JS Output

1. **Using innerHTML**
   To access an HTML element, JavaScript can use the document.getElementById(id) method. The id attribute defines the HTML element. The innerHTML property defines the HTML content. Changing the innerHTML property of an HTML element is a common way to display data in HTML.

   document.getElementById("demo").innerHTML = 5 + 6;

2. **Using document.write()**
   For testing purposes, it is convenient to use document.write(). Using document.write() after an HTML document is loaded, will delete all existing HTML.The document.write() method should only be used for testing.

   <button type="button" onclick="document.write(5 + 6)">Try it</button>

3. **Using window.alert()**
   You can use an alert box to display data.

   window.alert(5 + 6);

   You can skip the window keyword. In JavaScript, the window object is the global scope object. This means that variables, properties, and methods by default belong to the window object. This also means that specifying the window keyword is optional.

   alert(5 + 6);

4. **Using console.log()**

   console.log(5 + 6);

5. **JavaScript Print**

   JavaScript does not have any print object or print methods. You cannot access output devices from JavaScript. The only exception is that you can call the window.print() method in the browser to print the content of the current window.

   <button onclick="window.print()">Print this page</button>

JS Statements

JavaScript Keywords

var	: Declares a variable
let	: Declares a block variable
const	: Declares a block constant
if	: Marks a block of statements to be executed on a condition
switch	: Marks a block of statements to be executed in different cases
for	: Marks a block of statements to be executed in a loop
function	: Declares a function
return	: Exits a function
try	: Implements error handling to a block of statements

JS Variables

JavaScript Variables can be declared in 4 ways.

- Automatically
- Using var
- Using let
- Using const : These are constant values and cannot be changed.

When to Use var, let, or const?

1. Always declare variables
2. Always use const if the value should not be changed
3. Always use const if the type should not be changed (Arrays and Objects)
4. Only use let if you can't use const
5. Only use var if you MUST support old browsers.

Re-Declaring JavaScript Variables

- If you re-declare a JavaScript variable declared with var, it will not lose its value. The variable carName will still have the value "Volvo" after the execution of these statements.

```
var carName = "Volvo";
var carName;
```

- You cannot re-declare a variable declared with let or const. This will not work.

```
let carName = "Volvo";
let carName;
```

<u>JavaScript Underscore (_)</u>

Using the underscore is not very common in JavaScript, but a convention among professional programmers is to use it as an alias for "private (hidden)" variables.

```
let _lastName = "Johnson";
```

<u>JS Let</u>

- Variables declared with let have Block Scope
- Variables declared with let must be Declared before use
- Variables declared with let cannot be Redeclared in the same scope

<u>Block Scope</u>

- Before ES6 (2015), JavaScript did not have Block Scope.
- JavaScript had Global Scope and Function Scope.
- ES6 introduced the two new JavaScript keywords: let and const.
- These two keywords provided Block Scope in JavaScript.

Example :
Variables declared inside a { } block cannot be accessed from outside the block:

```
{
  let x = 2;
}
// x can NOT be used here
```

<u>Cannot be Redeclared</u>

Variables defined with let can not be redeclared.
You can not accidentally redeclare a variable declared with let.
With let you can not do this:

```
let x = "John Doe";
let x = 0;
```

Redeclaring a variable with let, in another block, IS allowed:

```
let x = 2;   // Allowed
{
let x = 3;   // Allowed
}
```

```
{
let x = 4;    // Allowed
}
```

JS Var

Global Scope

- Variables declared with the var always have Global Scope.
- Variables declared with the var keyword can NOT have block scope.

Example

Variables declared with var inside a { } block can be accessed from outside the block:

```
{
  var x = 2;
}
// x CAN be used here
```

Redeclaring Variables

- Redeclaring a variable using the var keyword can impose problems.
- Redeclaring a variable inside a block will also redeclare the variable outside the block.

Example

```
var x = 10;
// Here x is 10
{
var x = 2;
// Here x is 2
}
// Here x is 2
```

➔ Variables defined with var are hoisted to the top and can be initialized at any time.

