BASICS OF JAVASCRIPT & ES6 FUNCTIONS

ABSTRACT

This unit covers the fundamentals of JavaScript with an introduction to ES6 features. It explains variables (var, let, const), data types, arrays, objects, and commonly used array methods for data handling. Learners explore control structures such as loops and conditional statements, along with different types of functions, including arrow functions and modern concepts like default parameters, rest, and spread operators. The unit also introduces JavaScript pop-up boxes (alert, confirm, prompt) for basic user interaction.

Unit-8

Basics of JavaScript and ECMAScript (ES6)

Basics of Javascript: Client Side Scripting with JS

- JavaScript is one of the most used languages when it comes to building web applications as it allows developers to wrap HTML and CSS code in it to make web apps interactive.
- It enables the interaction of users with the web application. It is also used for making animations on websites
- JavaScript was initially created to **"make web pages alive".**
- The programs in this language are called scripts. They can be written right in a web page's HTML and run automatically as the page loads.
- Scripts are provided and executed as plain text. They don't need special preparation or compilation to run.
- It is lightweight and most commonly used as a part of web pages, whose implementations allow client-side script to interact with the user and make dynamic pages.
- It is an interpreted programming language with object-oriented capabilities.
- It is used both client-side and server-side to make web apps interactive.
- Uses built-in browser DOM. Extension of JavaScript file is **.js**

Why is it called JavaScript?

When JavaScript was created, it initially had another name: "**LiveScript**". But Java was very popular at that time, so it was decided that positioning a new language as a "younger brother" of Java would help.

But as it evolved, JavaScript became a fully independent language with its own specification called **ECMAScript**, and now it has no relation to Java at all.

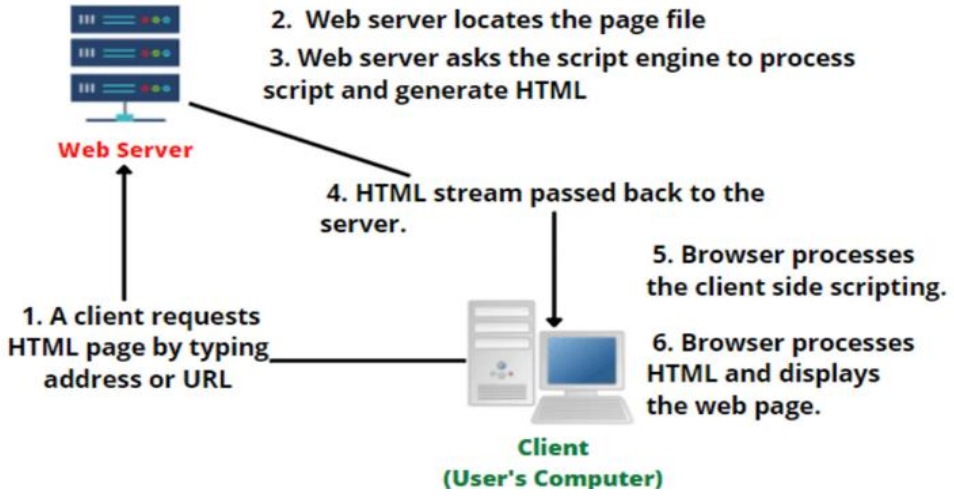
ECMAScript, also known as JavaScript, is a programming language adopted by the European Computer Manufacturer's Association as a standard for performing computations in Web applications.

History of JavaScript

In 1993, **Mosaic**, the first popular web browser, came into existence. In the **year 1994**, **Netscape** was founded by **Marc Andreessen**. He realized that the web needed to become more dynamic. Thus, a 'glue language' was believed to be provided to HTML to make web designing easy for designers and part-time programmers. Consequently, in 1995, the company recruited **Brendan Eich** intending to implement and embed Scheme programming language to the browser. But, before Brendan could start, the company merged with **Sun Microsystems** for adding Java into its Navigator so that it could compete with Microsoft over the web technologies and platforms. Now, two languages were there: Java and the scripting language. Further, Netscape decided to give a similar name to the scripting language as Java's. It led to 'Javascript'. Finally, in May 1995, Marc Andreessen coined the first code of Javascript named '**Mocha**'. Later, the marketing team replaced the name with '**LiveScript**'. But, due to trademark reasons and certain other reasons, in December 1995, the language was finally renamed to 'JavaScript'. From then, JavaScript came into existence.

Client-side scripting

- A client-side script is a tiny program (or collection of instructions) that is put into a web page. It is handled by the client browser rather than the web server.
- In other words, client-side scripting is a method for browsers to run scripts without having to connect to a server.
- The code runs on the client's computer's browser either while the web page is loading or after it has finished loading.
- Client-side scripting is mostly used for dynamic user interface components including pull-down menus, navigation tools, animation buttons, and data validation.
- The client refers to the script that runs on the user's computer system. It can either be integrated (or injected) into the HTML content or stored in a separate file (known as an external script).
- When the script files are requested, they are transmitted from the web server (or servers) to the client system. The script is run by the client's web browser, which subsequently displays the web page, including any visible script output.
- look at the diagram below.



Applications of Javascript Programming

- **Client side validation** - This is really important to verify any user input before submitting it to the server and Javascript plays an important role in validting those inputs at front-end itself.
- **Manipulating HTML Pages** - Javascript helps in manipulating HTML page on the fly. This helps in adding and deleting any HTML tag very easily using javascript and modify your HTML to change its look and feel based on different devices and requirements.
- **User Notifications** - You can use Javascript to raise dynamic pop-ups on the webpages to give different types of notifications to your website visitors.
- **Back-end Data Loading** - Javascript provides Ajax library which helps in loading back-end data while you are doing some other processing. This really gives an amazing experience to your website visitors.
- **Presentations** - JavaScript also provides the facility of creating presentations which gives website look and feel. JavaScript provides RevealJS and BespokeJS libraries to build a web-based slide presentations.

- **Server Applications** - Node JS is built on Chrome's Javascript runtime for building fast and scalable network applications. This is an event based library which helps in developing very sophisticated server applications including Web Servers.

Advantages of JavaScript

- **Less server interaction** − You can validate user input before sending the page off to the server. This saves server traffic, which means less load on your server.
- **Immediate feedback to the visitors** − They don't have to wait for a page reload to see if they have forgotten to enter something.
- **Increased interactivity** − You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard.
- **Richer interfaces** − You can use JavaScript to include such items as drag-and-drop components and sliders to give a Rich Interface to your site visitors.

Limitations of JavaScript

- We cannot treat JavaScript as a full-fledged programming language. It lacks the following important features:
- Client-side JavaScript does not allow the reading or writing of files. It has been kept for the security reason.
- JavaScript could not used for networking applications because there is no such support available.
- JavaScript doesn't have any multithreading or multiprocessor capabilities

Syntax of JavaScript

- JavaScript can be implemented using JavaScript statements that are placed within the **<script>... </script>** HTML tags in a web page.
- You can place the **<script>** tags, containing your JavaScript, anywhere within your web page, but it is normally recommended that you should keep it within the **<head>** tags.
- The script tag takes two important attributes –
  - **Language** − This attribute specifies what scripting language you are using. Typically, its value will be javascript. Although recent versions of HTML (and XHTML, its successor) have phased out the use of this attribute.
  - **Type** − This attribute is what is now recommended to indicate the scripting language in use and its value should be set to "text/javascript".
- The <script> tag alerts the browser program to start interpreting all the text between these tags as a script.

```
<script language = "javascript" type = "text/javascrit">
JavaScript code
</script>
```

Internal JavaScript

JavaScript can be added directly to the HTML file by writing the code inside the <script> tag. We can place the <script> tag either inside <head> or the <body> tag according to the need.

