PREFACE

ABOUT THE BOOK

This book contains around 140+ very important JavaScript interview questions.

ABOUT THE AUTHOR

Jagmohan Rai has around 3.5 years of experience in software development. He helps candidates in clearing technical interview in tech companies.

JS Chapters

Fundamentals

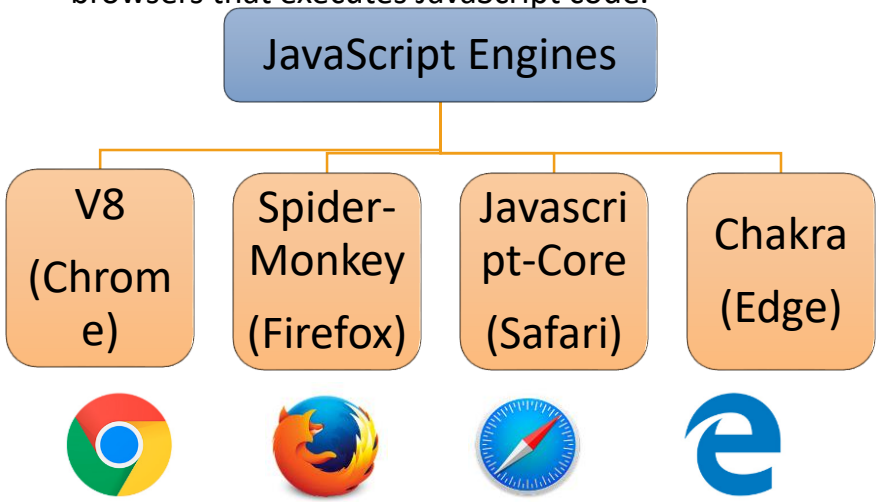
1. Basics & Fundamentals
2. Variables & Datatypes
3. Operators & Conditions
4. Arrays
5. Loops
6. Functions
7. Strings
8. DOM
9. Error Handling
10. Objects

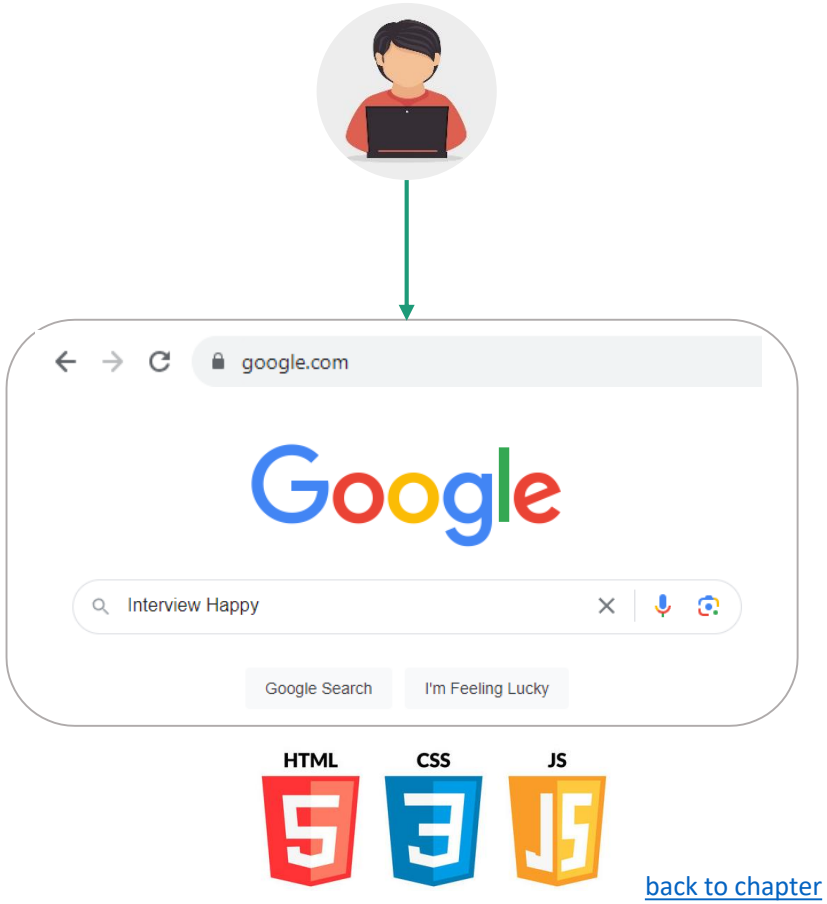
Advanced

11. Events
12. Closures
13. Asynchronous - Basics
14. Asynchronous - Promises
15. Asynchronous - Async Await
16. Browser APIs & Web Storage
17. Classes & Constructors
18. ECMAScript & Modules
19. Security & Performance

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>
```

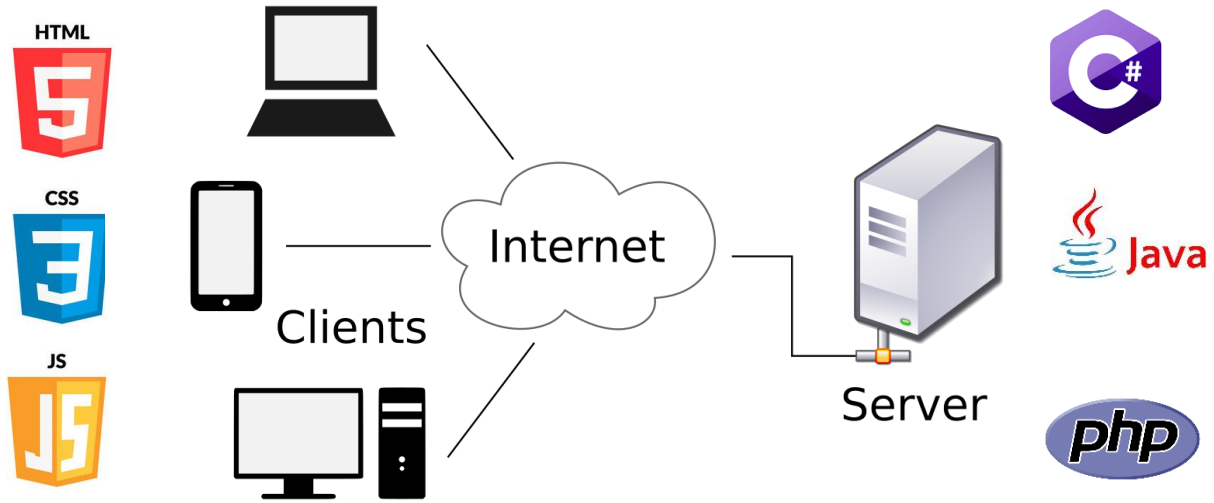
index.js > ...

```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.

- 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?

```
//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
```

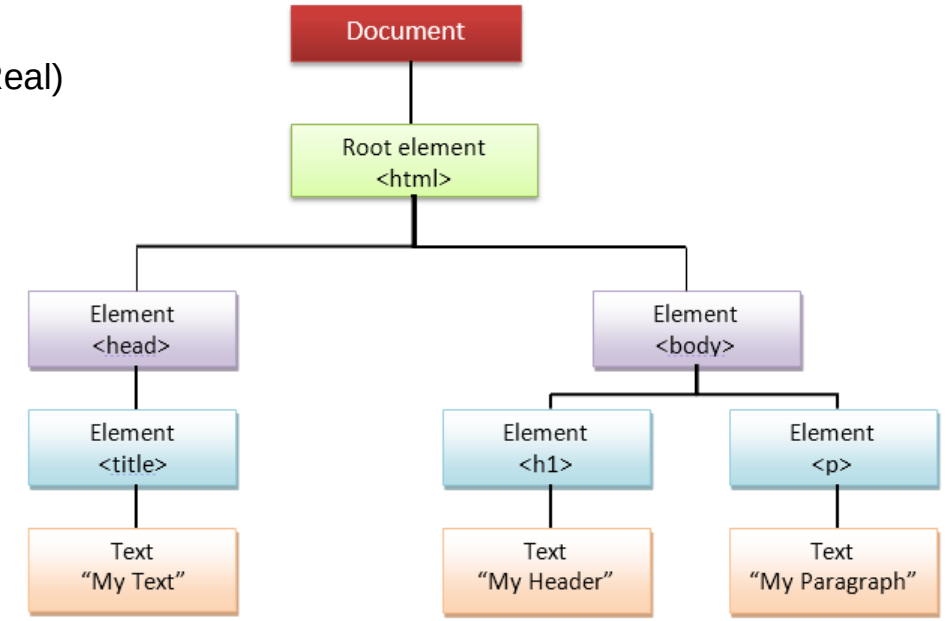
back to chapter index

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)

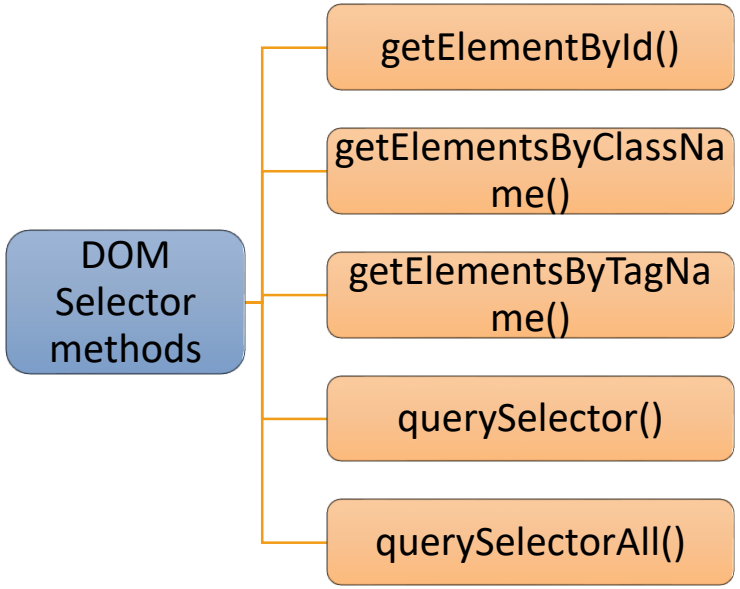


- The DOM(Document Object Model) represents the web page as a **tree-like structure** that allows JavaScript to dynamically access and manipulate the content and structure of a web page.

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.




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

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

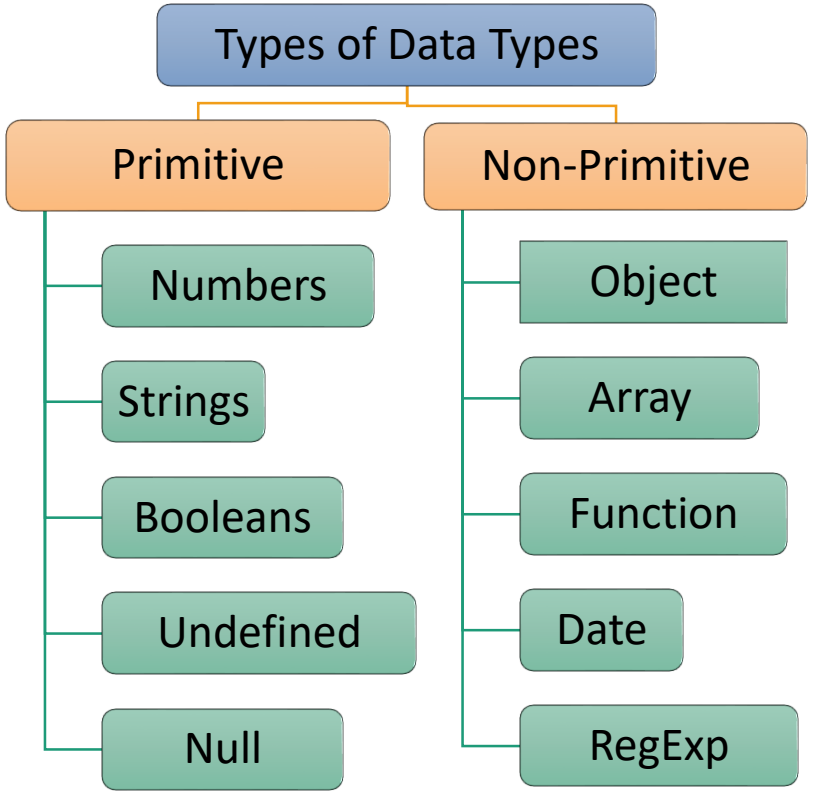
back to chapter index

Q. What are data types in JS?

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

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

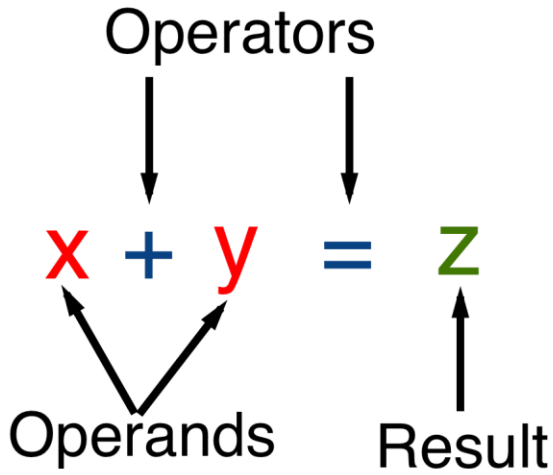
```
//String
let message = 'Hello!';

//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

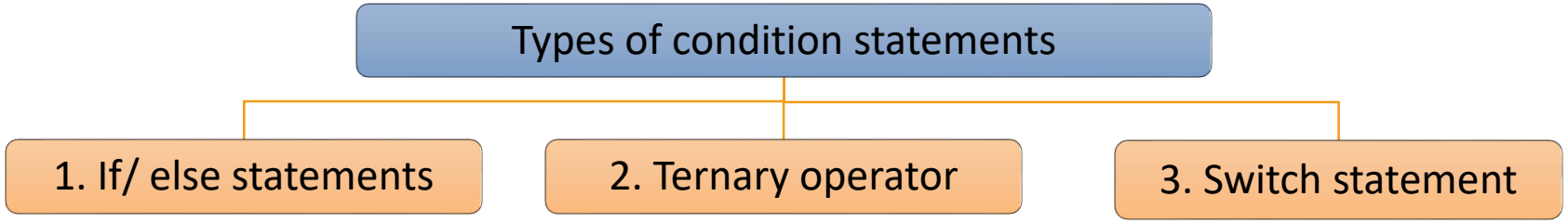
Q. What are operators? What are the types of operators in JS? V. IMP.

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



back to chapter index

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




```
let x = 5;

