;
PI = 3.14;      // This will give an error
PI = PI + 10;   // This will also give an error
```

Must be Assigned

JavaScript const variables must be assigned a value when they are declared:

```
const PI = 3.14159265359;  //correct
const PI;           //incorrect
PI = 3.14159265359;
```

When to use JavaScript const?

Always declare a variable with const when you know that the value should not be changed.

Use const when you declare:

- A new Array
- A new Object

- **Const with array:**

```
<p id="demo"></p>
<script>
// Create an Array:
const cars = ["Saab", "Volvo", "BMW"];
// Change an element:
cars[0] = "Toyota";
// Add an element:
cars.push("Audi");
// Display the Array:
document.getElementById("demo").innerHTML = cars;
</script>
```

- **Const with object:**

```
<p id="demo"></p>
<script>
// Create an object:
const car = {type:"Fiat", model:"500", color:"white"};
// Change a property:
car.color = "red";
// Add a property:
car.owner = "Johnson";
// Display the property:
document.getElementById("demo").innerHTML = "Car owner is " +
car.owner;
</script>
```

Logical operators in JavaScript are used to combine or invert Boolean values and expressions. They are essential for creating complex conditions in control flow statements like if, while, and for. The main logical operators in JavaScript are && (logical AND), || (logical OR), and ! (logical NOT).

1. Logical AND (&&)

The logical AND operator (&&) returns true if both operands are true; otherwise, it returns false

Syntex:- expr1 && expr2

```
let a = true;
let b = false;
console.log(a && b); // Outputs: false
```

console.log(5 > 3 && 10 < 20); // Outputs: true

console.log(5 > 3 && 10 > 20); // Outputs: false

2. Logical OR (||)

The logical OR operator (||) returns true if at least one of the operands is true; otherwise, it returns false.

Syntex:-expr1 || expr2

let a = true;

let b = false;

console.log(a || b); // Outputs: true

console.log(5 > 3 || 10 < 20); // Outputs: true

console.log(5 > 3 || 10 > 20); // Outputs: true

console.log(5 < 3 || 10 > 20); // Outputs: false

- **Logical AND (&&)**: Returns true if both operands are true.
- **Logical OR (||)**: Returns true if at least one operand is true.

❖ Conditional statement

1. if Statement

The if statement executes a block of code if a specified condition is true

Syntax:

javascript

if (condition) { // code to be executed if condition is true }

Ex.

let age = 18;

if (age >= 18) {

 console.log("You are an adult.");

 }

2. if...else Statement

The if...else statement executes one block of code if a condition is true and another block if it is false.

Syntax:

```
if (condition) {
 // code to be executed if condition is true
 } else
 { // code to be executed if condition is false }
```

Ex.

```
let age = 16;
 if (age >= 18) {
 console.log("You are an adult.");
} else {
console.log("You are a minor.");
}
```

3. if...else if...else Statement

The if...else if...else statement allows you to test multiple conditions sequentially.

Syntax:

```
if (condition1) {
 // code to be executed if condition1 is true
 }
else if (condition2) {
 // code to be executed if condition2 is true
} else {
 // code to be executed if both condition1 and condition2 are false
}
```

Ex.

```
let score = 85;
if (score >= 90) {
  console.log("Grade: A");
} else if (score >= 80) {
  console.log("Grade: B");
} else if (score >= 70) {
  console.log("Grade: C");
} else if (score >= 60) {
  console.log("Grade: D");
} else {
  console.log("Grade: F");
}
```

4. switch Statement

The switch statement is used to perform different actions based on different conditions. It is often more convenient than using many if...else if statements.

```
switch (expression) {
  case value1:
    // code to be executed if expression === value1
    break;
  case value2:
    // code to be executed if expression === value2
    break;
  // more cases...
  default:
    // code to be executed if expression doesn't match any case
}
```

```
let day = 3;
let dayName;
switch (day) {
  case 1:
    dayName = "Monday";
    break;
  case 2:
    dayName = "Tuesday";
    break;
  case 3:
    dayName = "Wednesday";
    break;
  case 4:
    dayName = "Thursday";
    break;
  case 5:
    dayName = "Friday";
    break;
  case 6:
    dayName = "Saturday";
    break;
  case 7:
    dayName = "Sunday";
    break;
  default:
    dayName = "Invalid day";
}
```

```
console.log(dayName); // Outputs
```

5. Ternary Operator

The ternary operator is a shorthand way of writing an if...else statement. It is also known as the conditional operator.

Syntax:-

```
condition ? exprIfTrue : exprIfFalse
```

Ex.

