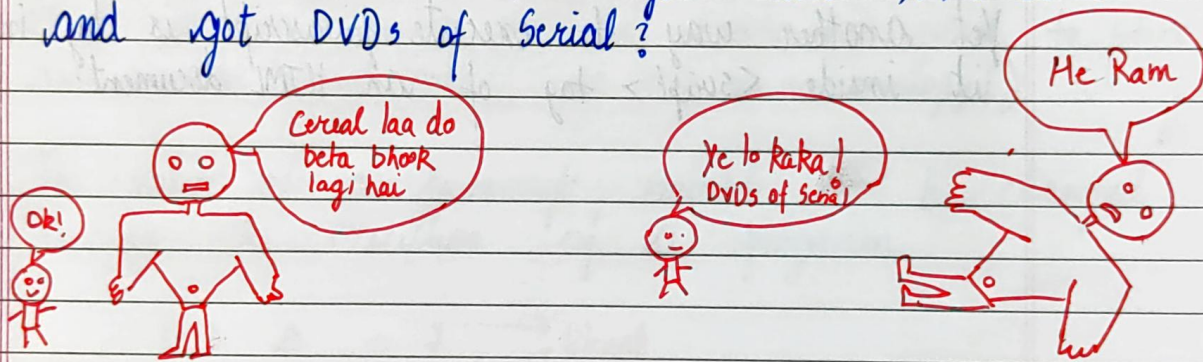


## Introduction to programming

Programming is a way to talk to computers.  
A language like Hindi, English or Bengali can be used to talk to a human but for computers we need straightforward instructions.

Computer is Dumb!

When was the last time you ordered some cereal and got DVDs of Serial?



Programming is the act of constructing a program, a set of precise instructions telling a computer what to do.

What is EcmaScript?

ECMAScript is a standard on which JavaScript is based! It was created to ensure that different documents on JavaScript are actually talking about the same language.

JavaScript & ECMAScript can <sup>almost</sup> always be used interchangeably. JavaScript is very liberal in what it allows.

How to execute JavaScript?

JavaScript can be executed right inside one's browser. You can open the javascript console and start writing javascript there.

Another way to execute javascript is a runtime like Node.js which can be installed and used to run javascript code.

Yet another way to execute javascript is by inserting it inside `<script>` tag of an HTML document.



## Chapter 1 - Variables & Data

Just like we follow some rules while speaking english (the grammar), we have some rules to follow while writing a javascript program. The set of these rules is called syntax in javascript.

### What is a Variable?

A variable is a container that stores a value. This is very similar to the containers used to store rice, water and oats (Treat this as an analogy!)

The value of a javascript variable can be changed during the execution of a program.

```
var a = 7;
let a = 7;
```

→ literal  
⇒ Declaring Variables  
↓ Identifier     ↓ assignment operator

### Rules for choosing variable names

- Letters, digits, underscores & \$ sign allowed.
- Must begin with a \$, \_ or a letter.
- JavaScript reserved words cannot be used as a variable name
- Harry & hARRY are different variable (case sensitive)

### Var vs let in JavaScript

- 1> var is globally scoped while let & const are block scoped
- 2> var can be updated & re-declared within its scope
- 3> let can be updated but not re-declared
- 4> const can neither be updated nor be re-declared.



- 5> var variables are initialized with undefined whereas let and const variables are not initialized.
- 6> const must be initialized during declaration unlike let and var

### Primitive Data Types & Objects

Primitive data types are a set of basic data types in javascript

Object is a non primitive datatype in javascript

These are the 7 primitive datatypes in javascript

- Null
- Undefined
- Number
- Boolean
- String
- BigInt
- Symbol

### Object

An object in JavaScript can be created as follows.

```
const item = {
```

```
  key ← name: "Led Bulb", → value
      price: "150" → value
}
```

Quick Quiz: Write a JavaScript program to store name, phone number and marks of a student using objects.



## Chapter 1 - Practice Set

- 1 Create a variable of type string and try to add a number to it.
- 2 Use typeof operator to find the datatype of the string in last question.
- 3 Create a const object in javascript. Can you change it to hold a number later?
- 4 Try to add a new key to the const object in Problem 3. Were you able to do it?
- 5 Write a Js program to create a word-meaning dictionary of 5 words.

## Chapter 2 - Expressions & Conditionals

A fragment of code that produces a value is called an expression. Every value written literally is an expression. For ex: 77 or "Harry"

### Operators in JavaScript

#### 1> Arithmetic Operators

+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation
/	Division
%	Modulus
++	Increment
--	Decrement

#### 2> Assignment Operators

=	$x = y$
+=	$x = x + y$
-=	$x = x - y$
*=	$x = x * y$
/=	$x = x / y$
%=	$x = x \% y$
**=	$x = x ** y$



### 3> Comparison Operators

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

### 4> Logical Operators

$\&\&$	logical and
$\ \ $	logical or
$!$	logical not

Apart from these, we also have type and bitwise operators. Bitwise operators perform bit by bit operations on numbers

$$\begin{array}{c} \text{operands} \\ \swarrow \searrow \\ 7 + 8 = 15 \rightarrow \text{Result} \\ \swarrow \searrow \\ \text{operator} \end{array}$$

### Comments in JavaScript

Sometimes we want our programs to contain a text which is not executed by the JS Engine

Such a text is called comment in JavaScript



A comment in javascript can be written as follows :

let a = 2; // this is a single line comment

```

/*
  I am a
  multiline comment
*/

```

→ Single line  
comment

} Multiline comment

Sometimes comments are used to prevent the execution of some lines of code

```

let switch = true;
// switch = false → commented line won't
                    execute

```

### Conditional Statements

Sometimes we might have to execute a block of code based off some condition.

For example a prompt might ask for the age of the user and if its greater than 18, display a special message.

In javascript we have three forms of if ... else statement.

- 1) if statement
- 2) if ... else statement
- 3) if ... else if ... else statement



### If statement

The if statement in JavaScript looks like this:

```
if (condition) {  
  // execute this code  
}
```

The if statement evaluates the condition inside the (). If the condition is evaluated to true, the code inside the body of if is executed else the code is not executed.