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

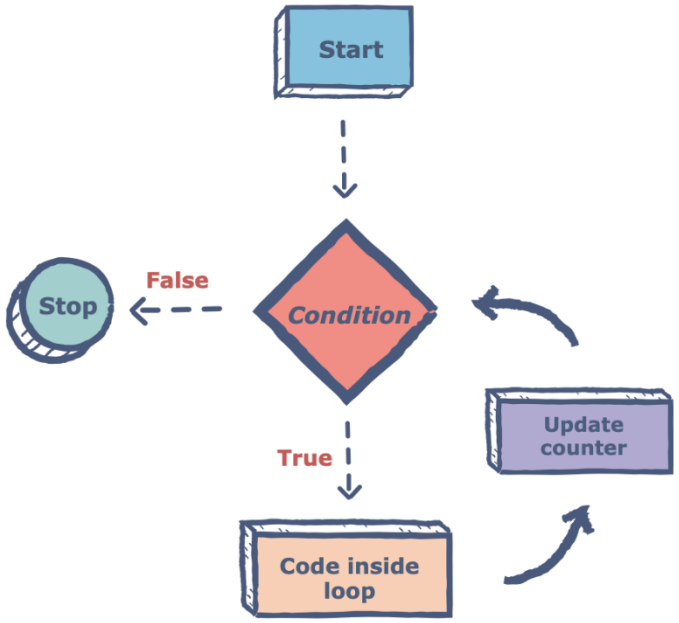
```
let y = 20;
let z = y > 10 ? "1" : "0"
console.log(z);
// Output: '1'
```

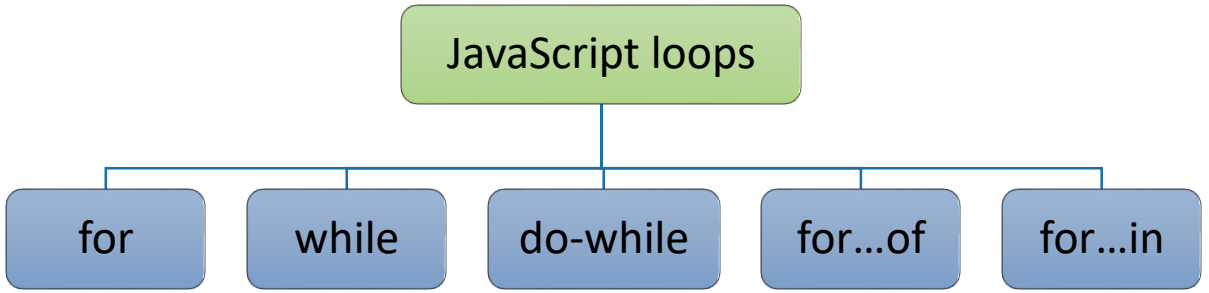
```
let a = 5;
switch (a) {
  case 1:
    console.log("1");
    break;
  case 5:
    console.log("2");
    break;
  default:
    console.log("3");
}
// Output: '2'
```

back to chapter index

Q. 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.





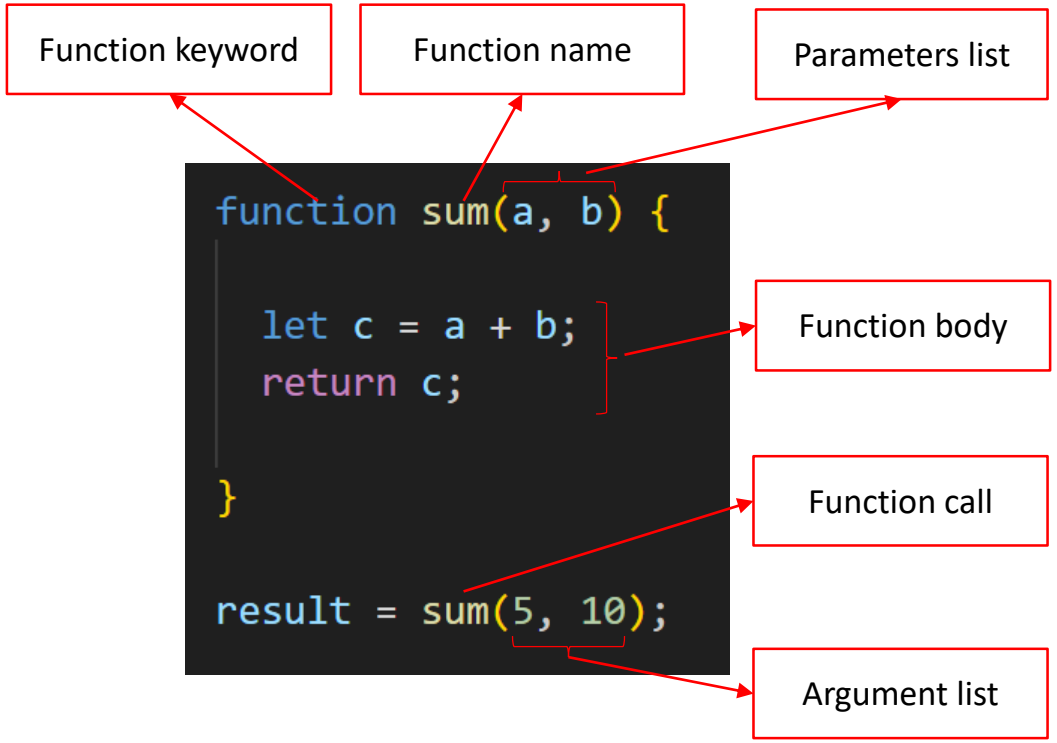
1. Initialization
2. Condition
3. Increment

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

    console.log(i);

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

back to chapter index

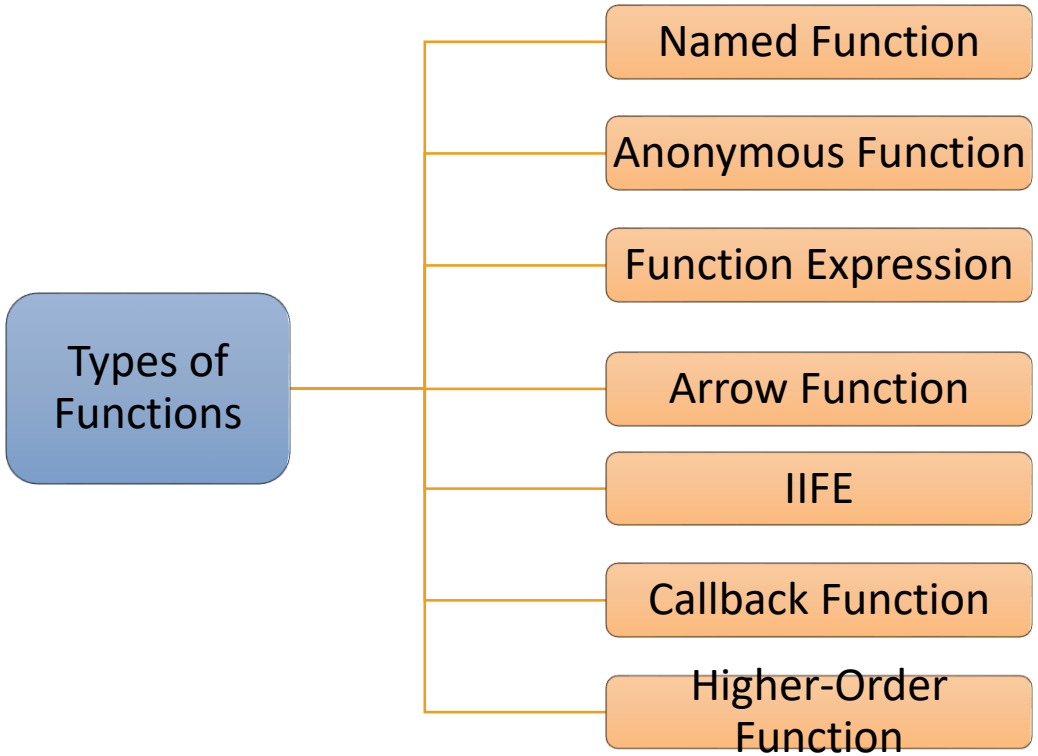
Q. What are Functions in JS? What are the types of function? V. IMP.

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



back to chapter index

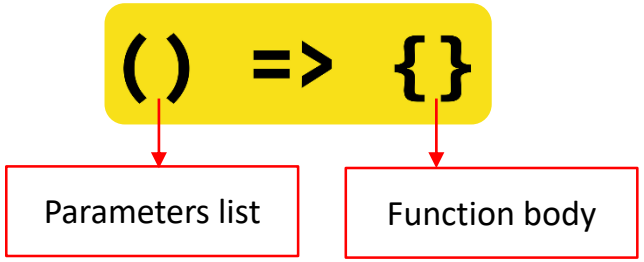
Q. What are Functions in JS? What are the types of function? V. IMP.



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
```

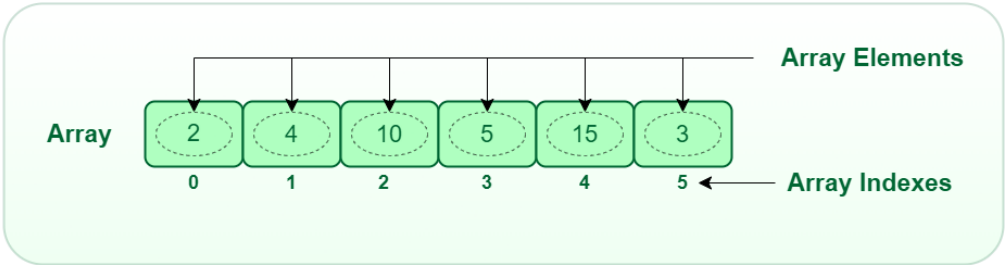
```
//Arrow function

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

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

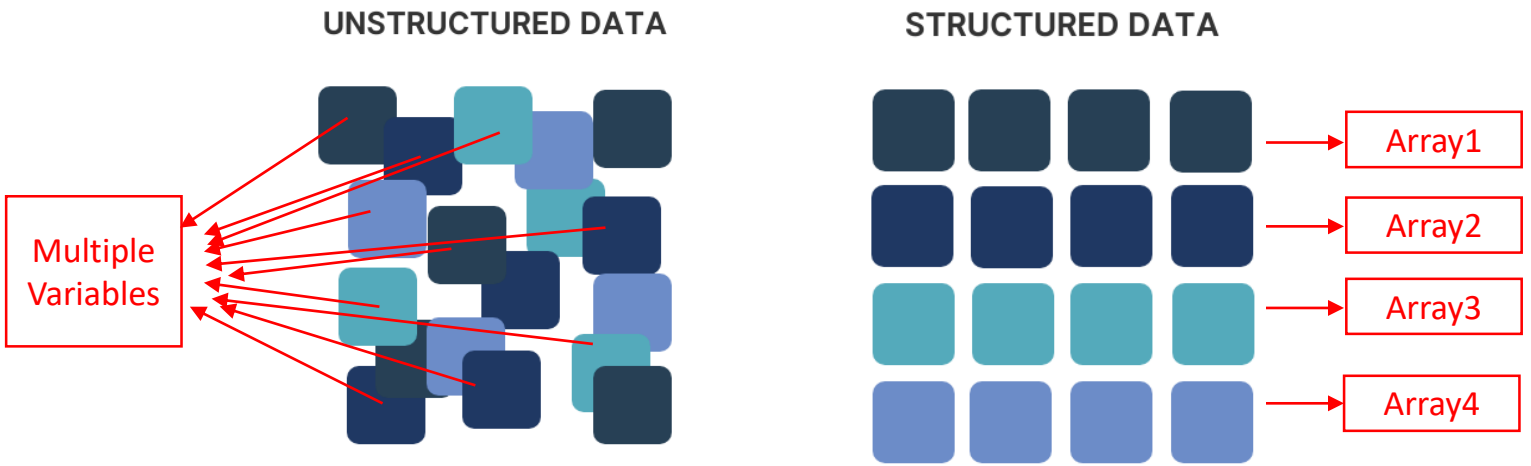
back to chapter index

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 Objects in JS? V. IMP.

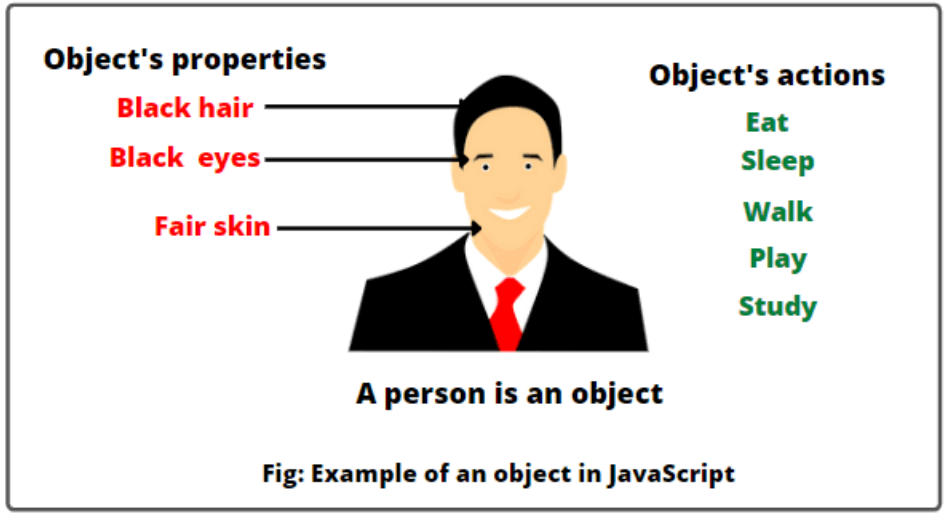
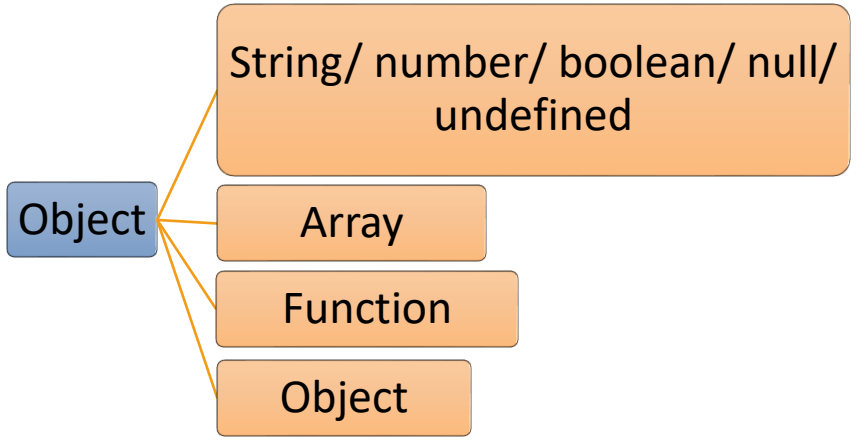


Fig: Example of an object in JavaScript

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**. There are 5 types scopes in js below is 3 types 2 types you have to explore.