Example:

```
<html>
  <head>
    <script language = "javascript" type = "text/javascript">
        document.write("Hello World!")
        console.log("Hello!")
    </script>
  </head>
</html>
```

External JavaScript

The other way is to write JavaScript code in another file having a .js extension and then link the file inside the **<head> or <body>** tag of the HTML file in which we want to add this code.

Example:

j1.html

```
<html>
  <head>
    <script src="j1.js"></script>
  </head>
</html>
```

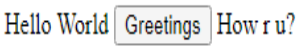
j1.js

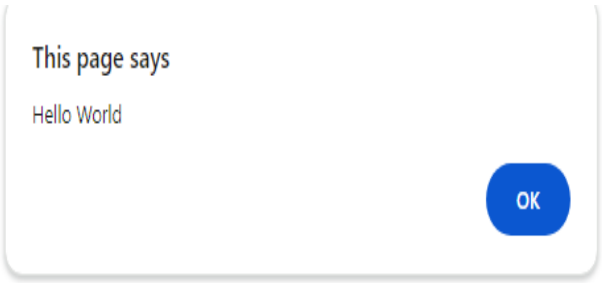
```
document.write("This is written in external file")
```

JavaScript in <body> and <head> sections together: Can execute as per need

```
<head>
   <script type="text/javascript" language="javascript">
      function sayHello(){
                  alert("Hello World")
         }
document.write("Hello World")
   </script>
</head>
<body>
   <input type="button" onClick="sayHello()" value="Greetings"/>
   <script type="text/javascript" language="javascript">
          document.write("How r u?")
   </script>
</body>
```

Output:





Understand the concept of document.write(),document.writeln(),console.log()

1) document.write()

- Writes output directly on the web page.
- Mostly used for learning.
- Can overwrite the page if used after load.

2) document.writeln()

- Same as document.write() but adds a newline.
- Newline is not visible in HTML unless
 is used.

3) console.log()

- Prints output in the browser console.
- Used for debugging.
- Does not display on the web page.

Example

```
<script>
document.writeln("Hello");
document.writeln("World");
</script>
```

Output on web page:

Hello World

What happens?

- document.writeln() adds a **newline in the source code**
- HTML ignores newlines

With
 (visible line break)

```
<script>
document.writeln("Hello<br>");
document.writeln("World");
</script>
```

Output:

Hello
World

ECMAScript (ES6) Overview

ECMAScript (ES) is a scripting language specification standardized by ECMAScript International. It is used by applications to enable client-side scripting. ECMAScript 2015 was the second major revision to JavaScript. ECMAScript 2015 is also known as ES6 and ECMAScript 6.

"ECMA" stood for "European Computer Manufacturers Association" until 1994.

ES6 allows one to write code in a clever way which makes the code more readable. In short,using ES6, we write less and do more, hence it work on principle “write less. Do more”.

In simple words:

ECMAScript = Rules

JavaScript = Language built using those rules

Key Features of ES6

- let and const
- Arrow functions
- Template literals
- Classes
- Spread and Rest operators
- Default parameters

- ✓ JavaScript is a scripting language used for interactive web development.
- ✓ ECMAScript is the standard that defines JavaScript.
- ✓ ES6 is an updated version of ECMAScript with modern features.
- ✓ ES6 makes JavaScript code shorter, cleaner, and more efficient.
- ✓ Modern applications use ES6 by default.

Variables

- **Variables are Containers for Storing Data**
- **Variables** are the building blocks of any programming language.
- In JavaScript, variables can be used to store reusable values.
- The values of the variables are allocated using the assignment operator("=").
- JavaScript is a dynamically typed language, so variable types are decided at runtime.

JavaScript Identifiers

JavaScript variables must have unique names. These names are called **Identifiers**.

There are some basic rules to declare a variable in JavaScript:

- Name must start with a letter (a to z or A to Z), underscore(_), or dollar($) sign.
- After first letter we can use digits (0 to 9), for example value1.
- JavaScript variables are case sensitive, for example x and X are different variables.
- A variable name cannot be a reserved keyword.

JavaScript is a dynamically typed language so the type of variables is decided at runtime. Therefore, there is no need to explicitly define the type of a variable. We can declare variables in JavaScript in four ways:

1. Automatically (Implicit)
2. var
3. let
4. const

All three keywords do the basic task of declaring a variable but with some differences Initially, **all the variables in JavaScript were written using the var keyword** but **in ES6 the keywords let and const were introduced.**

JavaScript local variable

A JavaScript local variable is declared inside block or function.
It is accessible within the function or block only.
For example:

```
<script>
function abc(){
var x=10;//local variable
document.write(x)
}
abc();//calling JavaScript function
</script>
```

Output: 10

JavaScript global variable

A **JavaScript global variable** is accessible from any function.
A variable i.e. declared outside the function or declared with window object is known as global variable.

Example:

```
<script>
var d=200;//global variable
function a(){
document.writeln(d);
}
document.writeln(d);
a();//calling JavaScript function
</script>
```

Output:

200 200

1. Implicit (Automatically)

In this first example, x, y, and z are undeclared variables. They are automatically declared when first used. Such variables become global by default, which can cause unexpected behavior, make debugging difficult, and lead to errors. Therefore, this practice should be avoided.

```
x = 5;
y = 6;
z = x + y;
```

2. Var Keyword

The **var** is the oldest keyword to declare a variable in JavaScript.

It has the Global scoped or function scoped which means variables defined outside the function can be accessed globally, and variables defined inside a particular function can be accessed within the function.

1. The below code example explains the use of the var keyword to declare the variables in JavaScript.

```
var a = 10
function f1() {
var b = 20
console.log(a, b)
}
f1();
console.log(a);
Output:
10 20
10
```

2. The below example explains the behavior of var variables when declared inside a function and accessed outside of it.

```
function f1() {
// It can be accessible anywhere within this function
var a = 10;
console.log(a)
}
f1();

// A cannot be accessible outside of function
console.log(a);
```

Output:

10
ReferenceError: a is not defined

3. The below code re-declare a variable with same name in the same scope using the var keyword, which gives no error in the case of var keyword.

```
var a = 10
// User can re-declare variable using var
var a = 8
// User can update var variable
a = 7
console.log(a);
```

Output:

7

4. The below code explains the initialized the variables after accessing.

```
console.log(a);
var a = 10; //initialized the variable after accessing
```

Output:

Undefined

3. Let Keyword

- The let keyword is an improved version of the var keyword.
- It is introduced in the ES6 or EcmaScript 2015.
- These variables has the block scope.
- It can't be accessible outside the particular code block ({block}).