### if-else statement

The if statement can have an optional else clause. The syntax looks something like this

```
if (condition) {  
  // block of code if condition true  
}  
else {  
  // block of code if condition false  
}
```

If the condition is true, code inside if is executed else code inside else block is executed

### if-else if statement

Sometimes we might want to keep rechecking a set of conditions one by one until one matches. We use if else if for achieving this.



Syntax of if...else if looks like this

```
if (age > 0) {  
    console.log("A valid age");  
}  
else if (age > 10 && age < 15) {  
    console.log("but you are a kid");  
}  
else if (age > 18) {  
    console.log("not a kid");  
}  
else {  
    console.log("Invalid Age");  
}
```

JavaScript Ternary Operator

Evaluates a condition and executes a block of code based on the condition

Condition ? exp1 : exp2

Example syntax of ternary operator looks like this:

(marks > 10) ? 'yes' : 'No'

↳ if marks are greater than 10, you are passed else not



## Chapter 2 - Practice Set

- 1 Use logical operators to find whether the age of a person lies between 10 and 20?
- 2 Demonstrate the use of switch case statements in JavaScript
- 3 Write a JavaScript program to find whether a number is Divisible by 2 and 3.
- 4 Write a JavaScript program to find whether a number is Divisible by either 2 or 3.
- 6 Print "you can Drive" or "you cannot Drive" based on age being greater than 18 using ternary operator.



## Chapter 3 - Loops & Functions

We use loops to perform repeated actions. For example - If you are assigned a task of printing numbers from 1 to 100, it will be very hectic to do it manually. Loops help us automate such tasks.

### Types of loops in JavaScript

for loop → loop a block of code no of times  
 for in loop → loops through the keys of an object  
 for of loop → loops through the values of an object  
 while loop → loops a block based on a specific condition  
 do-while loop → while loop variant which runs atleast once

### The for loop

The syntax of a for loop looks something like this

```
for ( statement 1; statement 2; statement 3 ) {
  // code to be executed
}
```

- Statement 1 is executed one time
- Statement 2 is the condition base on which the loop runs (loop body is executed)
- Statement 3 is executed everytime the loop body is executed

Quick Quiz : Write a sample for loop of your choice.



The for-in loop  
The syntax of for-in loop looks like this

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

Quick Quiz: Write a sample program demonstrating for-in loop

Note - for-in loops also work with arrays which will be discussed in the later videos.

The for-of loop  
The syntax of for-of loop looks like this

```
for (variable of iterable) {
  // code
}
```

→ For every iteration  
↳ Iterable data structure like Arrays, Strings etc.

Quick Quiz: Write a sample program demonstrating for-of loops

The while loop  
The syntax of while loop looks like this:

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

Note: If the condition never becomes false, the loop will never end and this might crash the runtime.



Quick Quiz: Write a sample program demonstrating while loop.

The do-while loop

The do while loop's syntax looks like this:

```
do {
  // code to be executed => Executed at least once
}
while (condition)
```

Quick Quiz: Write a sample program demonstrating do while loop

Functions in JavaScript

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

Syntax of a function looks something like this:

```
function myFunc () {
  // code
}
```

```
function bindFunc (parameter 1, parameter 2) {
  // code
}
```

↳ Function with parameters

↓ Here the parameters behave as local variables



binodFunc (7, 8)  $\Rightarrow$  Function Invocation

Function invocation is a way to use the code inside the function

A function can also return a value. The value is "returned" back to the caller

const sum = (a, b)  $\Rightarrow$  {

↓  
Another way to create & use the function

↑

let c = a + b;

return c;

}

$\Rightarrow$  Returns the sum

let y = sum(1, 3)

console.log(y)

$\rightarrow$  Prints 4



## Chapter 3 - Practice Set

1 Write a program to print the marks of a student in an object using for loop

obj = { harry : 98, rohan : 70, aakash : 73 }

2 Write the program in Q 1 using for in loop

3 Write a program to print "try again" until the user enters the correct number.

4 Write a function to find mean of 5 numbers.



## Chapter 4 - Strings

Strings are used to store and manipulate text. String can be created using the following syntax:

```
let name = "Harry" → Creates a string
name.length
```

↳ This property prints length of the string

Strings can also be created using single quotes

```
let name = 'Harry'
```

### Template Literals

Template literals use backticks instead of quotes to define a string

```
let name = `Harry`
```

With template literals, it is possible to use both single as well as double quotes inside a string

```
let sentence = `The name "is" Harry's`
```

↳ Single quote

↳ backtick

↳ double quote

We can insert variables directly in template literal. This is called string interpolation

```
let a = `This is ${name}` → Prints 'This is a Harry'`
```

↳ name is a variable



## Escape Sequence Characters

If you try to print the following string, JavaScript will misunderstand it

```
let name = 'Adam D'Angelo'
```

We can use single quote escape sequence to solve the problem

```
let name = 'Adam D\'Angelo'
```

Similarly we can use \" inside a string with double quotes

Other escape sequence characters are as follows

\n → Newline

\t → Tab

\r → Carriage Return

## String properties and Methods

17 let name = "Harry"  
name.length → prints 5

27 let name = "Harry"  
name.toUpperCase() → prints HARRY

37 let name = "Harry"  
name.toLowerCase() → prints harry



4> let name = "Harry"  
name.slice(2, 4) → prints rr  
(from 2 to 4, 4 not included)

5> let name = "Harry"  
name.slice(2) → prints rry  
(from 2 to end)

6> let name = "Harry Bhai"  
let newName = name.replace("Bhai", "Bhai")

7> let name1 = "Harry"  
let name2 = "Naman"  
let name3 = name1.concat(name2, "Yes")  
↳ We can even use + operator

8> let name = " Harry "  
let newName = name.trim()  
↳ Removes whitespaces

Strings are immutable. In order to access the character at an index we use the following syntax

```
let name = "Harry"  
name[0] → Prints H  
name[1] → Prints a
```



## Chapter 4 - Practice Set

- 1 What will the following print in JavaScript?  
`console.log("har".length)`
- 2 Explore the `includes`, `startsWith` & `endsWith` functions of a string
- 3 Write a program to convert a given string to lowercase
- 4 Extract the amount out of this string  
"Please give Rs 1000"
- 5 Try to change 4<sup>th</sup> character of a given string.  
Were you able to do it?