Global

Function

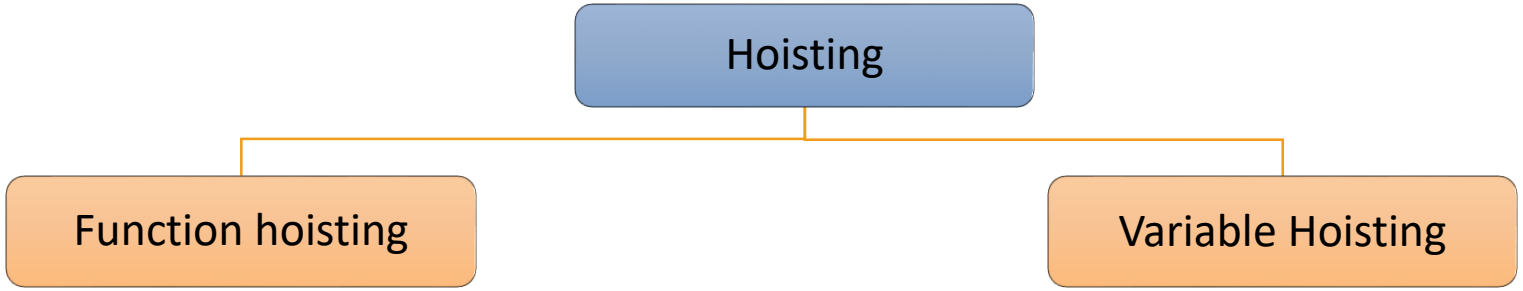
Block

```
//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. (It is not a correct definition go to MDN and get correct one but in interview you can tell. There is one more hoisting read it.



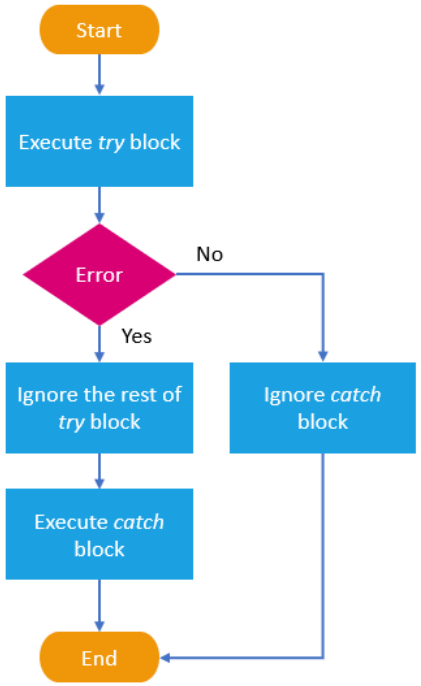

```
//Function hoisting
myFunction();

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**.

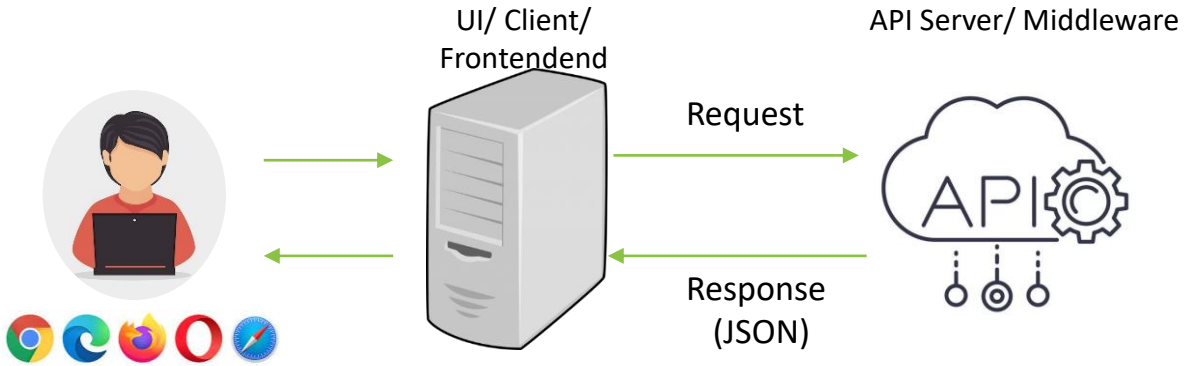



```
//try block contains the code that might throw an error
try {

  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?

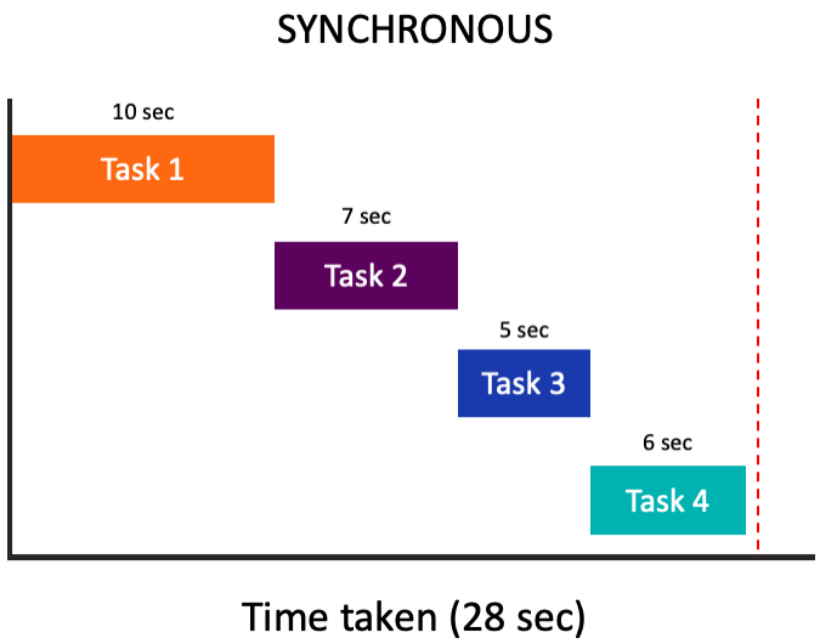
- JSON (JavaScript Object Notation) is a lightweight **data interchange format**.
- JSON consists of **key-value** pairs.

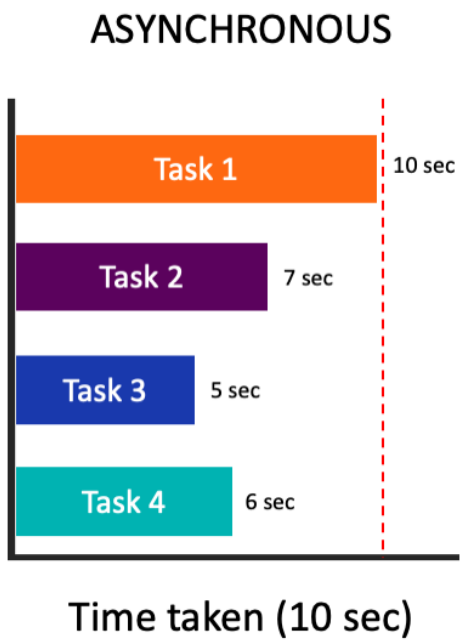



```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "country": "USA"
  }
}
```

back to chapter index

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

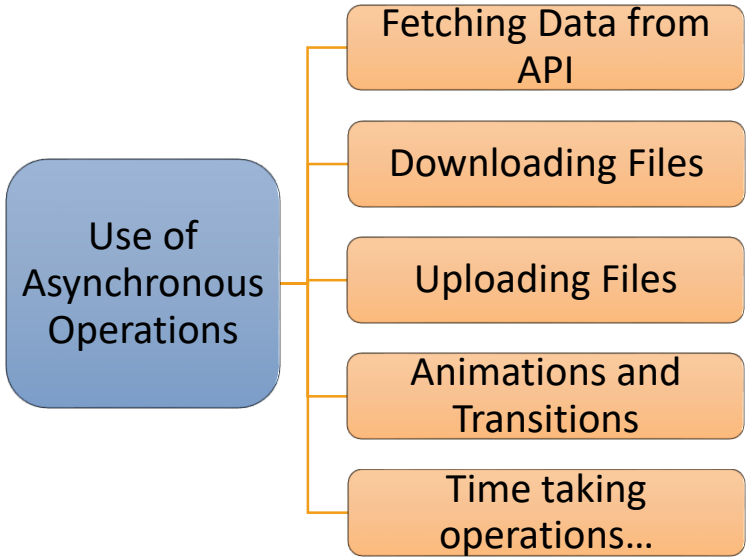




back to chapter index

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 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

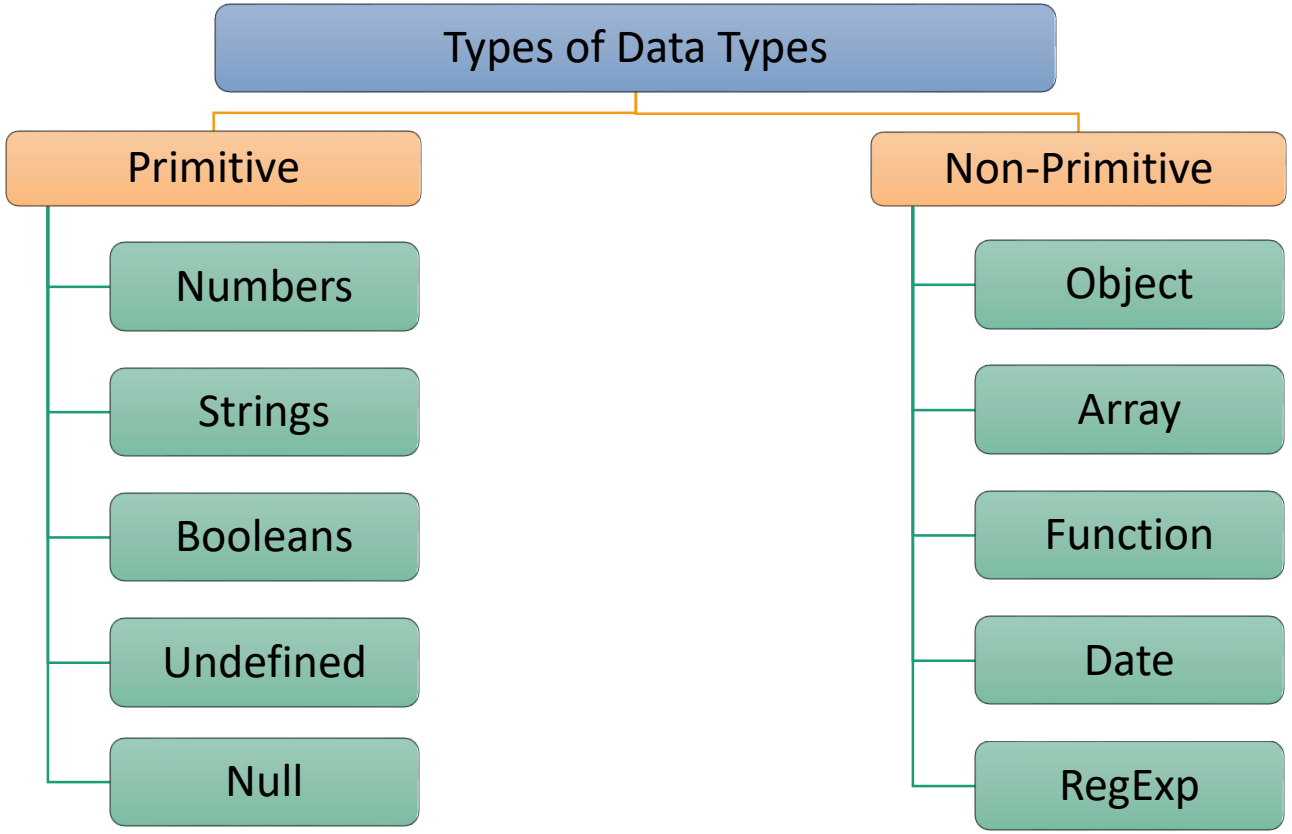
```
//using let
function example() {

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

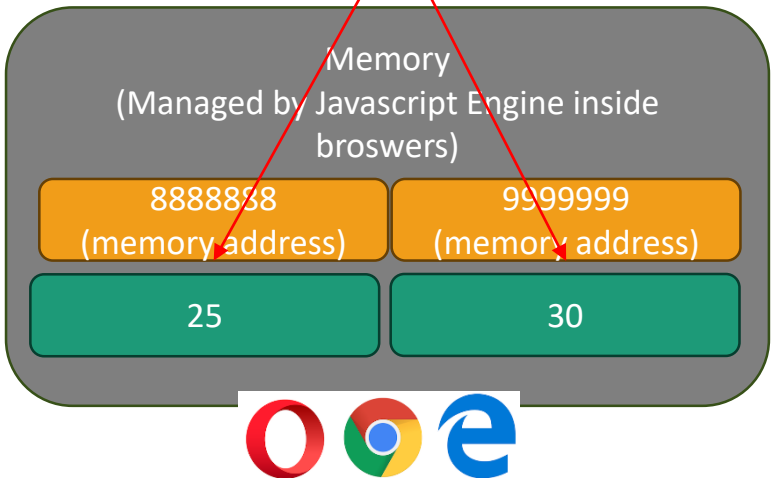


back to chapter index

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 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?

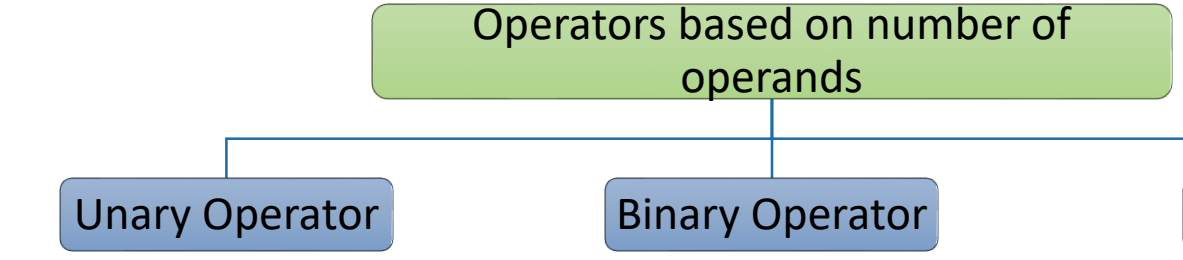
- typeof operator is used to determine the **type** of each variable.
- Real application use -> typeOf operator can be used to **validate the data** received from external sources(api).

```
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 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
```

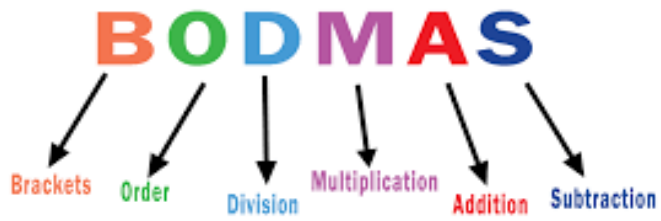
```
let x = 10;
let y = 3;

// 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 operator precedence?

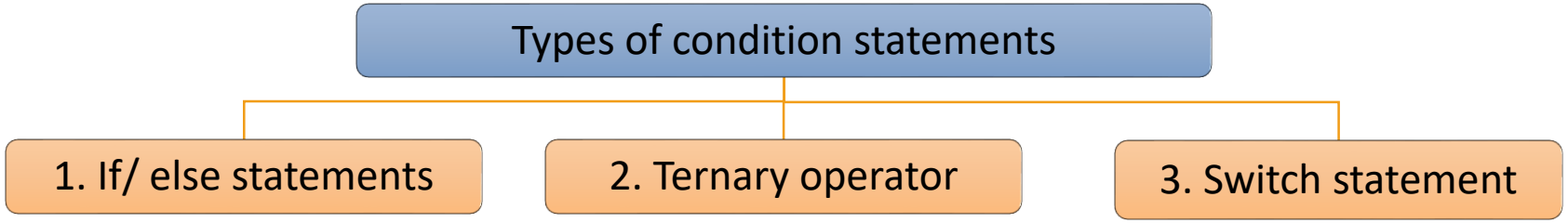
- As per operator precedence, operators with higher precedence are **evaluated first.**