1. The below code declares the variable using the let keyword.

```
let a = 10;
function f() {
let b = 9
console.log(b);  //local scope
console.log(a);  //global scope
}
f();
console.log(b);

Output:
9
10
Uncaught ReferenceError: b is not defined
```

Block scoped

```
function blockEx () {
var b=5
if (b>4) {
    let a = 1; // Declared with let inside an if block
    console.log(a); // Output: 1 (Accessible inside the block)
  }
  console.log(a); // Error: a is not defined (Not accessible outside the block)
}

blockEx();

Output:
1
ReferenceError: a is not defined
```

2. The below code explains the behavior of let variables when they are re-declared in the same scope.

```
let a = 10
// It is not allowed
let a = 10
// It is allowed
a = 10
Output:
Uncaught SyntaxError:Identifier 'a' has already been declared
```

3. The below code explains the behavior of let variable when it isre-declared in the different scopes.

```
let a = 10
function f1() {
let a = 9
console.log(a) // It prints 9
}
f1()
console.log(a) // It prints 10
```

Output:

9
10

4. The below code explains the initialized the variables after accessing.

```
console.log(a);
let a = 10; //initialized the variable after accessing
```

Output:

Uncaught **ReferenceError**: Cannot access 'a' before initialization

4. Const Keyword

The const keyword has all the properties that are the same as the let keyword, except the user cannot update it and have to assign it with a value at the time of declaration.
These variables also have the block scope.
It is mainly used to create constant variables whose values can not be changed once they are initialized with a value.

1. This code tries to change the value of the const variable.

```
const a = 10;
function f() {
a = 9
console.log(a)
}
f();
```

Output:

TypeError:**Assignment to constant variable.**

This does indeed throw a TypeError because you're trying to reassign a const variable.

2.The below code explains the behavior of const variables when they are redeclared in the same scope.

```
const a = 10
// It is not allowed
const a = 10
// It is not allowed
a = 10
```

Output:

Uncaught SyntaxError: **Identifier 'a' has already been declared**

3. No reassignment is happening—two independent const variables (a) exist in different scopes.

```
const a = 10;
function f() {
   const a = 9
   console.log(a)
}
f();
console.log(a);
```

Output:
9
10

This works without error because the const a = 9; inside the function creates a new variable in the function's local scope**, shadowing the global a.**

Note:

The variables declared with var and let are mutable that is their value can be changed but variables declared using const are immutable.

Comparison: var v/s let v/s const

	var	let	const
Scope	The scope of a *var* variable is functional or global scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
Re-declare	It can re-declared in the **same scope.** **var a=10;** **var a=20;** **a=30;** **let a=45;**	It cannot be re-declared in the **same scope.** **let b=20;** ***//not allowed*** ***let b=24;*** ***var b=33;***	It can not re-declared in **same scope.** **const b=20;** **//not allowed** ***const b=33;*** ***let b=24;*** ***var b=33;***
Re-assign	It can be updated in the **same scope.** **var a=10;** **a=30;**	It can be in the **same scope.** **let b=20;** **b=45;**	It can not updated in **same scope.** const b=20; **//not allowed** ***b=45;***
Without Init	**It can** be declared without initialization. **var a;**	**It can** be declared without initialization. **let a;**	It cannot be declared without initialization. **No** **Must use** **const a=20;**
Hoisting access	It can be accessed without initialization as its default value is **“undefined”.**	It cannot be accessed without initialization otherwise it will give **‘referenceError’.**	It cannot be accessed without initialization, as it cannot be declared without initialization. **‘referenceError’**

Data Types

JavaScript provides different data types to hold different types of values.

There are two types of data types in JavaScript.

1. **Primitive data type**
2. **Non-primitive (reference) data type**

- JavaScript is a dynamic type language, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine.
- You need to use var, let or const here to specify the data type.
- It can hold any type of values such as numbers, strings etc.

For example:

```
var a=20;//holding number

var b="LJU";//holding string
```

Primitive Data Types

Primitive data types are **predefined, built-in data types**. They store **single values** and are **immutable**.

An **immutable** value **cannot be changed after it is created**.
- Any modification creates a **new value**, not a change to the original.
- In JavaScript, **all primitive data types are immutable**.

Types of Primitive Data Types

1) String

Represents a sequence of characters.

```
let x = "LJU";  // String
```

2) Number

Represents numeric values (integers and floating-point numbers).

```
let x1 = 24.00; // with decimals
let x2 = 24;    // without decimals
```

3) Boolean

Represents logical values: **true** or **false**.

```
let x = 5;
let y = 5;
let z = 6;

(x == y); // true
(x == z); // false
```

Comparison Operators

- == → compares values after type coercion
- === → compares **both value and type** (strict comparison)

4) Undefined

A variable is declared but **not assigned** a value.

```
let name;
console.log(name); // undefined
```

5) Null

Represents **intentional absence of value**.

```
let name = null;
All primitive types, except null, can be tested by the typeof operator.
typeof null returns "object", so one has to use === null to test for null.
```

Difference Between undefined and null

Feature	undefined	null
Meaning	Declared but not assigned	Explicitly no value
Type	"undefined"	"object"
Use Case	Uninitialized variables	Intentional empty value

Example: typeof Operator Example

```
<script>
var a = undefined;
var b = "test";
var c = true;
var d = 5;
let e = null;

document.write(typeof(a)+"<br>");
document.write(typeof(b)+"<br>");
document.write(typeof(c)+"<br>");
document.write(typeof(d)+"<br>");
document.write(typeof(e)+"<br>");
</script>
```

Output:

```
undefined
string
boolean
number
object
```

Example: String and Number Combination Example

```
<script>
var x = 16 + 4 + "xyz";
var y = "xyz" + 16 + 4;

document.write(x + "<br>");
document.write(y);
</script>
```

Output:

20xyz
xyz164

Non-primitive data types

Non-primitive data types are called reference types because they refer to objects.

Non-primitive types are created by the programmer and is not defined by Java (except for String).

- Store **multiple values**
- Stored by **reference**
- Created by the programmer
- Mutable

Mutable Meaning:
A mutable **value** can be changed after it is created**.**
- The original value is **modified directly**.
- In JavaScript, **non-primitive (reference) types are mutable**.

The non-primitive data types are as follows:

1) **Object**
2) **Array**

Objects

It represents instance through which we can access members
- JavaScript objects are written with curly braces {}.
- Object properties are written as name:value pairs, separated by commas.

Example: const person = {firstName:"ABC", lastName:"DEF", age:30};

```
<script>
  const person = {firstName:"Khushbu", lastName:"Patel"};
  document.write(person.firstName);
document.write(person[“lastName”];
</script>
```

Output:
Khushbu

Note:
- **Dot notation** used when Property name is fixed and simple.

- **Bracket notation** used when Property name is in a variable or has spaces.

Example with variable:

```
let key = "firstName";
console.log(person[key]);   // Amit
```

Example with space/compiler-unfriendly name:

```
let obj = { "full name": "Amit Kumar" };
console.log(obj["full name"]);
```

Example

```
<script type="text/javascript" language="javascript">
var emp ={id:101, name:"xyz", salary:60000};
document.write(emp.id+" "+emp.name+" "+emp.salary);
</script>
```

Array

It represents group of similar values

- Arrays are a special kind of objects, with numbered indexes.
- An array can hold many values under a single name, and you can access the values by referring to an index number.
- It is a common practice to declare arrays with the const keyword.
- JavaScript arrays are written with square brackets.
- Array items are separated by commas.
- The following code declares (creates) an array called fruits, containing three items (car names):

Example:

```
const fruits = ["grapes", "mango", "watermelon"];
```

- There are two main ways to create arrays in JavaScript:
  - you can use a literal, or
  - you can use a constructor. (*Reference)
- The **literal** looks like this and can be defined with or without any array elements. The one above doesn't have any elements yet.

```
const myArray = [];
```

The length Property

The length property of an array returns the length of an array (the number of array elements).

