Chapter 1 – Introduction

JavaScript can change HTML content- getElementById().
e.g., document.getElementById("demo").innerHTML = "Hello JavaScript";
JavaScript can change HTML Attribute Values –
e.g., src attribute of img tag
JavaScript can change HTML style(CSS) –
document.getElementById("demo").style.fontSize = "35px";

Chapter 2 – JavaScript Where to

In HTML, JavaScript code is inserted between <script> and </script>
Tags.
JavaScript code can be inserted in head or body.
To use an external script, put the name of the script file in the src
attribute of a <script> tag

Chapter 3- JavaScript Output

JavaScript can "display" data in different ways:

- Using innerHTML.
- document.write()
- window.alert()
  window is global scope object so we can skip it.
- console.log()

JavaScript Print
window.print() method in the browser to print the content of the current window.

Chapter 4- JavaScript Statements

JavaScript statements are lines of codes separated by ;
Keywords – var, let, const, if, switch, for, function, return, try etc.
var, let and const to declare variables.

```
let x, y;
x = 5 + 6;
y = x * 10
```

- rule of constructing identifiers name must begins with a letter, a dollar sign or a underscore.
- Commenting- // or /* */
- JavaScript is a case sensitive language

Chapter 5- JavaScript Variables

```
let x, y;
x = 5 + 6;
y = x * 10
let z = x- y;
                or,
var x, y;
x = 5 + 6;
y = x * 10
var z = x – y;
```

let person = "John Doe", carName = "Volvo", price = 200;
let carName;
// carName has value = undefined

Re-declaring JavaScript Variable
var carNmae = 'Volvo'
var carName;
it will not loose its value. but we cannot re declare variable declared with let or const.

Before ES6(2015), JavaScript had Global Scope and Function Scope.
ES6 introduced two important new JavaScript keywords: let and const.
These two keyword provide Block Scope in JavaScript.
Variables declared inside a {} block cannot be accessed from outside the block.

Variables declared with var are hoisted to the top and can be initialized at any time.
We can use the variable before it is declared:
e.g., carName = "Volvo";
    var carName;

Variables defined with let are also hoisted to the top of the block, but not initialized.

JavaScript const variables must be assigned a value when they are declared:
const PI = 3.141592;
const keyword does not define a const value it defines a constant reference to a value.
so we can change the element of constant array and the properties of constant object.
We are not able to reassign the value of the array.

Redeclaring the var or let variable to const in the same scope, is not allowed.

Chapter 6 - JavaScript Operators

There are different types of JavaScript operators:

- Arithmetic (+, -, *, /, %, **, ++, --)
- Assignment (=, +=, -=, *=, /=, %=, **=)
- Comparison (<, >, <=, >=, ?, ==, ===, !=, !==)
- String (<, >, +, +=)
- Logical (&&, ||, !)
- Bitwise (&, |, ~, ^, <<, >>, >>>)
- Ternary
- Type (typeof, instanceof)

The ??= operator
If the first value is undefined or null, the second value is assigned.
Var x = 100;
x ??= 5

Chapter 7 - JavaScript Datatypes

JavaScript has 8 Datatypes(ssbbnnu)

1. String
   let color = “Yellow”;
2. Number
   let length = 16;
3. Bigint
4. Boolean
   let x = true;
5. Undefined
6. Null
7. Symbol
8. Object
   An object
   const person = {firstName:"John", lastName:"Doe"};
   An array
   const cars = ["Saab", "Volvo", "BMW"];
   A date
   const date = new Date("2022-03-25");

JavaScript evaluates expression from left to right.

```
let x = 16 + 4 + "Volvo";            // result – 20Volvo
let x = "Volvo" + 16 + 4;            // result – Volvo164
```

JavaScript types are dynamic

```
let x;         // Now x is undefined
x = 5;         // Now x is a Number
x = “John”;  // Now x is a string
```

Chapter 8 - JavaScript Functions

```
function myfunction(p1, p2) {
        return p1 * p2 ;
}
```

Chapter 9 - JavaScript Objects

Accessing Object Properties -
objectName.propertyName
or,
objectName[propertyName]

Object Method-

```
const person = {
  firstName: "John",
  lastName : "Doe",
  id       : 5566,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

This keyword refers to an object.

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

Accessing Object Methods
objectNmae.methodName( )

When a JavaScript variable is declared with the keyword “new”, the variable is created as an object:

```
X = new String();     // Declares X as a string object
Y = new Number();   // Declares Y as a Number object
Z = new Boolean();   // Declares Z as a Boolean object
```

Chapter 10 - JavaScript Events

HTML events are “things” that happens to HTML elements.
e.g., An HTML web page has finished loading
An HTML input field was changed
An HTML button was clicked
HTML allows event handler attribute, with JavaScript code, to be added to HTML elements.
<element event='some JavaScript'> double quotes are also used.
e.g., <button onclick="this.innerHTML = Date()">The time is?</button>
<button onclick="displayDate()">The time is?</button>

Chapter 11 - JavaScript String

JavaScript are for storing and manipulating text.
A string is zero or more characters written inside quotes.
e.g., let text = "John Doe";
use single or double quotes.

- To find length of a string, use the built-in length property:
  let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  let length = text.length;
- Escape character: The backslash (\) escape character turns special characters into string character.
  e.g., \’   single quote
  \”   double quote
  \\   backslash
  \b, \f, \n, \r, \t, \v
- You can also break up a code line within a text string with a single backslash:
  The \ method is not preferred method. It might not have universal support. A safer way to use string addition:
  e.g.,
  document.getElementById("demo").innerHTML = "Hello +
  “Dolly!";
- Normally, JavaScript strings are primitive values, created from literals:
  let x = “John”;

but strings can also be defined as objects with the keyword new:
let y = new string (“John”);
The new keyword complicates the code and slows down execution speed.

- JavaScript objects cannot be compared.
  (x == y) true or false?
  let x = "John";
  let y = new String("John");
  output: true

  **comparing two js object always returns false.
  let x = new String("John");
  let y = new String("John"); // x == y

  let x = new String("John");
  let y = new String("John"); // x === y

Chapter- 12 String Methods

length,

slice [start, end), **substring** [start, end) the difference between slice and substring is the starting and ending value less than 0 is treated as 0 in substring.

, **substr** (start, length) is similar to slice() the difference is the second parameter specifies the length of the extracted part.

, replace(“string1”, “String2”) this method returns a new string. Replaces only the first match.

By default, the replace() method is case sensitive. Writing MICROSOFT will not work

```
let text = "Please visit Microsoft!";
let newText = text.replace("MICROSOFT", "W3Schools");
```

To replace case insensitive, use regular expression with an /i flag(insensitive)

```
let text = "Please visit Microsoft!";
let newText = text.replace(/MICROSOFT/i, "W3Schools");
```

To replace all matches, use a regular expression with a /g flag (global match):

```
let text = "Please visit Microsoft and Microsoft!";
let newText = text.replace(/Microsoft/g, "W3Schools");
```

replaceAll(“string1”, “string2”) returns a new string.

```
text = text.replaceAll("Cats","Dogs");
text = text.replaceAll("cats","dogs");
```

If the parameter is a regular expression, the global flag (g) must be set, otherwise a typeError is thrown.

```
text = text.replaceAll(/Cats/g,"Dogs");
text = text.replaceAll(/cats/g,"dogs");
```

toUpperCase, toLowerCase,

```
let text1 = "Hello World!";
let text2 = text1.toUpperCase();
```

concat

```
let text1 = "Hello";
let text2 = "World";
let text3 = text1.concat(" ", text2);
```

trim, trimStart, trimEnd, trimStart, used to trim whitespace

padStart, padEnd - let text = "5";
let padded = text.padStart(4,"0");
pad a string with “0” until it reaches the length 4

```
let numb = 5;
let text = numb.toString();
let padded = text.padStart(4,"0");
```

charAt (position), charCodeAt(position)
returns the character at a specified index (position) in a string.
The method returns a UTF-16 code (an integer between 0 and 65535).

```
let text = "HELLO WORLD";
let char = text[0];          // char = H
```

If no character is found, [ ] returns undefined, while charAt() returns an empty string.
It is read only. str[0] = "A" gives no error (but does not work!)
split – A string can be converted to an array with the split( ) method:

```
text.split(“,”)   // split on commas
text.split(“ ”)   // split on spaces
text.split(“,”)   // split on pipe
```

Chapter- 13 String Search

indexOf() – returns the index(position) the first occurrence of a string in 8ua string.
lastIndexOf() – returns the index of the last occurrence of a specified text in a string. This method searches backwards,
Both methods accept a second parameter as the starting position for the search.
search() – searches a string for a string(or a regular expression) and returns the position of the match, doesn’t take second argument.

```
let text = "Please locate where 'locate' occurs!";
text.search("locate");
let text = "Please locate where 'locate' occurs!";
text.search(/locate/);
```

match() – returns an array containing the results of matching a string again a string (or a regular expression).

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match("ain");   // ain

let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/);   // ain

let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/g);   // ain, ain, ain

let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/gi);   // ain, AIN, ain, ain
```

matchAll() – method returns an iterator containing the results of matching a string against a string (or a regular expression),

```
const iterator = text.matchAll("Cats");
document.getElementById("demo").innerHTML =
Array.from(iterator);
```

*if a parameter is regular expression, the global flag(g) must be set.

includes() – method returns true if a string contains a specified value. Otherwise it returns false.

```
let text = "Hello world, welcome to the universe.";
text.includes("world");
let text = "Hello world, welcome to the universe.";
text.includes("world", 12);     // start at position 12
```

startsWith(), endsWith() // can take second argument i.e. starting position. Is case sensitive

Back-Tics Syntax

Template Literals use back-ticks(` `) rather than the quotes (“ ”) to define a string.

let text = `Hello World!`;

- We can use both single and double quotes inside a string.
- Allows multiline strings
- **String Interpolation**

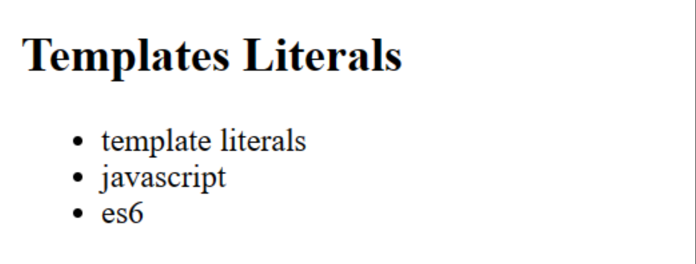
  ${ …….}

  Template literals allow variables in string.

  ```
  let firstName = "John";
  let lastName = "Doe";
  let text = ` Welcome ${firstName}, ${lastName}!`;
  ```
- Expression Substitution

  ```
  let price = 10;
  let VAT = 0.25;

  let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;
  // toFixed is used in float precision
  // rember back-ticks is used in interpolation
  ```

```
<script>
let header = "Templates Literals";
let tags = ["template literals", "javascript", "es6"];

let html = `<h2>${header}</h2><ul>`;

for (const x of tags) {
  html += `<li>${x}</li>`;
}

html += `</ul>`;
document.getElementById("demo").innerHTML = html;
</script>
```



```
Templates Literals

  • template literals
  • javascript
  • es6
```

Chapter- 14 JavaScript Numbers

JavaScript has only one type of number. Number can be written with or without decimals.

JavaScript numbers are always 64 bit floating point.

0-52 for mantissa, 53- 62 for exponent, 63 for sign

Integers are accurate up to 15 digits.

Numeric String

let x = “100”

let y = “10”

let z = x/y;

JavaScript will try to convert strings to numbers in all the numeric operations:

So, *, /, -, % all operators works on numeric String.

NaN is a JavaScript reserved word indicating that number is not a legal number.

Let x = 100 / ”apple”

We can skip ; because JavaScript will understand and execute the statement.

Let x = 100 / “10”

But this will work

isNaN() is a global JavaScript function to find out if a value is not a number.

NaN is a number.

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

Infinity is a number type.

toString() method to output numbers from base 2 to base 36.

```
let myNumber = 32;
myNumber.toString(32);
myNumber.toString(16);
myNumber.toString(12);
myNumber.toString(10);
myNumber.toString(8);
myNumber.toString(2);

let y  = new Number(123);
```

JavaScript Numbers as Objects.