```
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.




```
let x = 5;

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

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

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

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

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

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

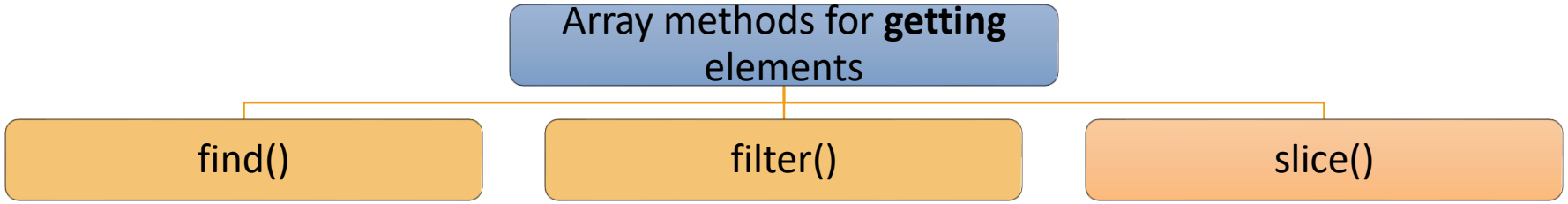
Q. What is the indexOf() method of an Array?

- 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.



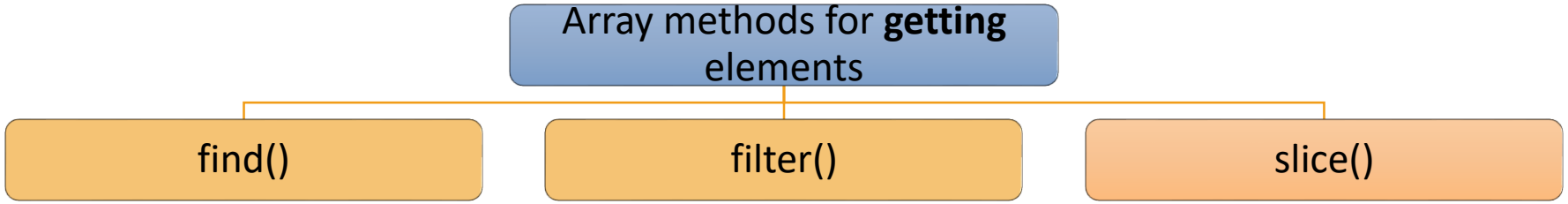

```
// Example array
const array = [1, 2, 3, 4, 5];

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.

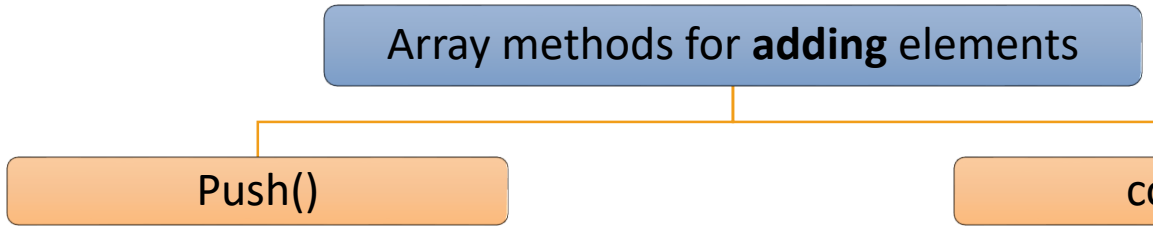



```
const array = ["a", "b", "c", "d", "e"];

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?



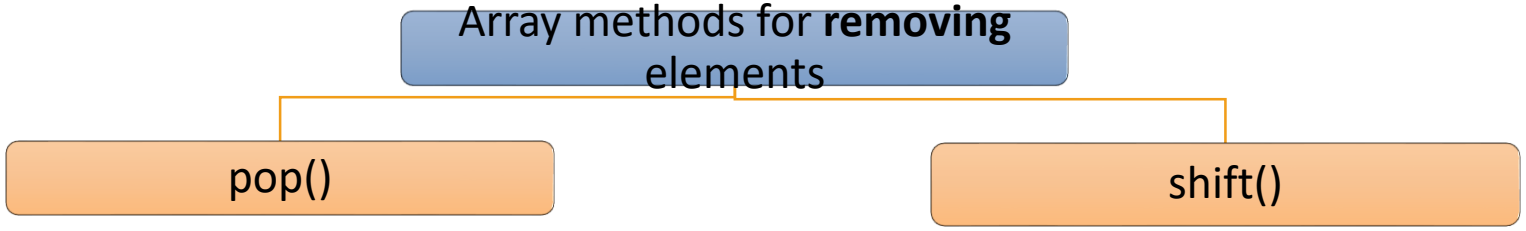

```
let array1 = [1, 2];
// Using push()
array1.push(3, 4);
console.log(array1);
// Output: [1, 2, 3, 4]
```

```
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]
```

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

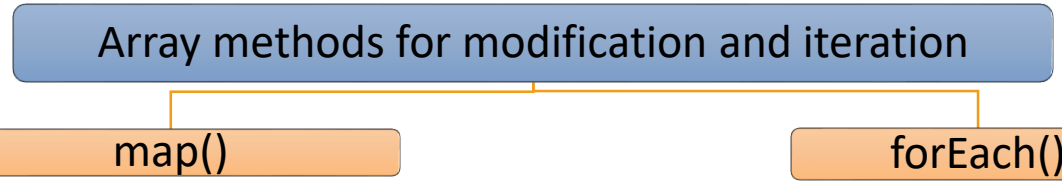
back to chapter index

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

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

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

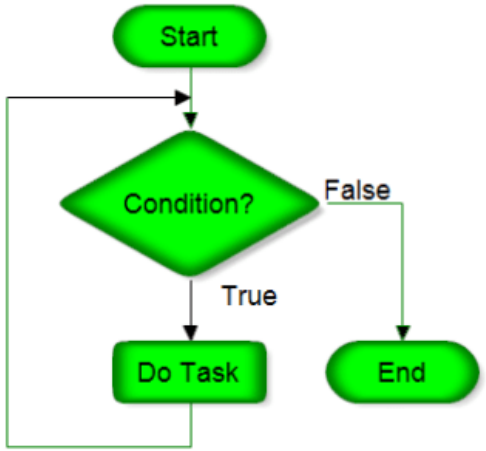
```
// Output
console.log(firstFruit); // Output: "apple"
console.log(secondFruit); // Output: "banana"
console.log(thirdFruit); // Output: "orange"
```

- 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

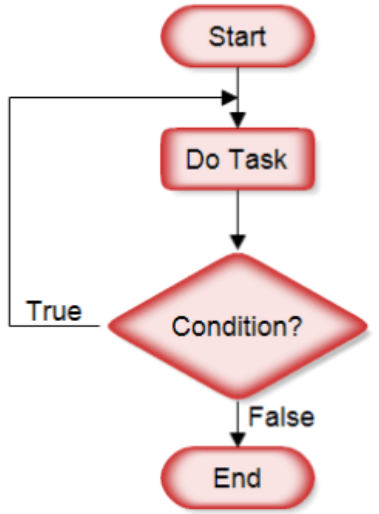
Q. What is the difference between while and do-while loops? V. IMP.

While Loop



- While loop execute a block of code while a certain **condition** is true.

Do While Loop



- 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];
```

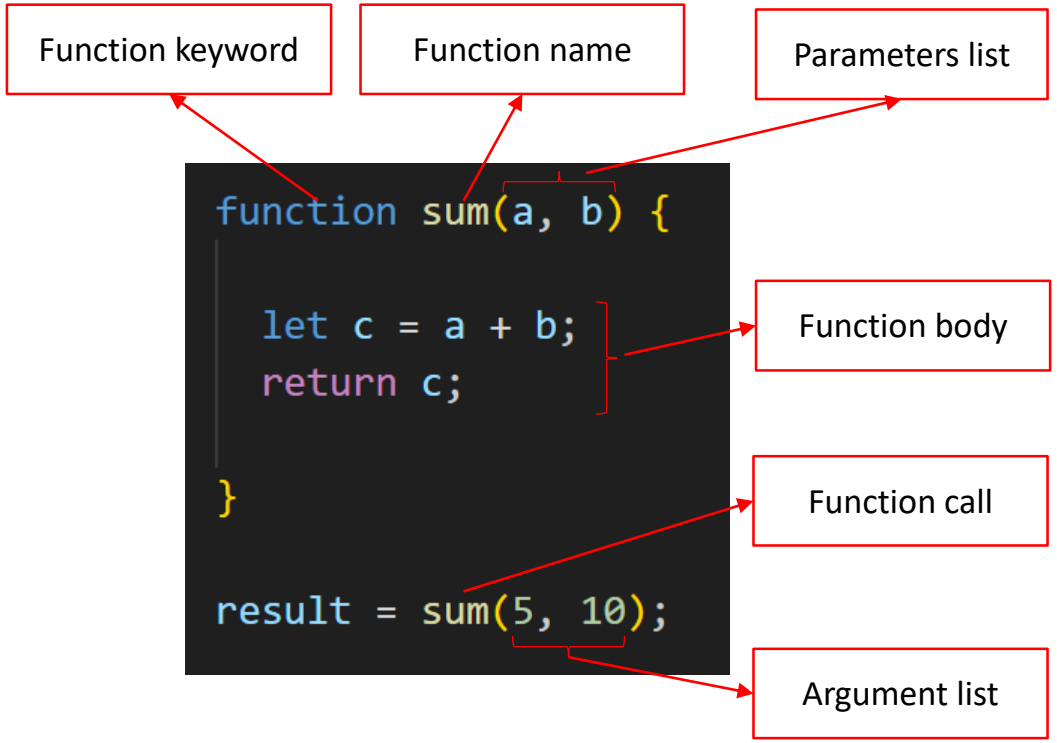
```
// for-of loop
for (let item of array) {
  console.log(item);

  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

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 is function expression in JS?

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

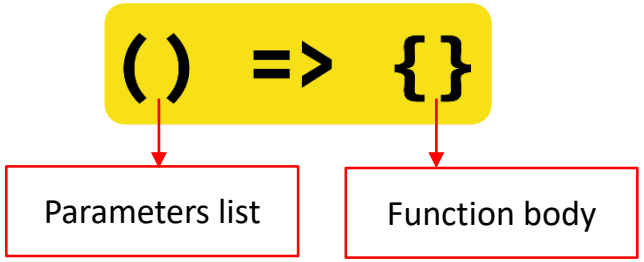
```
//Anonymous Function Expression

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
```

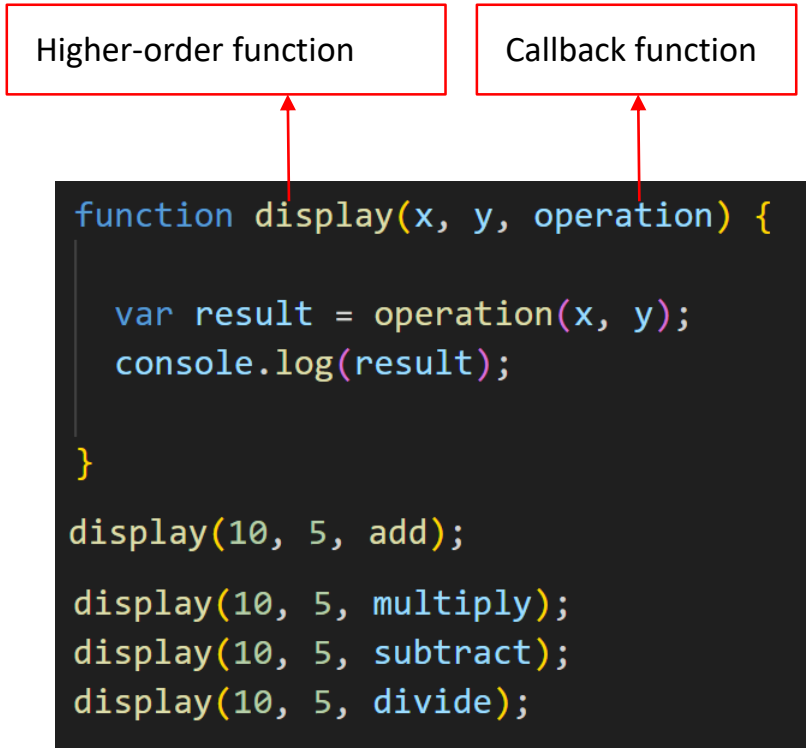
```
//Arrow function

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
```



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

```
//Take one or more functions
//as arguments
function hof(func) {

  func();
}

hof(sayHello);

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

Callback function

High-order function

```
//Return a function as a result
function createAdder(number) {

  return function (value) {
    return value + number;
  };

}

const addFive = createAdder(5);

console.log(addFive(2));

// Output: 7
```

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. 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

1.Click Event: addEventListener('click', handler)

1.Mouseover Event: addEventListener('mouseover', handler)

1.Keydown Event: addEventListener('keydown', handler)

1.Keyup Event: addEventListener('keyup', handler)

1.Submit Event: addEventListener('submit', handler)

1.Focus Event: addEventListener('focus', handler)

1.Blur Event: addEventListener('blur', handler)

1.Change Event: addEventListener('change', handler)

1.Load Event: addEventListener('load', handler)

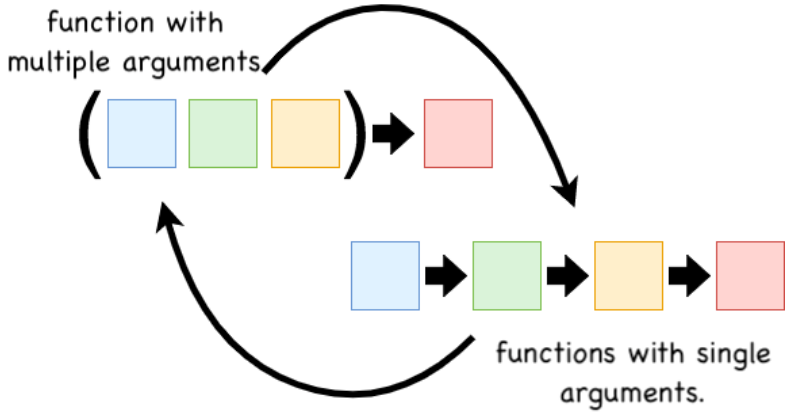
1.Resize Event: addEventListener('resize', handler)

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

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.


```
// 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?