```
let age = 20;

let status = (age >= 18) ? "adult" : "minor";

console.log(status); // Outputs: adult
```

- **if**: Executes a block of code if a specified condition is true.
- **if...else**: Executes one block of code if a condition is true, and another block if it is false.
- **if...else if...else**: Tests multiple conditions sequentially and executes the corresponding block of code.
- **switch**: Evaluates an expression and matches its value against multiple cases, executing the corresponding block of code.
- **Ternary Operator**: A shorthand way of writing a simple if...else statement.

❖ JavaScript Functions

A JavaScript function is a block of code designed to perform a particular task.

A JavaScript function is executed when "something" invokes it (calls it).

```
<p>Call a function which performs a calculation and returns the result:</p>


<p id="demo"></p>

<script>

function myFunction(p1, p2) {

  return p1 * p2;

}

let result = myFunction(4, 3);
```

```
document.getElementById("demo").innerHTML = result;
</script>
```

❖ JavaScript Function Syntax

A JavaScript function is defined with the function keyword, followed by a name, followed by parentheses ().

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas: (parameter1, parameter2, ...)

The code to be executed, by the function, is placed inside curly brackets: {}

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

❖ Why Functions?

With functions you can reuse code

You can write code that can be used many times.

You can use the same code with different arguments, to produce different results.

Function Invocation

The code inside the function will execute when "something" **invokes** (calls) the function:

- When an event occurs (when a user clicks a button)
- When it is invoked (called) from JavaScript code
- Automatically (self invoked)- **Immediately Invoked Function Expression (IIFE)**: A function that runs as soon as it is defined. This is a common pattern in JavaScript to create a local scope and avoid polluting the global scope.

Automatically:-

- The function is defined within the parentheses: (function() { ... }).
- (function() {
- console.log("This function runs immediately!");
- })();

Function Return

When JavaScript reaches a return statement, the function will stop executing.

If the function was invoked from a statement, JavaScript will "return" to execute the code after the invoking statement.

Functions often compute a **return value**. The return value is "returned" back to the "caller":

```
function checkNumber(number) {
 if (number > 10) {
    return "Number is too large";
    }
             return "Number is acceptable";
    }
    console.log(checkNumber(5)); // Outputs: Number is acceptable
    console.log(checkNumber(15)); // Outputs: Number is too large
```

In this example:

- The function checkNumber checks if the number is greater than 10.
- If the number is greater than 10, it returns "Number is too large" and stops further execution of the function.
- If the number is not greater than 10, it returns "Number is acceptable".

- **Local Variables**

Variables declared within a JavaScript function, become **LOCAL** to the function.

Local variables can only be accessed from within the function

```
<p>Outside myFunction() carName is undefined.</p>
<p id="demo1"></p>
         <p id="demo2"></p>
    <script>
```

```
let text = "Outside: " + typeof carName;
document.getElementById("demo1").innerHTML = text;
function myFunction() {
  let carName = "Volvo";
  let text = "Inside: " + typeof carName + " " + carName;
  document.getElementById("demo2").innerHTML = text;
}
myFunction();
</script>
```

❖ JavaScript Function Definitions

JavaScript functions are **defined** with the function keyword.

You can use a function **declaration** or a function **expression.**

Function Expressions

A JavaScript function can also be defined using an **expression**.

A function expression can be stored in a variable:

```
<h2>JavaScript Functions</h2>
<p>A function can be stored in a variable:</p>
<p id="demo"></p>
<script>
const x = function (a, b) {return a * b};
document.getElementById("demo").innerHTML = x;
</script>
```

The function above is actually an **anonymous function** (a function without a name).

Functions stored in variables do not need function names. They are always invoked (called) using the variable name.

```
<h2>JavaScript Functions</h2>
<p>After a function has been stored in a variable, the variable can be used as a
function:</p>
```

```
<p id="demo"></p>
<script>
const x = function (a, b) {return a * b};
document.getElementById("demo").innerHTML = x(4, 3);
</script>
```

The arguments.length property returns the number of arguments received when the function was invoked:

```
<p>The arguments.length property returns the number of arguments received by the function:</p>
<p id="demo"></p>
<script>
function myFunction(a, b) {
  return arguments.length;
}
document.getElementById("demo").innerHTML = myFunction(4, 3);
</script>
```

❖ Arrow Functions

Arrow functions allows a short syntax for writing function expressions.

You don't need the function keyword, the return keyword, and the **curly brackets**.