Chapter- 15 JavaScript BigInt

To create a BigInt, append n to the end of an integer or call BigInt():

let x = 9999999999999999;

let y = 9999999999999999n;
or,
let x = 1234567890123456789012345n;
let y = BigInt(1234567890123456789012345)

Operators that can be used on a JavaScript Number can also be used on a BigInt.
Notes: Arithmetic between a BigInt and a Number is not allowed (type conversion lose information).
Unsigned right shift (>>>) can not be done on a BigInt (it does not have a fixed width).

A BigInt can not have decimals.
let x = 5n;
let y = x / 2;
// Error: Cannot mix BigInt and other types, use explicit conversion.
let x = 5n;
let y = Number(x) / 2;

BigInt can also be written in hexadecimal, octal, or binary notation:
let hex = 0x20000000000003n;
let oct = 0o400000000000000003n
let bin = 0b10000000000000000000000000000000000000000000000000011n;

ES6 full form is ECMA Script 6 which is standard for JavaScript
ES6 added max and min properties to the Number Object:
MAX_SAFE_INTEGER -- let y = Number.MAX_SAFE_INTEGER;
MIN_SAFE_INTEGER -- let x = Number.MIN_SAFE_INTEGER;
ES6 also added 2 new method to the Number object:

- Number.isInteger()
- Number.isSafeInteger()

Safe integers are all integers from -(253 - 1) to +(253 - 1).

Chapter- 16 JavaScript Number Methods

toString() - returns a number as a string.
Eg. (123).toString();
toExponential() - returns a string, with the number rounded and written using exponential notation.
e.g. let x = 9.656;
x.toExponential(4)
output - 9.6560e+0
toFixed() - rounds a number to a given number of digits.
e.g. let x = 9.656;
x.toFixed(2)
output - 9.66
toPrecision() – returns a string, with a number written with a specified length.
e.g. let x = 9.656;
x.toPrecision(4)
output - 9.656
valueOf() – returns a number

Converting Variables to Numbers –
There are 3 JavaScript methods that can be used to convert a variable to a number – Number(), parseFloat(), parseInt()
They are global JavaScript methods.

```
Number(true)  - 1      Number(" 10  ") - 10
Number(false) - 0      Number("10.33")  - 10.33
Number("10") - 10      Number("10,33")  - NaN
Number("  10")  - 10   Number("10 33") - NaN
Number("10  ") - 10    Number("John"); - NaN
```

The Number() Method Used on Dates
Number() can also convert a date to a number.
The Date() method returns the number of milliseconds since 1.1.1970.
Number(new Date("1970-01-02")) ---- 86400000

E.g of parseInt()

```
parseInt("-10")      – 10
parseInt("-10.33")  – 10
parseInt("10")          10
parseInt("10.33")      10
parseInt("10 6")        10
parseInt("10 years")  NaN
```

like parseInt(), parseFloat() parses a string and returns a number. Spaces are allowed. Only the first number is returned :

```
parseFloat("10")       10
parseFloat("10.33")  10.33
parseFloat("10 6")     10
parseFloat("10 years") NaN
```

Chapter- 17 JavaScript Number Properties

Number properties belong to the JavaScript Number Object.
These properties can only be accessed as Number.MAX_VALUE.
Using x.MAX_VALUE, where x is a variable or a value, will return undefined.
Number.EPSILON – The difference between 1 and the smallest number > 1
Number.MAX_VALUE - The largest number possible in JS
Number.MIN_VALUE – The minimum number possible in JS
Number.MAX_SAFE_INTEGER - The maximum safe integer (2^53 - 1)
Number.MIN_SAFE_INTEGER - The minimum safe integer -(2^53 - 1)
Number.POSITIVE_INFINITY - Infinity (returned on overflow)
Number.NEGATIVE_INFINITY - Negative infinity (returned on overflow)

Number.NaN - A "Not-a-Number" value

Chapter- 18 JavaScript Arrays

Creating an Array –

```
const array_name = [item1, item2, ...];
```

You can also create an array, and then provide the elements:

```
const cars = [];
cars[0]= "Saab";
cars[1]= "Volvo";
cars[2]= "BMW";
```

The following example also creates an Array, and assigns values to it:

```
const cars = new Array("Saab", "Volvo", "BMW");
```

Accessing Array Elements –

You access an array element by referring to the index number:

```
const cars = ["Saab", "Volvo", "BMW"];
let car = cars[0];
```

With JavaScript, the full array can be accessed by referring to the array name:

```
const cars = ["Saab", "Volvo", "BMW"];
document.getElementById("demo").innerHTML = cars;
```

Array Elements Can Be Objects –

```
myArray[0] = Date.now; // object
myArray[1] = myFunction;  // function
myArray[2] = myCars; // array
```

Array Properties and Methods-

1. **cars.length** // Returns the number of elements
2. **cars.sort()** // Sorts the array
3. Looping Array –
   e.g.
   ```
   const fruits = ["Banana", "Orange", "Apple", "Mango"];
   let fLen = fruits.length;
   let text = "<ul>";
   for (let i = 0; i < fLen; i++) {
       text += "<li>" + fruits[i] + "</li>";
   }
   text += "</ul>";
   ```

   or,
   ```
   let text = "<ul>";
   fruits.forEach(myFunction);
   text += "</ul>";
   document.getElementById("demo").innerHTML = text;

   function myFunction(value) {
     text += "<li>" + value + "</li>";}
   ```

   Array.forEach() calls a function for each array element.
4. Adding Array Elements -
   const fruits = ["Banana", "Orange", "Apple"];
   fruits.**push**("Lemon"); // Adds a new element (Lemon) to fruits
   The push() method returns the new array length
5. JavaScript does not support arrays with named indexes or Associative array.
6. In JS array use numbered indexes.
7. In JS object use named indexed.

e.g.
```
const points = new Array(40, 100, 1, 5, 25, 10);
const points = [40, 100, 1, 5, 25, 10]; // both are same

const points = new Array(40);
// creates array of 40 undefined values.
```

8. How to Recognize an Array
   The problem is that the JavaScript operator typeof returns "object".
   Sol. 1 - Array.**isArray**(fruits);
   Sol. 2 - const fruits = ["Banana", "Orange","Appl"];
   fruits instanceof Array;
9. The JavaScript method toString() converts an array to a string of (comma separated) array values.
   e.g.
   const fruits = ["Banana", "Orange", "Apple", "Mango"];
   fruits.**toString()**;
   Banana,Orange,Apple,Mango // result
10. The join() method also joins all array elements into a string.
    It behaves just like toString(), but in addition you can specify the separator:
    fruits.**join**(" * ");
    Banana * Orange * Apple * Mango // result

11. The pop() method removes the last element from an array -
    fruits.**pop**();

12. The shift() method removes the first array element and "shifts" all other elements to a lower index.
    fruits.**shift**();
    returns the value that was shifted out.

13. The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements: fruits.unshift("Lemon");
    The **unshift**() method returns the new array length
    Lemon,Banana,Orange,Apple,Mango // final array

14. Array elements can be deleted using the JavaScript operator delete.
    Using delete leaves undefined holes in the array.
    Use pop() or shift() instead.
15. The concat() method creates a new array by merging (concatenating) existing arrays:

```
const myChildren = myGirls.concat(myBoys, adopted);
```

The concat() method can also take strings as arguments.

16. Flattening an array is the process of reducing the dimensionality of an array.
    The flat() method creates a new array with sub-array elements concatenated to a specified depth.
17. The splice() method can be used to add new item to an array:
    ```
    const fruits = ["Banana", "Orange", "Apple", "Mango"];
    fruits.splice(2, 0, "Lemon", "Kiwi");
    ```

```
Banana,Orange,Lemon,Kiwi,Apple,Mango // output
```

The first parameter (2) defines the position where new elements should be added (spliced in).

The second parameter (0) defines how many elements should be removed.

The splice method edits the original array and returns deleted items.

18. With clever parameter setting, you can use splice() to remove elements without leaving "holes" in the array:
    ```
    fruits.splice(0, 1);
    ```

19. The slice() method slices out a piece of an array into a new array.
    ```
    fruits.slice(1, 3); // slice array from 1 to 2
    // if second argument not provided it slice array
    // from starting point to end of the array.
    ```
    The slice() method creates a new array.
    The slice() method does not remove any elements from the source array.
20. sort() method sorts an array alphabetically:
    ```
    const fruits = ["Banana", "Orange", "Apple", "Mango"];
    fruits.sort();
    //output-  Apple,Banana,Mango,Orange
    ```

    Numeric sort
    By default, the sort() function sorts values as strings.
    ```
    const points = [40, 100, 1, 5, 25, 10];
    points.sort(function(a, b){return a - b});
    // output- 1, 5,10, 25, 40, 100
    ```
    Use this trick to sort an array descending.

    Sorting in random

```
points.sort(function(){return 0.5 - Math.random()});
```

The above example is not accurate. It will favour some numbers over the others.

The most popular correct method, is called the Fisher Yates shuffle.

```
const points = [40, 100, 1, 5, 25, 10];
for (let i  = points.length – 1; i > 0; i--){
    let j = Math.floor(Math.random() * (i+1));
    let k= points[i];
    points[i] = points[j];
    points[j] = k;
}
```

Note – there is no builtins function to find the min and max in an array. But we can use Math.max.apply(null, arr); to find the highest number in an array. null is optional here
On the other hand using Math.min.apply(arr) to find the minimum value in an array.

Home made Solution-

```
function myArrayMax(arr) {
  let len = arr.length;
  let max = -Infinity;
  while (len--) {
    if (arr[len] > max) {
      max = arr[len];
    }
  }
  return max;
}
```

Sorting Object Arrays

```
const cars = [ {type:"Volvo", year:2016},……….};
cars.sort( function(a, b) { return a.year – b.year} )
```

Sorting string is little complex-

```
cars.sort(function(a, b){
        let x = a.type.toLowerCase();
        let y = a.type.toLowerCase();
        if(x<y) {return -1;}
        if(x>y) {return 1;}
        return 0;
        }
);
```

21. The reverse() method reverses the elements in an array.
    Array_name.reverse
22. **Array Iteration**
    array iteration methods operate on every array item.
    The forEach() method calls a function (a callback function) once for each array element.

    ```
    const numbers = [45, 4, 9, 16, 25];
    numbers.forEach(myFunction);
    function myFunction(value, index, array){
            ……// value is 45, 4 etc.
    }
    ```
23. **Array map**
    The map() method creates a new array by performing a function on each array element.
    The map() method does not change the original array.
    ```
    const numbers1 = [45, 4, 9, 16, 25];
    const numbers2 = numbers1.map(myFunction);
    ```

```
function myFunction(value, index, array) {
  return value * 2;
}
```

24. **JavaScript Array flatMap()**
The flatMap() method first maps all elements of an array and then creates a new array by flattening the array.
```
const myArr = [1, 2, 3, 4, 5,6];
const newArr = myArr.flatMap((x) => x * 2);
document.getElementById("demo").innerHTML = newArr;
// 2, 4, 6, 8, 10, 12
```

25. **JavaScript Array filter()**
The filter() method creates a new array with array elements that pass a test.
```
const numbers = [45, 4, 9, 16, 25];
const over18 = numbers.filter(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

26. **JavaScript Array reduce()**
The reduce() method runs a function on each array element to produce a single value.
The reduce() method works from left-to-right in the array. reduceRight() works from left-to-right in the array.
```
const numbers = [45, 4, 9, 16, 25];
let sum = numbers.reduce(myFunction);

function myFunction(total, value, index, array) {
  return total + value;
}
```

27. JavaScript Array every()
The every() method checks if all array values pass a test. Return true or false.
```
const numbers = [45, 4, 9, 16, 25];
let allOver18 = numbers.every(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

28. **JavaScript Array some()**
The some() method checks if some array value pass a test.
```
const numbers = [45, 4, 9, 16, 25];
let someOver18 = numbers.some(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

29. **JavaScript Array indexOf()**
The indexOf() method searches an array for an element value and returns its position.
```
const fruits = ["Apple", "Orange", "Apple", "Mango"];
let position = fruits.indexOf("Apple") + 1;

array.indexOf(item, start)
```
returns -1 if the item is not found.

30. JavaScript Array lastIndexOf()
Array.lastIndexOf() is the same as Array.indexOf(), but returns the position of the last occurrence of the specified element.
```
Array.lastIndexOf(item, start)
```

31. **JavaScript Array find()**
The find() method returns the value of the first array element that passes a test function.
```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.find(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

32. **JavaScript Array findIndex()**
The findIndex() method returns the index of the first array element that passes a test function.
```
const numbers = [4, 9, 16, 25, 29];
let first = numbers.findIndex(myFunction);

function myFunction(value, index, array) {
  return value > 18;
}
```

33. **JavaScript Array.from()**
The Array.from() method returns an Array object from any object with a length property or any iterable object.
```
Array.from("ABCDEFG");
```

34. **JavaScript Array keys()**
The Array.keys() method returns an Array Iterator object with the keys of an array.
```
const fruits = ["Banana", "Orange", "Apple", "Mango"];
const keys = fruits.keys();

for (let x of keys) {
  text += x + "<br>";
}
```

35. **JavaScript Array entries()**
entries() returns an Array Iterator object with key/value pairs:
```
const f = fruits.entries();
for (let x of f){
……
Text = Text + x + <br>; // x is array
}
```

The entries() method returns an Array Iterator object with key/value pairs:

[0, "Banana"]
[1, "Orange"]
[2, "Apple"]
[3, "Mango"]

The entries() method does not change the original array.