➔ **Meaning:** You can use the variable before it is declared:

Example

This is OK:

```
carName = "Volvo";
var carName;
```

JS Const

- The const keyword was introduced in ES6 (2015)
- Variables defined with const cannot be Redeclared
- Variables defined with const cannot be Reassigned
- JavaScript const variables must be assigned a value when they are declared:
- Variables defined with const have Block Scope

When to use JavaScript const?

- Always declare a variable with const when you know that the value should not be changed.
- Use const when you declare:

  A new Array
  A new Object
  A new Function
  A new RegExp

Constant Objects and Arrays

The keyword const is a little misleading. It does not define a constant value. It defines a constant reference to a value. Because of this you can NOT:

- Reassign a constant value
- Reassign a constant array
- Reassign a constant object

But you CAN:

- Change the elements of constant array
- Change the properties of constant object

Difference Between var, let and const

	Scope	Redeclare	Reassign	Hoisted	Binds this
var	No	Yes	Yes	Yes	Yes
let	Yes	No	Yes	No	No
const	Yes	No	No	No	No

What is Good?

- let and const have block scope.
- let and const can not be redeclared.
- let and const must be declared before use.
- let and const does not bind to this.

- let and const are not hoisted.

What is Not Good?

- var does not have to be declared.
- var is hoisted.
- var binds to this.

Browser Support

The let and const keywords are not supported in Internet Explorer 11 or earlier.

JS Operators

Adding Strings and Numbers

Adding two numbers, will return the sum, but adding a number and a string will return a string:

let z = "Hello" + 5; => ***Hello5***

JavaScript Type Operators

Operator	Description
typeof	Returns the type of a variable
instanceof	Returns true if an object is an instance of an object type

JavaScript Bitwise Operators

Bit operators work on 32 bits numbers. Any numeric operand in the operation is converted into a 32 bit number. The result is converted back to a JavaScript number.

Operator	Description	Example	Same as	Result	Decimal
&	AND	5 & 1	0101 & 0001	0001	1
|	OR	5 | 1	0101 | 0001	0101	5
~	NOT	~ 5	~0101	1010	10
^	XOR	5 ^ 1	0101 ^ 0001	0100	4
<<	left shift	5 << 1	0101 << 1	1010	10
>>	right shift	5 >> 1	0101 >> 1	0010	2
>>>	unsigned right shift	5 >>> 1	0101 >>> 1	0010	2

JS Data Types

JavaScript has 8 Data Types

- String
- Number
- Bigint
- Boolean
- Undefined
- Null
- Symbol
- Object

Note:

Most programming languages have many number types:

- Whole numbers (integers): byte (8-bit), short (16-bit), int (32-bit), long (64-bit)
- Real numbers (floating-point): float (32-bit), double (64-bit)

Javascript numbers are always one type: double (64-bit floating point).
All JavaScript numbers are stored in a 64-bit floating-point format.

JS Functions

```
function name(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

Local Variables

Variables declared within a JavaScript function, become LOCAL to the function. Local variables can only be accessed from within the function.

```
// code here can NOT use carName
function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}
// code here can NOT use carName
```

JS Objects

How to Define a JavaScript Object

- Using an Object Literal
- Using the new Keyword
- Using an Object Constructor

1. **JavaScript Object Literal**
   An object literal is a list of key:value pairs inside curly braces {}.
   Ex : ***{firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"}***

2. **Creating a JavaScript Object**
   ```
   // Create an Object
   const person = {};

   // Add Properties
   person.firstName = "John";
   person.lastName = "Doe";
   person.age = 50;
   person.eyeColor = "blue";
   ```

3. **Using the new Keyword**
   ```
   // Create an Object
   const person = new Object();

   // Add Properties
   person.firstName = "John";
   person.lastName = "Doe";
   person.age = 50;
   person.eyeColor = "blue";
   ```

Accessing Object Properties
1. objectName.propertyName
2. objectName["propertyName"]

JS Object Properties

Deleting Properties

The delete keyword deletes a property from an object:
Ex : ***delete person.age;***

JS Object Methods

Adding a Method to an Object

Ex : ***person.name = function () {***
return this.firstName + " " + this.lastName;
};

JS Object Display

- **Object.values() creates an array from the property values:**