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let length = fruits.length;
```

The length property is always one more than the highest array index.

Accessing the First Array Element

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits[0]
```

Accessing the Last Array Element

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
let fruit = fruits[fruits.length - 1];
```

Example: Display all elements of an array in new line.

```
<script type="text/javascript" language="javascript">
    var emp =["abc", "pqr", "xyz"];
    for(i=0; i<emp.length; i++) {
        document.write(emp[i]+"<br>");
    }
</script>
```

Example: Multiply each elements of an array by 2 and display in new line.

```
<script>
  var num = [2,4,3,1,5];
  for(i=0; i<num.length; i++) {
    let mul = num[i]*2;
  document.write("multiplication is: "+ mul+"<br>");
  }
  </script>
```

Array of Objects

An **array of objects** is an array where **each element is an object**.

Example

let students = [{name: "Rahul", age: 20},{name: "Anita", age: 22}, {name: "Amit", age: 21}];

students → array having three {} → object & Each object has **key–value pairs**

Accessing values

```
console.log(students[0].name); // Rahul
console.log(students[1].age);  // 22
```

When to Use Arrays. When to use Objects.

- JavaScript does not support associative arrays.
- You should use **objects** when you want the element names to be **strings (text)**.
- You should use **arrays** when you want the element names to be **numbers (indexes)**.

Conditions

Conditions help your program **make decisions** based on true/false logic.

Condition	WHY we use it	WHEN to use it
if	To check a single condition	One decision needed
if...else	To choose between two paths	Yes / No situations
else if	To handle multiple conditions	More than two outcomes
switch	To match one value with many options	Fixed values (menu, roles, days)
ternary (? :)	Short decision making	Simple conditions

◈ if Statement

- To execute code **only when condition is true**
- When you have **one condition** to check

Example

```
let isLoggedIn = true;

if (isLoggedIn) {

  console.log("Welcome user");

}
```

◈ if...else

- To choose **one of two paths**
- When decision has **only two outcomes**

Example

```
let age = 16;
if (age >= 18) {
  console.log("Allowed");
} else {
  console.log("Not allowed");
}
```

◈ else if

To handle **multiple conditions**
When you have **more than two possibilities**

Example

```
let score = 82;
if (score >= 90) {
  console.log("Grade A");
} else if (score >= 75) {
  console.log("Grade B");
} else {
  console.log("Grade C");
}
```

◈ switch Statement

- Cleaner than many else if
- Easier to read for **fixed values**
- When checking **one variable** against many exact values

Example

```
let role = "admin";
switch (role) {
  case "admin":
    console.log("Full access");
    break;
  case "editor":
    console.log("Edit access");
    break;
  case "viewer":
    console.log("Read only");
    break;
  default:
    console.log("No access");
}
```

Use break to stop execution. Use default as fallback

◈ Ternary Operator (? :)

- Short and clean code
- Simple condition with one line output
- **Avoid ternary for complex logic**

Example

```
let age = 20;
let result = age >= 18 ? "Adult" : "Minor";
console.log(result);
```

Loops

Loop	Syntax	Used For	Best When
for	for(init; condition; step)	Repeat code fixed number of times	You know how many times
while	while(condition)	Repeat while condition is true	Count unknown
do...while	do { } while(condition)	Runs at least once	Must execute once
for...of	for (value of iterable)	Loop through values	Arrays, strings
for...in	for (key in object)	Loop through keys	Objects

Loop Examples

◈ for loop

Used when number of iterations is known.

```
for (let i = 1; i <= 5; i++) {
  console.log(i);
}

Output:
1 2 3 4 5
```

◈ while loop

Runs while condition is true.

```
let i = 1;

while (i <= 5) {
  console.log(i);
  i++;
}

Output:
1 2 3 4 5
```

◈ do...while loop

Runs **at least once**, even if condition is false.

```
let i = 6;
do {
  console.log(i);
  i++;
} while (i <= 5);

Output:
6
```

◈ for...of Loop

- Used to loop through iterable values
- Best for **arrays, strings**

Syntax:

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

Example

```
let fruits = ["apple", "banana", "orange"];

// for...of loop iterates over the VALUES of the array
for (let fruit of fruits) {
  console.log(fruit);
}
```

Output

```
apple
banana
orange
```

◈ for...in Loop (with Objects)

- Used to loop through **keys (properties)** of an object
- Best for **objects**

Syntax

```
for (key in object) {
// code block to be executed
}
```

Example

```
let user = {
  name: "Sonali",
  age: 25,
  city: "Delhi"
};

// for...in loop iterates over the KEYS of the object
for (let key in user) {
  console.log(key + ": " + user[key]);
}
```

Output

```
name: Sonali
age: 25
city: Delhi
```

◈ for...in Loop with Arrays

- for...in **can** work with arrays
- It iterates over **indexes (keys)**, not values
- Not recommended for arrays

Example

```
let students = ["ABC", "PQR", "XYZ"];
// for...in loop iterates over the INDEXES (keys) of the array
for (let i in students) {
// 'i' represents the index (0, 1, 2)
 console.log(i);
// Access the value using the index
  console.log(students[i]);
}
```

Output

0
ABC
1
PQR
2
XYZ

Why for...in Is NOT Preferred for Arrays

It loops over **indexes**, not values.
Can include **extra properties** if added to array.
Order is **not guaranteed.**
Arrays depend on **sequence and values**

array.entries() Mehtod

The .entries() method returns an array containing arrays of an **object's key-value** pairs in the following format: [[key, value], [key, value], ...].

Example:

```
const object = { name: 'abc', age: 42 };
console.log(Object.entries(object));
```

Output: [[name, 'abc'],[age, 42]]

To access the index of the array elements inside the loop, you can use the for...of statement with the entries() method of the array. The array.entries() method returns a pair of [index, element] in each iteration.

Example:

```
<html>
 <body>
<script>
```

```
let colors = ['Red', 'Green', 'Blue'];
for (let [index, color] of colors.entries())
{
console.log(`${color} is at index ${index}`);
}
</script>
</body>
</html>
```

Output:

```
Red is at index 0
Green is at index 1
Blue is at index 2
```

Comparison of var and let (Overwrite Effect)

Case	Code using var	Output	Code using let	Output

Condition (if)	var x=10;
	if(true){ var x=3; }
	console.log(x);
		3	let x=10; if(true){ let x=3; }
			console.log(x);
				10

Loop (for)	var i=5;
	for(var i=0;i<2;i++){
	console.log(i);}
	console.log(i);
		0
		1
		2
			let i=5; for(let i=0;i<2;i++){
			console.log(i);}
			console.log(i);
				0
				1
				5

Array Methods

Method	Syntax	What it Does	Returns
push()	array.push(item)	Adds element(s) to the **end** of an array	New array length
pop()	array.pop()	Removes the **last** element from an array	Removed element
map()	array.map(callback)	Creates a **new array** by transforming each element	New array
filter()	array.filter(callback)	Creates a **new array** with elements that pass a condition	New array
forEach()	array.forEach(callback)	Executes a function for each element	undefined
includes()	array.includes(value)	Checks if array contains a value	true or false
find()	array.find(callback)	Finds the **first element** that matches a condition	Element or undefined

Examples

◈ push()

Adds elements to the **end** of an array.

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

◈ pop()

Removes the **last element** from an array.