## Chapter 5 - Practice Set

- 1 Create an array of numbers and take input from the user to add numbers to this array.
- 2 Keep adding numbers to the array in ① until 0 is added to the array.
- 3 Filter for numbers divisible by 10 from a given array.
- 4 Create an array of square of given numbers.
- 5 Use reduce to calculate factorial of a given number from an array of first  $n$  natural numbers. ( $n$  being the number whose factorial needs to be calculated)



## Chapter 5 - Arrays

Arrays are variables which can hold more than one value.

```
const fruits = ["banana", "apple", "grapes"]
```

```
const a1 = [7, "Harry", false]
```

↳ can be different types

### Accessing values

```
let numbers = [1, 2, 7, 9]
```

numbers[0] → 1

numbers[1] → 2

### Finding the length

```
let numbers = [1, 7, 9, 21]
```

numbers[0] → 1

numbers.length → 4

### Changing the values

```
let numbers = [7, 2, 40, 9]
```

```
numbers[2] = 8
```

↳ "numbers" now becomes [7, 2, 8, 9]

Arrays are mutable

Arrays can be changed



In JavaScript, arrays are objects. The type of operator on arrays returns object

```
const n = [1, 7, 9]
```

typeof n → returns "object"

Arrays can hold many values under a single name

### Array methods

There are some important array methods in JavaScript. Some of them are as follows:

1. `toString()` → Converts an array to a string of comma separated values

```
let n = [1, 7, 9]
n.toString() → 1, 7, 9
```

2. `join()` → joins all the array elements using a separator

```
let n = [7, 9, 13]
n.join("-") → 7-9-13
```

3. `pop()` → removes last element from the array

```
let n = [1, 2, 4]
n.pop() → updates the original array
returns the popped value
```



4> `push()` → Adds a new element at the end of the array

let `a = [7, 1, 2, 8]`

`a.push(9)` → modifies the original array  
↳ returns the new array length

5> `shift()` → Removes first element and returns it

6> `unshift()` → Adds element to the beginning  
Returns new array length

7> `delete` → Array elements can be deleted using the `delete` operator

let `d = [7, 8, 9, 10]`

`delete d[1]` → `delete` is an operator

8> `concat()` → Used to join arrays to the given array

let `a1 = [1, 2, 3]`

let `a2 = [4, 5, 6]`

let `a3 = [9, 8, 7]`

`a1.concat(a2, a3)` → Returns `[1, 2, 3, 4, 5, 6, 9, 8, 7]`

↓  
Returns a new array  
Does not change existing arrays



9> `Sort()` → `sort()` method is used to sort an array alphabetically.

let `a = [7, 9, 8]`  
`a.sort()`

↳ `a` changes to `[7, 8, 9]`  
 [modifies the original array]

`Sort()` takes an optional compare function. If this function is provided as the first argument, the `Sort()` function will consider these values (the values returned from the compare function) as the basis of sorting.

10> `splice()` → `splice` can be used to add new items to an array

const `numbers = [1, 2, 3, 4, 5]`  
`numbers.splice(2, 1, 23, 24)`

Returns deleted items, modifies the Array ←

position to add ↓

No of elements to remove ↓

Elements to be added →

11> `slice()` → slices out a piece from an array. It creates a new array

const `num = [1, 2, 3, 4]`

`num.slice(2)` → `[3, 4]`

`num.slice(1, 3)` → `[2, 3]`



12. `reverse()` → Reverses the elements in the source array.

### Looping through Arrays

Arrays can be looped through using the classical JavaScript `for` loop or through some other methods discussed below

1. `forEach` loop → calls a function, once for each array element

```
const a = [1, 2, 3]
a.forEach ( (value, index, array) => {
  // function logic
});
```

2. `map()` → creates a new array by performing some operation on each array element.

```
const a = [1, 2, 3]
a.map ( (value, index, array) => {
  return value * value;
});
```

3. `filter()` → filters an array with values that passes a test. Creates a new array

```
const a = [1, 2, 3, 4, 5]
a.filter ( greater-than-5 );
```

4, reduce method → Reduces an array to a single value

```
const n = [1, 8, 7, 11]
let sum = numbers.reduce(add)
           ↓
           1+8+7+11
```

↳ A function

5, Array.from → used to create an array from any other object

```
Array.from("Harry")
```

6, for... of → For-of loop can be used to get the values from an array

7, for... in → for-in loop can be used to get the keys from an array.



## Chapter 6 - JavaScript in the browser

JavaScript was initially created to make web pages alive. JS can be written right in a web page's HTML to make it interactive.

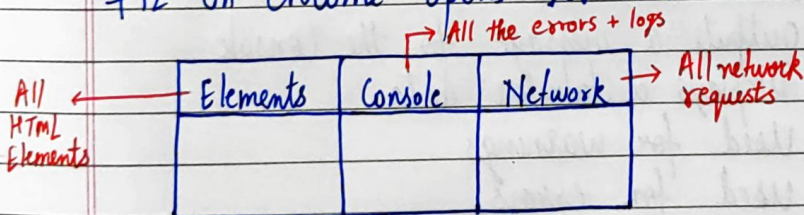
The browser has an embedded engine called the JavaScript engine or the JavaScript runtime.

JavaScript's ability in the browser is very limited to protect the user's safety. For example a webpage on <http://google.com> cannot access <http://codeswear.com> and steal information from there.

### Developer tools

Every browser has some developer tools which makes a developer's life a lot easier.

