Java Script Essentials

- JS Variable
- Operators
- Decision Making (If, if-else and if-else-if)
- Loop Controls (for, while and do-while)
- Function & Arrow function
- Arrays & String
- Array methods- Map, Filter, Reduce, Sort
- Class-Objects
- Events & Listeners
- Intervals
- Callbacks, Callbacks hell
- Promises, Promise API
- Async-Await
- Try-Catch

Function

- Function is a block of code designed to perform a particular task.

```
// Function to compute the product of p1 and p2
function myFunction(p1, p2) {
  return p1 * p2;
}
```

Arrow Function

- ES6, introduced Arrow function.
- Arrow function is used to write shorter function syntax:

Before Arrow:

```
function square(param) {
        return param * 2;
}

square(4);
```

After Arrow:

```
var square = (param) => {
        return param * 2;
}

square(4);
```

Note: Arrow function is anonymous.

Arrow Function

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

Arrow Functions Return Value by Default:

```
hello = () => "Hello World!";
```

Arrow Function With Parameter:

```
hello = (val) => "Hello " + val;
```

Arrow Function Without Parentheses:

```
hello = val => "Hello " + val;
```

Array

Without Array

var m1=89;
var m2=68;
var m3=98;
var m4=78;
var m5=45;

With Array
var m=[89,68,98,78,45];

Array Variable

Array Elements

Accessing Array:
var sum=m[0]+m[1] +m[2] +m[3] +m[4];

Accessing Array using loop:

var sum=0;
for(var x=0; x<m.length ; x++){
  sum += m[x];
}

Note:

Array indexing starts from 0.

'array.length' property return the number of elements in array.

Array Functions

Properties/Methods	Description
join(“characters”)	Joins all elements into a string with characters in between.
pop()	Removes the last element from an array.
push(“data")	Adds a new element to an array (at the end).
shift()	Removes the first array element and "shifts" all other elements to a lower index. And returns the value that was "shifted out"
unshift()	Adds a new element to an array (at the beginning), and "unshifts" older elements. And returns the new array length.
array1.concat(array2,…)	Creates a new array by merging (concatenating) existing arrays.
sort()	Sorts an array in increasing order.
reverse()	Reverses the elements in an array.

Array Functions

Properties/Methods	Description
forEach()	Executes a function for each array element.
map()	Creates a new array by applying a function to each element.
filter()	Creates a new array with elements that pass a condition.
reduce()	Reduces array to a single value.

String

String is the collection of characters in double or single quotes.

```
var name="Rahul Chauhan";
```

Properties/Methods	Description
length()	Returns the length of a string.
charAt(index)	Returns the character at specified index number.
concat(data)	Returns the String by concatenating the data.
indexOf(data)	Returns the index number of specified data.
lastIndexOf(data)	Returns the index number of specified data from the last.
replace(data1, data2)	Returns the String by replacing data1 to data2.
substr(index, number)	Returns the String of specified number of character from specified index.
substring(start-index, end-index)	Returns the String in between the specified indexes. End-index is excluded.

String

Properties/Methods	Description
toLowerCase()	Returns the String in small letters.
toUpperCase()	Returns the String in capital letters.
trim()	Returns the String by removing spaces from start and end.
split(“data”)	Returns the array of Strings by splitting according to specified data.

Class and Object

- ES6, introduced JavaScript Classes.
- JavaScript Classes are templates for JavaScript Objects.

Use the keyword **class** to create a class.
Always add a method named **constructor()**

Class Declaration:

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
}
```

Object Creation:

```
let objectName = new ClassName () ;
```

Class and Object

Constructor with no parameter:

```
class Person {
  constructor() {
    this.name = “Ram”;
    this.age = 20;
  }
  show() {
    return (this.name + this.age);
  }
}

let person1 = new  Person();
```

Class and Object

Constructor with parameter:

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  show() {
    return (this.name + this.age);
  }
}

let person1 = new  Person("Rohan", 23);
let person2 = new  Person("Himesh", 21);
```

Class and Object

Constructor overloading is NOT allowed.