```
let numbers = [10, 20, 30];
let removed = numbers.pop();
console.log(numbers); // [10, 20]
console.log(removed); // 30
```

◈ map()

Used to **transform each element** and return a new array.

```
let nums = [1, 2, 3, 4];
let squares = nums.map(num => num * num);
console.log(squares); // [1, 4, 9, 16]
```

Original array is **not changed**

◈ filter()

Returns a new array with elements that satisfy a condition.

```
let ages = [12, 18, 25, 14];
let adults = ages.filter(age => age >= 18);
console.log(adults);// [18, 25]
```

◈ forEach()

Runs a function for each element (does **not** return a new array).

```
let names = ["A", "B", "C"];
names.forEach(name => {
  console.log("Hello " + name);
});
```

Used for **logging, updating UI, side effects**

◈ includes()

Checks if a value exists in an array.

```
let colors = ["red", "green", "blue"];
console.log(colors.includes("green")); // true
console.log(colors.includes("yellow")); // false
```

◈ find()

Returns the **first element** that matches a condition.

```
let users = [
  { id: 1, name: "A" },
  { id: 2, name: "B" },
  { id: 3, name: "C" }
];
let user = users.find(u => u.id === 2);
console.log(user); // { id: 2, name: "B" }
```

Stops searching after first match

Situation	Use
Add item to list	push()
Remove last item	pop()
Modify every item	map()
Select some items	filter()
Perform action only	forEach()
Check if value exists	includes()
Find one matching item	find()

++ and -- Operators in JavaScript

Operator	Name	Meaning	When value is updated	Example

++a	Pre-increment	Increases value by 1 **before** using it	Immediately	let a = 5;
				let b = ++a;
				Result:
				a = 6
				b = 6

a++	Post-increment	Uses current value, then increases by 1	After use	let a = 5;
				let b = a++;
				Result:
				a = 6
				b = 5

--a	Pre-decrement	Decreases value by 1 **before** using it	Immediately	let a = 5;
				let b = --a;
				Result:
				a = 4
				b = 4

a--	Post-decrement	Uses current value, then decreases by 1	After use	let a = 5; let b = a--; **Result:** a = 4 b = 5

Combined Example

```
let x = 3;
let y = x++ + ++x;
console.log(x, y);
Output:
5 8
```

Explanation:

- x++ → uses 3, then x becomes 4
- ++x → increases to 5, then uses 5
- y = 3 + 5 = 8

Functions

Functions: Syntax, Calling function on some event

- A function is a group of reusable code which can be called anywhere in your program.
- This eliminates the need of writing the same code again and again.
- Functions allow a programmer to divide a big program into a number of small and manageable functions.
- Before we use a function, we need to define it.
- 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: **{}**

Syntax of a JavaScript Function

```
function Name(Parameter1, Parameter2, ...)
{
   // Function body
}
```

- A function in JavaScript is similar to a procedure—a set of statements that performs a task or calculates a value, but for a procedure to qualify as a function, it should take some input and return an output where there is some obvious relationship between the input and the output.
- To use a function, you must define it somewhere in the scope from which you wish to call it.

Function Invocation:

- Triggered by an event (e.g., a button click by a user).
- When explicitly called from JavaScript code.
- Automatically executed, such as in self-invoking functions.

1. Function Triggered by an Event (Button Click)

A function is executed when a **user performs an action**, such as clicking a button.

Example of function without passing parameter

```
<head>
    <script>
        function fun() {
            document.write ("Hello! How are you?");
        }
    </script>
</head>
<body> <input type = "button" onclick = "fun()" value = "Click"></body>
```

Example 2 : Alert

```
<head>
<script>
function showMessage() {
   alert("Button Clicked!");
}
</script>
</head>
<body>
<input type="button" value="Click Me" onclick="showMessage()">
</body>
```

Example of function with passing parameter

```
<head>
    <script>
        function fun(f_name,l_name) {
              document.write ("Hello " + f_name  + " " + l_name + " !");
         }
     </script>
 </head>
 <body> <input type = "button" onclick = "fun('ABC','XYZ')" value = "Click"></body>
```

2.Function Called Explicitly from JavaScript Code

A function is executed by calling it **directly from JavaScript code**.

Example

```
<html>
<body>
<script>
function add(a, b) {
   return a + b;
}

let result = add(5, 10);
document.write(result);
</script>
</body>
</html>
```

The function add() is called **explicitly** in the code.

JavaScript return statement

- The return statement is used to **send a value back** from a function.
- Once return is executed, the function **stops executing**.
- Code written after return is **not executed**.
- A function can return:
  - **Primitive values** (number, string, boolean)
  - **Objects** (array, object, function)

```
return value;
```

Example:

```
function Product(a, b) {
// Return the product of a and b
return a * b;
};
console.log(Product(6, 10));
```

Assigned function to the variable

```
    var a = function ( x, y ) {
      return x + y;
    }
    let result = a(3, 5);
document.write(result);
```

or

```
var res = fun(12, 30);
function fun(x,y)
{
return x * y;
}
document.write(res);
```

Example: One Function Calling Another Function

```
<html>
<head>
<script>
function info(first, last) {
   var full = first + " " + last;
   return full;
}
function hello() {
   var result = info('abc', 'xyz');
   document.write(result);
}
</script>
</head>
<body>
<input type="button" onclick="hello()" value="info">
</body>
</html>
```

3. Automatically Executed Function (Self-Invoking Function)

A self-invoking function runs **automatically without being called**.

Example

```
<html>
<body>
<script>
(function () {            //  This is an anonymous function because it has no name.
   document.write("This function runs automatically");
})();
</script>
</body>
</html>
```

This function executes **immediately when the page loads**.

Anonymous Function

```
(function() {
  var a = 20;
})();
```

- This is an anonymous function because it has no name.
- It is immediately executed — this is called an IIFE (Immediately Invoked Function Expression).
- var a = 20 is local to the function.
- After the function finishes, a is destroyed and cannot be accessed outside.

Template Literals

Template Literals are a modern way to create strings in JavaScript. They were introduced in ES6 and provide more flexibility than normal strings. Template literals make it easier to work with variables, expressions, and multiline text.

Syntax Difference

Normal Strings

Normal strings use single quotes (' ') or double quotes (" ").

```
Example:
let message = "Hello World";
```

Template Literals

Template literals use backticks (`).

```
Example:
let message = `Hello World`;
```

Variable Interpolation

Normal Strings

Variables must be joined using the + operator.

```
Example:
let name = "abc";
let text = "Hello " + name + "!";
```

Template Literals

Variables can be inserted directly using ${ }.

```
Example:
let name = "abc";
let text = `Hello ${name}!`;
```

Using Expressions

Template literals allow JavaScript expressions inside ${ }.

```
Example:
let a = 5;
let b = 3;
let result = `Sum is ${a + b}`;
```

Normal strings require extra brackets and + operators.

Multiline Strings

Normal Strings

Multiline text requires special characters like \n.

```
Example:
let text = "Hello\nHow are you?\nGoodbye";
Console.log(text);
```

Template Literals

Multiline strings work naturally.

```
Example:
let text = `Hello
How are you?
Goodbye`;
Console.log(text);
```

Creating HTML Content

Normal Strings

HTML strings require heavy concatenation.