F12 on Chrome opens Dev tools



We can also write JavaScript commands in the Console

### The script tag

The script tag is used to insert JavaScript into an HTML page



The script tag can be used to insert external or internal scripts

```
<script>  
  alert("Hello")  
</script>  
// or ...  
<script src = " /Js/thisone.js" > </script >
```

The benefit of a separate javascript file is that the browser will download it and store it in its cache

### Console object methods

The console object has several methods, log being one of them. Some of them are as follows:

assert () → Used to assert a condition

clear () → clears the console

log () → Outputs a message to the console

table () → Displays a tabular data

warn () → Used for warnings

error () → Used for errors

info () → Used for special information

You will naturally remember some or all of these with time

Comprehensive list can be looked up on MDN



Interaction: alert, prompt and confirm

alert: Used to invoke a mini window with a msg.

`alert("hello")`

prompt: Used to take user input as string

`inp = prompt("Hi", "No")`

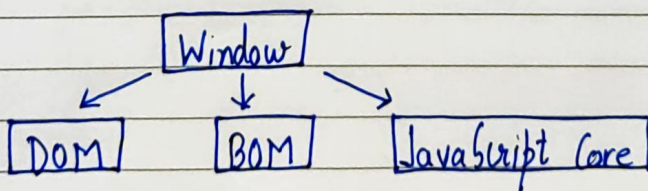
↳ optional default value

Confirm: Shows a message and waits for the user to press OK or cancel. Returns true for OK and false for cancel.

The exact location & look is determined by the browser which is a limitation

Window object, BOM & DOM

We have the following when JavaScript runs in a browser



Window object represents browser window and provides methods to control it. It is a global object

## Document Object Model (DOM)

Dom represents the page content as HTML

`document.body` → Page body as JS object

`document.body.style.background = "green"`

↳ Changes page background to green

## Browser Object Model (BOM)

The Browser Object Model (BOM) represents additional objects provided by the browser (host environment) for working with everything except the document.

The functions `alert` / `confirm` / `prompt` are also a part of the BOM

`location.href = "https://codewithharry.com"`

↳ Redirect to another URL



## Chapter 6 - Practice Set

- 1 Write a program using prompt function to take input of age as a value from the user and use alert to tell him if he can drive!
- 2 In Q1 use confirm to ask the user if he wants to see the prompt again
- 3 In the previous question, use console.error to log the error if the age entered is negative
- 4 Write a program to change the url to google.com (Redirection) if user enters a number greater than 4
- 5 Change the background of the page to yellow, red or any other color based on user input through prompt.

## Chapter 7 - Walking the DOM

DOM tree refers to the HTML page where all the nodes are objects. There can be 3 main types of nodes in the DOM tree :

- 1> text nodes
- 2> element nodes
- 3> comment nodes

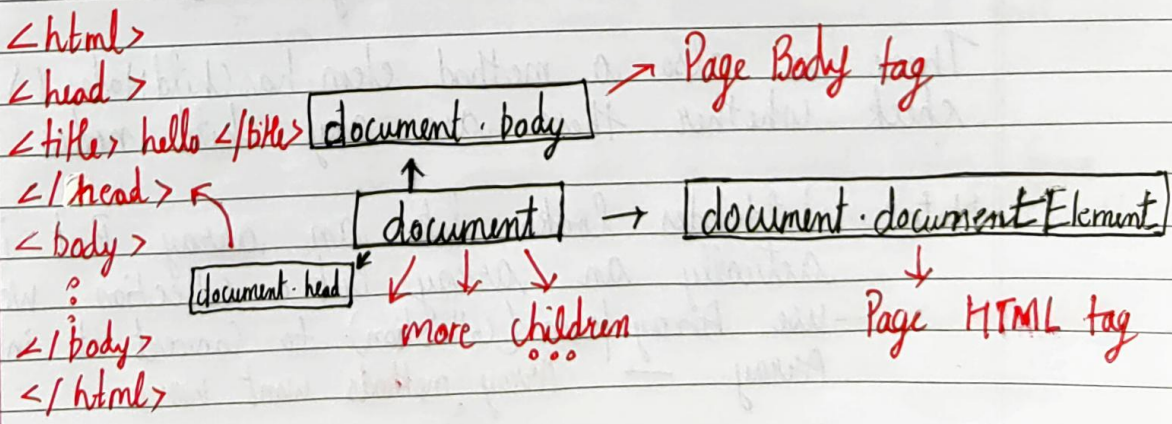
In an HTML page, `<html>` is at the root and `<head>` and `<body>` are its children, etc.

A text node is always a leaf of the tree

### Auto Correction

If an erroneous HTML is encountered by the browser, it tends to correct it. For example, if we put something after the body, it is automatically moved inside the body. Another example is `<table>` tag which must contain `<tbody>`

### Walking the DOM





Note: document.body can sometimes be null if the javascript is written before the body tag.

Children of an element

Direct as well as deeply nested elements of an element are called its children

Child nodes → Elements that are direct children  
For example head & body are children of `<html>`

Descendant nodes → All nested elements, children, their children and so on...

firstChild, lastChild & childNodes

element.firstChild → first child element

element.lastChild → last child element

element.childNodes → All child nodes

Following is always true:

$elem.childNodes[0] === elem.firstChild$

$elem.childNodes[elem.childNodes.length - 1] === elem.lastChild$

There is also a method `elem.hasChildNodes()` to check whether there are any child nodes.

Note: childNodes looks like an array. But it's not actually an array but a collection. We can use `Array.from(collection)` to convert it into an Array. → Array methods won't work



## Notes on DOM collections

- They are read-only
- They are live. `elem.childNodes` variable (reference) will automatically update if `childNodes` of `elem` is changed.
- They are iterable using `for...of` loop

## Siblings and the parent

Siblings are nodes that are children of the same parent.

- For example: `<head>` and `<body>` are siblings. Siblings have same parent. In the above example its `html`
- `<body>` is said to be the "next" or "right" sibling of `<head>`, `<head>` is said to be the "previous" or "left" sibling of `<body>`
- The next sibling is in `nextSibling` property, and the previous one in `previousSibling`.  
The parent is available as `parentNode`.

```
< alert ( document. documentElement. parentNode ); //document
  alert ( document. documentElement. parentElement ); // null
```

## Element only Navigation

Sometimes, we don't want text or comment nodes. Some links only take Element nodes into account. For example

`document.previousElementSibling` → Previous sibling which is an Element



document.nextElementSibling → next sibling (Element)

document.firstElementChild → first Element child

document.lastElementChild → last Element child.