36. **JavaScript Array includes()**
    Check if the fruit array contains "Mango":
    fruits.includes("Mango");
37. **JavaScript Array spread()**
    The "spread" operator spreads elements of iterable objects:

```
const q1 = ["Jan", "Feb", "Mar"];
const q2 = ["Apr", "May", "Jun"];
const q3 = ["Jul", "Aug", "Sep"];
const q4 = ["Oct", "Nov", "May"];

const year = [...q1, ...q2, ...q3, ...q4];

//year = Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,May
```

Note - An array declared with const must be initialized when it is declared.
Arrays declared with var can be initialized at any time.

Chapter – 19 JavaScript Date Object

const d = new Date(); // create a date object with the current date and time.
or, const d = new Date("2023-05-22"); // creates a date object from a date string.
By default, JavaScript will use the browser's time zone to display a date as a full text string:
Mon May 22 2023 20:59:28 GMT+0530 (India Standard Time)

const d = new Date(2018, 11, 24, 10, 33, 30, 0);
// year, month, day, hours, minutes, seconds, ms
JavaScript counts months from 0 to 11
Specifying a day higher than max, will not result in an error but add the overflow to the next month.
If you supply only one parameter it will be treated as milliseconds.
One and two digit years will be interpreted as 19xx.
Zero time is January 01, 1970 00:00:00 UTC

The **toDateString()** method converts a date to a more readable format
const d = new Date();
d.toDateString();
The toUTCString() method converts a date to a string using the UTC standard.
toISOString()

ISO dates can be written with added hours, minutes, and seconds (YYYY-MM-DDTHH:MM:SSZ):
const d = new Date("2015-03-25T12:00:00Z");

Date and time is separated with a capital T.
UTC time is defined with a capital letter Z.
If you want to modify the time relative to UTC, remove the Z and add +HH:MM or -HH:MM instead:

new Date("2015-03-25T12:00:00+00:00");
Wed Mar 25 2015 17:30:00 GMT+0530 (India Standard Time)

new Date("2015-03-25T12:00:00+06:00");
Wed Mar 25 2015 11:30:00 GMT+0530 (India Standard Time)

new Date("2015-03-25T12:00:00-06:00");
Wed Mar 25 2015 23:30:00 GMT+0530 (India Standard Time)
Date.parse("March 21, 2012) returns the number of milliseconds between the date and January 1, 1970.

JavaScript Get Date Methods

1. getFullYear() - xxxx
2. getMonth() – 0-11
3. getDate() – 1- 31
4. getDay() – weekend as a number
5. getHours()
6. getMinutes()
7. getSeconds()
8. getMilliseconds()
9. getTime()

The get methods return information from existing date objects.In a date object, the time is static. The "clock" is not "running".The time in a date object is NOT the same as current time.
Date.now() returns the number of milliseconds since January 1, 1970.

UTC Date Methods

getUTCDate(), getUTCFullYear()…etc
The difference between Local time and UTC time can be up to 24 hours.
The getTimezoneOffset() method returns the difference (in minutes) between local time an UTC time:
let diff = d.getTimezoneOffset();

JavaScript Set Date Methods

Set Date methods let you set date values (years, months, days, hours, minutes, seconds, milliseconds) for a Date Object.
setDate(), setFullYear(), setHours()…etc.

Chapter- 20 JavaScript Math Object

Unlike other object, The Math object has no constructor.

The Math object is static.

All methods and properties can be used without creating a Math object first.

Syntax – Math.property

e.g. E, PI, SQRT2, SQRT1_2, LN2, LN10, LOG2E, LOG10E

Syntax – Math.method(number)

round, ceil, floor, traunc, sign, pow(x, y), sqrt, abs,
 sin(Angle in radians = Angle in degrees x PI / 180.),
cos, min(x1, x2, x3,..), max(x1, x2, x3…), random(), log, log2, log10, ….

Math.random() returns a number between 0 (inclusive), and 1 (exclusive).

```
// Returns a random integer from 0 to 9:
Math.floor(Math.random() * 10);

function getRndInteger(min, max) {
  return Math.floor(Math.random() * (max - min) ) + min;
}
```

chapter – 21 The Boolean Function

Boolean() function is used to find out if an expression(or a variable) is true.

Boolean(10>9)

Or 10 > 9

Every things are except 0, -0, “”, undefined, null, false, NaN

Let x = false // type Boolean

Let y = new Boolean(false) // type object

The optional Chaining Operator(?.)

The ?. operator returns undefined if an object is undefined or null(instead throwing error)

Conditional Statement

```
if(condition){
   // block of code
}
else{
   // block of code
}
else if(condition){
   //block of code
}
Switch(expression) {
   case x:
      // code block
      break;
   case y:
      // code block;
      break;
   default:
      // code block;
}
```

In this example case 4 and 5 share the same code block, and 0 and 6 share another code block:

```
switch (new Date().getDay()) {
  case 4:
  case 5:
    text = "Soon it is Weekend";
    break;
  case 0:
  case 6:
    text = "It is Weekend";
    break;
  default:
    text = "Looking forward to the Weekend";
}
```

Switch cases uses strict comparison (===)

The value must be of the same type to match

Chapter – 22 Loop

1. **For loop**

```
for( let I = 0; i< cars.length; i++)  {
    text += cars[i] + “<br>”;
}
```

Remember scope of var and let in a loop

1.a **for/in loop**

```
 For(key in object) {
    // code block
}
```

Loops through the properties of the object

e.g. const person = { fname : “John”, lname : “Doe”……};

```
let text = “”;
for ( let x in person) {
    text += person[x]
}
// output – John Doe 25
```

Note - Do not use for in over an Array if the index order is important.

1.b **for/of loop**

Loops through the value of an iterable objects like string, array, Maps, NodeLists and more

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

For every iteration the value of the next property is assigned to the variable. Variable can be declared with const, let or var.

```
const cars = ["BMW", "Volvo", "Mini"];
let text = "";
for (let x of cars) {
  text += x;
}
// output – BMW Volvo Mini
```

2. **While loop**

```
While (condition) {
    // code block}
```

3. **Do while loop**

```
do {
   // code block
}while ( condition);
```

The **break** statement “jumps out” of a loop.
The **continue** statement “jump over” one iteration in the loop.

JavaScript labels

label:
statements

The break and the continue statements are the only JavaScript statements that can "jump out of" a code block.
The continue statement (with or without a label reference) can only be used to skip one loop iteration.

The break statement, without a label reference, can only be used to jump out of a loop or a switch.
With a label reference, the break statement can be used to jump out of any code block
e.g.

```
const cars = ["BMW", "Volvo", "Saab", "Ford"];
list: {
  text += cars[0] + "<br>";
  text += cars[1] + "<br>";
  break list;
  text += cars[2] + "<br>";
  text += cars[3] + "<br>";
}
// text = BMW Volvo
```

Chapter – 23 JavaScript Sets

Set is a collection of unique values.
Each value can only occurs once in a Set

Set Methods

new Set(), add(), delete(), has(), forEach(), values(), set.size

e.g.1 const letters = new Set([“a”, “b”, “c”]);

```
or, const letters = new Set();
letters.add(“a”);
letters.add(“b”);
letters.add(“c”);
```

```
const letters = new Set();
// Create Variables
const a = "a";
const b = "b";
const c = "c";
// Add Variables to the Set
letters.add(a);
letters.add(b);
```

The values() method returns a new iterator object containing all the values in a set.
e.g.

```
let text = "";
for (const x of letters.values()) {
  text += x + “<br>”;
}
```

Set Properties

letters.size use to get the size of set.

Chapter – 24 JavaScript Maps

Map holds key-value pairs where the keys can be any datatype.
Map remembers the original insertion order of the keys.

Map methods

new Map(), set(), get(), delete(), has(), forEach(), entries(), map.size

Map Properties

Map.size method used to get the size of map.

How to create a Map

1. Passing an Array to new Map()

```
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

2. Create a Map and use Map.set()

```
// Create a Map
const fruits = new Map();
// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);
```

The set() method can also be use to change existing map value.
The get() method gets the value of a key in a Map.
Fruits.get(“apples”);
The delete() method removes a Map element.
Fruits.delete(“apples”);
The has() method returns true if a key exists in a Map.
fruits.has(“apples”);

Object vs Map

Object are not directly iterable and Maps are directly iterable.
Object do not have a size property and Map has a size property.
In Object keys must be String (or Symbols) and Maps key can be any datatype.
In Object keys are not well ordered and In Map keys are ordered by insertion.

Objects have default keys and Maps do not have default keys.

The forEach() Method

```
let text = "";
fruits.forEach (function(value, key) {
  text += key + ' = ' + value;
})
```

The **entries()** method returns an iterator object with the [key, values] in a Map:

```
let text = "";
for (const x of fruits.entries()) {
  text += x;
}
// text = apples, 500
         Bananas, 300
         Oranges, 200
```

Chapter – 25 JavaScript typeof

In JavaScript there are 5 different data types that can contain values:

string, number, boolean, object, function

There are 6 types of objects:

Object, Date, Array, String, Number, Boolean

And 2 data types that cannot contain values:

null, undefined

```
typeof "John"   // Returns "string"
```

- The data type of NaN is number
- The data type of an array is object
- The data type of a date is object
- The data type of null is object
- The data type of an undefined variable is undefined *
- The data type of a variable that has not been assigned a value is also undefined *

The instanceof Operator

The instanceof operator returns true if an object is an instance of the specified object:

```
const cars = ["Saab", "Volvo", "BMW"];

(cars instanceof Array);   // true
(cars instanceof Object); // true
(cars instanceof String);  // false
(cars instanceof Number);  // false
```

The void operator

The void operator evaluates an expression and returns undefined.

```
<a href="javascript:void(0);">
  Useless link
</a>
```

```
<a
href="javascript:void(document.body.style.backgroundColor='red');">
  Click me to change the background color of body to red
</a>
```

Chapter- 26 JavaScript Type Conversion

- Converting Strings to Numbers

  The global method Number() converts a variable (or a value) into a number.
  A numeric string (like "3.14") converts to a number (like 3.14).
  An empty string (like "") converts to 0.
  A non numeric string (like "John") converts to NaN (Not a Number).
  If the variable cannot be converted, it will still become a number, but with the value NaN.
  The unary + operator

  ```
  let y = "5";      // y is a string
  let x = + y;      // x is a number (NaN)
  ```

- Converting Numbers to, Dates, Booleans to String

  The global method String() can convert numbers to strings, dates to string and Booleans to strings.
  It can be used on any type of numbers, literals, variables, or expressions:

  ```
  String(x)       // returns a string from a number variable x
  String(123)  // returns a string from a number literal 123
  String(100 + 23)  // returns a string from a number from an expression
  ```

  The Number method toString() does the same.

  ```
  x.toString()
  (123).toString()
  (100 + 23).toString()
  ```

  More Methods

  toExponential()   Returns a string, with a number rounded and written using exponential notation.
  toFixed()   Returns a string, with a number rounded and written with a specified number of decimals.
  toPrecision()   Returns a string, with a number written with a specified length

- Converting Dates to Numbers

  The global method Number() can be used to convert dates to numbers.