```
<h2>JavaScript Arrow Functions</h2>
<p>With arrow functions, you don't have to type the function keyword, the return keyword, and the curly brackets.</p>
<p>Arrow functions are not supported in IE11 or earlier.</p>
<p id="demo"></p>
<script>
const x = (x, y) => x * y;
document.getElementById("demo").innerHTML = x(5, 5);
</script>
```

❖ alert and prompt

In JavaScript, alert and prompt are methods used to interact with the user. Both are part of the window object and are often used for simple user interactions, debugging, or gathering input. Here's an explanation of each:

alert

The alert method displays a dialog box with a specified message and an OK button. It is typically used to convey information or warnings to the user. The user must click the OK button to close the alert box and continue with the script execution.

```
alert("This is an alert message!");
```

The prompt method displays a dialog box that asks the user for input. The dialog box contains a text field where the user can enter a value, along with OK and Cancel buttons. The method returns the input value as a string if the user clicks OK, or null if the user clicks Cancel.

```
var userInput = prompt("Please enter your name");

Console.log(userInput);
```

Example combining both:

```
var name = prompt("What is your name?");

 if name () {

 alert("Hello, " + name + "!");

 } else {

alert("You didn't enter your name.");

 }
```

❖ String methods

JavaScript provides a variety of built-in string methods that allow you to manipulate and work with strings. Here are some of the most commonly used string methods:

Ex: consol.log();

Methods declears with parenthesis.

These methods provide powerful tools for string manipulation, enabling you to perform a wide variety of tasks when working with strings in JavaScript.

trim()

Removes whitespace from both ends of a string.

```
let str = " Hello World ";
 console.log(str.trim()); // "Hello World"
```

In JavaScript, strings are immutable, meaning that once a string is created, its characters cannot be changed. Any operation that appears to modify a string actually creates a new string instead of altering the original one.

```
let str = "hello";
let newStr = str.toUpperCase(); // "HELLO“
 console.log(str); // Output: "hello" (original string remains unchanged)
console.log(newStr); // Output: "HELLO" (new string)
```

toLowerCase()
Returns the calling string value converted to lower case.

```
let str = "Hello World";
console.log(str.toLowerCase()); // "hello world"
```

toUpperCase()
Returns the calling string value converted to upper case.

```
let str = "Hello World";
console.log(str.toUpperCase()); // "HELLO WORLD"
```

charAt()
Returns the character at a specified index.

```
let str = "Hello";
console.log(str.charAt(0)); // "H"
```

charCodeAt()
Returns the Unicode value of the character at a specified index.

```
let str = "Hello";
console.log(str.charCodeAt(0)); // 72
```

concat()
Joins two or more strings and returns a new concatenated string.

```
let str1 = "Hello";
let str2 = "World";
console.log(str1.concat(" ", str2)); // "Hello World"
```

replace()
Returns a new string with some or all matches of a pattern replaced by a replacement.

```
let str = "Hello World";
console.log(str.replace("World", "Everyone")); // "Hello
Everyone"
```

startsWith()
Determines whether a string begins with the characters of a specified string, returning true or false.