### Table links

Certain DOM elements may provide additional properties specific to their type for convenience.

Table element supports the following properties :

table.rows → collection of tr elements

table.caption → reference to <caption>

table.tHead → reference to <thead>

table.tFoot → reference to <tfoot>

table.tBodies → collection of <tbody> elements

tbody.rows → collection of <tr> inside

tr.cells → collection of td and th

tr.sectionRowIndex → index of tr inside enclosing element

tr.rowIndex → Row number starting from 0

td.cellIndex → no of cells inside enclosing <tr>

Quick Quiz: Print `typeof document` and `typeof window` in the console & see what it prints



## Searching the DOM

DOM navigation properties are helpful when the elements are close to each other. If they are not close to each other, we have some more methods to search the DOM

→ `document.getElementById`

This method is used to get the element with a given "id" attribute

```
let span = document.getElementById('span')  
span.style.color = "red"
```

→ `document.querySelectorAll`

Returns all elements inside an element matching the given CSS selector

→ `document.querySelector`

Returns the first element for the given CSS Selector.  
A efficient version of `elem.querySelectorAll(css)[0]`

→ `document.getElementsByTagName`

Returns elements with the given tag name

→ `document.getElementsByClassName`

Returns elements that have the given CSS class

↳ Don't forget the "s" letter

→ `document.getElementsByName`

Searches elements by the name attribute.



matches, closest & contains methods

There are three important methods to search the DOM

- 1) `elem.matches(css)` → To check if element matches the given CSS selector
- 2) `elem.closest(css)` → To look for the nearest ancestor that matches the given CSS-selector. The elem itself is also checked
- 3) `elemA.contains(elemB)` → Returns true if `elemB` is inside `elemA` (a descendant of `elemA`) or when `elemA == elemB`

## Chapter 7 - Practice Set

- 1 Create a navbar and change the color of its first element to red.
- 2 Create a table without tbody. Now use "View page source" button to check whether it has a tbody or not.
- 3 Create an element with 3 children. Now change the color of first and last element to green.
- 4 Write a javascript code to change background of all `<li>` tags to cyan.
- 5 Which of the following is used to look for the farthest ancestor that matches a given CSS selector  
(a) matches    (b) closest    (c) contains    (d) none of these



## Chapter 8 - Events & other DOM properties

### Console.dir function

console.log shows the element DOM tree

console.dir shows the element as an object with its properties

### tagName / nodeName

Used to read tag name of an element

tagName → only exists for Element nodes

nodeName → defined for any node (text, comment etc.)

### innerHTML and outerHTML

The innerHTML property allows to get the HTML inside the element as a string.

↳ Valid for element nodes only

The outerHTML property contains the full HTML, innerHTML + the element itself.

innerHTML is valid only for element nodes. For other node types we can use nodeValue or data.

### textContent

Provides access to the text inside the element: only text, minus all tags.

### The hidden property

The "hidden" attribute and the DOM property specifies whether the element is visible or not.



< div hidden > I am hidden </div>

< div id = "element" > I can be hidden </div>

```
< script >
  element.hidden = true;
</ script >
```

### Attribute methods

- 1> elem.hasAttribute (name) → Method to check for existence of an attribute
- 2> elem.getAttribute (name) → Method used to get the value of an attribute
- 3> elem.setAttribute (name, value) → Method used to set the value of an attribute.
- 4> elem.removeAttribute (name) → Method to remove the attribute from elem.
- 5> elem.attributes → Method to get the collection of all attributes

### data - \* attributes

We can always create custom attributes but the ones starting with "data-" are reserved for programmers use. They are available in a property named dataset.



If an element has an attribute named "data-one", its available as `element.dataset.one`

### Insertion methods

We looked at some ways to insert elements in the DOM. Here is another way:

```
let div = document.createElement('div') // create
```

```
div.className = "alert" // set class
```

```
div.innerHTML = "<span>hello </span>"
```

```
document.body.appendChild(div)
```

Here are some more insertion methods:

1. `node.appendChild(e)` → append at the end of node
2. `node.prepend(e)` → Insert at the beginning of node
3. `node.before(e)` → Insert before node
4. `node.after(e)` → Insert after node
5. `node.replaceWith(e)` → replaces node with the given node.

Quick Quiz: Try out all these methods with your own webpage.

## insert Adjacent HTML / Text / Element

This method is used to insert HTML. The first parameter is a code word, specifying where to insert. Must be one of the following:

- 1> "beforebegin" - Insert HTML immediately before element
- 2> "afterbegin" - Insert HTML into element at the beginning
- 3> "beforeend" - Insert HTML into element at the end.
- 4> "afterend" - Insert HTML immediately after element.

The second parameter is an HTML string

Example :

```
<div id="div"> </div>
<script>
  div.insertAdjacentHTML('beforebegin', '<p> Hello </p>');
  div.insertAdjacentHTML('afterend', '<p> Bye </p>');
</script>
```

The output would be :

```
<p> Hello </p>
<div id="div"> </div>
<p> Bye </p>
```



## Node removal

To remove a node, there's a method `node.remove()`

```
let id1 = document.getElementById("id1")
```

```
id1.remove()
```

## className and classList

If we assign something to `elem.className`, it replaces the whole string of classes.

Often we want to add/remove/toggle a single class.

1> `elem.classList.add/remove("class")` - Adds/removes a class

2> `elem.classList.toggle("class")` - Adds the class if it doesn't exist, otherwise removes it.

3> `elem.classList.contains("class")` - Checks for the given class, returns true/false

## setTimeout and setInterval

`setTimeout` allows us to run a function once after the interval of time.