```
// Create an Array
const myArray = Object.values(person);
```

- **Object.entries() makes it simple to use objects in loops:**

```
const fruits = {Bananas:300, Oranges:200, Apples:500};

let text = "";
for (let [fruit, value] of Object.entries(fruits)) {
  text += fruit + ": " + value + "<br>";
```

- **JavaScript objects can be converted to a string with JSON method JSON.stringify().**

JSON.stringify() is included in JavaScript and supported in all major browsers.
The result will be a string written in JSON notation:
Ex : ***{"name":"John","age":50,"city":"New York"}***

let myString = JSON.stringify(person);

JS Object Constructors

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

Adding a Property to a Constructor

You can NOT add a new property to an object constructor. To add a new property, you must add it to the constructor function prototype:

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

Adding a Method to a Constructor

You cannot add a new method to an object constructor function. The code will produce a TypeError:
Adding a new method must be done to the constructor function prototype:

```
Person.prototype.changeName = function (name) {
  this.lastName = name;
}
```

JS Events

HTML events are "things" that happen to HTML elements. When JavaScript is used in HTML pages, JavaScript can "react" to these events.

HTML Events

An HTML event can be something the browser does, or something a user does.

Ex : An HTML web page has finished loading
An HTML input field was changed
An HTML button was clicked

```
<button onclick="document.getElementById('demo').innerHTML = Date()">The time is?</button>
```

Common HTML Events

Event	Description
onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element

onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

JS Strings

Template Strings

Templates are strings enclosed in backticks (`This is a template string`).

JS String Methods

String Length

let text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
let length = text.length;

Extracting String Characters

- The charAt() method returns the character at a specified index (position) in a string. If no character is found, this returns an empty string.
- The charCodeAt() method returns the code of the character at a specified index in a string. The method returns a UTF-16 code (an integer between 0 and 65535).
- JavaScript String at() method returns the character at the given position. This allows the use of negative indexes while charAt() do not.
- Property Access [ ]. This is read only. str[0] = "A" gives no error (but does not work!). If no character is found, [ ] returns undefined.

let char = text.charAt(0);
let char = text.charCodeAt(0);
let letter = name.at(2);
let char = text[0];

JS String Search

- The indexOf() method returns the index (position) of the first occurrence of a string in a string, or it returns -1 if the string is not found.

- The lastIndexOf() method returns the index of the last occurrence of a specified text in a string. return -1 if the text is not found.
  Both methods accept a second parameter as the starting position for the search.
  let index = text.indexOf("locate", 15);
  meaning: if the second parameter is 15, the search starts at position 15, and searches to the beginning of the string.
- The search() method searches a string for a string (or a regular expression) and returns the position of the match.
- The match() method returns an array containing the results of matching a string against a string (or a regular expression).
  Perform a global search for "ain": ***text.match(/ain/g);***
  Perform a global, case-insensitive search for "ain": ***text.match(/ain/gi);***
  If a regular expression does not include the g modifier (global search), match() will return only the first match in the string.
- The includes() method returns true if a string contains a specified value. Otherwise it returns false.
  Check if a string includes "world". Start at position 12: ***text.includes("world", 12);***
- The startsWith() method returns true if a string begins with a specified value. A start position for the search can be specified:
  text.startsWith("world", 5)
- The endsWith() method returns true if a string ends with a specified value.

JS String Templates/ Template Strings/Template Literals

Template Strings use back-ticks (``)

- **Quotes Inside Strings**
  let text = `He's often called "Johnny"`;
- **Multiline Strings**
  let text =
  `The quick
  brown fox
  jumps over
  the lazy dog`;
- **Variable Substitutions**
  let text = `Welcome ${firstName}, ${lastName}!`;
- **Expression Substitution**
  let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;

JS Numbers

- JavaScript has only one type of number. JavaScript Numbers are Always 64-bit Floating Points.
- The maximum number of decimals is 17.
- NaN is a JavaScript reserved word indicating that a number is not a legal number. NaN is a number. typeof NaN returns number.
- You can use the global JavaScript function isNaN() to find out if a value is a not a number:
- JavaScript interprets numeric constants as hexadecimal if they are preceded by 0x.

JS BigInt

JavaScript BigInt variables are used to store big integer
JavaScript integers are only accurate up to 15 digits:

To create a BigInt, append n to the end of an integer or call BigInt():
let y = 9999999999999999n;
let y = BigInt(1234567890123456789012345)

Arithmetic between a BigInt and a Number is not allowed (type conversion lose information).

JS Number Methods

- The toString() method returns a number as a string.
- toExponential() returns a string, with a number rounded and written using exponential notation.
- toFixed() returns a string, with the number written with a specified number of decimals.
- toPrecision() returns a string, with a number written with a specified length.
- valueOf() returns a number as a number.