  ```
  d = new Date();
  Number(d)        // returns 1404568027739
  ```

  The date method getTime() does the same.

  ```
  d = new Date();
  d.getTime()       // returns 1404568027739
  Date().toString()  // returns "Thu Jul 17 2014 15:38:19 GMT+0200 (W. Europe Daylight Time)"
  ```

- Converting Booleans to Numbers

  The global method Number() can also convert booleans to numbers.

  ```
  Number(false)     // returns 0
  Number(true)      // returns 1
  ```

Automatic String Conversion

JavaScript automatically calls the variable's toString() function when you try to "output" an object or a variable:
document.getElementById("demo").innerHTML = myVar;

// if myVar = {name:"Fjohn"} // toString converts to "[object Object]"
// if myVar = [1,2,3,4] // toString converts to "1,2,3,4"
// if myVar = new Date() // toString converts to "Fri Jul 18 2014 09:08:55 GMT+0200"

Chapter – 27 Bitwise Operator

&, |, ^, ~, << (zero fill left shift), >> Signed right shift (Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off), >>> (Zero fill right shift)

JavaScript stores numbers as 64 bits floating point numbers, but all bitwise operations are performed on 32 bits binary numbers.
Before a bitwise operation is performed, JavaScript converts numbers to 32 bits signed integers.S
After the bitwise operation is performed, the result is converted back to 64 bits JavaScript numbers.

Conversion of decimal to binary and Binary to decimal

```
 function dec2bin(dec){
   return (dec >>> 0).toString(2);
}

function bin2dec(bin){
  return parseInt(bin, 2).toString(10);
}
```

Chapter – 28 Regular Expression

A regular expression is a sequence of characters that forms a search pattern.
The search pattern can be used for text search and text replace operations.
Syntax - /pattern/modifiers;
E.g., /w3school/i
i is a modifier (modifies the search to be case-insensetive)
Regular expressions are often used with the two string methods: **search()** and **replace()**.
The search() method uses an expression to search for a match, and returns the position of the match.

The replace() method returns a modified string where the pattern is replaced.
e.g.,
let text = "Visit W3Schools!";
let n = text.search("W3Schools"); // n = 6

let text = "Visit Microsoft!";
let result = text.replace("Microsoft", "W3Schools");
// result = visit Microsoft

Modifiers: i(case-insensetive**), g(**global search**), m (**multiple matching**)**
Brackets are used to find a range of characters:

- **[abc]** find any of the characters between the brackets

e.g. let text = "Is this all there is?";
let result = text.match(/[h]/g); // result = h, h

- **[0-9]** Find any of the digits between the brackets

e.g. let text = "123456789";
let result = text.match(/[1-4]/g); // result = 1, 2, 3, 4

- **(x|y)** Find any of the alternatives separated with |

e.g. let text = "re, green, red, green, gren, gr, blue, yellow";
let result = text.match(/(red|green)/g);
// result = green,red, green

Metacharacter are characters with a special meaning:

- **\d** Find a digit
  let text = "Give 100%!";
  let result = text.match(/\d/g); // result = 1, 0, 0
- **\s** Find a whitespace character
  let text = "Give 100%!";
  let result = text.match(/\d/g); // result = , , ,
- **\b** Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b let text = "HELLO, LOOK AT YOU!";
  let result = text.search(/\bLO/); // result = 7
  let text = "HELLO, LOOK AT YOU!";
  let result = text.search(/LO\b/); // result = 3
- **\uxxxx** Find the Unicode character specified by the hexadecimal number xxxx
  let text = "Visit W3Schools. Hello World!";
  let result = text.match(/\u0057/g); // result= W,W

Quantifiers define quantities:

- **n+** Matches any string that contains at least one n

let text = "Hellooo World! Hello W3Schools!";
let result = text.match(/o+/g); // result = ooo, o, o, oo

- **n*** Matches any string that contains zero or more occurrences of n

let text = "Hellooo World! Hello W3Schools!";
let result = text.match(/lo*/g); // result = l, looo, l, l, lo, l

- **n?** Matches any string that contains zero or one occurrences of n

let text = "1, 100 or 1000?";
let result = text.match(/10?/g); // result =1, 10,10

The **test()** method is a RegExp expression method.
It searches a string for a pattern, and returns true or false.
e.g. <p id="p01">The best things in life are free!</p>
let text = document.getElementById("p01").innerHTML;
const pattern = /best/;
document.getElementById("demo").innerHTML=
pattern.test(text); // result = true

The **exec()** method is a RegExp expression method
It searches a string for a specified pattern, and returns the found text as an object.
If no match is found, it returns an empty (null) object.
e.g.

```
const obj = /e/.exec("The best things in life are free!");
let result = "Found " + obj[0] + " in position " + obj.index +
" in the text: " + obj.input;
//  result = Found e in position 2 in the text: The best things
in life are free!
```

Chapter – 29 Operator Precedence

val	Op.	Desc.	val	Op.	Desc.
18	()	Exp. grouping	13	**	Exponential
17	.	Member of	12	*	Multiplication
17	[]	Member of	12	/	Division
17	?.	Optional Chaining	12	%	Remainder
17	()	Function call	11	+	Addition
17	new	With arg	11	-	Substraction
16	new	Without Arg	11	+	Concatination
15	++	postfix	10		Shift Operator
15	--	postfix	9	Ins.of in	Relational Operator
14	++	Prefix	9		Comparison Operator
14	--	Prefix	8	==, ===, !=, !==	Equal Operator
14	!	Logical Not	7	&	BitwiseAND
14	~	Bitwise Not	6	^	Bitwise XOR
14	+	Unary plus	5	|	Bitwise OR
14	-	Unary minus	4	&&	Logical AND
14	typeof	Data type	3	||, ??	
14	void	Evaluate Void	2		Assignment Operator
14	delete	Property Delete	1	,	Comma

Chapter – 30 Errors Handling

Throw, and Try...Catch...Finally

The **try** statement defines a code block to run (to try).

The **catch** statement defines a code block to handle any error.
The **finally** statement defines a code block to run regardless of the result.
The **throw** statement defines a custom error.
you can throw an exception (throw an error).
The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big";    // throw a text
throw 500;          // throw a number

try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch
result
}
```

When an error occurs, JavaScript will actually create an Error object with two properties: name and message.

Input Validation Example

```
try {
   if(x.trim() == "") throw "empty";
   if(isNaN(x)) throw "not a number";
   x = Number(x);
   if(x < 5) throw "too low";
   if(x > 10) throw "too high";
  }
  catch(err) {
   message.innerHTML = "Input is " + err;
  }
}
```

Chapter – 31 Hoisting

In JavaScript, a variable can be declared after it has been used.
Hoisting is JavaScript's default behaviour of moving all declarations to the top of the current scope (to the top of the current script or the current function).

Variables defined with let and const are hoisted to the top of the block, but not initialized.
Meaning: The block of code is aware of the variable, but it cannot be used until it has been declared.

Using a let variable before it is declared will result in a ReferenceError.
Using a const variable before it is declared, is a syntax error, so the code will simply not run.

JavaScript only hoists declarations, not initializations.

Chapter – 32 JavaScript Use Strict

The purpose of "use strict" is to indicate that the code should be executed in "strict mode".

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string

e.g.

```
"use strict";
 x = 3.14;  // This will cause an error because x is not declared
```

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

```
x = 3.14;      // This will not cause an error.
myFunction();

function myFunction() {
  "use strict";
  y = 3.14;   // This will cause an error
}
```

1. **Using an object, without declaring it, is not allowed:**

```
"use strict";
x = {p1:10, p2:20};     // This will cause an error
```

2. **Deleting a variable (or object) is not allowed.**

```
"use strict";
let x = 3.14;
delete x;
```

3. **Deleting a function is not allowed.**

```
"use strict";
function x(p1, p2) {};
delete x;              // This will cause an error
```

4. **Duplicating a parameter name is not allowed:**

```
"use strict";
function x(p1, p1) {};   // This will cause an error
```

5. **Octal numeric literals are not allowed:**

```
"use strict";
let x = 010;           // This will cause an error
```

6. **Octal escape characters are not allowed:**

```
"use strict";
let x = "\010";          // This will cause an error
```

7. **Writing to a read-only property is not allowed:**

```
"use strict";
const obj = {};
Object.defineProperty(obj, "x", {value:0, writable:false});

obj.x = 3.14;          // This will cause an error
```

8. **Writing to a get-only property is not allowed:**

```
"use strict";
const obj = {get x() {return 0} };

obj.x = 3.14;          // This will cause an error
```

9. **Deleting an undeletable property is not allowed:**

```
"use strict";
delete Object.prototype; // This will cause an error
```

10. **The word eval cannot be used as a variable:**

```
"use strict";
let eval = 3.14;        // This will cause an error
```

11. **The word arguments cannot be used as a variable:**

```
"use strict";
let arguments = 3.14;    // This will cause an error
```

12. **The with statement is not allowed**

```
"use strict";
with (Math){x = cos(2)}; // This will cause an error
```

13. **For security reasons, eval() is not allowed to create variables in the scope from which it was called.**

```
"use strict";
eval ("x = 2");
alert (x);     // This will cause an error

"use strict";
eval ("var x = 2");
alert (x);    // This will cause an error

eval ("let x = 2");
alert (x);       // This will cause an error
```

14. The this keyword in functions behaves differently in strict mode.
    The this keyword refers to the object that called the function.
    If the object is not specified, functions in strict mode will return undefined and functions in normal mode will return the global object (window):

Chapter – 32 this keyword

The this keyword refers to different objects depending on how it is used:

In an object method, this refers to the object.

1. In an event, this refers to the element that received the event.
2. Methods like call(), apply(), and bind() can refer this to any object.

3. **this** in a method

```
fullName : function() {
  return this.firstName + " " + this.lastName;
}
```

4. When used alone, this refers to the global object.
    Because this is running in the global scope.
   In a browser window the global object is [object Window]

```
    let x = this;
```

5. In strict mode, when used alone, this also refers to the global object:

```
    "use strict";
    let x = this;
```

6. In a function, the global object is the default binding for this.

7. JavaScript strict mode does not allow default binding.
   So, when used in a function, in strict mode, this is undefined.

8. Object Method Binding

```
const person = {
  firstName  : "John",
  lastName   : "Doe",
  id         : 5566,
  myFunction : function() {
    return this;
  }
};
```

// this is the person object

9. Explicit Function Binding

The call() and apply() methods are both used to call an object method with another object as argument.

9.1 call()

```
const person1 = {
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
}

const person2 = {
  firstName:"John",
  lastName: "Doe",
}

// Return "John Doe":
let x = person1.fullName.call(person2); // x is variable
```

9.2 apply()

The apply() method is similar to the call() method
The difference is:
The call() method takes arguments separately.
The apply() method takes arguments as an array.

```
const person = {
  fullName: function(city, country) {
    return this.firstName + " " + this.lastName + "," + city +
"," + country;
  }
}

const person1 = {
  firstName:"John",
  lastName: "Doe"
}
let x = person.fullName.apply(person1, [‘Raniganj’, ‘India’]
// x is a variable
```

9.3 bind()

With the bind() method, an object can borrow a method from another object.

```
const person = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}

const member = {
  firstName:"Hege",
  lastName: "Nilsen",
}

let fullName = person.fullName.bind(member);
// fullName is a function name should be same
```

this Precedence
To determine which object this refers to; use the following precedence of order.

1	bind()
2	apply() and call()
3	Object method
4	Global scope

Chapter – 33 Arrow Function

Arrow functions allows us to write shorter function syntax:

```
let myFunction = (a, b) => a*b;
```

Before Arrow

```
hello = function() {
  return "Hello World!";
}
```

With Arrow Function

```
hello = () => {
  return "Hello World!";
}
```

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

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

hello = (val) => "Hello " + val; (With Parameter)
hello = val => "Hello " + val; (Without Parameter)

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

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

Chapter – 34 Class

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

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

Example

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
}
```

It is a **template** for JavaScript objects.
Creating Objects:

```
const myCar1 = new Car("Ford", 2014);
const myCar2 = new Car("Audi", 2019);
```

The constructor method is a special method:

- It has to have the exact name "constructor"
- It is executed automatically when a new object is created
- It is used to initialize object properties

Class Methods

```
class ClassName {
  constructor() { ... }
  method_1() { ... }
  method_2() { ... }
  method_3() { ... }
}
```

The syntax in classes must be written in "strict mode".
You will get an error if you do not follow the "strict mode" rules.

Chapter – 35 Modules

JavaScript modules allow you to break us your code into separate files.
Modules are imported from external files with the **import** statement.
Modules also rely on type="module" in the <script> tag.