```
//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
```

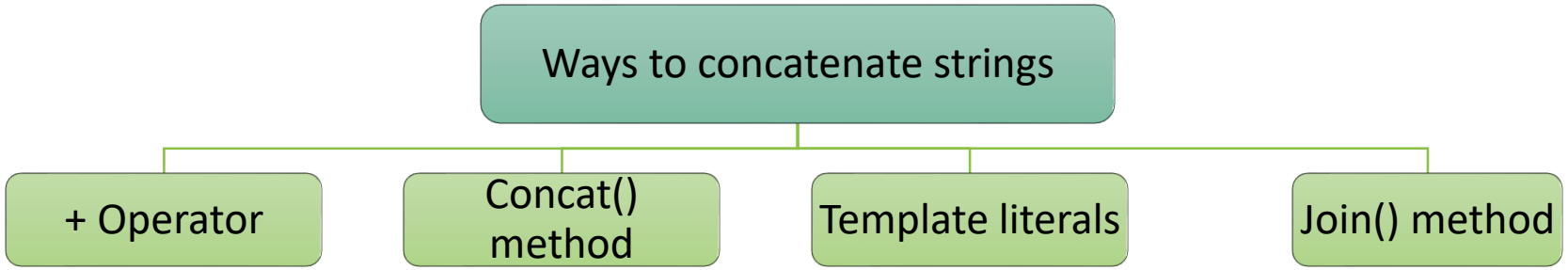
```
//Split a string into an array of substrings
//based on a delimiter
let arr = result.split(" ");
console.log(arr);
// Output: ["Hello", "World"]
```

```
//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. 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
```

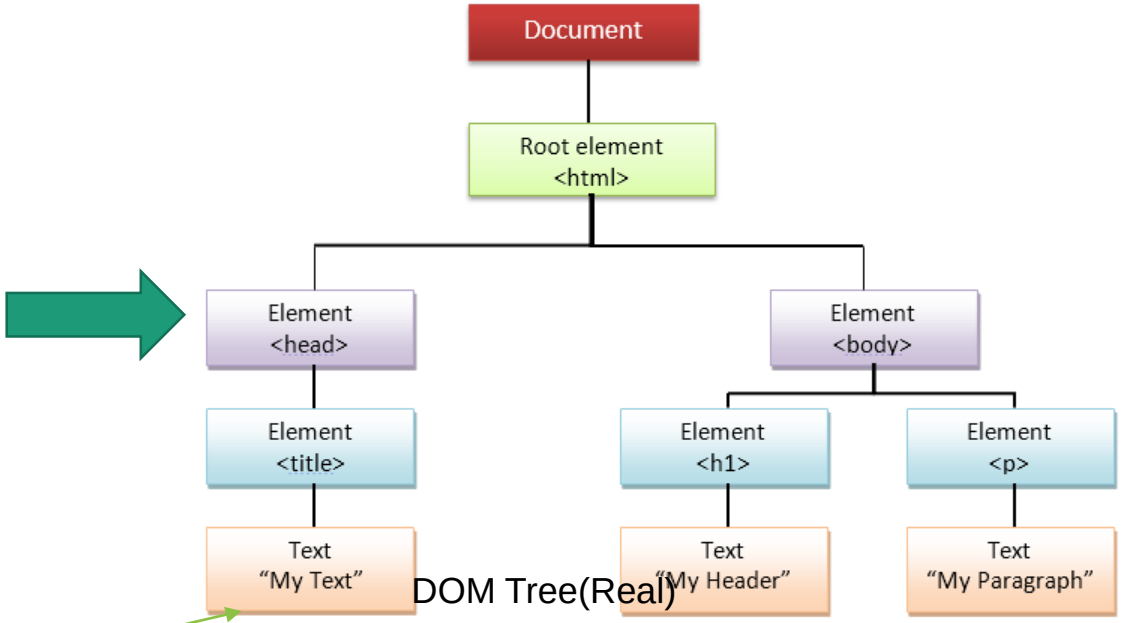
```
// join() method
let strings = [s1, s2];
let r4 = strings.join(' ');
console.log(r4);
// Output: Hello World
```

back to chapter index

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



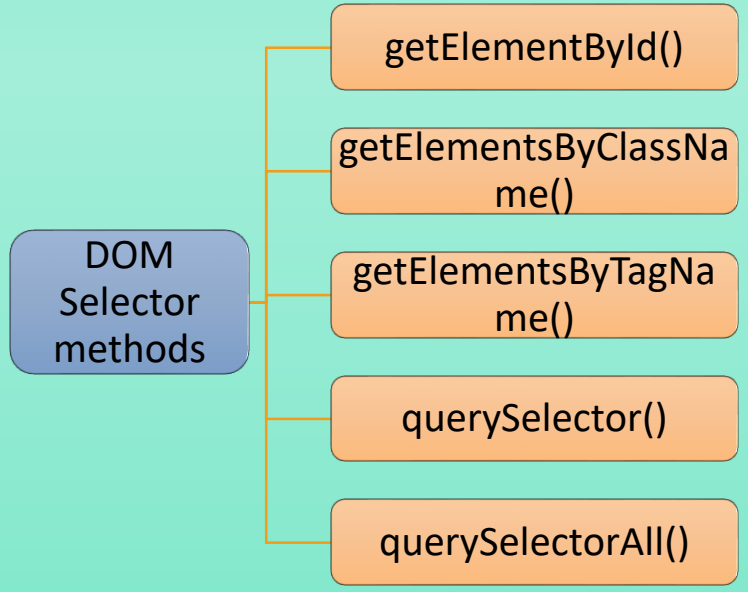
DOM Tree(Real)

- 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.




```html
<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.

```html
<!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>
```

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

```js
//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
```

```js
//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()?

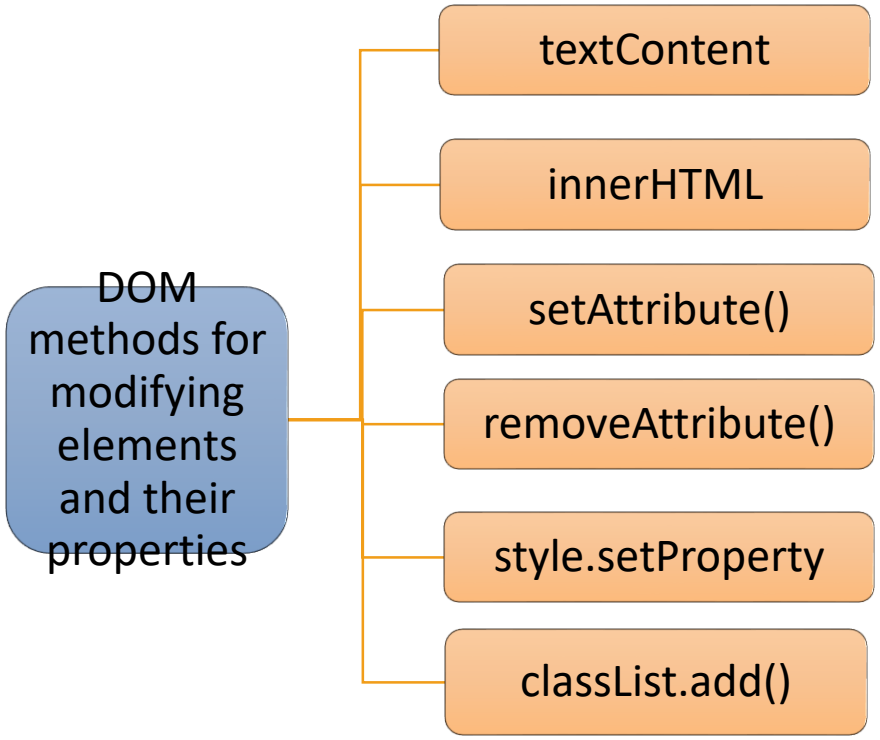
```html
<html>
  <head>
    <title></title>
  </head>
  <body>
    <div class="myClass">Element 1</div>
    <div class="myClass">Element 2</div>
    <div class="myClass">Element 3</div>
    <script src="index.js"></script>
  </body>
</html>
```

```js
// Using querySelector() - returns the first element
var element = document.querySelector('.myClass');
console.log(element.textContent);
// Output: Element 1
```

```js
// 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

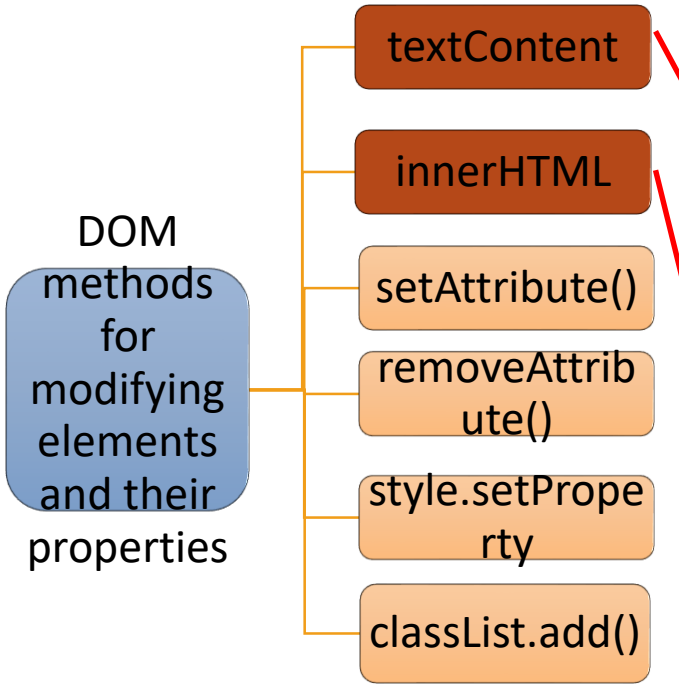
Q. What are the methods to modify elements properties and attributes?




```html
<!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

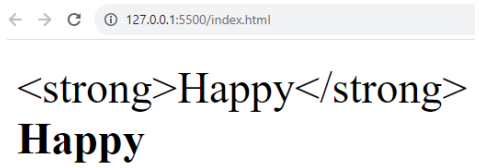
Q. What is the difference between innerHTML and textContent? V. IMP.




```html
<div id="myElement1">Hello</div>
<div id="myElement2">World</div>
```

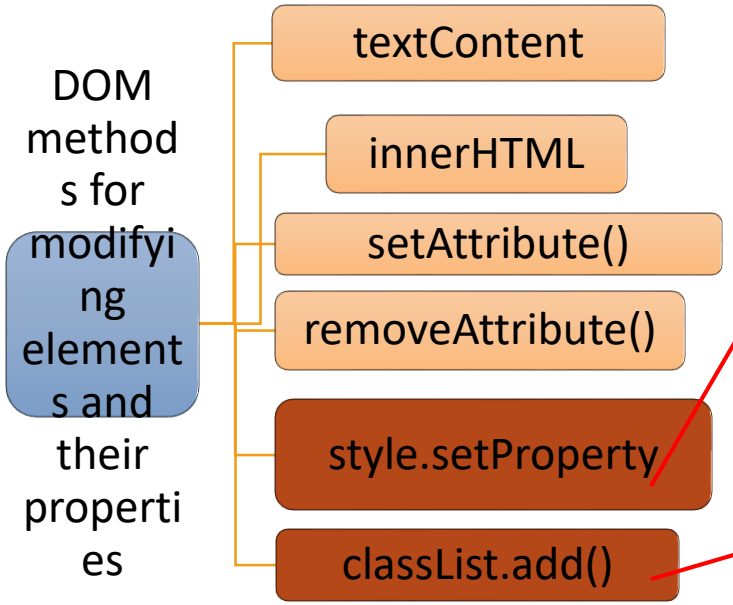
```js
// textContent property used to assign plain text to element
var element1 = document.getElementById("myElement1");
element1.textContent = "<strong>Happy</strong>";
```

```js
// 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 style from HTML elements in DOM using JS?




```
<div id="myElement">Hello, World!</div>
```

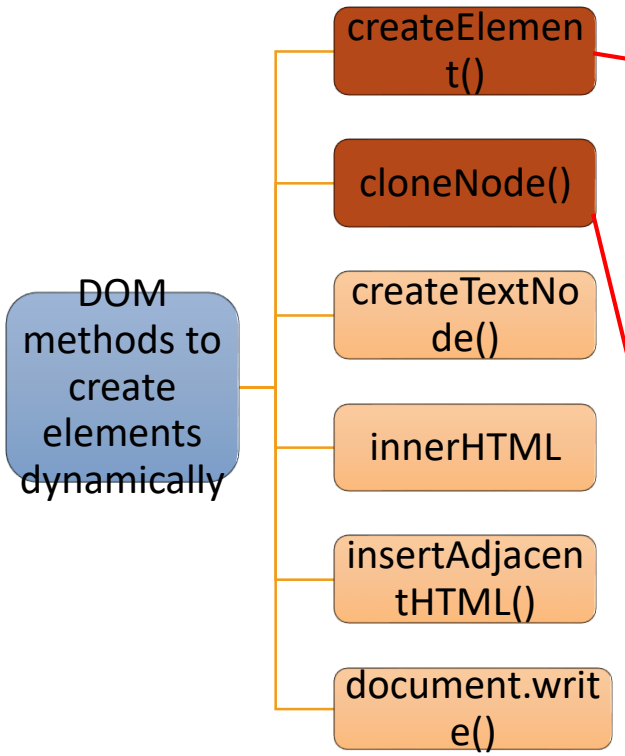
```
var element = document.getElementById("myElement");

// 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()?




```js
// createElement() - create new element
var newDiv = document.createElement('div');
newDiv.textContent = 'Newly created div';
document.body.appendChild(newDiv);
```

```html
<div>Newly created div</div>
```

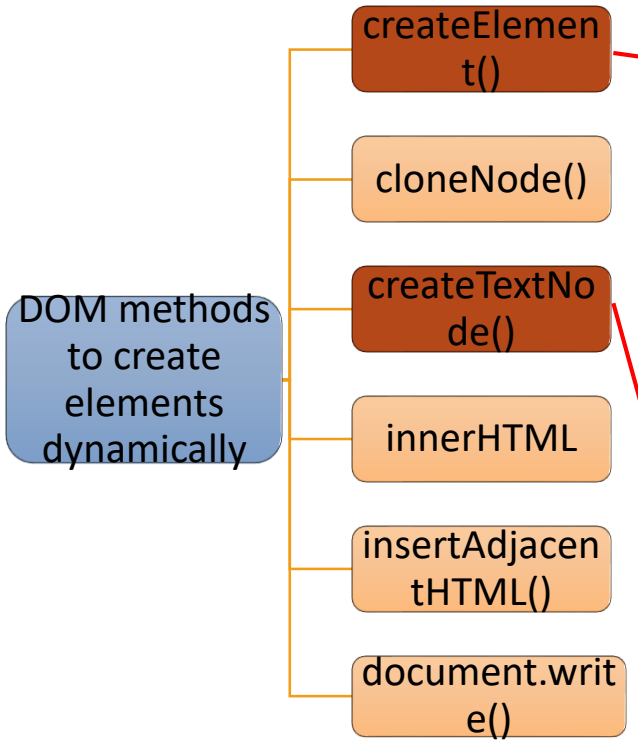
```html
<div id="parentElement">
  <h1>Existing Element</h1>
</div>
```

```js
// 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);
```

```html
<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>
```

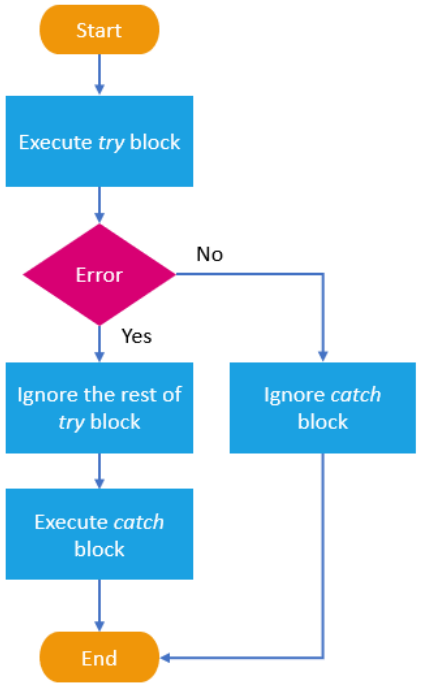
```
var parentElement = document.getElementById("parentElement");
// Create a new text node
var newText = document.createTextNode(" This is some text.");
// Append the text node to the parent
parentElement.appendChild(newText);
```

```
<div id="parentElement">
  <h1>Existing Element</h1>
  " This is some text."
</div>
```

back to chapter index

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

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

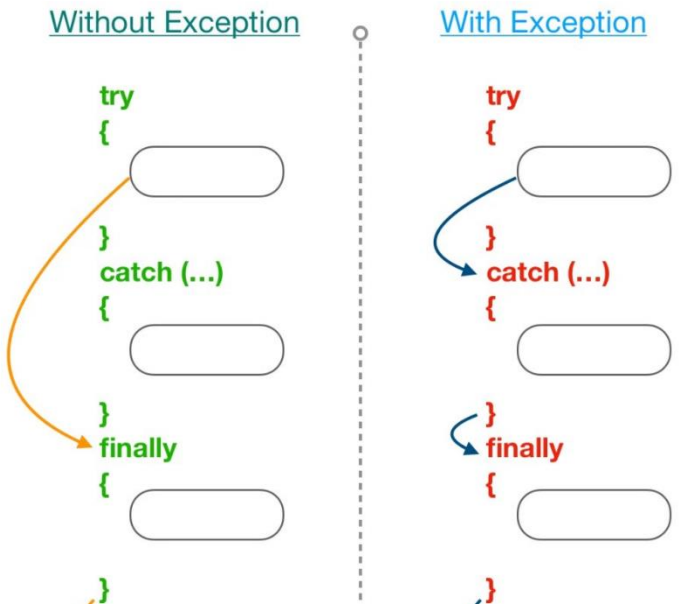



```
//try block contains the code that might throw an error
try {

  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**.