```
let str = "Hello World";
console.log(str.startsWith("Hello")); // true
```

endsWith()
Determines whether a string ends with the characters of a specified string, returning true or false.

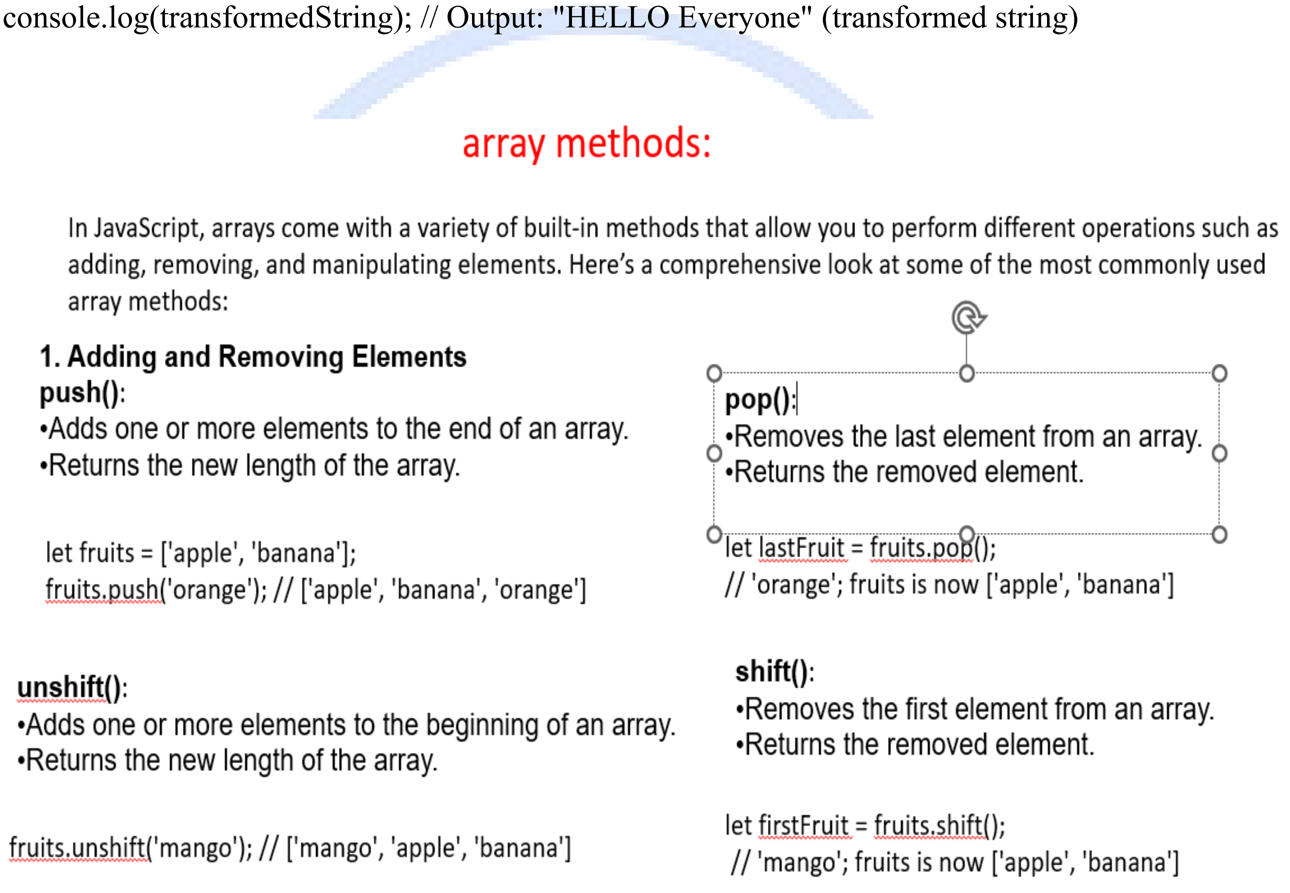
```
let str = "Hello World";
console.log(str.endsWith("World")); // true
```

repeat()
Constructs and returns a new string which contains the specified number of copies of the string on which it was called, concatenated together.

```
let str = "Hello ";
console.log(str.repeat(3)); // "Hello Hello Hello "
```

Method chaining in JavaScript is a technique in which multiple methods are called on the same object sequentially. Each method call returns an object, allowing the calls to be chained together. This is particularly useful with strings and arrays.

```
let originalString = "   hello world   ";

// Method chaining to perform multiple operations

let transformedString = originalString .trim().toUpperCase() .replace("WORLD",
"Everyone");

// Remove whitespace from both ends

// Convert to uppercase

// Replace 'WORLD' with 'Everyone'

console.log(originalString);    // Output: "   hello world   " (original string remains
unchanged)

console.log(transformedString); // Output: "HELLO Everyone" (transformed string)
```



array methods:

In JavaScript, arrays come with a variety of built-in methods that allow you to perform different operations such as adding, removing, and manipulating elements. Here's a comprehensive look at some of the most commonly used array methods:

1. Adding and Removing Elements

push():
•Adds one or more elements to the end of an array.
•Returns the new length of the array.

```
let fruits = ['apple', 'banana'];
fruits.push('orange'); // ['apple', 'banana', 'orange']
```

pop():
•Removes the last element from an array.
•Returns the removed element.

```
let lastFruit = fruits.pop();
// 'orange'; fruits is now ['apple', 'banana']
```

unshift():
•Adds one or more elements to the beginning of an array.
•Returns the new length of the array.

```
fruits.unshift('mango'); // ['mango', 'apple', 'banana']
```

shift():
•Removes the first element from an array.
•Returns the removed element.