```
Example:
let html = "<div>"+
"<h1>Title</h1>" +
"<p>Text</p>" +
"</div>";
document.write(html);
```

Template Literals

HTML is easier to read and write.

```
Example:
let html = `<div>
<h1>Title</h1>
<p>Text</p>
</div>`;
document.write(html);
```

ES6 Functions

Arrow functions, introduced in ES6 (ECMAScript 2015), provide a more concise syntax for writing functions in JavaScript.

Syntax

```
const  functionname = (param1, param2, ..., paramN) => {
                        // body of function expression
                                }
```

- **Parameters**: Listed in parentheses. If there is only one parameter, the parentheses can be omitted.
- **Expression**: After the arrow (=>), if there's a single expression, it is implicitly returned.

Different cases

1. Arrow Function with Multiple Parameters

```
const add = (a, b) => a + b;
console.log(add(3, 4)); // Output: 7
```

If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword.

2. Arrow Function with a Single Parameter

```
const square = x => x * x;
console.log(square(5)); // Output: 25
```

OR

```
const square = (x) => x * x;
console.log(square(5)); // Output: 25
```

3. Arrow Function with No Parameters

```
const sayHello = () => "Hello!";
console.log(sayHello()); // Output: "Hello!"
```

Note: This works only if the function has only one statement.

4. Arrow Function with a Block Body

For functions with multiple statements, use a block body with curly braces {}. You must use an explicit return statement to return a value.

```
const multiply = (a, b) => {
   const result = a * b;
   return result;
};
console.log(multiply(2, 3)); // Output: 6
```

If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword:

Arrow Functions Return Value by Default:

```
fun1 = () => "Hello World!";
console.log(fun1())
```

Example 1

```
<script>
const newfun = p => `val=${p}`;
console.log(newfun(30));

fun1 = (val) => "Hello " + val;
console.log(fun1('world'))
</script>
```

Example 2

```
<script>
const add = (j,k) => `${j+k}`;
console.log(`Addition = ${add(5,9)}`);
</script>
```

Example 3

```
<script>
const add = (j,k) =>{
let ans = j+k;
return ans;
}
console.log(`Addition = ${add(10,15)}`);
</script>
```

Understand concept of below example

Case-1

The function doesn't explicitly return anything because there is no return statement. In JavaScript, when a function doesn't return a value explicitly, it implicitly returns undefined. add(8, 9) is called, but it returns undefined. So, console.log concatenates "Ans=" with undefined.

```
<script>
const add=(j,k)=> { ans=`${j+k}`; }
console.log("Ans="+add(8,9));
</script>
```

Output : Ans=undefined

Case-2

The function uses a concise arrow function syntax (without curly braces). In such cases, the value of the expression after the arrow (=>) is implicitly returned.
The expression ans = \${j + k}`` is evaluated and returned.

```
<script>
const add=(j,k)=>  ans=`${j+k}`;
console.log("Ans="+add(8,9));
</script>
```

Output : Ans=17

Case-3

The function uses the full arrow function syntax with curly braces {}.
Inside the function body, the return statement explicitly returns ans = \${j + k}``.

```
<script>
const add=(j,k)=> { return ans=`${j+k}`; }
console.log("Ans="+add(8,9));
</script>
```

Output : Ans=17

Implicit means the return happens automatically without writing return, while **explicit** means the return value is clearly specified using the return keyword.

In arrow functions, when curly braces {} are used, the return keyword is required; when curly braces are not used, the expression value is returned automatically.

Rest parameter

- **The rest parameter is an improved way to handle function parameters, allowing us to more easily handle various inputs as parameters in a function.**
- The rest parameter syntax allows us to represent an indefinite number of arguments as an array.
- With the help of a rest parameter, a function can be called with any number of arguments, no matter how it was defined.
- Rest parameter is added in ES2015 or ES6 which improved the ability to handle parameter.

Syntax:

```
//... is the rest parameter (triple dots)
function functionname(...parameters)
{
statement;
}
```

Note: When … is added with parameter then it is the rest parameter. It stores n number of parameters as an array.

Example: without rest parameter. (passing arguments more than the parameters**)**

```
  function fun(a, b){
      console.log(a);
      console.log(b);
   }
   fun(1,2,3,4,5);
```

Output:

```
1
2
```

- ✓ In the above code, no error will be thrown even when we are passing arguments more than the parameters, but only the first two arguments will be evaluated.
- ✓ It's different in the case of the rest parameter. With the use of the rest parameter, we can gather any number of arguments into an array and do what we want with them.

Example using rest parameter

```
function fun(a, b, ...c){
      console.log(a);
      console.log(b);
      console.log(c);
   }
   fun(1, 2);
   fun(1, 2, 3, 6, 5);
```

Output:

```
1
2
```

```
[3,6,5]
```

Note: The rest parameter has to be the last argument, as its job is to collect all the remaining arguments into an array.

Example: In this example, we are using the rest parameter with some other arguments inside a function.

```
function fun(a, b, ...c) {
   console.log(`${a} ${b}`); //abc xyz
   console.log(c);  //[ 'pqr', 'mno', 'stu']
   console.log(c[0]); //pqr
   console.log(c.length); //3
   console.log(c.indexOf('pqr')); //0
   console.log(c.indexOf('abc')); //-1
}
fun('abc', 'xyz', 'pqr', 'mno', 'stu');
```

In the above code sample, we passed the rest parameter as the third parameter, and then we basically called the function fun() with five arguments the first two were treated normally and the rest were all collected by the rest parameter hence we get ‘'pqr' when we tried to access c[0] and it is also important to note that the rest parameter gives an array in return and we can make use of the array methods that the javascript provides us.

The rest parameter allows us to represent an indefinite number of arguments as an array. By using the rest parameter, a function can be called with any number of arguments.

```
<script>
function f(...args)
{ console.log(args); }
f(1,2,3,4,5,6,7,8,9,10);
</script>
```

Output:

[1,2,3,4,5,6,7,8,9,10]

Spread Operator

- The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.
- **The Spread operator** allows an iterable to expand in places where zero plus arguments are expected.
- It is mostly used in the variable array where there is more than 1 value is expected. It allows us the privilege to obtain a list of parameters from an array.
- **The syntax of the Spread operator is the same as the** Rest parameter **but it works opposite of it.**

Example:

```
<html> <body> <script>
odd = [1,2,4];
combined = [5,6,7,...odd];
console.log(combined);
</script> </body></html>
```

Output:

(6) [5, 6, 7, 1, 2, 4]

Example:

```
arr1 = [10, 42, 35];
arr2 = [14, 56];

// normally used expand method
let arr_test = [arr1, arr2];
console.log(arr_test); //[[10, 42, 35],[14, 56]]

// spread
arr = [...arr1, ...arr2];
console.log(...arr);  //10 42 35 14 56 //shows value
console.log(arr); //[10,42,35,14,56]  // shows array
console.log(Math.min(arr)); //NAN
console.log('Using spread operator: ' + Math.min(...arr)); //10
console.log(`Sorted: ${arr.sort()}` ); //Sorted: 10,14,35,42,56
```

Default parameters

Function parameters with default values are initialized with default values if they contain no value or are undefined.

JavaScript function parameters are defined as undefined by default.

However, it may be useful to set a different default value. That is where default parameters come into play.

Syntax:

```
function name(parameters_n, parameter_n=value) {
//Code
}
```

Example 1: If we multiply two numbers in below example without passing a second argument and without using the default parameter, the answer that this function will return is NAN(Not a Number), since if we do not pass the second parameter, the function will multiply the first number with undefined.