```
<script type="module">
import message from "./message.js";
</script>
```

Export

Modules with functions or variables can be stored in any external file.
There are two types of exports: **Named Exports** and **Default Exports.**

Named Exports

Let us create a file named person.js, and fill it with the things we want to export.
You can create named exports two ways. In-line individually, or all at once at the bottom.

In-line individually:

```
export const name = "Jesse";
export const age = 40;
```

All at once at the bottom:

```
const  name = "Jesse";
const age = 40;
export {name, age};
```

Default Export

```
const message = () => {
const name = "Jesse";
const age = 40;
return name + ' is ' + age + 'years old.';
};

export default message;
```

Import

You can import modules into a file in two ways, based on if they are named exports or default exports

Import from named exports

```
import { name, age } from "./person.js";
```

Imports from default exports

```
import message from "./message.js";
```

Chapter – 36 JSON

JSON is a format for storing and transporting data.
JSON is often used when data is sent from a server to a web page.

- JSON stands for JavaScript Object Notation
- JSON is a lightweight data interchange format
- JSON is language independent *

- JSON is "self-describing" and easy to understand

* The JSON syntax is derived from JavaScript object notation syntax, but the JSON format is text only. Code for reading and generating JSON data can be written in any programming language.

Example - employees object: an array of 3 employee records (objects):

```
{
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
}
```

JSON Data

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
"firstName":"John"
```

JSON names require double quotes. JavaScript names do not.

JSON Objects

JSON objects are written inside curly braces.

Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

JSON Array

JSON arrays are written inside square brackets.

Just like in JavaScript, an array can contain objects:

```
"employees":[
  {"firstName":"John", "lastName":"Doe"},
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]
```

Converting a JSON Text to a JavaScript Object

A common use of JSON is to read data from a web server, and display the data in a web page.

For simplicity, this can be demonstrated using a string as input.

First, create a JavaScript string containing JSON syntax:

```
let text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

Then, use the JavaScript built-in function JSON.parse() to convert the string into a JavaScript object:

```
const obj = JSON.parse(text);
```

now, use the object

```
document.getElementById("demo").innerHTML =
obj.employees[1].firstName + " " +
obj.employees[1].lastName;
```

Chapter – 37 Programming paradigms

Like we speak in different styles programming can also be done using different styles.

In programming there are two common programming paradigm:

1. **Functional Programming (FP)**
   In functional programming there is a clear distinction between function and data. In functional programming data can exist outside the function.
2. **Object Oriented Programming (OOP)**
   Here, both data and functions are combined to form objects

Many functions return undefined by default like console.log()

Features of JavaScript -

First-class functions

It means that a function in JavaScript is just another value that we can:

- pass to other functions
- save in a variable
- return from other functions

Higher-order functions

A higher-order function is a function that has either one or both of the following characteristics:

- It accepts other functions as arguments
- It returns functions when invoked

Example -

```
// Task 1: Build a function-based console log message generator
function consoleStyler(color, background, fontSize, txt) {
  var message = "%c" + txt;
  var style = `color: ${color};`
  style += `background: ${background};`
  style += `font-size: ${fontSize};`
  console.log(message, style)
}

// Task 2: Build another console log message generator
function celebrateStyler(reason) {
  var fontStyle = "color: tomato; font-size: 50px"
  if (reason == "birthday") {
    console.log(`%cHappy birthday`, fontStyle)
  } else if (reason == "champions") {
    console.log(`%cCongrats on the title!`, fontStyle)
  } else {
    console.log(message, style)
  }

}

// Task 3: Run both the consoleStyler and the celebrateStyler functions
consoleStyler('#1d5c63', '#ede6db', '40px', 'Congrats!')
celebrateStyler('birthday')


// Task 4: Insert a congratulatory and custom message
function styleAndCelebrate(color, background, fontSize, txt, reason) {
  consoleStyler(color, background, fontSize, txt)
  celebrateStyler(reason)

}
// Call styleAndCelebrate

styleAndCelebrate('#1d5c63', '#ede6db', '40px', 'Congrats!', 'champions')
```

The Benefit of OOP

OOP help developers to mimic the relationship between objects in the real world. OOP is an effective approach come up with solutions in the code you write. OOP also:

- Allows you to write modular code
- Make your code more flexible and
- Make your code reusable

```
class Animal { /* ...class code here... */ }
var myDog = Object.create(Animal)
console.log (Animal) // [class Animal]
```

A more common method of creating objects from classes is to use **new** keyword.

```
class Animal { /* ...class code here... */ }
var myDog = new Animal()
console.log (Animal)
```

OOP Principles:

1. **Inheritance -** There is a base class of a "thing".
   There is one or more sub-classes of "things" that inherit the properties of the base class (sometimes also referred to as the "super-class").
   There might be some other sub-sub-classes of "things" that inherit from those classes in point 2.
   To setup the inheritance relation between classes in JavaScript, I can use the **extends** keyword, as in class B extends A.
   ```
   class Animal { /* ...class code here... */ }
   class Bird extends Animal { /* ...class code here... */
   }
   class Eagle extends Bird { /* ...class code here... */ }
   ```
2. **Encapsulation**
   In the simplest terms, encapsulation has to do with making a code implementation "hidden" from other users, in the sense that they don't have to know how my code works in order to "consume" the code.

3. **Abstraction**
   Abstraction is all about writing code in a way that will make it more generalized.
   - An abstraction is about extracting the concept of what you're trying to do, rather than dealing with a specific manifestation of that concept.
   - Encapsulation is about you not having access to, or not being concerned with, how some implementation works internally.
4. **Polymorphism**
   Polymorphism is a word derived from the Greek language meaning "multiple forms". An alternative translation might be: "something that can take on many shapes".
   e.g.
   the concatenation operator, used by calling the built-in concat() method.
   If I use the concat() method on two strings, it behaves exactly the same as if I used the + operator.
   `"abc".concat("def"); // 'abcdef'`
   I can also use the concat() method on two arrays. Here's the result:
   `["abc"].concat(["def"]); // ['abc', 'def']`
   Consider using the + operator on two arrays with one member each:
   `["abc"] + ["def"]; // ["abcdef"]`
   This means that the concat() method is exhibiting polymorphic behavior since it behaves differently based on the context - in this case, based on what data types I give it.
   e.g.
   ```
   class Bird {
     useWings() {
       console.log("Flying!")
     }
   }
   class Eagle extends Bird {
     useWings() {
       super.useWings()
       console.log("Barely flapping!")
     }
   }
   class Penguin extends Bird {
     useWings() {
       console.log("Diving!")
     }
   }
   var baldEagle = new Eagle();
   ```

```
        var kingPenguin = new Penguin();
        baldEagle.useWings(); // "Flying! Barely flapping!"
        kingPenguin.useWings(); // "Diving!"
```

Constructor

Constructor functions, commonly referred to as just "constructors", are special functions that allow us to build instances of these built-in native objects. All the constructors are capitalized.

To use a constructor function, I must prepend it with the operator new.

For example, to create a new instance of the Date object, I can run: new Date(). What I get back is the current datetime, such as:

```
Thu Feb 03 2022 11:24:08 GMT+0100 (Central European Standard Time)
```

However, not all the built-in objects come with a constructor function. An example of such an object type is the built-in Math object.

Running new Math() throws an Uncaught TypeError, informing us that Math is not a constructor.

The most common use case of new is to use it with one of the built-in object types.

Note that using constructor functions on all built-in objects is sometimes not the best approach.

This is especially true for object constructors of primitive types, namely: String, Number, and Boolean.

For example, using the built-in String constructor, I can build new strings:

```
let apple = new String("apple");
apple; // --> String {'apple'}

let pear = "pear";
pear; // --> "pear"
```

The pear variable, being a primitive value, will always be more performant than the apple variable, which is an object.

Besides being more performant, due to the fact that each object in JavaScript is unique, you can't compare a String object with another String object, even when their values are identical.

In other words, if you compare new String('plum') === new String('plum'), you'll get back false, while "plum" === "plum" returns true. You're getting the false when comparing objects because it is not the values that you pass to the constructor that are being compared, but rather the memory location where objects are saved.

Alternatively, you can use a pattern literal instead of RegExp. Here's an example of using /d/ as a pattern literal, passed-in as an argument to the match method on a string.

```
"abcd".match(/e/); // null
"abcd".match(/a/); // ['a', index: 0, input: 'abcd', groups: undefined]
```

In JavaScript, the **prototype** is an object that can hold properties to be shared by multiple other objects. And this is the basis of how inheritance works in JavaScript. This is why it's sometimes said that JavaScript implements a prototype of inheritance model.

Example

```
var birds = {
  hasWings : true,
  canFly : true,
  hasFeathers : true
};
var eagle1 = Object.create(bird)
console.log(eagle1); // {}
console.log(eagle1.hasWings, eagle1.canFly,
eagle1.hasFeathers)  // true true true
```

I can build as many objects as I want and they will all have the bird objects as their prototype.

```
var penguin = Object.create(bird);
penguin.canFly = false;
console.log(penguin);  // {canFly : false}
console.log(penguin.canFly);  //  false
console.log(penguin.hasFeathers);  //  true
```

canFly property is changed in penguin object only it doesn't has any effect on the prototype object.

JavaScript Class (chapter – 34)

```
class Train {
  constructor(color, lightsOn) {
    this.color = color;
    this.lightsOn = lightsOn;
  }
  toggleLights() {
    this.lightsOn = !this.lightsOn;
  }
  lightsStatus() {
    console.log('Lights on?', this.lightsOn);
  }
  getSelf() {
    console.log(this);
  }
  getPrototype() {
    var proto = Object.getPrototypeOf(this);
    console.log(proto);
  }
}

class HighSpeedTrain extends Train {
  constructor(passengers, highSpeedOn, color, lightsOn) {
    super(color, lightsOn);
    this.passengers = passengers;
    this.highSpeedOn = highSpeedOn;
  }
}
```

In JavaScript classes, super is used to specify what property gets inherited from the super-class in the sub-class.
In this case, I choose to inherit both the properties from the Train super-class in the HighSpeedTrain sub-class.
These properties are color and lightsOn.

Using class instance as another class' constructor's property

```
class StationaryBike {
  constructor(position, gears) {
    this.position = position
    this.gears = gears
  }
}

class Treadmill {
  constructor(position, modes) {
    this.position = position
    this.modes = modes
  }
}

class Gym {
  constructor(openHrs, stationaryBikePos, treadmillPos) {
    this.openHrs = openHrs
    this.stationaryBike = new
StationaryBike(stationaryBikePos, 8)
    this.treadmill = new Treadmill(treadmillPos, 5)
  }
}

var boxingGym = new Gym("7-22", "right corner", "left
corner")

console.log(boxingGym.openHrs) //
console.log(boxingGym.stationaryBike) //
console.log(boxingGym.treadmill) //
```

programming an OO program

```
// Task 1: Code a Person class
class Person{
  constructor(name="Tom", age=20, energy=100) {
    this.name = name;
    this.age = age;
    this.energy = energy;
  }
  sleep() {
    this.energy += 10;
  }
  doSomethingFun() {
    this.energy -= 10;
  }
}
```

```
// Task 2: Code a Worker class
class Worker extends Person{
  constructor(name = "Tom", age = 20, energy = 100, xp =
0, hourlyWage = 10) {
    super(name, age, energy)
    this.xp = xp;
    this.hourlyWage = hourlyWage;
  }

  doSomethingFun(){
    super.doSomethingFun;
  }

  sleep() {
    super.sleep;
  }

  goToWork() {
    this.xp += 10;
  }
}


// Task 3: Code an intern object, run methods
function intern() {
  let intern = new Worker();
  intern.name = "Bob";
  intern.age = 21
  intern.energy = 110
  intern.xp = 0
  intern.hourlyWage = 10
  intern.goToWork()
  return (intern);
}

// Task 4: Code a manager object, methods
function manager() {
  let manager = new Worker("Alice", 30, 120, 100, 30);
  manager.doSomethingFun();
  return manager;

}

console.log(intern())
console.log(manager())
```

Getter and Setter in Class

```
class Car {
  constructor(brand) {
    this._carname = brand;
  }
  get carname() {
    return this._carname;
  }
  set carname(x) {
    this._carname = x;
  }
}

const myCar = new Car("Ford");
myCar.carname = ‘Toyota’;
document.getElementById("demo").innerHTML =
myCar.carname; // return Toyota
```

Unlike functions, and other JavaScript decleration, class declerations are not **hoisted**.

Static Methods

Static class methods are defined on the class itself.

You cannot call a static method on an object, only on an object class.