```
let firstFruit = fruits.shift();
// 'mango'; fruits is now ['apple', 'banana']
```

The **slice()** method

in JavaScript is used to extract a section of an array or a string and return it as a new array or string without modifying the original array or string. Here's a detailed explanation of how slice() works:

Syntax:

```
array.slice(start, end);
```

- **start**: The index at which to begin extraction. If negative, it indicates an offset from the end of the array. Default is 0.
- **end**: The index before which to end extraction. The slice extracts up to but not including this index. If omitted, extracts through the end of the array. If negative, it indicates an offset from the end of the array.

```
let fruits = ['Apple', 'Banana', 'Mango', 'Orange', 'Peach'];

// Extract from index 1 to 3 (not including 3)

let citrus = fruits.slice(1, 3);

console.log(citrus); // ['Banana', 'Mango']

// Extract from index 2 to the end

let tropical = fruits.slice(2);

console.log(tropical); // ['Mango', 'Orange', 'Peach']

// Extract from the second to last element to the end

let lastTwo = fruits.slice(-2);

console.log(lastTwo); // ['Orange', 'Peach']

// Extract from the second to last element to the last element (not including)

let lastOne = fruits.slice(-2, -1);

console.log(lastOne); // ['Orange']

console.log(fruits); // Original array remains unchanged


//

let fruits = ['Apple', 'Banana', 'Mango', 'Orange', 'Peach'];

// Extract from index 1 to 3 (not including 3)

let citrus = fruits.slice(1, 3);

console.log(citrus); // ['Banana', 'Mango']

// Extract from index 2 to the end

let tropical = fruits.slice(2);

console.log(tropical); // ['Mango', 'Orange', 'Peach']

// Extract from the second to last element to the end

let lastTwo = fruits.slice(-2);
```

```
console.log(lastTwo); // ['Orange', 'Peach']
// Extract from the second to last element to the last element (not including)
let lastOne = fruits.slice(-2, -1);
console.log(lastOne); // ['Orange']
console.log(fruits); // Original array remains unchanged
```

Key Points

- slice() does not modify the original array or string; it returns a new array or string.
- Negative indices can be used to start the extraction from the end of the array or string.
- When end is omitted, slice() extracts through the end of the array or string.
- If start is greater than the length of the array or string, slice() returns an empty array or string.

substring()

The substring() method extracts characters from a string between two specified indices and returns the new substring.

Syntax:

```
string.substring(start, end);
```

- **start**: The index at which to begin extraction. If greater than end, the two values are swapped.
- **end**: The index before which to end extraction. If omitted, extracts to the end of the string.

```
let text = "Hello, World!";
// Extract from index 7 and the next 5 characters
let world = text.substr(7, 5);
console.log(world); // 'World'
// Extract from index 0 and the next 5 characters
let hello = text.substr(0, 5);
console.log(hello); // 'Hello'
```

```
// Extract the last 6 characters

let lastPart = text.substr(-6);

console.log(lastPart); // 'World!'
```

Differences Between slice(), substring(), and substr()

1. **Parameters and Functionality**:
   - **slice(start, end)**: Extracts from start to end (not including end). Negative indices count from the end.
   - **substring(start, end)**: Extracts from start to end (not including end). Swaps start and end if start > end. Does not accept negative indices.
   - **substr(start, length)**: Extracts from start for length characters. Negative start counts from the end.

Behavior with Negative Indices:

slice(): Supports negative indices.

substring(): Does not support negative indices.

substr(): Supports negative start indices but not negative length.

The splice() method

in JavaScript it is a versatile array method used to add, remove, or replace elements in an array. It directly modifies the

original array and can return an array containing the removed elements, if any. Here's an in-depth explanation of how splice() works:

Syntax

```
array.splice(start, deleteCount, item1, item2, ...);
```

- **start**: The index at which to start changing the array. If negative, it starts that many elements from the end of the array.
- **deleteCount**: The number of elements to remove from the array. If 0, no elements are removed.

item1, item2, ...: The elements to add to the array, starting from the start index. If omitted, no elements are added.

Removing Elements: To remove elements from an array, specify the start index and the deleteCount.

```
let fruits = ['apple', 'banana', 'mango', 'orange', 'peach'];
let removed = fruits.splice(2, 2); // Remove 2 elements starting from index 2
console.log(fruits); // ['apple', 'banana', 'peach']
console.log(removed); // ['mango', 'orange']
```

Adding Elements: To add elements to an array, specify the start index and set deleteCount to 0, then provide the new elements.