```
function multiply(a, b) {
   return a * b;
}
let num1 = multiply(5);
console.log(num1);
let num2 = multiply(5, 8);
console.log(num2);
```

Output:

```
NaN
40
```

Example 2: If we do not pass a number as the second argument and take the default parameter as the second parameter, it will multiply the first number with the default parameter, and if we pass two numbers as parameters, it will multiply the first number with the second number.

```
function multiply(a, b = 2) {
   return a * b;
}
let num1 = multiply(5);
console.log(num1);
let num2 = multiply(5, 8);
console.log(num2);
```

Output:

```
10
40
```

Default argument must be filled from rear(right) side only

Example to understand above sentence.

```
<script>
defaultfun = (p=2,q)=>{
return(p+q);
}
console.log(`Add =${defaultfun(30)}`);
</script>
```

Output:

NaN

Passed argument assigned to the p so here q is undefined. Answer is NaN

The simple solution is to have all the parameters with default values on the right (rear side).

Example: Write Es6 function to be called on button click. Take one division with some text and no decoration initially. Take font size and style as default argument and pass background color and text color while passing argument to function style should change on button click.

```
<html>
<head>
</head>
<body>
<script>
function fun(bg,tx,size="50px",fs="italic")
{
i=document.getElementById("d1");
i.style.color=tx;
i.style.fontStyle=fs;
i.style.backgroundColor=bg;
i.style.fontSize=size;
}
</script>
<div id="d1">Hello</div>
<input type="submit" onclick="fun('pink','blue')" value="CLICK HERE"/>
</body>
```

Pop up Boxes: Alert, Confirm, Prompt

Alert Box

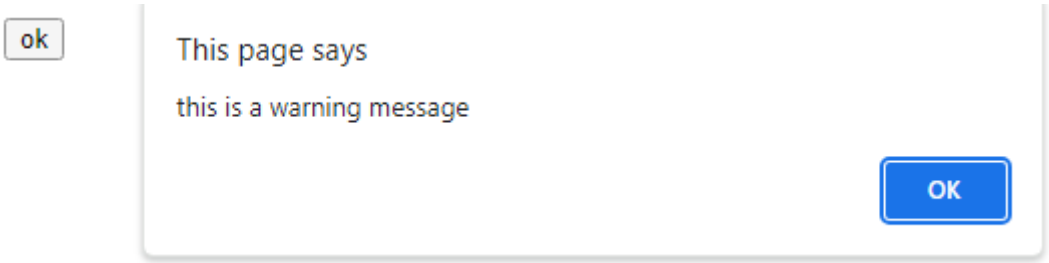
- An alert dialog box is mostly used to give a warning message to the users. For example, if one input field requires to enter some text but the user does not provide any input, then as a part of validation, you can use an alert box to give a warning message.

- Nonetheless, an alert box can still be used for friendlier messages. Alert box gives only one button "OK" to select and proceed.

Example

```
<html>
<head>
<script>
function warn()
{
   alert("this is a warning message");
   document.write("Warning!!!!");    //It will be printed after closing alert box.//
}
</script>
</head>
<body>
   <input type="button" value="ok" onclick="warn()"/>
</body>
</html>
```

Output:



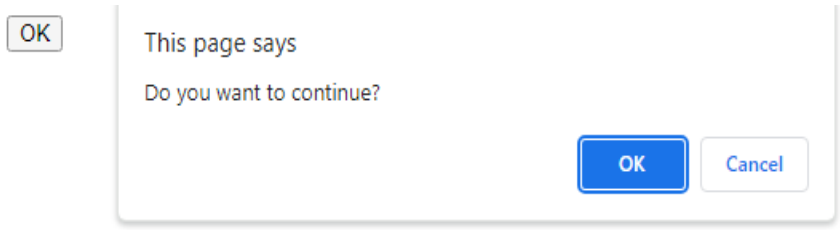
Confirmation Box

- A confirmation dialog box is mostly used to take user's consent on any option. It displays a dialog box with two buttons: OK and Cancel.
- If the user clicks on the OK button, the window method confirm() will return true. If the user clicks on the Cancel button, then confirm() returns false. You can use a confirmation dialog box as follows.

Example

```
<head>
<script>
function conf()
{
res = confirm("Do you want to continue?");     //It returns value as true/false
if(res==true)
{
   document.write("user wants");
   return true;
}
else
{
   document.write("user does not want");
   return false;
}
}
</script>
</head>
<body>
   <input type="button" onclick="conf()" value="OK"/>
</body>
```

Output:



Prompt Dialog Box

- The prompt dialog box is very useful when you want to pop-up a text box to get user input.
- Thus, it enables you to interact with the user. The user needs to fill in the field and then click OK. If user clicks on cancle button, it returns Null.

This dialog box is displayed using a method called **prompt()** which takes two parameters:

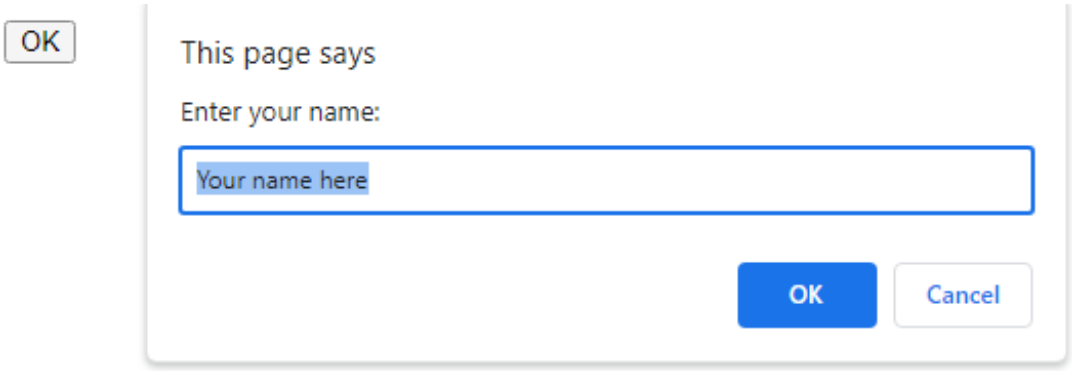
(i) a label which you want to display in the text box and

(ii) a default string to display in the text box.

Example:

```
<html>
<head>
<script type="text/javascript">
function getval()
{
   var res = prompt("Enter your name: ", "Your name here");
   document.write("you entered "+res);
}
</script>
</head>
<body>
<input type="button" onclick="getval()" value="OK"/>
</body>
</html>
```

Output:



Methods

Method	Purpose	Syntax	Return Type	Example	Output
toFixed()	Formats number to fixed decimal places	num.toFixed(n)	String	(12.345).toFixed(2)	"12.35"
isNaN()	Checks if value is Not-a-Number	isNaN(value)	Boolean	isNaN("abc")	true
parseInt()	Converts string to integer	parseInt(value)	Number	parseInt("45.6")	45
parseFloat()	Converts string to decimal number	parseFloat(value)	Number	parseFloat("45.6")	45.6
entries()	Returns index-value pairs of array	array.entries()	Iterator	[10,20].entries()	[0,10], [1,20]

1. toFixed()

Purpose: Formats a number to a fixed number of decimal places.