```
class Car {
  constructor(name) {
    this.name = name;
  }
  static hello() {
    return "Hello!!";
  }
}

const myCar = new Car("Ford");

// You can call 'hello()' on the Car Class:
document.getElementById("demo").innerHTML =
Car.hello();
// But NOT on a Car Object:
// document.getElementById("demo").innerHTML =
myCar.hello();
// this will raise an error.

class Car {
  constructor(name) {
    this.name = name;
  }
  static hello(x) {
    return "Hello " + x.name;
  }
}
const myCar = new Car("Ford");
document.getElementById("demo").innerHTML =
Car.hello(myCar);
```

Chapter – 38 JavaScript Promises

A promise is a JavaScript object that link “Producing Code” and “Consuming Code”.

“Producing Code” can take some time and “Consuming Code” must wait for the result.

```
const myPromise = new Promise(function(resolve, reject) {
// "Producing Code" (May take some time)

  resolve(); // when successful
  reject();  // when error
});
// "Consuming Code" (Must wait for a fulfilled Promise)
myPromise.then(
  function(value) { /* code if successful */ },
  function(error) { /* code if some error */ }
);
```

Then method takes two arguments. a callback for success and another for failure

When the producing code obtains the result, it should call one of the two callbacks:

Result	Call
Success	resolve(result value)
Error	reject(error object)

A JavaScript Promise object can be:

- Pending
- Fulfilled
- Rejected

The Promise object supports two properties: **state** and **result**.

myPromise.state	myPromise.result
"pending"	undefined
"fulfilled"	a result value
"rejected"	an error object

You cannot access the Promise properties state and result.

You must use a Promise method to handle promises.

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

let myPromise = new Promise(function(resolve, reject) {
  let x = 0;
// The producing code (this may take some time)
  if (x == 0) {
    resolve("OK");
  } else {
    reject("Error");
  }
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

Waiting for a timeout

```
setTimeout(function() { myFunction("I love You !!!"); }, 3000);

function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
}
```

Example Using Promise

```
let myPromise = new Promise(function(resolve, reject) {
  setTimeout(function() { resolve("I love You !!"); }, 3000);
});

myPromise.then(function(value) {
  document.getElementById("demo").innerHTML = value;
});
```

Waiting for a file

Example using Callback

```
function getFile(myCallback) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.html");
  req.onload = function() {
    if (req.status == 200) {
      myCallback(req.responseText);
    } else {
      myCallback("Error: " + req.status);
    }
  }
  req.send();
}

getFile(myDisplayer);
```

Example using Promise

```
let myPromise = new Promise(function(resolve, reject) {
  let req = new XMLHttpRequest();
  req.open('GET', "mycar.htm");
  req.onload = function() {
    if (req.status == 200) {
      resolve(req.response);
    } else {
      reject("File not Found");
    }
  };
  req.send();
});

myPromise.then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);
```

The Symbol Type

A JavaScript Symbol is a primitive datatype just like Number, String, or Boolean.

It represents a unique "hidden" identifier that no other code can accidentally access.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};

let id = Symbol('id');
person[id] = 140353;
// Now person[id] = 140353
// but person.id is still undefined
```

Symbols are always unique.

If you create two symbols with the same description they will have different values:

Symbol("id") == Symbol("id"); // false

Default Parameter Values

```
function myFunction(x, y = 10) {
  // y is 10 if not passed or undefined
  return x + y;
}
myFunction(5); // will return 15
```

Function Rest Parameter

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

```
function sum(initial, ...args) {
  let sum = initial;
  for (let arg of args) sum += arg;
  return sum;
}
```

Rest parameter must be the last parameter in function definition

```
let x = sum(0, 4, 9, 16, 25, 29, 100, 66, 77);
```

Chapter- 39 JavaScript Objects

All JavaScript values, except primitives are objects.

Object values are written as name : value pairs (name and value separated by a colon).

const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

There are different ways to create new objects:

- **Create a single object, using an object literal.**
  const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};
  or,
  const person = {};
  person.firstName = "John";..........

- **Create a single object, with the keyword new.**
  const person = new Object();
  person.firstName = "John";
  ……….
- Define an object constructor, and then create objects of the constructed type.
- Create an object using Object.create().

Objects are **mutable**: They are addressed by reference, not by value.
const x = person; // Will not create a copy of person.
The object x is not a copy of person. It is person. Both x and person are the same object.
Any changes to x will also change person, because x and person are the same object.

Accessing JavaScript Properties
objectName.property   // person.age
or,
objectName["property"]   // person["age"]
or,
objectName[expression]   // x = "age"; person[x]

for in loop
```
const person = {
  fname:" John",
  lname:" Doe",
  age: 25
};

for (let x in person) {
  txt += person[x];
}
```

The **delete** keyword deletes a property from an object:
delete person.age;
or, delete person["age"];

Nested Objects
Values in an object can be another object:
```
myObj = {
  name:"John",
  age:30,
  cars: { car1:"Ford", car2:"BMW", car3:"Fiat" }
}
```
You can access nested objects using the dot notation or the bracket notation:
myObj.cars.car2;
or, myObj.cars["car2"];
or, myObj["cars"]["car2"];
Nested Arrays and Objects
Values in objects can be arrays, and values in arrays can be objects:

```
const myObj = {
  name: "John",
  age: 30,
  cars: [
    {name:"Ford", models:["Fiesta", "Focus", "Mustang"]},
    {name:"BMW", models:["320", "X3", "X5"]},
    {name:"Fiat", models:["500", "Panda"]}
  ]
}
```
To access arrays inside arrays, use a for-in loop for each array:
```
for (let i in myObj.cars) {
  x += "<h1>" + myObj.cars[i].name + "</h1>";
  for (let j in myObj.cars[i].models) {
    x += myObj.cars[i].models[j];
  }
}
```

Property Attributes
All properties have a name. In addition they also have a value.
The value is one of the property's attributes.
Other attributes are: **enumerable**, **configurable**, and **writable**.
These attributes define how the property can be accessed (is it readable?, is it writable?)
Object.defineProperty() is a new Object method in ES5.
```
var person = {
  firstName: "John",
  lastName : "Doe",
  language : "NO",
};

// Change a Property:
Object.defineProperty(person, "language", {
  value: "EN",
  writable : true,
  enumerable : true, // used to hide from enumeration
  configurable : true
});
```
It lets you define an object property and/or change a property's value and/or metadata.
This example creates a setter and a getter to secure upper case updates of language:

```
Object.defineProperty(person, "language", {
  get : function() { return language },
  set : function(value) { language = value.toUpperCase()}
});

// Change Language
person.language = "en";
```

Protyoype properties
JavaScript objects inherit the properties of their prototype.

The delete keyword does not delete inherited properties, but if you delete a prototype property, it will affect all objects inherited from the prototype.

JavaScript Object Methods

A JavaScript method is a property containing a function definition.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  id: 5566,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

this keyword refers to an object.
this is not a variable. It is a keyword. You cannot change the value of this.

You access an object method with the following syntax:

```
objectName.methodName()
person.name = function () {
  return this.firstName + " " + this.lastName;
};
```

Some common solutions to display JavaScript objects are:

- Displaying the Object Properties by name
- Displaying the Object Properties in a Loop
  ```
  for (let x in person) {
  txt += person[x] + " ";
  };
  ```
  You must use **person[x]** in the loop.
  person.x will not work (Because **x** is a variable).

- Displaying the Object using Object.values()
  const myArray = Object.values(person);
  myArray is now a JavaScript array, ready to be displayed.
  // myArray = ["John", 30, "New York"]
- Displaying the Object using JSON.stringify()
  let myString = JSON.stringify(person);
  myString is now a JavaScript string, ready to be displayed.
  // {"name":"John","age":30,"city":"New York"}

Stringify Functions

JSON.stringify will not stringify functions:

```
const person = {
  name: "John",
  age: function () {return 30;}
};

let myString = JSON.stringify(person);
document.getElementById("demo").innerHTML = myString;
// {"name":"John"}
```

This can be fixed using:

```
person.age = person.age.toString();

let myString = JSON.stringify(person);
document.getElementById("demo").innerHTML = myString;
// {"name":"John","age":"function () {return 30;}"}
```

Arrays can be stringfy too.

Object Accessors

1. Getter (The get keyword)
   This example uses a lang property to get the value of the language property.

```
// Create an object:
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "en",
  get lang() {
    return this.language;
  }
};

// Display data from the object using a getter:
document.getElementById("demo").innerHTML=
person.lang;
```

2. Setter (The set Keyword)
   This example uses a lang property to set the value of the langage property.

```
const person = {
  firstName: "John",
  lastName: "Doe",
  language: "",
  set lang(lang) {
    this.language = lang;
  }
};

// Set an object property using a setter:
person.lang = "en";
```

What is the difference between function and getter ?

Getter provides a simpler syntax.

```
person.fullName()  // declared using function keyword
person.fullName    // declared using get keyword
```

Why Using Getters and Setters?

It gives simpler syntax
It allows equal syntax for properties and methods
It can secure better data quality
It is useful for doing things behind-the-scenes

Object.defineProperty()

A counter example

```
// Define object
const obj = {counter : 0};

// Define setters and getters
Object.defineProperty(obj, "reset", {
  get : function () {this.counter = 0;}
});
Object.defineProperty(obj, "increment", {
  get : function () {this.counter++;}
});
Object.defineProperty(obj, "decrement", {
  get : function () {this.counter--;}
});
Object.defineProperty(obj, "add", {
  set : function (value) {this.counter += value;}
});
Object.defineProperty(obj, "subtract", {
  set : function (value) {this.counter -= value;}
});

// Play with the counter:
obj.reset;
obj.add = 5;
obj.subtract = 1;
obj.increment;
obj.decrement;
```

Constructors

Constructor function for person objects:

```
function Person( first, last, age, eye) {
   this.firstName = first;
   this.lastName = last;
   this.age = age;
   this.eyeColor = eye;
}
const myFather = new Person("John", "Doe", 50, "blue");
```

In a constructor function this does not have a value. It is a substitute for the new object. The value of this will become the new object when a new object is created.

You cannot add a new property to an object constructor the same way you add a new property to an existing object.

Prototype Inheritance

All JavaScript object inherit properties and methods from a prototype:

- Date objects inherit from Date.prototype
- Array objects inherit from Array.prototype
- Person objects inherit from Person.prototype

The Object.prototype is on the top of the prototype inheritance chain:

Date objects, Array objects, and Person objects inherit from Object.prototype.

Using the **prototype** Property

The prototype property allows you to add new properties to object constructors:

```
function Person(first, last, age, eyecolor) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eyecolor;
}

Person.prototype.nationality = "English";
```

Note – We can access objectName.nationality which gives “English” and we cannot explicitly change this property when creating new objects.

The JavaScript prototype property also allows you to add new methods to objects constructors:

```
Person.prototype.name = function() {
  return this.firstName + " " + this.lastName;
};
```

A **JavaScript iterable** is an object that has a **Symbol.iterator**

The Symbol.iterator is a function that returns a next() function.

Chapter – 40 JavaScript Sets

A JavaScript Set is a collection of unique values.

A Set can hold any value of any data type.