```
let x = 123;
x.toString();
x.toExponential(2);
x.toFixed(0);
x.toPrecision(2);
x.valueOf();
```

Converting Variables to Numbers

- The Number() method can be used to convert JavaScript variables to numbers.
- parseInt() parses a string and returns a whole number. Spaces are allowed. Only the first number is returned.
- parseFloat() parses a string and returns a number. Spaces are allowed. Only the first number is returned.

Number Object Methods

- **Number.isInteger()** : Returns true if the argument is an integer

  Number.isInteger(10);
- **Number.isSafeInteger()** : Returns true if the argument is a safe integer
- **Number.parseFloat() :** Converts a string to a number
- **Number.parseInt()** : Converts a string to a whole number

JS Number Properties

- Number.EPSILON is the difference between the smallest floating point number greater than 1 and 1.
- Number.MAX_VALUE is a constant representing the largest possible number in JavaScript.
- Number.MIN_VALUE is a constant representing the lowest possible number in JavaScript.
- Number.MAX_SAFE_INTEGER represents the maximum safe integer in JavaScript. Number.MAX_SAFE_INTEGER is (253 - 1).
- Number.MIN_SAFE_INTEGER represents the minimum safe integer in JavaScript. Number.MIN_SAFE_INTEGER is -(253 - 1).
- NaN is a JavaScript reserved word for a number that is not a legal number.

  let x = Number.NaN;

JS Arrays

- Arrays are a special type of object. The typeof operator in JavaScript returns "object" for arrays.
- You can have variables of different types in the same Array.

Ex:

1. ***const cars = ["Saab", "Volvo", "BMW"];***
2. ***const cars = [];***
   cars[0]= "Saab";

cars[1]= "Volvo";

3. ***const cars = new Array("Saab", "Volvo", "BMW");***
4. *// Create an array with 40 undefined elements*
   const points = new Array(40);

<u>Converting an Array to a String</u>

The JavaScript method toString() converts an array to a string of (comma separated) array values.

const fruits = ["Banana", "Orange", "Apple", "Mango"];
document.getElementById("demo").innerHTML = fruits.toString();

Result:
Banana,Orange,Apple,Mango

<u>Array Properties and Methods</u>

- The length property of an array returns the length of an array (the number of array elements).
- cars.sort() // Sorts the array
- The easiest way to add a new element to an array is using the push() method.
- Array.isArray()
- The at() method returns an indexed element from an array : ***fruits.at(2);***
- The join() method also joins all array elements into a string : ***fruits.join(" * ");***
- The pop() method removes the last element from an array.
- The push() method adds a new element to an array (at the end).
- The shift() method removes the first array element and "shifts" all other elements to a lower index.
- The unshift() method adds a new element to an array (at the beginning), and "unshifts" older elements.
- The concat() method creates a new array by merging (concatenating) existing arrays.
  const myChildren = myGirls.concat(myBoys);
  const myChildren = arr1.concat(arr2, arr3);
  const myChildren = arr1.concat("Peter");
- The copyWithin() method copies array elements to another position in an array. The copyWithin() method overwrites the existing values. The copyWithin() method does not add items to the array. The copyWithin() method does not change the length of the array.
  fruits.copyWithin(2, 0); : Copy to index 2, all elements from index 0.
  fruits.copyWithin(2, 0, 2); : Copy to index 2, the elements from index 0 to 2.
- The flat() method creates a new array with sub-array elements concatenated to a specified depth.
- The flatMap() method first maps all elements of an array and then creates a new array by flattening the array.
  const newArr = myArr.flatMap(x => [x, x * 10]);

- The splice() method can be used to add new items to an array.

The Difference Between Arrays and Objects

- In JavaScript, arrays use numbered indexes.
- In JavaScript, objects use named indexes.

JS Array Search

- The indexOf() method searches an array for an element value and returns its position.
- Array.lastIndexOf() returns the position of the last occurrence of the specified element.
- Array.includes() allows us to check if an element is present in an array.
- The find() method returns the value of the first array element that passes a test function.
- The findIndex() method returns the index of the first array element that passes a test function.
- The findLast() method that will start from the end of an array and return the value of the first element that satisfies a condition.
  ```
  const temp = [27, 28, 30, 40, 42, 35, 30];
  let high = temp.findLast(x => x > 40);
  ```
- The findLastIndex() method finds the index of the last element that satisfies a condition.

JS Array Sort

- The sort() method sorts an array alphabetically : ***fruits.sort();***
  This alters the original array
- The reverse() method reverses the elements in an array : ***fruits.reverse();***
- The toSorted() method creates a new array, keeping the original array unchanged :
  ```
  const sorted = months.toSorted();
  ```
- ***const reversed = months.toReversed();***

Numeric Sort

By default, the sort() function sorts values as strings. This works well for strings ("Apple" comes before "Banana").
If numbers are sorted as strings, "25" is bigger than "100", because "2" is bigger than "1". Because of this, the sort() method will produce incorrect results when sorting numbers.