```
let fruits = ['apple', 'banana', 'peach'];
fruits.splice(2, 0, 'mango', 'orange'); // Add 'mango' and 'orange' at index 2
console.log(fruits); // ['apple', 'banana', 'mango', 'orange', 'peach']
```

Replacing Elements: To replace elements in an array, specify the start index and the deleteCount, then provide the new elements to replace the old ones.

```
let fruits = ['apple', 'banana', 'mango', 'orange', 'peach'];
fruits.splice(2, 2, 'kiwi', 'grape'); // Replace 2 elements starting from index 2
console.log(fruits); // ['apple', 'banana', 'kiwi', 'grape', 'peach']
```

Removing All Elements from a Given Index: To remove all elements from a specific index to the end of the array, omit the deleteCount or set it to a value greater than the number of remaining elements.

```
let fruits = ['apple', 'banana', 'mango', 'orange', 'peach'];
fruits.splice(2); // Remove all elements starting from index 2
console.log(fruits); // ['apple', 'banana']
```

Negative Indices
The start index can be negative, which means it counts back from the end of the array.

```
let fruits = ['apple', 'banana', 'mango', 'orange', 'peach'];
fruits.splice(-2, 1); // Start 2 elements from the end, remove 1
element
console.log(fruits); // ['apple', 'banana', 'mango', 'peach']
```

Here's a practical example that combines adding, removing, and replacing elements in a single array using splice()

```
let tasks = ['task1', 'task2', 'task3', 'task4'];

// Remove 'task3' and 'task4', and add 'taskA' and 'taskB'

tasks.splice(2, 2, 'taskA', 'taskB');

console.log(tasks); // ['task1', 'task2', 'taskA', 'taskB']

// Add 'taskC' at the beginning without removing any element

tasks.splice(0, 0, 'taskC');

console.log(tasks); // ['taskC', 'task1', 'task2', 'taskA', 'taskB']

// Remove the first two elements
```

```
let removedTasks = tasks.splice(0, 2);

console.log(tasks); // ['task2', 'taskA', 'taskB']

console.log(removedTasks); // ['taskC', 'task1']
```

Ascending Numerical Sort:

```
let numbers = [10, 1, 21, 2];
numbers.sort((a, b) => a - b);
console.log(numbers); // [1, 2, 10, 21]
```

Descending Numerical Sort:

```
numbers.sort((a, b) => b - a);
console.log(numbers); // [21, 10, 2, 1]
```

❖ The for loop

The for loop in JavaScript is a control flow statement used to execute a block of code repeatedly until a specified condition evaluates to false. It's one of the most commonly used loops in JavaScript for iterating over arrays, ranges, or performing repetitive tasks.

Syntax:

```
for (initialization; condition; increment) {

 // code block to be executed

}
```

- **initialization**: Executed once before the loop starts. Typically used to initialize a counter variable.
- **condition**: Evaluated before each iteration. If true, the loop continues. If false, the loop stops.
- **increment**: Executed after each iteration. Typically used to update the counter variable.

```
// Print numbers from 0 to 4

for (; let i = 0 i < 5; i++) {

  console.log(i);
}
// Output: 0 1 2 3 4
```

- **Variations of for Loop**

  Iterating Over Arrays: The for loop is often used to iterate over array elements.

```
let fruits = ['apple', 'banana', 'mango', 'orange', 'peach'];
for (let i = 0; i < fruits.length; i++) {
  console.log(fruits[i]);
}
// Output: apple banana mango orange peach

// Print even numbers from 1 to 10
for (let i = 1; i <= 10; i++) {
  if (i % 2 === 0) {
    console.log(i);
  }
}
// Output: 2 4 6 8 10
```

- **Nested for Loops:** You can nest for loops to iterate over multi-dimensional arrays or perform complex iterations.

```
for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    console.log(`i = ${i}, j = ${j}`);
  }
}
/* Output:
i = 0, j = 0
i = 0, j = 1
```

```
i = 0, j = 2
i = 1, j = 0
i = 1, j = 1
i = 1, j = 2
i = 2, j = 0
i = 2, j = 1
i = 2, j = 2
*/
```

The **while loop** in JavaScript is a control flow statement that allows code to be executed repeatedly based on a given boolean condition.

The loop continues to execute as long as the condition evaluates to true.

It's useful when the number of iterations is not known beforehand.

Syntax:

```
while (condition) { // code block to be executed }

let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
// Output: 0 1 2 3 4
```

Using break to Exit the Loop: The break statement can be used to exit the loop early.