Set Methods

1.	new Set()	Creates a new Set
2.	add()	Adds a new element to the Set
3.	delete()	Removes an element from a Set
4.	has()	Returns true if a value exists
5.	clear()	Removes all elements from a Set
6.	forEach()	Invokes a callback for each element
7.	values()	Returns an Iterator with all the values in a Set
8.	keys()	Same as values()
9.	entries()	Returns an Iterator with the [value,value] pairs from a Set

Set Property

Size   returns the number elements in a Set

Examples

1.   // Create a Set

```
const letters = new Set(["a","b","c"]);
```

2. // Create a Set

```
const letters = new Set();

// Add Values to the Set
letters.add("a");
letters.add("b");
letters.add("c");
```

3. // Create Variables

```
const a = "a";
const b = "b";
const c = "c";

// Create a Set
const letters = new Set();

// Add Variables to the Set
letters.add(a);
letters.add(b);
letters.add(c);
```

4. forEach() Method

```
letters.forEach (function(value) {
  text += value;
})
```

5. // Create an Iterator

```
const myIterator = letters.values();

// List all Values
let text = "";
for (const entry of myIterator) {
  text += entry;
}
```

6. A Set has no keys.
keys() returns the same as values().
This makes Sets compatible with Maps.

7. A Set has no keys.
entries() returns [value,value] pairs instead of [key,value] pairs.
This makes Sets compatible with Maps:
8. typeof set is object

Chapter – 41 JavaScript Maps

A Map holds key-value pairs where the keys can be any datatype.
A Map remembers the original insertion order of the keys.
A Map has a property that represents the size of the map.

Map Method

1.	new Map()	Creates a new Map object
2.	set()	Sets the value for a key in a Map
3.	get()	Gets the value for a key in a Map
4.	clear()	Removes all the elements from a Map
5.	delete()	Removes a Map element specified by a key
6.	has()	Returns true if a key exists in a Map
7.	forEach()	Invokes a callback for each key/value pair in a Map
8.	entries()	Returns an iterator object with the [key, value] pairs in a Map
9.	keys()	Returns an iterator object with the keys in a Map
10.	values()	Returns an iterator object of the values in a Map

Property

size	Returns the number of Map elements

You can create a JavaScript Map by:

1. Passing an Array to new Map()

```
// Create a Map
const fruits = new Map([
  ["apples", 500],
  ["bananas", 300],
  ["oranges", 200]
]);
```

2. Create a Map and use Map.set()

```
// Create a Map
const fruits = new Map();

// Set Map Values
fruits.set("apples", 500);
fruits.set("bananas", 300);
fruits.set("oranges", 200);
```

The set() method can also be used to change existing Map values:
fruits.set("apples", 200);

Examples-

1. fruits.get("apples");   // Returns 200
2. fruits.size;
3. fruits.delete("apples");
4. fruits.clear();
5. fruits.has("apples");
6. Maps are object
7. // List all entries
```
let text = "";
fruits.forEach (function(value, key) {
  text += key + ' = ' + value;
})
```
8. // List all entries
```
let text = "";
for (const x of fruits.entries()) {
  text += x;
}
// x = key, value
```
9. // List all keys
```
let text = "";
for (const x of fruits.keys()) {
  text += x;
}
```
10. // List all values
```
let text = "";
for (const x of fruits.values()) {
  text += x;
}
```

Objects as Keys

```
// Create Objects
const apples = {name: 'Apples'};
const bananas = {name: 'Bananas'};
const oranges = {name: 'Oranges'};

// Create a Map
const fruits = new Map();

// Add new Elements to the Map
fruits.set(apples, 500);
fruits.set(bananas, 300);
fruits.set(oranges, 200);
```

JavaScript ES5 object methods

Managing Object

```
// Create object with an existing object as prototype
Object.create()

// Adding or changing an object property
Object.defineProperty(object, property, descriptor)

// Adding or changing object properties
Object.defineProperties(object, descriptors)

// Accessing Properties
Object.getOwnPropertyDescriptor(object, property)

// Returns all properties as an array
Object.getOwnPropertyNames(object)

// Accessing the prototype
Object.getPrototypeOf(object)

// Returns enumerable properties as an array
Object.keys(object)
```

Protecting Objects

```
// Prevents adding properties to an object
Object.preventExtensions(object)

// Returns true if properties can be added to an object
Object.isExtensible(object)

// Prevents changes of object properties (not values)
Object.seal(object)

// Returns true if object is sealed
Object.isSealed(object)

// Prevents any changes to an object
Object.freeze(object)

// Returns true if object is frozen
Object.isFrozen(object)
```

Examples –

Changing a property value

```
const person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};

// Change a property
Object.defineProperty(person, "language", {value : "NO"});
```

Adding a Property

```
// Create an object:
const person = {
  firstName: "John",
  lastName : "Doe",
  language : "EN"
};

// Add a property
Object.defineProperty(person, "year", {value:"2008"});
```

Chapter – 42 JavaScript Functions

JavaScript function declarations are hoisted.

Self Invocation functions

A self-invoking expression is invoked (started) automatically, without being called.
Function expressions will execute automatically if the expression is followed by ().

```
(function () {
  let x = "Hello!!";  // I will invoke myself
})();
```

The arguments.length property returns the number of arguments received when the function was invoked.
The toString() method returns the function as a string.

Arrow function

```
const x = (x, y) => x * y;
```

Arameter Rules

JavaScript function definitions do not specify data types for parameters.
JavaScript functions do not perform type checking on the passed arguments.
JavaScript functions do not check the number of arguments received.

Default Parameters

If a function is called with missing arguments (less than declared), the missing values are set to undefined.

Function Rest Parameter
The rest parameter (...) allows a function to treat an indefinite number of arguments as an array:
Example
```
function sum(...args) {
  let sum = 0;
  for (let arg of args) sum += arg;
  return sum;
}

let x = sum(4, 9, 16, 25, 29, 100, 66, 77);
```

JavaScript arguments are passed by value.
Changes to arguments are not visible (reflected) outside the function.
In JavaScript, object references are values.
If a function changes an object property, it changes the original value.

Max method on array
Math.max(1,2,3); // Will return 3
Since JavaScript arrays do not have a max() method, you can apply the Math.max() method instead.
Math.max.apply(null, [1,2,3]); // Will also return 3

Counter Dilemma
Solution – JavaScript Closure
```
const add = (function () {
  let counter = 0; // this exexutes only one time
  return function () {counter += 1; return counter}
})();

add();
add();
add();
// the counter is now 3
```

This is called a JavaScript closure. It makes it possible for a function to have "private" variables.
The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

De-Structuring arrays and object
let {PI} = Math
PI; // 3.1415
PI === Math.PI; //true
Destructuring is case sensetive.
extract the properties from objects into distinct variables using destructuring.

<u>for – of loops and for – in loop in objects</u>
```
const car = {
  engine: true
}
Const sportsCar = Object.create(car);
sportsCar.speed = “fast”;
console.log(“The sportsCar object: “, sportsCar);
// The sportsCar object : {speed: ‘fast’}

for ( prop in sportsCar) {
  console.log(prop);
}
```
// speed engine (For in loop iterates properties of object and its prototype)
```
for ( prop of Object.keys(sportsCar)){
  console.log(prop + “:” + sportsCar[prop]);
}
```
// speed : fast (whereas for of loop itertes on the propery of object only)

The **rest** operator can be used to destructure existing array items, rather than typing them out again.
```
top7 = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’]
const [] = top7;
const [first, second, third, …secondVisit] = top7;
```

Using Spread and Rest

1. Join arrays, objects using the rest operator
   Example -
   const fruits = ['apple', 'pear', 'plum']
   const berries = ['blueberry', 'strawberry']
   const fruitsAndBerries = [...fruits, ...berries]
   Example –
   const flying = { wings: 2 }
   const car = { wheels: 4 }
   const flyingCar = {...flying, ...car}

2. Add new members to arrays without using the push() method
   Example –
   let veggies = ['onion', 'parsley'];
   veggies = [...veggies, 'carrot', 'beetroot'];
3. Convert a string to an array using the spread operator
   Example –
   const greeting = "Hello";
   const arrayOfChars = [...greeting];
4. Copy either an object or an array into a separate one
   Example -
```
const car1 = {
        speed: 200,
  color: 'yellow'
}
const car 2 = {...car1}
```

Chapter – 43 JavaScript Async

Callback function

```
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

function myCalculator(num1, num2, myCallback) {
  let sum = num1 + num2;
  myCallback(sum);
}

myCalculator(5, 5, myDisplayer);
```

In the example above, myDisplayer is a called a callback function.
It is passed to myCalculator() as an argument.

Asynchronous

Functions running in parallel with other functions are called asynchronous
A good example is JavaScript setTimeout().
When using the JavaScript function **setTimeout()**, you can specify a callback function to be executed on time-out:

```
setTimeout(myFunction, 3000);

function myFunction() {
  document.getElementById("demo").innerHTML = "I love You !!";
}
```

In the example above, myFunction is used as a callback.
myFunction is passed to setTimeout() as an argument.
3000 is the number of milliseconds before time-out, so myFunction() will be called after 3 seconds.

```
setTimeout(function() { myFunction("I love You !!!"); }, 3000);

function myFunction(value) {
  document.getElementById("demo").innerHTML = value;
```

Using **setInterval()** to display the time every second (1000 milliseconds).

```
setInterval(myFunction, 1000);

function myFunction() {
  let d = new Date();
  document.getElementById("demo").innerHTML=
  d.getHours() + ":" +
  d.getMinutes() + ":" +
  d.getSeconds();
}
// This code display a clock
```

async makes a function return a Promise

await makes a function wait for a Promise

Async Syntax

The keyword async before a function makes the function return a promise:

```
async function myFunction() {
  return "Hello";
}
```

Is same as,

```
function myFunction() {
  return Promise.resolve("Hello");
}
```

Example -

```
async function myFunction() {
  return "Hello";
}
myFunction().then(
  function(value) {myDisplayer(value);}
);
```

Await Syntax

The await keyword can only be used inside an async function.
The await keyword makes the function pause the execution and wait for a resolved promise before it continues:

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

Basic Syntax

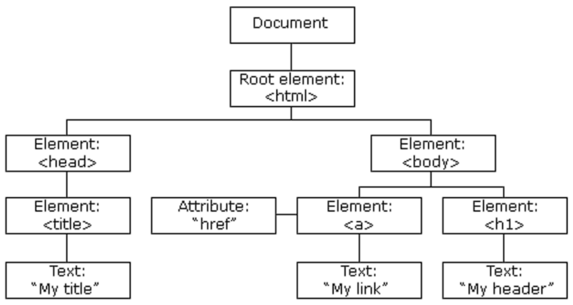
```
async function myDisplay() {
  let myPromise = new Promise(function(resolve, reject) {
    resolve("I love You !!");
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
```

Waiting for a Timeout

```
async function myDisplay() {
  let myPromise = new Promise(function(resolve) {
    setTimeout(function() {resolve("I love You !!");}, 3000);
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
```

Chapter – 44 HTML DOM(Document Object Model)

With the HTML DOM, JavaScript can access and change all the elements of an HTML document.
When a web page is loaded, the browser creates a Document Object Model of the page.
The HTML DOM model is constructed as a tree of Objects:



With the object model, JavaScript gets all the power it needs to create dynamic HTML:

- JavaScript can change all the HTML elements and attributes in the page
- JavaScript can change all the CSS styles in the page
- JavaScript can remove existing HTML elements and attributes
- JavaScript can add new HTML elements and attributes
- JavaScript can react to all existing HTML events in the page
- JavaScript can create new HTML events in the page

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The DOM Programming Interface

In the DOM, all HTML elements are defined as objects.
The programming interface is the properties and methods of each object.
A **property** is a value that you can get or set (like changing the content of an HTML element).
A **method** is an action you can do (like add or deleting an HTML element).