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  constructor() {
    this.name = “Ram”;
    this.age = 20;
  }
  show() {
    return (this.name + this.age);
  }
}
```

Note: Only one constructor is allowed at a time.

Event and Listener

Events:

An **event** is a signal that something has happened. Examples:

- Clicking a button
- Pressing a key
- Moving the mouse
- Page loading

Event Listeners :

An **event listener** is a function that waits for an event to happen on a particular element.

Syntax

```
element.addEventListener(event, function, useCapture);
```

Event and Listener

Html file

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

<script>
        document.getElementById("myBtn").addEventListener("click",
        function () {
        alert("Button Clicked!");
        });
</script>
```

Types of Common Events:

Event Type	Description
click	When the element is clicked
mouseover	When mouse moves over element
keydown	When a key is pressed
submit	When a form is submitted
load	When a page finishes loading

Event Object:

```
document.addEventListener("click", function (event) {
        console.log(event.target); // Element that was clicked
});
```

setTimeout

setTimeout() **executes a function once after a delay** (in milliseconds).

```
setTimeout(() => {
            console.log("This message appears after 3 seconds");
}, 3000);
```

Example with argument:

```
function greet(name) {
      console.log("Hello, " + name);
}
setTimeout(greet, 2000, "Rahul");
```

Cancel the timeout: clearTimeout()

```
let timeoutID = setTimeout(() => {
      console.log("This will not run");
}, 3000);
clearTimeout(timeoutID); // cancels the timeout
```

setInterval

setInterval() **repeatedly executes a function** at every interval (in milliseconds) until it's stopped.

```
setInterval(() => {
        console.log("Repeating every 2 seconds");
}, 2000);
```

Stop the Repetition: clearInterval()

```
let intervalID = setInterval(() => {
        console.log("Running...");
}, 1000);
setTimeout(() => {
        clearInterval(intervalID);
        console.log("Stopped!");
}, 5000);
```

setTimeout & setInterval

Real-World Use Cases

Use Case	Method
Show splash screen	setTimeout()
Auto-slide banners	setInterval()
Delayed popup	setTimeout()
Live clock	setInterval()
Countdown timer	Both combined

Note:

- setTimeout() and setInterval() are **asynchronous**, meaning they don't block other code.
- The delay is **not guaranteed to be exact**, especially when the browser is busy.

Callbacks

A **callback** is a function passed as an argument to another function, which is then **called later**.

```
function greet(name, callback) {
        console.log("Hi " + name);
        callback();
}
function sayBye() {
        console.log("Bye!");
}

greet("Rahul", sayBye);
```

Callback Hell (a.k.a. Pyramid of Doom)

Problem:
When callbacks are nested within callbacks, leading to unreadable and hard-to-maintain code.

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

You can fix callback hell with:

- **Named functions**
- **Promises**
- **Async/Await** (to be taught after this)

try-catch

The try-catch block in JavaScript is used to **handle runtime errors** gracefully without stopping the program.
It lets you:

- “Try” a block of code that might fail
- “Catch” and handle errors if they occur
- (Optionally) run cleanup code in finally

Real-World Example:
Imagine you're withdrawing cash from an ATM:

- try: Insert card and withdraw
- catch: If ATM has no cash, show error
- finally: Remove card (always happens)

try-catch

Syntax:

```
try {
// Code that may throw an error
} catch (error) {
// Code to handle the error
} finally {
// Optional - always runs
}
```

try-catch

```
try {
        let result = 10 / 2;
        console.log("Result:", result);
} catch (err) {
        console.log("Error:", err.message);
} finally {
        console.log("This block always executes, regardless of errors.");
}
// No error, so catch is skipped.

try {
        let result = 10 / 0; // This will cause an error
        console.log("Result:", result);
} catch (err) {
        console.log("Error:", err.message);
} finally {
        console.log("This block always executes, regardless of errors.");
}
```

finally Block

Always executes, whether or not there was an error.

```
try {
        console.log("Trying...");
        throw new Error("Oops!");
} catch (e) {
        console.log("Caught:", e.message);
} finally {
        console.log("Cleanup done!");
}
```

Catching Specific Error Properties

You can access:

- error.message: only the message
- error.name: error type
- error.stack: full error trace

```
try {
        let num = x * 2;
} catch (err) {
        console.log("Name:", err.name);
        console.log("Message:", err.message);
        console.log("Stack:", err.stack);
}
// This will catch the error and log the name, message, and stack trace of the error.
```

Synchronous & Asynchronous

Synchronous

Synchronous code is executed **line-by-line**, one after the other.
Each task must **finish before the next one starts**.