```
const points = [40, 100, 1, 5, 25, 10];
points.sort(function(a, b){return a - b});
```

Find the Lowest (or Highest) Array Value

```
function myArrayMin(arr) {
  return Math.min.apply(null, arr);
}

function myArrayMax(arr) {
  return Math.max.apply(null, arr);
}
```

Sorting Object Arrays

```
const cars = [
  {type:"Volvo", year:2016},
  {type:"Saab", year:2001},
  {type:"BMW", year:2010}
];

cars.sort(function(a, b){return a.year - b.year});
```

JS Array Iteration

JS Array Const

- An array declared with const cannot be reassigned.

```
const cars = ["Saab", "Volvo", "BMW"];
cars = ["Toyota", "Volvo", "Audi"];    // ERROR
```

- But we can still change the elements of a constant array.

```
// You can change an element:
cars[0] = "Toyota";
```

// You can add an element:
cars.push("Audi");

- An array declared with const must be initialized when it is declared. Arrays declared with var can be initialized at any time. You can even use the array before it is declared:
  cars = ["Saab", "Volvo", "BMW"];
  var cars;

JS Dates

- new Date() creates a date object with the current date and time : ***const d = new Date();***
- new Date(date string) creates a date object from a date string :
  const d = new Date("2022-03-25");
- 6 numbers specify year, month, day, hour, minute, second :
  const d = new Date(2018, 11, 24, 10, 33, 30);
- When you display a date object in HTML, it is automatically converted to a string, with the toString() method.
- The toDateString() method converts a date to a more readable format:
  const d = new Date();
  d.toDateString();

JS Date Get Methods

Method	Description
getFullYear()	Get year as a four digit number (yyyy)
getMonth()	Get month as a number (0-11)
getDate()	Get day as a number (1-31)
getDay()	Get weekday as a number (0-6)
getHours()	Get hour (0-23)
getMinutes()	Get minute (0-59)
getSeconds()	Get second (0-59)
getMilliseconds()	Get millisecond (0-999)
getTime()	Get time (milliseconds since January 1, 1970)

- The ***getTimezoneOffset()*** method returns the difference (in minutes) between local time in UTC time:

JS Date Set Methods

Method	Description
setDate()	Set the day as a number (1-31)
setFullYear()	Set the year (optionally month and day)
setHours()	Set the hour (0-23)
setMilliseconds()	Set the milliseconds (0-999)
setMinutes()	Set the minutes (0-59)
setMonth()	Set the month (0-11)
setSeconds()	Set the seconds (0-59)
setTime()	Set the time (milliseconds since January 1, 1970)

const d = new Date();
d.setFullYear(2020);

JS Math

● Math.round(x)	:	Returns x rounded to its nearest integer
● Math.ceil(x)	:	Returns x rounded up to its nearest integer
● Math.floor(x)	:	Returns x rounded down to its nearest integer
● Math.trunc(x)	:	Returns the integer part of x (new in ES6)
● Math.sign(x)	:	Returns if x is negative, null or positive
● Math.pow(x, y)	:	Returns the value of x to the power of y
● Math.sqrt(x)	:	Returns the square root of x
● Math.abs(x)	:	Returns the absolute (positive) value of x
● Math.min()	:	Returns the minimum value in a list of arguments
● Math.max()	:	Returns the highest value in a list of arguments
● Math.random()	:	Returns a random number between 0 (inclusive), and 1 (exclusive)