Example

The following example changes the content (the innerHTML) of the <p> element with id="demo":

```
<html>
<body>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

In the example above, getElementById is a method, while innerHTML is a property.

The most common way to access an HTML element is to use the id of the element.

In the example above the getElementById method used id="demo" to find the element.

The easiest way to get the content of an element is by using the innerHTML property.

The innerHTML property is useful for getting or replacing the content of HTML elements.

The HTML DOM document objects

The document object represents your webpage
If you want to access any element in an HTML page, you always start with accessing the document object.

Finding HTML elements

Property
document.getElementBy(id) // Element not Elements
document.getElementsByTagName(name)
document.getElementsByClassName(name)

Changing HTML elements

Property
element.innerHTML = new html content
element.attribute = new value
element.style.property = new style
Method
element.setAttribute(attribute, value)

Adding and Deleting Elements

Method
document.createElement(element) //create an HTML element
document.revoveChild(element)
document.appendChild(element) //Add an HTML element
document.replaceChild(new, old)
document.write(text) // write into HTML output stream

Adding Events Handler

Method
document.getElementById(id).onclick = function(){code}

Finding HTML elements by CSS Selectors

```
const x = document.querySelectorAll("p.intro");
```

Finding HTML elements by HTML Object Collections

```
<form id="frm1" action="/action_page.php">
  First name: <input type="text" name="fname" value="Donald"><br>
  Last name: <input type="text" name="lname" value="Duck"><br><br>
  <input type="submit" value="Submit">
</form>
```

```
<p>These are the values of each element in the form:</p>

<p id="demo"></p>

<script>
const x = document.forms["frm1"];
let text = "";
for (let i = 0; i < x.length ;i++) {
  text += x.elements[i].value + "<br>";
}
document.getElementById("demo").innerHTML = text;
</script>

// Output
Donald
Duck
Submit
```

The following HTML objects (and object collections) are also accessible:

```
document.anchors
document.body
document.documentElement
document.embeds
document.forms
document.head
document.images
document.links
document.scripts
document.title
```

JavaScript Form Validation

```
function validateForm() {
  let x = document.forms["myForm"]["fname"].value;
  if (x == "") {
    alert("Name must be filled out");
    return false;
  }
}
```

The function can be called when the form is submitted:

```
<form name="myForm" action="/action_page.php"
onsubmit="return validateForm()" method="post">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

DOM animation

```
<!DOCTYPE html>
<html>
<style>
#container {
  width: 400px;
  height: 400px;
  position: relative;
  background: yellow;
}
#animate {
  width: 50px;
  height: 50px;
  position: absolute;
  background-color: red;
}
</style>
<body>

<p><button onclick="myMove()">Click Me</button></p>

<div id ="container">
  <div id ="animate"></div>
</div>

<script>
function myMove() {
  let id = null;
  const elem = document.getElementById("animate");
  let pos = 0;
  clearInterval(id);
  id = setInterval(frame, 5);
  function frame() {
    if (pos == 350) {
      clearInterval(id);
    } else {
      pos++;
      elem.style.top = pos + "px";
      elem.style.left = pos + "px";
    }
  }
}
</script>

</body>
</html>
```

JavaScript HTML DOM Events

e.g

```
<!DOCTYPE html>
<html>
<body>
<h1 onclick="changeText(this)">Click on this text!</h1>
<script>
function changeText(id) {
  id.innerHTML = "Ooops!";
}
</script>

</body>
</html>
```

Examples of HTML events

When a user clicks the mouse
When a web page has loaded
When an image has been loaded
When the mouse moves over an element
When an input field is changed
When an HTML form is submitted
When a user strokes a key
Note - Learn various HTML Events

Event Lister

The addEventListener() method

Syntax

element.addEventListener(event, function, useCapture);

Note that you don't use the "on" prefix for the event; use "click" instead of "onclick".

The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

The addEventListener() method attaches an event handler to the specified element.

The addEventListener() method attaches an event handler to an element without overwriting existing event handlers.

- You can add many event handlers to one element.
  example –
  element.addEventListener("click", myFunction);
  element.addEventListener("click", mySecondFunction);

- You can add event listeners to any DOM object not only HTML elements. i.e the window object.
  window.addEventListener("resize", function(){
    document.getElementById("demo").innerHTML = Math.random();
  });

You can easily remove an event listener by using the removeEventListener() method.

When passing parameter values, use an "anonymous function" that calls the specified function with the parameters:
element.addEventListener("click", function(){ myFunction(p1, p2); });

Event Bubbling or Event Capturing

There are two ways of event propagation in the HTML DOM, bubbling and capturing.

Event propagation is a way of defining the element order when an event occurs. If you have a <p> element inside a <div> element, and the user clicks on the <p> element, which element's "click" event should be handled first?

In bubbling the inner most element's event is handled first and then the outer: the <p> element's click event is handled first, then the <div> element's click event.

In capturing the outer most element's event is handled first and then the inner: the <div> element's click event will be handled first, then the <p> element's click event.

The default value is false, which will use the bubbling propagation, when the value is set to true, the event uses the capturing propagation.

```
document.getElementById("myP").addEventListener("click",
myFunction, true);
document.getElementById("myDiv").addEventListener("click",
myFunction, true);
```

removeEventListener()

```
document.getElementById("myDIV").addEventListener("mousemove", myFunction);

function myFunction() {
  document.getElementById("demo").innerHTML = Math.random();
}

function removeHandler() {

document.getElementById("myDIV").removeEventListener("mousemove", myFunction);
}
```

DOM Nodes

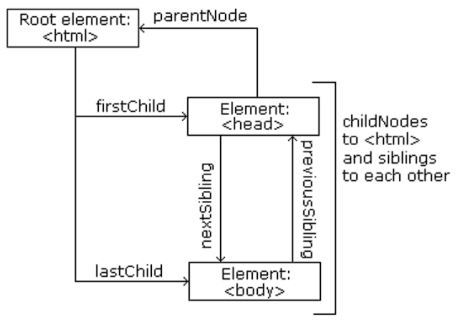
According to the W3C HTML DOM standard, everything in an HTML document is a node:

The entire document is a document node.
Every HTML element is an element node.
The text inside HTML elements are text nodes.
Every HTML attribute is an attribute node (deprecated).
All comments are comment nodes.

Node Relationships

The nodes in the node tree have a hierarchical relationship to each other.
The terms parent, child, and sibling are used to describe the relationships.
In a node tree, the top node is called the root (or root node)
Every node has exactly one parent, except the root (which has no parent)
A node can have a number of children
Siblings (brothers or sisters) are nodes with the same parent

```
<html>

  <head>
    <title>DOM Tutorial</title>
  </head>

  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>

</html>
```



<head> has one child: <title>
<title> has one child (a text node): "DOM Tutorial"

<body> has two children: <h1> and <p>
<h1> has one child: "DOM Lesson one"
<p> has one child: "Hello world!"
<h1> and <p> are siblings

Navigating Between Nodes
You can use the following node properties to navigate between nodes with JavaScript:

- parentNode
- childNodes[nodenumber]
- firstChild
- lastChild
- nextSibling
- previousSibling

Child Nodes and Node Values
```
<title id="demo">DOM Tutorial</title>
```
The element node <title> (in the example above) does not contain text.
It contains a text node with the value "DOM Tutorial".
The value of the text node can be accessed by the node's innerHTML property.
Accessing the innerHTML property is the same as accessing the nodeValue of the first child:
```
myTitle =
document.getElementById("demo").firstChild.nodeValue;
```

Accessing the first child can also be done like this:
```
myTitle =
document.getElementById("demo").childNodes[0].nodeValue;
```

All the (3) following examples retrieves the text of an <h1> element and copies it into a <p> element:
```
<h1 id="id01">My First Page</h1>
<p id="id02"></p>
```

1.
   ```
   <script>
   document.getElementById("id02").innerHTML =
   document.getElementById("id01").innerHTML;
   </script>
   ```

2.
   ```
   <script>
   document.getElementById("id02").innerHTML =
   document.getElementById("id01").firstChild.nodeValue;
   </script>
   ```

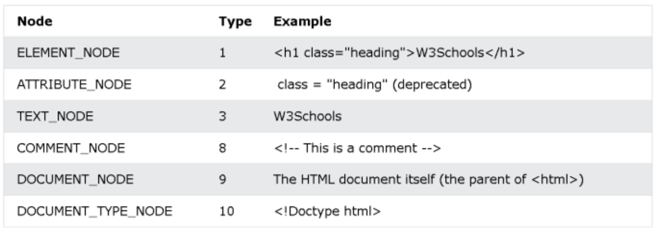
3.
   ```
   <script>
   document.getElementById("id02").innerHTML =
   document.getElementById("id01").childNodes[0].nodeValue;
   </script>
   ```

DOM Root Nodes
There are two special properties that allow access to the full document:
- document.body - The body of the document
- document.documentElement - The full document

The nodeName Property
The nodeName property specifies the name of a node.
nodeName is read-only
nodeName of an element node is the same as the tag name
nodeName of an attribute node is the attribute name
nodeName of a text node is always #text
nodeName of the document node is always #document

The nodeValue Property
The nodeValue property specifies the value of a node.
nodeValue for element nodes is null
nodeValue for text nodes is the text itself
nodeValue for attribute nodes is the attribute value

The nodeType Property
The nodeType property is read only. It returns the type of a node.



Node	Type	Example
ELEMENT_NODE	1	<h1 class="heading">W3Schools</h1>
ATTRIBUTE_NODE	2	class = "heading" (deprecated)
TEXT_NODE	3	W3Schools
COMMENT_NODE	8	<!-- This is a comment -->
DOCUMENT_NODE	9	The HTML document itself (the parent of <html>)
DOCUMENT_TYPE_NODE	10	<!Doctype html>

Creating New HTML Elements (Nodes)
To add a new element to the HTML DOM, you must create the element (element node) first, and then append it to an existing element.
```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const element = document.getElementById("div1");
element.appendChild(para);
</script>
```

insertBefore()
```
<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const element = document.getElementById("div1");
const child = document.getElementById("p1");
element.insertBefore(para, child);
</script>
// between div1 and p1
```

Removing Existing HTML Elements
```
<script>
const elmnt = document.getElementById("p1"); elmnt.remove();
</script>
```

Removing a Child Node
For browsers that does not support the remove() method, you have to find the parent node to remove an element:
```
<div id="div1">
  <p id="p1">This is a paragraph.</p>
```

```
  <p id="p2">This is another paragraph.</p>
</div>

<script>
const parent = document.getElementById("div1");
const child = document.getElementById("p1");
parent.removeChild(child);
</script>
```

or,

```
const child = document.getElementById("p1");
child.parentNode.removeChild(child);
```

Replacing HTML Elements

```
<script>
const para = document.createElement("p");
const node = document.createTextNode("This is new.");
para.appendChild(node);

const parent = document.getElementById("div1");
const child = document.getElementById("p1");
parent.replaceChild(para, child);
</script>
```

The HTMLCollection Object

The getElementsByTagName() method returns an HTMLCollection object.
An HTMLCollection object is an array-like list (collection) of HTML elements.

```
<!DOCTYPE html>
<html>
<body>

<h2>JavaScript HTML DOM</h2>

<p>Hello World!</p>

<p>Hello Norway!</p>

<p id="demo"></p>    //Hello Norway!

<script>
const myCollection = document.getElementsByTagName("p");

document.getElementById("demo").innerHTML = "The innerHTML of the second paragraph is: " + myCollection[1].innerHTML;

</script>

</body>
</html>
```

The length property defines the number of elements in an HTMLCollection:
myCollection.length

The HTML DOM NodeList Object

A NodeList object is a list (collection) of nodes extracted from a document.

A NodeList object is almost the same as an HTMLCollection object.

Some (older) browsers return a NodeList object instead of an HTMLCollection for methods like getElementsByClassName().

All browsers return a NodeList object for the property childNodes.

Most browsers return a NodeList object for the method querySelectorAll().

The following code selects all <p> nodes in a document:

```
<script>
const myNodelist = document.querySelectorAll("p");

document.getElementById("demo").innerHTML = "The innerHTML of the second paragraph is: " + myNodelist[0].innerHTML;

</script>
// Hello world
```

HTML DOM Node List Length

The length property defines the number of nodes in a node list:
myNodelist.length

The Difference Between an HTMLCollection and a NodeList

A NodeList and an HTMLcollection is very much the same thing.

Both are array-like collections (lists) of nodes (elements) extracted from a document. The nodes can be accessed by index numbers. The index starts at 0.

Both have a length property that returns the number of elements in the list (collection).

An HTMLCollection is a collection of document elements.

A NodeList is a collection of document nodes (element nodes, attribute nodes, and text nodes).

HTMLCollection items can be accessed by their name, id, or index number.

NodeList items can only be accessed by their index number.

An HTMLCollection is always a live collection. Example: If you add a <li> element to a list in the DOM, the list in the HTMLCollection will also change.

A NodeList is most often a static collection. Example: If you add a <li> element to a list in the DOM, the list in NodeList will not change.

The getElementsByClassName() and getElementsByTagName() methods return a live HTMLCollection.

The querySelectorAll() method returns a static NodeList.

The childNodes property returns a live NodeList.