Real World Example:

You stand in a queue at a bank. You **wait** for the person ahead to finish before it's your turn.

```
console.log("Start");
console.log("Step 1");
console.log("End");
```

Output:

```
Start
Step 1
End
```

Synchronous & Asynchronous

Asynchronous

Asynchronous code allows **non-blocking** execution.
It runs **in the background**, and the rest of the code **continues executing**.
Real World Example:
You place a food order, and **don't wait** there. You go back to work, and the food comes **later**.

```
console.log("Start");
setTimeout(() => {
      console.log("Step 1 (after 2s)");
}, 2000);
console.log("End");
```

Output:

```
Start
End
Step 1 (after 2s)
```

- Even though setTimeout() is written in the middle, the last line executes **first** because setTimeout() is **asynchronous**.

Synchronous & Asynchronous

Asynchronous Examples

Function
setTimeout()
setInterval()
Promises
async/await
fetch() (API calls)

Promise

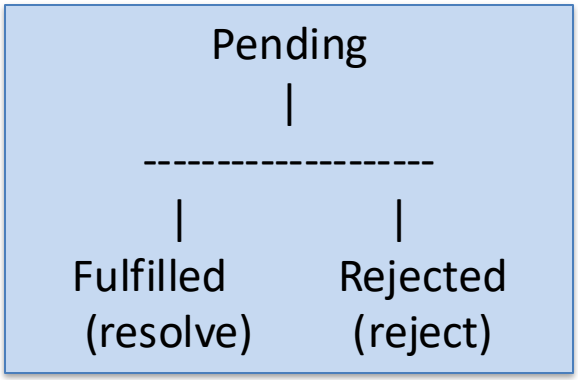
A **Promise** in JavaScript is a **placeholder for a value that will be available in the future** — usually after an asynchronous task (like API call, file load, etc.).

Real-life Example: Ordering food from Swiggy or Zomato:

- Promise: "We will deliver your food."
- Fulfilled: Food delivered ✅
- Rejected: Delivery failed ❌
- Pending: Still being prepared ⏳

States of a Promise:

State	Meaning
Pending	Still working
Fulfilled	Success – got result
Rejected	Failed – got error



Promise Syntax

```
let promise = new Promise((resolve, reject) => {
  // async code
  if (/* success */) {
    resolve(result);
  } else {
    reject(error);
  }
});
```

Note:

- resolve, reject are handers.
- resolve, reject are the callbacks provided by JS

Promise Example

```
let orderPizza = new Promise((resolve, reject) => {
let pizzaAvailable = true;
setTimeout(() => {
        if (pizzaAvailable) {
                resolve("Pizza Delivered!");
        } else {
                reject("Out of Stock");
        }
        }, 2000);
});
orderPizza
                .then((msg) => console.log("Success:", msg))
                .catch((err) => console.log("Error:", err));
```

- This code creates a promise that simulates a pizza delivery process.
- If the pizza is available, it resolves with a success message after 2 seconds.
- If it is out of stock, it rejects with an error message.
- The ".then()" method handles the success case, while ".catch()" handles any errors.

Promise Example

```
function checkNumber(num) {
	return new Promise((resolve, reject) => {
		if (num > 0) {
			resolve("Number is positive");
		} else {
			reject("Number is not positive");
		}
	});
}

//Caller
checkNumber(-9)
	.then((message) => {
		console.log("Success:", message);
	})
	.catch((error) => {
		console.log("Error:", error);
	});
```