```
let price = 99.5678;
let formattedPrice = price.toFixed(2);
console.log(formattedPrice);
Output:
99.57
```

Explanation:
Rounds the number to **2 decimal places** and returns it as a **string**.

2. isNaN()

Purpose: Checks whether a value is **Not-a-Number**.

```
let value1 = "abc";
let value2 = 25;
console.log(isNaN(value1));
console.log(isNaN(value2));
Output:
true
false
```

Explanation:

- "abc" is not a number → true
- 25 is a valid number → false

3. parseInt()

Purpose: Converts a string into an **integer**.

```
let data = "45.89";
let result = parseInt(data);
console.log(result);
```

Output:

45

Explanation:

Stops conversion at the decimal point.

4. parseFloat()

Purpose: Converts a string into a **decimal number**.

```
let data = "45.89";
let result = parseFloat(data);
console.log(result);
```

Output:

45.89

Explanation:

Keeps the decimal part while converting.

5. entries()

Purpose: Returns **index-value pairs** from an array.

```
let marks = [70, 80, 90];

for (let [index, value] of marks.entries()) {
  console.log("Index:", index, "Value:", value);
}
```

Output:

Index: 0 Value: 70
Index: 1 Value: 80
Index: 2 Value: 90

Example

Example-1

You are given a dataset of student test scores.

```
const students = [
   { name: 'Alice', scores: { math: 85, science: 90, history: 78 } },
   { name: 'Bob', scores: { math: 92, science: 88, history: 95 } },
   { name: 'Charlie', scores: { math: 70, science: 75, history: 80 } },
   { name: 'Diana', scores: { math: 95, science: 98, history: 93 } }
];
```

Write code to perform the following tasks:

1. Find the student with the highest average score across all subjects.
2. Return a new object containing only that student's name and their average score, formatted to two decimal places.

You must calculate the average score for each student to find the highest, and your final result should be a single object.

```
const students = [
   { name: 'Alice', scores: { math: 85, science: 90, history: 78 } },
   { name: 'Bob', scores: { math: 92, science: 88, history: 95 } },
   { name: 'Charlie', scores: { math: 70, science: 75, history: 80 } },
   { name: 'Diana', scores: { math: 95, science: 48, history: 93 } }
];

let topStudent = {};
let maxAvg = 0;

for (let s of students) {
  let avg = (s.scores.math + s.scores.science + s.scores.history) / 3;

  if (avg > maxAvg) {
    maxAvg = avg;
    topStudent = {
      name: s.name,
      average: avg.toFixed(2)
    };
  }
}

console.log(topStudent);
```

Example-2

Write a function gradeStudents that takes an array of student objects (each with a name and FSD marks property).

The function should:
1. Iterate through the student array.
2. For each student, it should use the prompt() pop-up box to ask the user to enter FSD marks. The prompt message should clearly state the student's name (e.g., "Enter FSD marks for Alice:").
3. It should validate the user input. If the input is not a valid number (use isNaN()), display an alert() message "Invalid grade. Please enter a number." and continue to the next student without updating the current one.
4. If the input is valid, update the student's FSD marks property with the new value.
5.Finally, after the loop finishes, it should use alert() to display a summary of the FSD marks for all students in a clear, multi-line format using template literals.

```
<script>
const students = [
  { name: "test", FSD: 85 },
  { name: "nisha", FSD: "abc" },   // invalid
  { name: "megha", FSD: 92 }
];
    function gradeStudents(students) {
  let summary = "";
  for (let student of students) {
    let marks = prompt(`Enter FSD marks for ${student.name}:`);

    // validation
    if (marks === "" || isNaN(marks)) {
      alert("Invalid grade. Please enter a number.");
      continue;
    }
    // update FSD marks
    student.FSD = Number(marks);
    // build summary WITHOUT +=
    summary = summary + `${student.name}: ${student.FSD}\n`;
  }
  // final output
  alert(`FSD Marks Summary:\n${summary}`);
}
gradeStudents(students);
</script>
```

Example -3

Create a function registerUser() that uses pop-up boxes to gather and validate user input for a new user registration.

The function should:

1. Use a while loop to repeatedly prompt the user for a username using the prompt() pop-up box. The loop should continue to run as long as the user's input is an empty string, null, or undefined.
2. Once a valid username is provided, use another while loop to prompt the user for a password. This loop should continue until the user provides a password that is at least 8 characters long.
3. After a valid password is entered, use the confirm() pop-up box to ask the user, "Are you sure you want to register with this username and password?".
4. If the user clicks "Cancel" on the confirmation box, use alert() to display a message "Registration canceled." and end the function.
5. If the user clicks "OK", use a template literal to display a final alert() message confirming the successful registration, showing both the username and the password.

```
<script>
function registerUser() {
  let username;
  // 1. Ask for username until valid
  while (username === "" || username === null || username === undefined) {
    username = prompt("Enter username:");
  }
  let password;
  // 2. Ask for password until length >= 8
  while (password === undefined || password === null || password.length < 8) {
    password = prompt("Enter password (minimum 8 characters):");
  }

  // 3. Confirmation
  let confirmRegister = confirm(
    "Are you sure you want to register with this username and password?"
  );

  // 4. If Cancel clicked
  if (confirmRegister === false) {
    alert("Registration canceled.");
    return;
  }
  // 5. If OK clicked
  alert(`Registration successful!
Username: ${username}
Password: ${password}`);
}
registerUser();
</script>
```

Example -4

Write a JavaScript program that first uses the alert box to display a welcome message to the user (Example: Welcome User). Then, use the confirm box to ask the user if they want to continue.
1. If the user clicks "OK", proceed to ask for their name using the prompt box and display a personalized greeting (Example: If user enters “Jack”, then display “Hello Jack”). After displaying the greeting using alert box, prompt the user to enter their age.
Based on the user's age, use a conditional statement to check if they are 18 or older. If the user is 18 or older, display an alert saying "You are an adult!". If the user is under 18, display an alert saying "You are a minor!".
2. If the user clicks "Cancel" in the confirm box at the beginning, display an alert saying "Goodbye".

```
<script>

  // Welcome message
  alert("Welcome User");

  // Ask user if they want to continue
  let choice = confirm("Do you want to continue?");

  if (choice === true) {
    // Ask for name
    let name = prompt("Enter your name:");

    // Display greeting
    alert("Hello " + name);

    // Ask for age
    let age = prompt("Enter your age:");

    // Check age condition
    if (age >= 18) {
      alert("You are an adult!");
    } else {
      alert("You are a minor!");
    }

  } else {
    // If Cancel is clicked
    alert("Goodbye");
  }
</script>
```

.

Example -5

Write a JavaScript program that displays an alert message **"welcome user"** when the page loads. Then, display a confirmation box asking **"Do you want to continue?"**

- If the user clicks **OK**, prompt the user to enter two numbers.
- Calculate the sum of the two numbers.
- Display the result in an alert box.
- If the user clicks **Cancel**, display an alert message **"bye bye"**.

```
alert("welcome user");

/* Confirmation box */
var res=confirm("Do you want to continue?");
if (res===true) {

   /* Prompt boxes */
   var num1 = prompt("Enter Number 1:");
   var num2 = prompt("Enter Number 2:");

   /* Convert to numbers and add */
   var sum = parseInt(num1) + parseInt (num2);

   /* Display result */
   alert("Addition of two numbers is: " + sum);
}else{
  alert("bye bye")
}
```