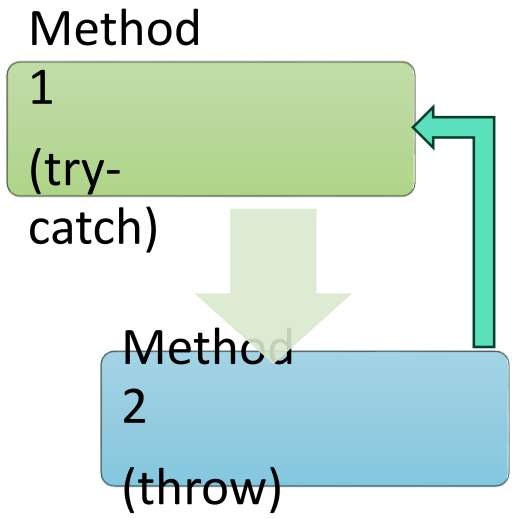



```
//try block contains the code that might throw an error
try {

  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**.

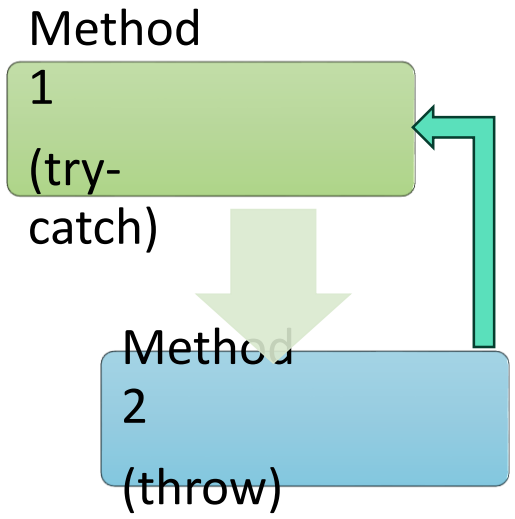



```
function UserData() {
  try {
    validateUserAge(25);
    validateUserAge("invalid"); // This will throw
    validateUserAge(15); // This will not execute
  } catch (error) {
    console.error("Error:", error.message);
  }
}
function validateUserAge(age) {
  if (typeof age !== "number") {
    throw 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
}
```

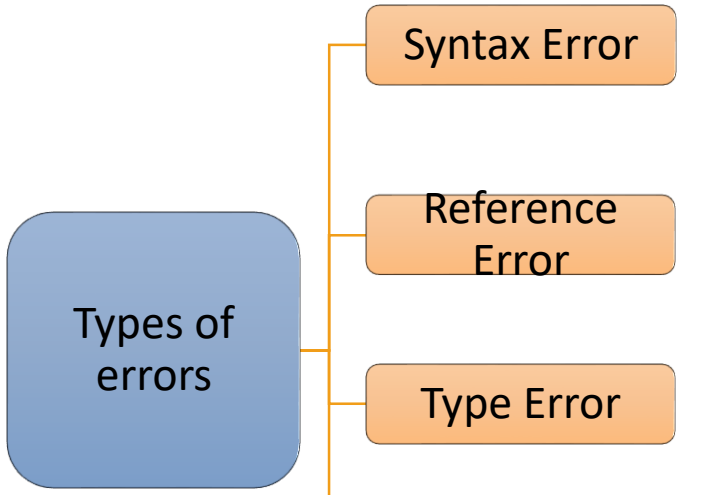
```
// 2. Use Descriptive Error Messages
throw new Error("Cannot divide by zero");
```

```
// 3. Avoid Swallowing Errors
try {
  // Code that may throw an error
} catch (error) {
  // Do not leave the catch blank
}
```

```
// 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
```

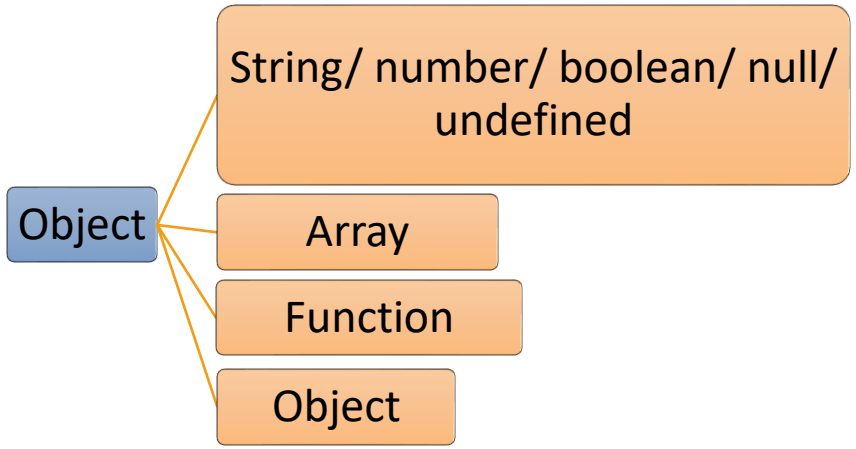
```
// Type Error
const number = 42;
console.log(number.toUpperCase());
// number.toUpperCase is not a function
```

```
// Range Error
const array = [1, 2, 3];
console.log(array[10]);
// Index 10 is out of bounds
```

back to chapter index

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. 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'
```

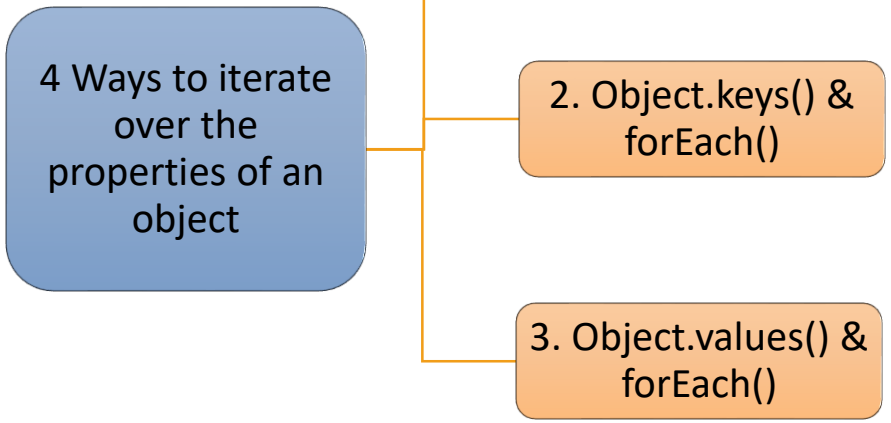
```
// Bracket notation:
console.log(person['name']);
// Output: 'Happy'
```

- 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. 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

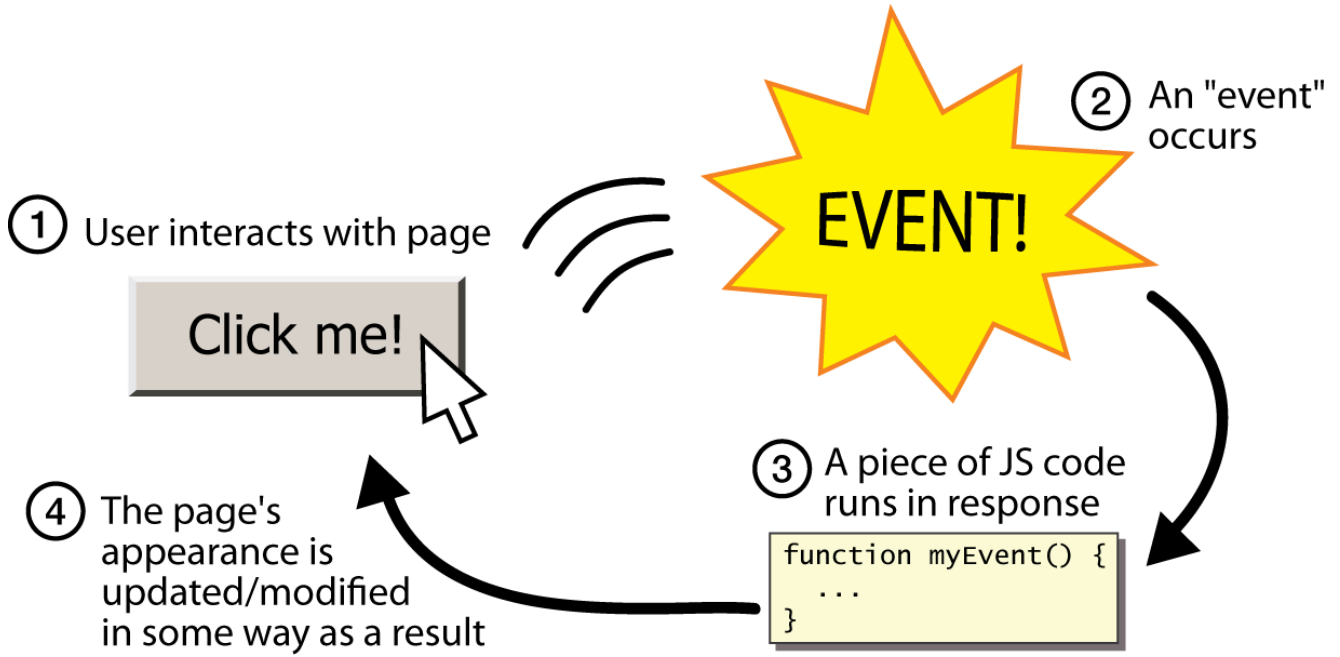
Q. What is Map Object in JS?

- 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 are Events? How are events triggered? V. IMP.



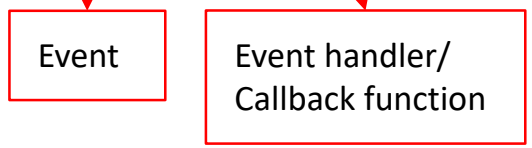
back to chapter index

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.

```html
<button id="myButton">Click Me</button>
```

```javascript
// 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");
}
```



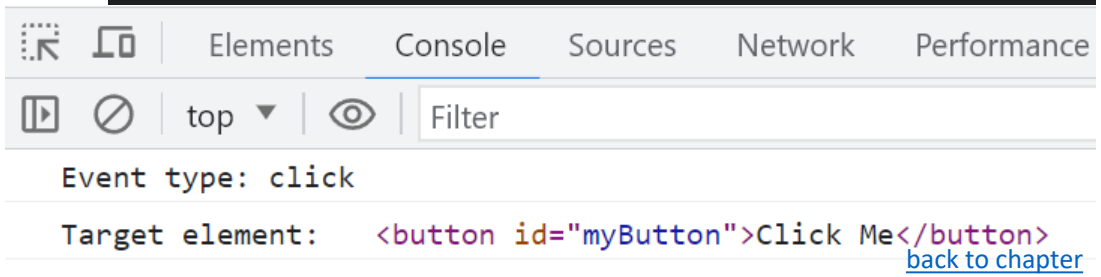
back to chapter index

Q. 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.

```html
<button id="myButton">Click Me</button>
```

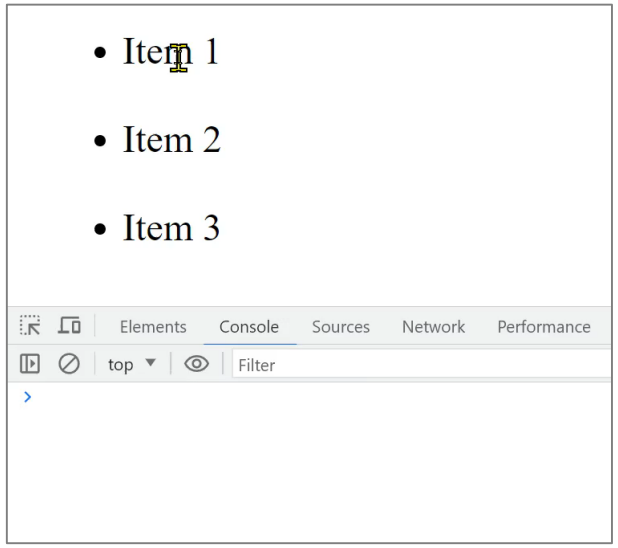
```js
// Get the button element
var button = document.getElementById("myButton");

// 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.

- 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**.



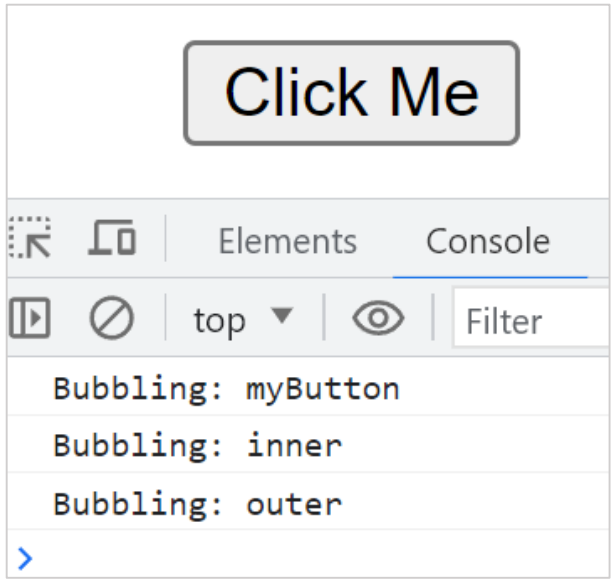

```html
<ul id="myList">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

```javascript
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.

- 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.



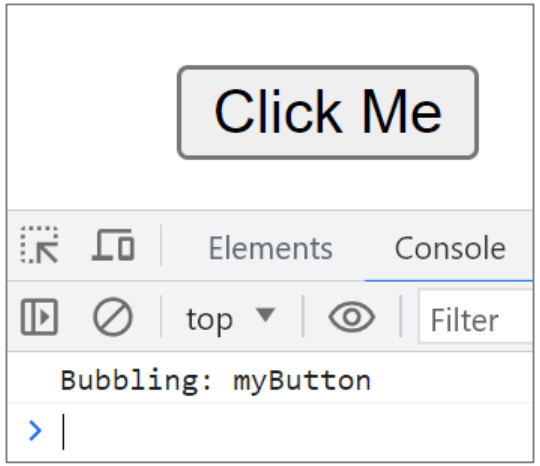

```html
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```

```javascript
// 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.