Syntax of `setTimeout` is as follows:

```
let timerId = setTimeout(function, <delay>, <arg1>, <arg2>)
```

↓  
returns a timerId

↓  
in ms



clearTimeout is used to cancel the execution (in case we change our mind). For example:

```
let timerId = setTimeout(() => alert("never"), 1000);
```

```
clearTimeout(timerId)
```

↳ cancel the execution

setInterval method has a similar syntax as setTimeout:

```
let timerId = setInterval(function, <delay>, <arg1>, <arg2>);
```

All arguments have the same meaning. But unlike setTimeout, it runs the function not only once, but regularly after the given interval of time.

To stop further calls, we can use clearInterval(timerId).

## Browser Events

An event is a signal that something has happened. All the DOM nodes generate such signals.

Some important DOM events are:

Mouse events: click, contextmenu (right click), mouseover/mouseout, mousedown/mouseup, mousemove

Keyboard events: keydown and keyup



form element events: submit, focus etc.

Document events: DOMContentLoaded

## Handling Events

Events can be handled through HTML attributes

```
<input value = "Hey" onclick = "alert('hey')" type = "button" >
```

↳ can be another JS function

Events can also be handled through the onclick property

```
elem.onclick = function() {  
    alert("yes")  
};
```

Note: Adding a handler with JavaScript overwrites the existing handler

addEventListener and removeEventListener

addEventListener is used to assign multiple handlers to an event.

```
element.addEventListener(event, handler)
```

```
element.removeEventListener(event, handler)
```

↳ handler must be the same function object for this to work

## The Event Object

When an event happens, the browser creates an event object, puts details into it and passes it as an argument to the handler

```
elem.onclick = function (event) {  
    ...  
}
```

event.type : Event type

event.currentTarget : Element that handled the event

event.clientX / event.clientY : Coordinates of the cursor



## Chapter 8 - Practice Set

- 1 Write a program to show different alerts when different buttons are clicked
- 2 Create a website which is capable of storing bookmarks of your favorite websites using href
- 3 Repeat Q2 using event listeners
- 4 Write a javascript program to keep fetching contents of a website (Every 5 seconds)
- 5 Create a glowing bulb effect using classlist toggle method in JavaScript



## Chapter 9 - Callbacks, promises & async/await

Asynchronous actions are the actions that we initiate now and they finish later. eg. setTimeout

Synchronous actions are the actions that initiate and finish one-by-one

### Callback functions

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete an action.

Here is an example of a callback:

```
function loadScript (src, callback) {  
  let script = document.createElement('script')  
  script.src = src  
  script.onload = () => callback(script)  
  document.head.appendChild(script)  
}
```

Now we can do something like this:

```
loadScript ('https://cdn.harry.com', (script) => {  
  alert('script is loaded')  
  alert(script.src)  
});
```



This is called "callback-based" style of async programming. A function that does something asynchronously should provide a callback argument where we put the function to run after its complete.

### Handling errors

We can handle callback errors by supplying error argument like this:

```
function loadScript (src, callback) {
```

```
  ...
```

```
  ...
```

```
  script.onload = () => callback (null, script);
```

```
  script.onerror = () => callback (new Error ('failed'));
```

```
  ...
```

```
}
```

Then inside of loadScript call:

```
loadScript ('cdn/harry', function (error, script) {
```

```
  if (error) {
```

```
    // handle error
```

```
  }
```

```
  else {
```

```
    // script loaded
```

```
  }
```

```
});
```



## Pyramid of Doom

When we have callback inside callbacks, the code gets difficult to manage

```
loadScript((...)) {
```

```
  loadScript ..
```

← Pyramid of Doom

```
    loadScript ...
```

```
      loadScript
```

```
        ..
```

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have real code instead of ...

This is sometimes called "callback hell" or "pyramid of doom"

The "pyramid" of these calls grows towards the right with every asynchronous action. Soon it spirals out of control. So this way of coding isn't very good!

## Introduction to Promises

The solution to the callback hell is promises.

A promise is a "promise of code execution". The code either executes or fails, in both the cases the subscriber will be notified.



The syntax of a Promise looks like this :

```
let promise = new Promise (function (resolve, reject) {
    // executor
});
```

predefined  
in JS engine

resolve and reject are two callbacks provided by javascript itself. They are called like this :

resolve (value) → If the job is finished successfully  
 reject (error) → If the job fails

The 'promise object returned by the new Promise constructor has these properties

- 1> state : Initially pending, then changes to either "fulfilled" when resolve is called or "rejected" when reject is called
- 2> result : Initially undefined, then changes to value if resolved or error when rejected  
           resolve(value)                      reject(error)

Consumers : then & catch

The consuming code can receive the final result of a promise through then & catch

The most fundamental one is then  
 promise . then ( function (result) { /\* handle \*/ },  
                   function (error) { /\* handle error \*/ }  
                   );



If we are interested only in successful completions, we can provide only one function argument to `.then()`:

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

```
promise.then (alert);
```

If we are interested only in errors, we can use `null` as the first argument: `.then(null, f)` or we can use `catch`:

```
promise.catch (alert)
```

`promise.finally ( () => { } )` is used to perform general cleanups

Quick Quiz: Rewrite the `loadScript` function we wrote in the beginning of this chapter using promises.

### Promises Chaining

We can chain promises and make them pass the resolved values to one another like this

```
p.then (function (result) => {
  alert (result); return 2;
}).then ...
```

// p is a promise



The idea is to pass the result through the chain of .then handlers.

Here is the flow of execution

1. The initial promise resolves in 1 seconds (Assumption)
2. The next .then() handler is then called, which returns a new promise (resolved with 2 value)
3. The next .then() gets the result of previous one and this keeps on going

Every call to .then() returns a new promise whose value is passed to the next one and so on. We can even create custom promises inside .then()

### Attaching multiple handlers

We can attach multiple handlers to one promise. They don't pass the result to each other; instead they process it independently.