```
let i = 0;
while (i < 10) {
  if (i === 5) {
    break; // Exit the loop when i is 5
  }
  console.log(i);
  i++;
}
// Output: 0 1 2 3 4
```

The **for...of loop** in JavaScript is used to iterate over iterable objects, such as arrays, strings, maps, sets, and more.

It allows you to loop through

the values of an iterable object in a straightforward manner. This loop is particularly useful when you want to work directly

with the values of an iterable without needing to manage an index or key.

Syntax:

for (variable of iterable) { // code block to be executed }

- **variable**: A variable that holds the value of the current element in each iteration.
- **iterable**: An iterable object (e.g., an array, string, map, set, etc.).

```
let fruits = ['apple', 'banana', 'mango'];
for (let fruit of fruits) {
  console.log(fruit);
}
// Output: apple banana mango
```

Difference Between for...of and for...in

- **for...of**: Iterates over the values of an iterable object.
- **for...in**: Iterates over the enumerable properties of an object (including array indices as properties).

```
let array = ['a', 'b', 'c'];
for (let index in array) {
  console.log(index); // 0, 1, 2 (indexes)
}
for (let value of array) {
  console.log(value); // 'a', 'b', 'c' (values)
}
```

❖ JavaScript Events

In JavaScript, events are actions or occurrences that happen in the browser, which the browser can detect and respond to. Events are an essential part of JavaScript and web development because they enable interaction between the user and the web page. Here's a breakdown of the key concepts related to JavaScript events:

❖ Types of Events

Mouse Events:

- click: Triggered when the user clicks on an element.
- dblclick: Triggered when the user double-clicks on an element.
- mouseover: Triggered when the user moves the mouse over an element.
- mouseout: Triggered when the user moves the mouse out of an element.
- mousemove: Triggered when the user moves the mouse within an element.
- mousedown: Triggered when the user presses a mouse button over an element.
- mouseup: Triggered when the user releases a mouse button over an element.

Keyboard Events:

- keydown: Triggered when the user presses a key.
- keypress: Triggered when the user presses a key (deprecated in favor of keydown and keyup).
- keyup: Triggered when the user releases a key.

Form Events:

- submit: Triggered when a form is submitted.
- change: Triggered when the value of an input element changes.

- focus: Triggered when an element gains focus.
- blur: Triggered when an element loses focus.

```
<h1>HTML DOM Events</h1>

<h2>The ondblclick Event</h2>

<p ondblclick="myFunction()">Double-click this paragraph to trigger a function.</p>

<p id="demo"></p>

<script>

function myFunction() {

  document.getElementById("demo").innerHTML += "Hello World ";

}

</script>
```

```
<h1>HTML DOM Events</h1>

<h2>The onmouseover Event</h2>

<img onmouseover="bigImg(this)" onmouseout="normalImg(this)" border="0"
src="smiley.gif" alt="Smiley" width="32" height="32">


<p>The function bigImg() is triggered when the user moves the mouse pointer over the
image.</p>

<p>The function normalImg() is triggered when the mouse pointer is moved out of the
image.</p>

<script>

function bigImg(x) {

  x.style.height = "64px";

  x.style.width = "64px";

}

function normalImg(x) {

  x.style.height = "32px";
  x.style.width = "32px";
}
</script>
```

Description onmouseout

The onmouseout event occurs when the mouse **pointer moves out of an element**.

The onmouseout event is often used together with the onmouseover event, which occurs when

the pointer is moved over an element.

```
<h1>HTML DOM Events</h1>
<h2>The onmouseover Event</h2>
<img onmouseover="bigImg(this)" onmouseout="normalImg(this)" border="0"
src="smiley.gif" alt="Smiley" width="32" height="32">
<p>The function bigImg() is triggered when the user moves the mouse pointer over the
image.</p>
<p>The function normalImg() is triggered when the mouse pointer is moved out of the
image.</p>
<script>
function bigImg(x) {
  x.style.height = "64px";
  x.style.width = "64px";
}
function normalImg(x) {
  x.style.height = "12px";
  x.style.width = "12px";
}
```

Description onmousemove

The onmousemove event occurs when the pointer moves over an element.