```html
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```

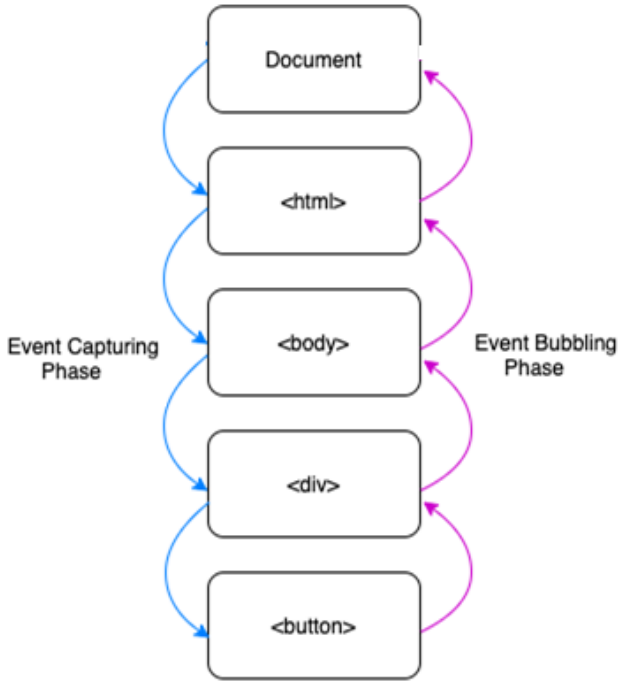
```javascript
// Get the reference of elements
var outer = document.getElementById("outer");
var inner = document.getElementById("inner");
var button = document.getElementById("myButton");

// Attach event handlers with elements
outer.addEventListener("click", handleBubbling);
inner.addEventListener("click", handleBubbling);
button.addEventListener("click", handleBubbling);
```

```javascript
function handleBubbling(event) {
  console.log("Bubbling: " + this.id);
  event.stopPropagation(); // Stop event propagation
}
```

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**.




```html
<div id="outer">
  <div id="inner">
    <button id="myButton">Click Me</button>
  </div>
</div>
```

```javascript
// Get the reference of elements
var outer = document.getElementById("outer");
var inner = document.getElementById("inner");
var button = document.getElementById("myButton");
```

```javascript
// Attach event handlers with elements
outer.addEventListener('click', handleCapture, true);
inner.addEventListener('click', handleCapture, true);
button.addEventListener('click', handleCapture, true);
```

```javascript
function handleCapture(event) {
  console.log("Capturing: " + this.id);
}
```

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.

```html
<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.

```html
<button id="myButton">Click Me</button>
```

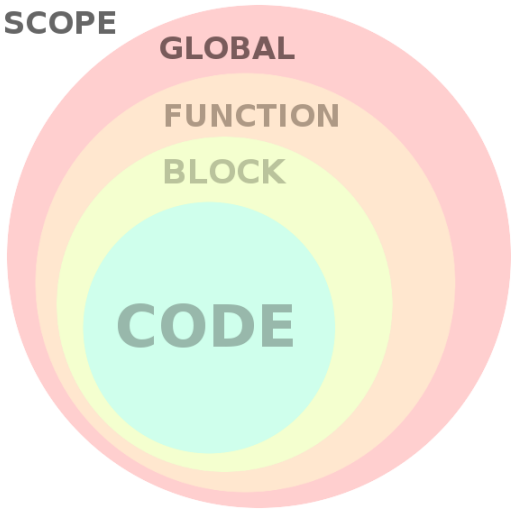
```javascript
var button = document.getElementById("myButton");

// Attach the event handler
button.addEventListener("click", handleClick);

function handleClick() {
  console.log("Button clicked!");
}
```

```javascript
// Remove the event handler
button.removeEventListener("click", handleClick);
```

back to chapter 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**.

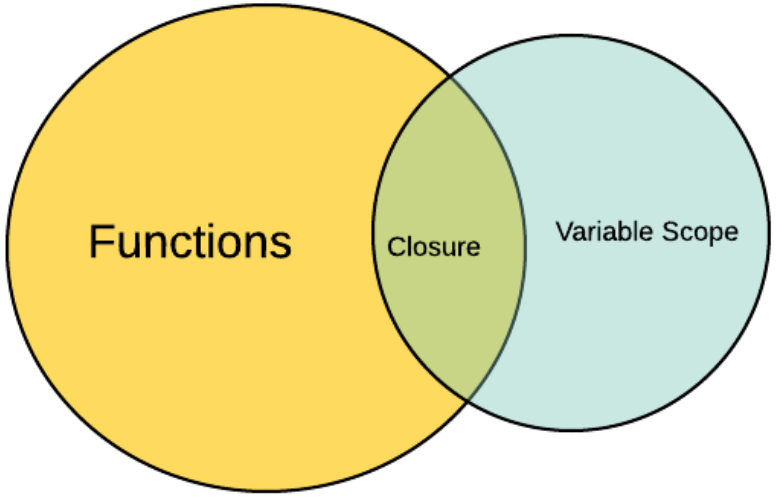



```
function outerFunction() {
  const outerVariable = "outer scope";

  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 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?

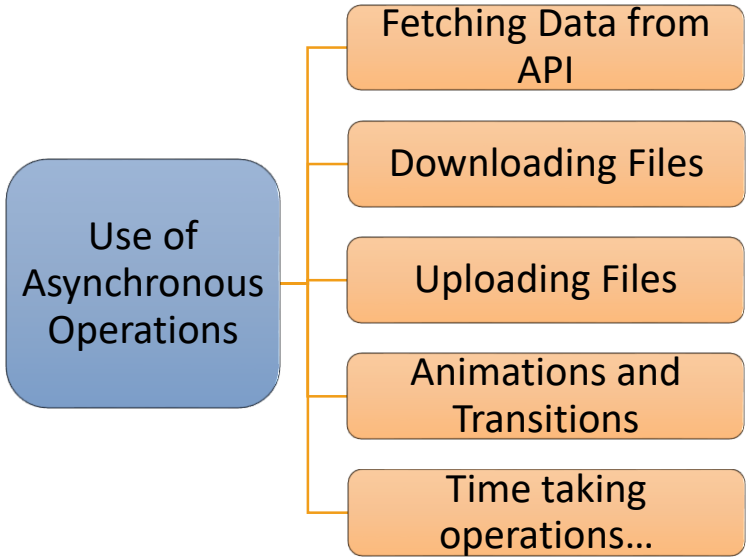
- You can release the reference to the closure by setting **closure to null**.

```
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 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);
}
```

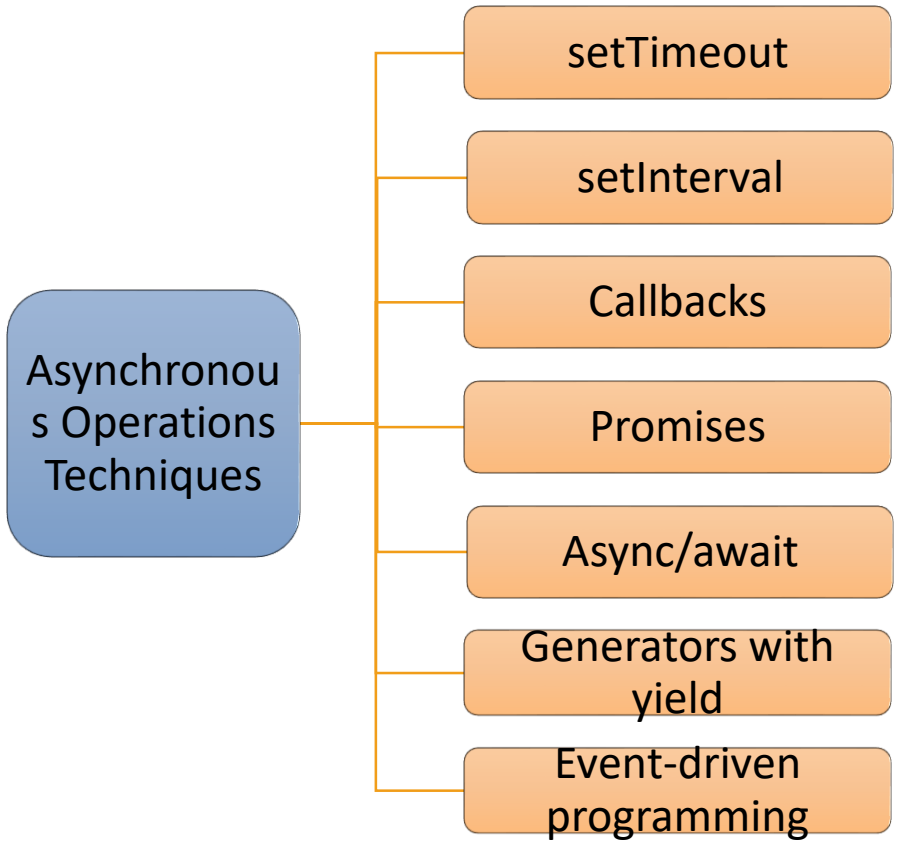
back to chapter index

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");
```

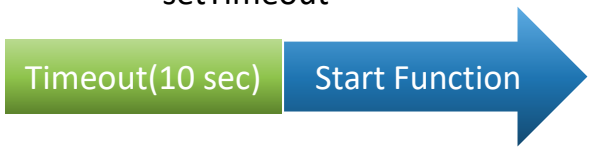
```
// Output:
// start
// not blocked
// I am not stopping anything
```

back to chapter index

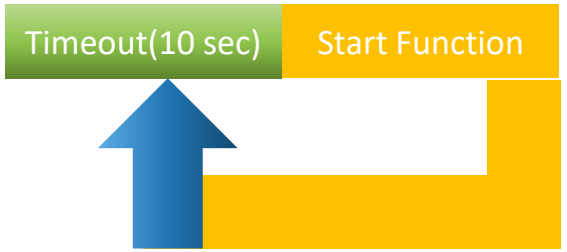
Q. What is setInterval()? How is it used to handle asynchronous operations?

- setInterval() is a built-in JavaScript function that allows you to **repeatedly execute a function at a specified interval asynchronously**.

setTimeout



setInterval




```
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 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);
  });
```

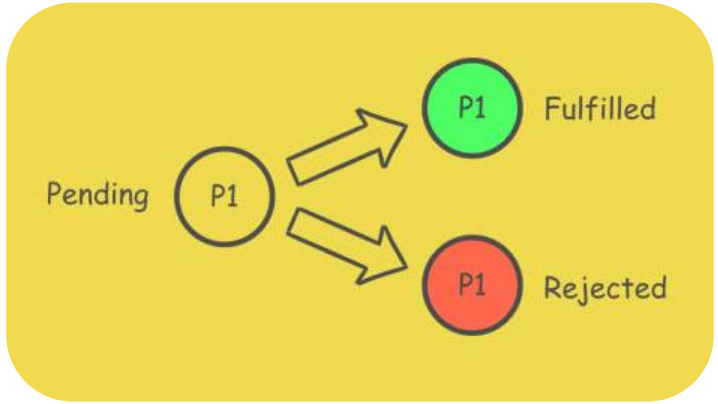
```
// Callback hell problem - multiple nested callbacks
asyncOperation1(function(error, result1) {
    if (error) {
      console.error('Error:', error);
    } else {
      asyncOperation2(function(error, result2) {
        if (error) {
          console.error('Error:', error);
        } else {
          asyncOperation3(function(error, result3) {
            if (error) {
              console.error('Error:', error);
            } else {
              // ... more nested callbacks
            }
          });
        }
      });
    }
  });
```

back to chapter index

Q. What are Promises in JavaScript? V. IMP.

❖ Important points about promises:

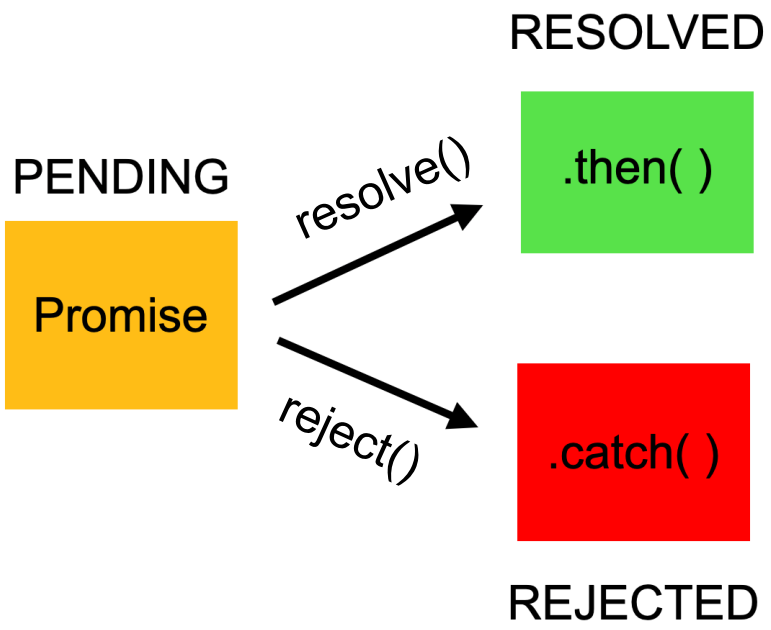
1. Promises in JavaScript are a way to handle **asynchronous operations**.
2. A Promise can be in one of three states: **pending, resolved, or rejected**.
3. A promise represents a value that **may not be available yet** but will be available at some point in the **future**.