Let  $p$  is a promise

$p$ .then(handler 1)

$p$ .then(handler 2)

$p$ .then(handler 3)



Runs Independently



## Promise API

There are 6 static methods of Promise class:

- 1> `Promise.all (promises)` → Waits for all promises to resolve and returns the array of their results. If any one fails, it becomes the error & all other results are ignored.
- 2> `Promise.allSettled (promises)` → Waits for all the promises to settle and returns their results as an array of objects with status and value.
- 3> `Promise.race (promises)` → Waits for the first promise to settle and its result/error becomes the outcome.
- 4> `Promise.any (promises)` → Waits for the first promise to fulfill (& not rejected), and its result becomes the outcome. Throws `AggregateError` if all the promises are rejected.
- 5> `Promise.resolve (value)` → Makes a resolved promise with the given value.
- 6> `Promise.reject (error)` → Makes a rejected promise with the given error.

Quick Quiz : Try all these promise APIs on your custom promises.



## Async / Await

There is a special syntax to work with promises in javascript

A function can be made async by using async keyword like this :

```
async function harry() {  
  return 7;  
}
```

An async function always returns a promise. Other values are wrapped in a promise automatically.

We can do something like this :

```
harry().then(alert)
```

So, async ensures that the function returns a promise and wraps non promises in it.

## The await keyword

There is another keyword called await that works only inside async functions

```
let value = await promise;
```

The await keyword makes javascript wait until the promise settles and returns its value.



It's just a more elegant syntax of getting the promise result than `promise.then` + it's easier to read & write

## Error Handling

We all make mistakes. Also sometimes our script can have errors. Usually a program halts when an error occurs.

The `try...catch` syntax allows us to catch errors so that the script instead of dying can do some thing more reasonable

## The `try...catch` syntax

The `try catch` syntax has two main blocks: `try` and then `catch`

```
try {
  // try the code
```

```
} catch (err) {
  // error handling
}
```

⇒ The `err` variable contains an error object

It works like this

1. first the code in `try` is executed
2. If there is no error, `catch` is ignored else `catch` is executed



try catch works synchronously  
 If an exception happens in scheduled code, like in setTimeout, then try... catch wont catch it :

```
try {
  setTimeout ( function () {
    // error code
  }
}
catch ...
```

→ Script dies and catch wont work

Thats because the function itself is executed later, when the engine has already left the try... catch construct.

The error object  
 For all the built in errors, the error object has two main properties :

```
try {
  key; // error variable not defined
} catch (err) {
  alert (err.name)
  alert (err.message)
  alert (err.stack)
}
```



## Throwing Custom Error

We can throw our own error by using the throw syntax

```
if (age > 180) {  
    throw new Error("Invalid Age")  
    ...  
}
```

We can also throw a particular error by using the built in constructor for standard errors:

```
let error = new SyntaxError(message)  
           or  
           new ReferenceError(message)
```

## The finally clause

The try...catch construct may have one more code clause: finally

If it exists it runs in all cases:

after try if there were no errors  
after catch if there were errors

If there is a return in try, finally is executed just before the control returns to the outer code.



## Chapter 9 - Practice Set

- 1 Write a program to load a JavaScript file in a browser using Promises. Use `.then()` to display an alert when the load is complete.
- 2 Write the same program from previous question and use `async/await` syntax.
- 3 Create a promise which rejects after 3 seconds. Use an `async/await` to get its value. Use a `try catch` to handle its error.
- 4 Write a program using `Promise.all()` inside an `async/await` to await 3 promises. Compare its results with the case where we await these promises one by one.



## Chapter 11 - Practice Set

- 1 Create a JavaScript class to create a complex number. Create a constructor to set the real and the complex part
- 2 Write a method to add two complex numbers in the above class
- 3 Create a class Student from a class Human. Override a method & see changes
- 4 See if Student is an instance of Human using instanceof keyword.
- 5 Use getters & setters to set and get the real and imaginary parts of the complex number

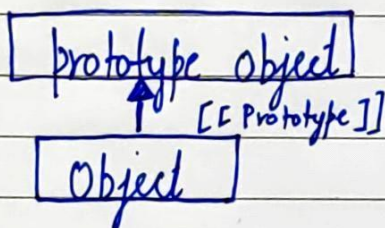


## Chapter 11 - Object Oriented Programming

In programming we often take something and then extend it. For example we might want to create a user object and "Admin" and "guest" will be slightly modified variants of it.

### [[ Prototype ]]

JavaScript objects have a special property called prototype that is either null or references another object.



When we try to read a property from a prototype and it's missing, JavaScript automatically takes it from the prototype. This is called "prototypal inheritance".

### Setting Prototype

We can set prototype by setting `-- proto --`. Now if we read a property from the object which is not in object and is present in the prototype, JavaScript will take it from prototype.

If we have a method in object, it will be called from the object. If it's missing in object and present in prototype, it's called from the prototype.



## Classes and Objects

In object-oriented programming, a class is an extensible program-code template for creating objects, providing initial values for state (member variables) and implementation of behavior (member functions)

The basic syntax for writing a class is :

```
class MyClass {  
    // class methods  
    constructor () { ... }  
    method 1 () { ... }  
    method 2 () { ... }  
}
```

We can then use `new MyClass()` to create a new object with all the listed methods

### The constructor method

The `constructor()` method is called automatically by `new`, so we can initialize the object there

**Quick Quiz :** Create a class `user` and create a few methods along with a constructor.

### Class Inheritance

Class Inheritance is a way for one class to extend another class. This is done by using the `extends` keyword.



The extends keyword extends keyword is used to extend another class.

class Child extends Parent

We can create a class Monkey that inherits from Animal

```
class Monkey extends Animal {
    hide () {
        alert (` ${this.name} hides !`);
    }
}
```

```
let monkey = new Monkey ("Monu")
monkey.run (7); // From Animal
monkey.hide ();
```

### Method Overriding

If we create our own implementation of run, it will not be taken from the Animal class.

This is called Method Overriding

### super keyword

When we override a method, we don't want the method of the previous class to go in vain.