```
<style>
div {width: 200px;  height: 100px;  border: 1px solid black;
}
</style>
<body>
<h2>The onmousemove Event</h2>
<div onmousemove="myFunction(event)" onmouseout="clearCoor()"></div>
<p>Mouse over the rectangle above, and get the coordinates of your mouse pointer.</p>
<p>When the mouse is moved over the div, the p element will display the horizontal and
vertical coordinates of your mouse pointer, whose values are returned from the clientX and
clientY properties on the
MouseEvent object.</p>
<p id="demo"></p>
<script>
function myFunction(e) {
  let x = e.clientX;
  let y = e.clientY;
  let coor = "Coordinates: (" + x + "," + y + ")";
  document.getElementById("demo").innerHTML = coor;
}
function clearCoor() {
  document.getElementById("demo").innerHTML = "";
}
</script>
```

- **Description onmousedown**

  The onmousedown event occurs when a user **presses a mouse button** over an **HTML element.**

  The onmouseup event occurs when a mouse button is released over an element.

  ```
  <body>
  <p>Clock the text below!</p>
  <p id="myP" onmousedown="mouseDown()" onmouseup="mouseUp()">
  ```

```
The mouseDown() function sets the color of this text to red.
The mouseUp() function sets the color of this text to blue.
</p>

<script>
function mouseDown() {
  document.getElementById("myP").style.color = "red";
}
function mouseUp() {
  document.getElementById("myP").style.color = "blue";
}
</script>
</body>
</html>
```

- **Description**

The onkeydown event occurs when the user **presses a key** on the keyboard.

The onkeyup event occurs when the user releases a key on the keyboard.

```
<h2>The onkeydown Event</h2>
<p>Press a key in the input field:</p>
<input type="text" onkeydown="myFunction()">
<p id="demo"></p>
<script>
function myFunction() {
  document.getElementById("demo").innerHTML =
"You pressed a key inside the input field";
}
</script>
```

```
<h2>The keyup Event</h2>
<p>A function is triggered when the user releases a key in the input field.</p>
<p>The function transforms the input field to upper case:</p>
Enter your name: <input type="text" id="fname" onkeyup="myFunction()">
<script>
function myFunction() {
  let x = document.getElementById("fname");
  x.value = x.value.toUpperCase();
}
</script>
```

The onsubmit event occurs when a form is submitted.

```
<p>When you submit the form, a function is triggered which alerts some text.</p>
<form action="/action_page.php" onsubmit="myFunction()">
  Enter name: <input type="text" name="fname">
  <input type="submit" value="Submit">
</form>
<script>
function myFunction() {
  alert("The form was submitted");
}
</script>
```

The onchange event occurs when **the value of an HTML element is changed**.

```
<p>Select a new car from the list.</p>
<select id="mySelect" onchange="myFunction()">
  <option value="Audi">Audi</option>
  <option value="BMW">BMW</option>
  <option value="Mercedes">Mercedes</option>
```

- **Description**

The onfocus event occurs when an element gets focus.

The onfocus event is often used on input fields.

```
<h1>HTML DOM Events</h1>

<h2>The focus Event</h2>

Enter your name: <input type="text" onfocus="myFunction(this)">

<p>When the input field gets focus, a function changes the background-color.</p>

<script>

function myFunction(x) {

  x.style.background = "yellow";

}

</script>
```

- **Description**

The onblur event occurs when an **HTML element loses focus**.

The onblur event is often used on input fields.

The onblur event is often used with form validation (when the user leaves a form field).

```
<h1>HTML DOM Events</h1>

<h2>The blur Event</h2>

Enter your name: <input type="text" id="fname" onblur="myFunction()">

<p>When you leave the input field, a function is triggered which transforms the input text
to upper case.</p>

<script>

function myFunction() {

  let x = document.getElementById("fname");

  x.value = x.value.toUpperCase();

}

</script>

<h1>The Document Object</h1>
```

```
<h2>The getElementsByClassName() Method</h2>
<p>Change the text of the first element with class="example":</p>
<div class="example">Element1</div>
<div class="example">Element2</div>
<script>
const collection = document.getElementsByClassName("example");
collection[0].innerHTML = "Hello World!";
</script>


<h1>The Document Object</h1>
<h2>The getElementsByTagName() Method</h2>
<p>An unordered list:</p>
<ul>
  <li>Coffee</li>
  <li>Tea</li>
  <li>Milk</li>
</ul>
<p>The innerHTML of the second li element is:</p>
<p id="demo"></p>
<script>
const collection = document.getElementsByTagName("li");
document.getElementById("demo").innerHTML = collection[1].innerHTML;
</script>
```