```
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

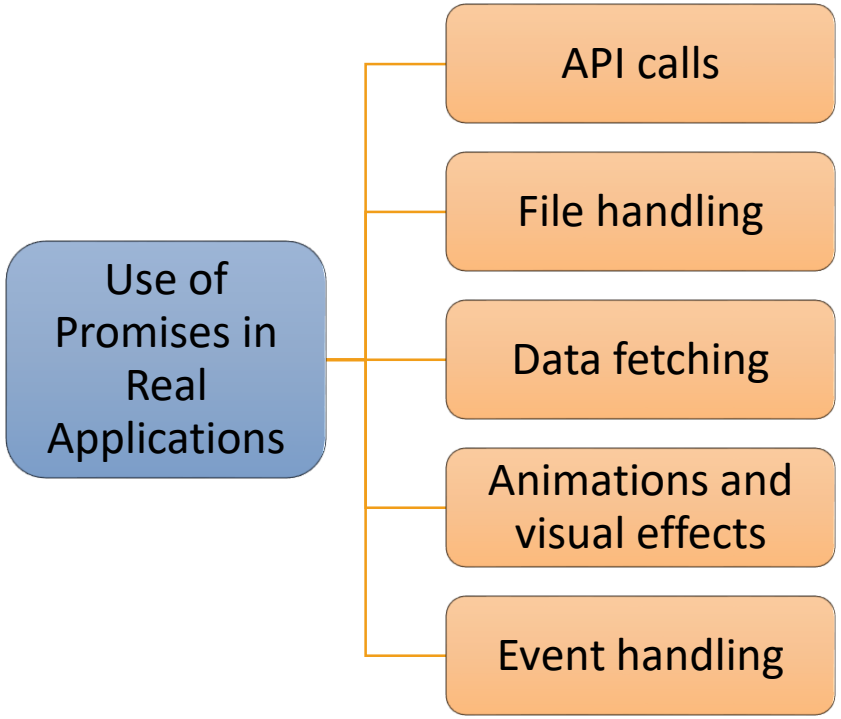
Q. How to implement Promises in JavaScript? V. IMP.



back to chapter index

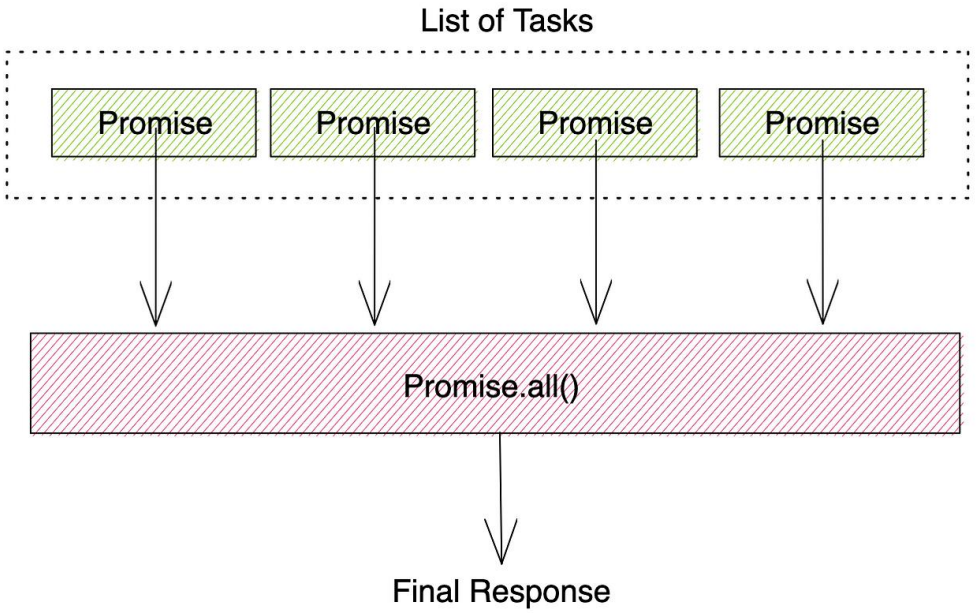
Q. When to use Promises in real applications?

- Promises are useful when you need to perform **time taking operations in asynchronous manner** and later handle the results when the result is available.



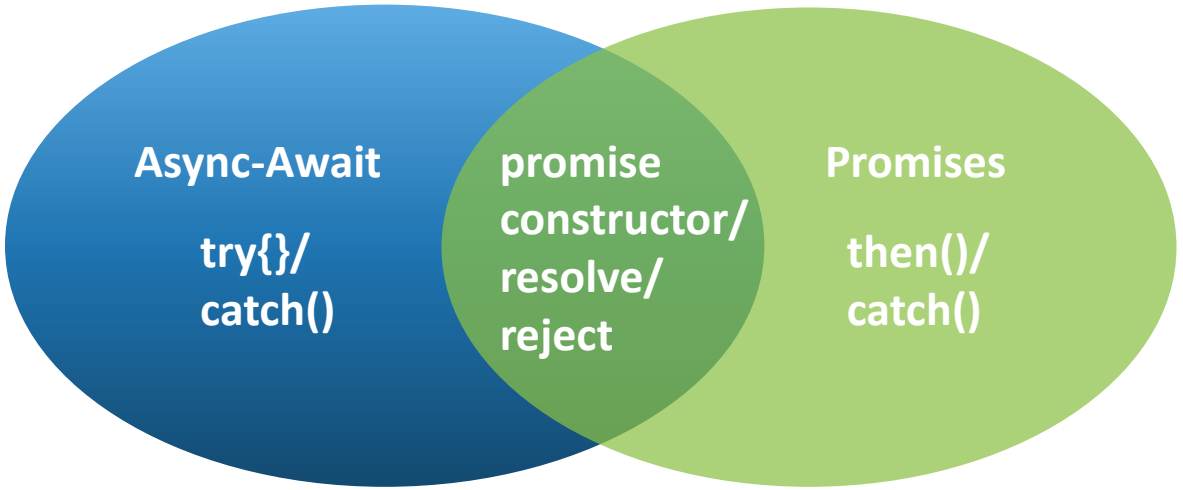
back to chapter index

Q. What is the use of Promise.all() method? V. IMP.



back to chapter index

Q. What is the 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.

```
// Output:
// Starting...
// Not Blocked
// Running (after 1 sec)
// Blocked
// Running (after 2 sec)
```

```
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();
```

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

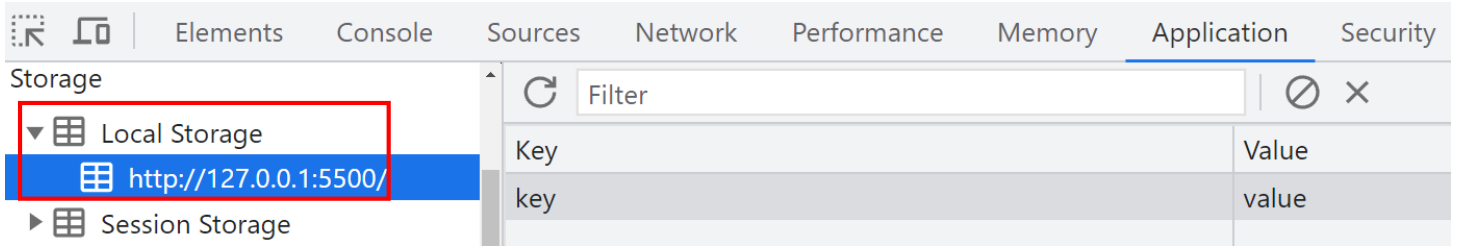
Q. What is Local Storage? How to store, retrieve and remove data from it?

- 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 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?

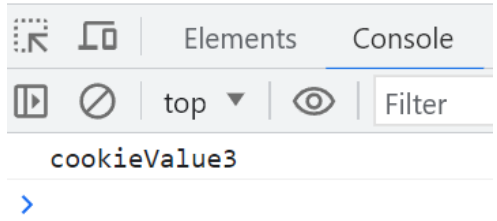
- **5-10MB per origin(approx)**
- It varies with browsers

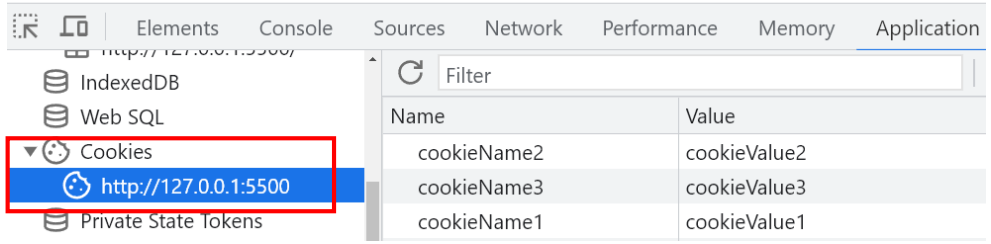
1. Google Chrome: 10MB per origin
2. Mozilla Firefox: 10MB per origin
3. Safari: 5MB per origin
4. Microsoft Edge: 10MB per origin.

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. 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.

```
// 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);
```

```
//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...');
  }
};
```

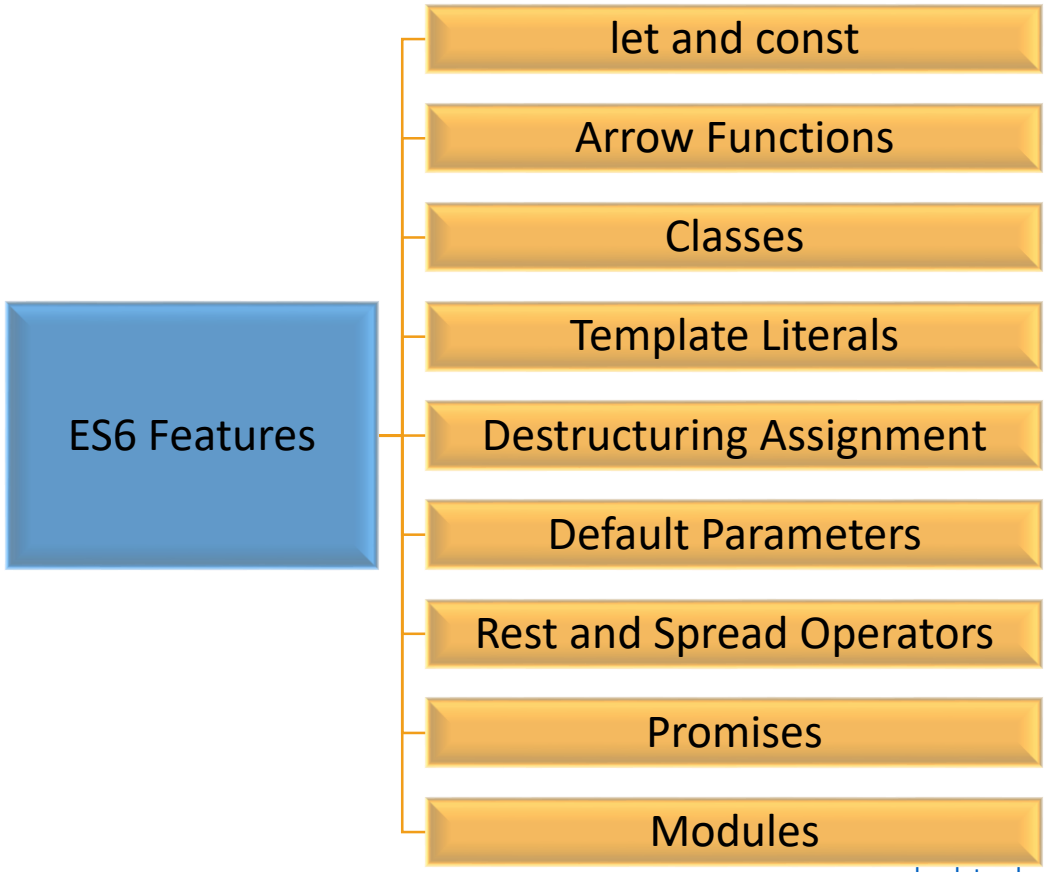
```
// Creating a child object using Object.create()
const bmw = Object.create(vehicle);
```

```
console.log(bmw.type); // Output: Car
bmw.drive(); // Output: Driving...
```

back to chapter index

Q. What is ES6? What are the new features introduced by it? V. IMP.

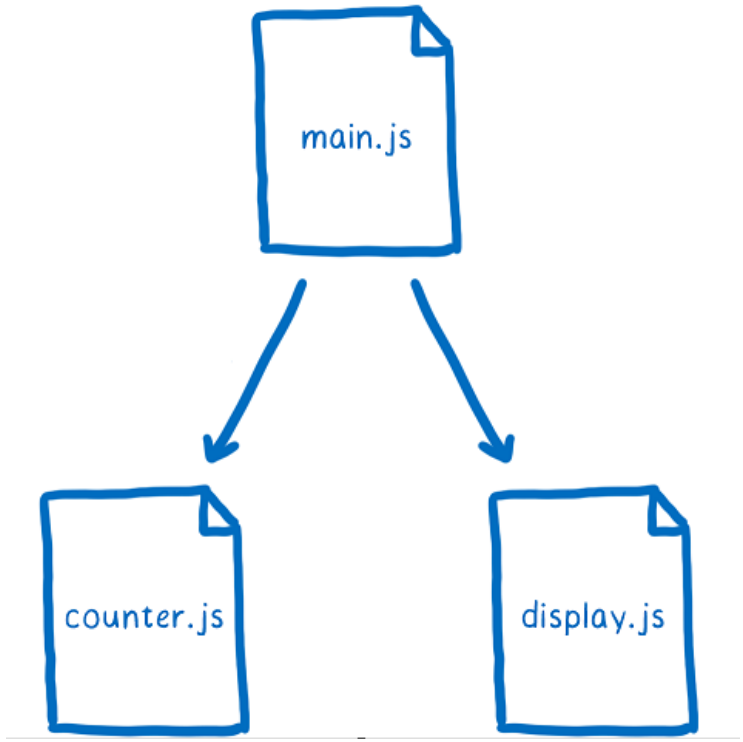
- ECMAScript(ES6) is the **standard** which JavaScript follows.



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

```js
import { add } from "./add.js";
import { subtract } from "./subtract.js";
import { multiply } from "./multiply.js";

console.log(add(2, 3)); // Output: 5
console.log(subtract(5, 2)); // Output: 3
console.log(multiply(4, 3)); // Output: 12
```

add.js

```js
export function add(a, b) {
  return a + b;
}
```

subtract.js

```js
export function subtract(a, b) {
  return a - b;
}
```

multiply.js

```js
export function multiply(a, b) {
  return a * b;
}
```

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 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. 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 validate user input as they type in a form? V. IMP.

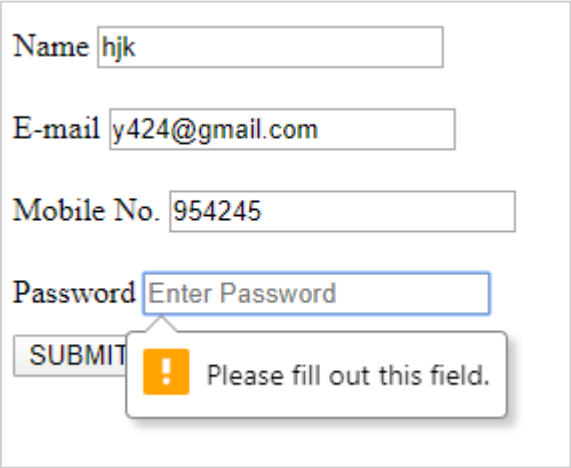
- By using event handling or event listeners on **input event**.

```
// 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 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

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 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 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

All the best for your interviews

Never ever give up

Jagmohan Rai