We can execute it using super keyword.

super (a, b) → call parent constructor



```
run () {
    super.run ()
    this.hide ()
}
```

### Overriding Constructor

With a constructor, things are a bit tricky / different. According to the specification, if a class extends another class and has no constructor, then the following empty constructor is generated

```
class Monkey extends Animal {
    // auto generated
    constructor (...args) {
        super (...args);
    }
}
```

⇒ Happens if we don't write our own constructor

Constructors in inheriting classes must call `super (...)` and do it before using `this`.

We can also use `super.method()` in a child method to call Parent Method

### Static method

Static methods are used to implement functions that belong to a class as a whole and not to any particular object



We can assign a static method as follows:

```
class Employee {
    static sMethod () {
        alert ("Hey");
    }
}
```

Employee.sMethod()

Static methods aren't available for individual objects

### Getters and Setters

Classes may include getters and setters to get & set the computed properties

Example :

```
class Person {
    ..
    get (name) {
        return this._name;
    }
    set name (newName) {
        this._name = newName;
    }
}
```

First the name property is changed to `_name` to avoid the name collision with the getter & setter. Then the getter uses the `get` keyword as shown above



## Instance of Operator

The instance of operator allows to check whether an object belongs to a certain class

The syntax is:

`<obj> instanceof <class>`

It returns true if obj belongs to the class or any other class inheriting from it



## Chapter 12 - Advanced JavaScript

There are some JavaScript concepts which make the life of a developer extremely simple. We will discuss some of those in this chapter.

### IIFE

IIFE is a JavaScript function that runs as soon as it is defined.

```
( function () {
```

```
  ...
```

```
  ...
```

```
  } ) ();
```

⇒ IIFE Syntax

It is used to avoid polluting the global namespace, execute an async-await, etc.

### Destructuring

Destructuring assignment is used to unpack values from an array, or properties from objects, into distinct variables.

```
let [x, y] = [7, 20]
```

x will be assigned 7 and y, 20

```
[10, x, ...rest] = [10, 80, 7, 11, 21, 88]
```

x will be 80 rest will be [7, 11, 21, 88]



Similarly we can destructure objects on the left hand side of the assignment

```
const obj = { a: 1, b: 2 }
const { a, b } = obj;
```

Some more examples can be found on MDN docs.

### Spread Syntax

Spread syntax allows an iterable such as an array or string to be expanded in places where zero or more arguments are expected. In an object literal, the spread syntax enumerates the properties of an object and adds the key-value pairs to the object being created.

Example :

```
① const arr = [ 1, 7, 11 ]
   const obj = { ...arr }; // { 0: 1, 1: 7, 2: 11 }
```

```
② const nums = [ 1, 2, 7 ]
   console.log (sum (...nums)) // 10
```

Other examples can be found on MDN docs

Quick Quiz : Output of the following ??

```
const a = "the", b = "no"
```

```
const c = { a, b }
```

```
console.log (c)
```



local, global & block scopes  
JavaScript has three types of scopes:

- 1> Block Scope
- 2> Function Scope
- 3> Global Scope

let & const provide block level scope which means that the variables declared inside a `{ }` cannot be accessed from outside the block

```
{
```

```
  let a = 27;
```

```
}
```

```
// a is not available here
```

Variables declared within a JavaScript function, become local to the function

A variable declared outside a function, becomes global

### Hoisting

Hoisting refers to the process whereby the interpreter appears to move the declarations to the top of the code before execution

Variables can thus be referenced before they are declared in JavaScript



```
hello("Harry")
```

```
function hello(name) {  
    ...  
}
```

⇒ Works!

Important Note: JavaScript only hoists declarations, not initializations. The variable will be undefined until the line where its initialized is reached.

Hoisting with let and var

With let and var hoisting is different

```
console.log(num)  
let num = 6;
```

→ Error if let or const

→ with var undefined is printed

Function expressions and class expressions are not hoisted



## Chapter 12 - Practice Set

1 Write a JavaScript program to print the following after 2 second delay

Hello  
World

2 Write a JavaScript program to find average of numbers in an array using spread syntax.

3 Write a JavaScript function which resolves a Promise after  $n$  seconds. The function takes  $n$  as the parameter. Use an IIFE to execute the functions with different values of  $n$ .

4 Write a simple interest calculator using JavaScript.



## Exercise 1 - Guess the number

Write a JavaScript program to generate a random number and store it in a variable. The program then takes an input from the user to tell them whether the guess was correct, greater or lesser than the original number.

100 - (no of guesses) is the score of the user. The program is expected to terminate once the number is guessed. Number should be between 1 - 100.



## Exercise 2 - Snake Water Gun

Use Javascript to create a game of Snake Water & Gun. The game should ask you to enter S, W or G. The computer should be able to randomly generate S, W or G and declare Win or Loss using alert. Use confirm and prompt wherever required.



### Exercise 3 - Tell me a Joke

elem.innerHTML is used to populate a div with HTML. Search online about this method and create a website with a div tag containing a random joke given an array of jokes. Use Math.random and fetch jokes from the internet (use any website to create the array). Your website should show a random joke on every reload. Min length of your jokes array should be 10.



## Exercise 4 - Digital Clock

- 1 Create a Digital Seconds clock using `setInterval` and `Date` object in JavaScript. The `Date` object can be used to get the date, time, hours and seconds which can be updated using `setInterval`. Try to keep the UI good looking.



## Exercise 5 - Hackerman

Write a javascript program to pretend to look like a hacker. Write an async function which will simply display the following output:

Initializing Hack program ...

Hacking Ashish's username ...

Username found ashish17 ...

Connecting to facebook ...

Try to use HTML & Styling if possible



## Exercise 6 - TODO List

Create a TODO List app capable of storing your TODOs in local storage. Add an option to create, delete and access all the TODOs.

Try to make UI as good as possible



## Exercise 7 - Password Generator

Create a JavaScript program capable of generating a password which contains atleast one lowercase, one uppercase and one special characters.

Create a Password class to achieve the same



## Exercise 8 - Alarm Clock

The HTML Audio Element Interface can be used to play audio in the browser. Create an alarm clock which displays time and plays sound at a user specified time.