● Math.log(x)	:	Returns the natural logarithm of x
● Math.log2(x)	:	Returns the base 2 logarithm of x

JS Booleans

You can use the Boolean() function to find out if an expression (or a variable) is true.

Boolean(10 > 9)

JS Comparisons

Conditional (Ternary) Operator

```
let voteable = (age < 18) ? "Too young":"Old enough";
```

Nullish Coalescing Operator (??)

The ?? operator returns the first argument if it is not nullish (null or undefined).

```
let name = null;
let text = "missing";
let result = name ?? text;
```

The Optional Chaining Operator (?.)

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

```
// Create an object:
const car = {type:"Fiat", model:"500", color:"white"};
// Ask for car name:
document.getElementById("demo").innerHTML = car?.name;
```

JS If Else

```
if (time < 10) {
  greeting = "Good morning";
} else if (time < 20) {
  greeting = "Good day";
} else {
  greeting = "Good evening";
}
```

JS Switch

- Switch cases use strict comparison (===).

```
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";
}
```

JS For Loop

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

JS For In

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

If we use this for loop method to access array elements, it won't access the array elements in the order. Therefore when the index, order is required, use forEach() method to access array elements.

Array.forEach()

The forEach() method calls a function (a callback function) once for each array element.

```
const numbers = [45, 4, 9, 16, 25];

let txt = "";
numbers.forEach(myFunction);

function myFunction(value) {
  txt += value;
}
```

JS For Of

The JavaScript for-of statement loops through the values of an iterable object. It lets you loop over iterable data structures such as Arrays, Strings, Maps, NodeLists, and more.

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

let text = "";
for (let x of cars) {
  text += x;
}
```

JS While Loop

```
while (i < 10) {
  text += "The number is " + i;
  i++;
}
```

The Do While Loop

The do while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

```
do {
  text += "The number is " + i;
  i++;
}
while (i < 10);
```

JS Break and Continue

- The **break** statement "jumps out" of a loop.

  Ex :
```
for (let i = 0; i < 10; i++) {
     if (i === 3) { break; }
     text += "The number is " + i + "<br>";
}
```

- The **continue** statement "jumps over" one iteration in the loop.

```
Ex : for (let i = 0; i < 10; i++) {
         if (i === 3) { continue; }
         text += "The number is " + i + "<br>";
     }
```

JavaScript Labels

JavaScript labels are identifiers that allow you to name loops or blocks of code, enabling more control over your code flow, especially with break and continue statements. This can be helpful when working with nested loops, where you may want to break or continue a specific loop rather than the innermost one.

- In this example, the break outerLoop; stops both the inner and outer loops when the condition is met. Without the label, only the inner loop would stop.

```
outerLoop: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (i === 1 && j === 1) {
      break outerLoop;  // Exits the outer loop
    }
    console.log(i, j);
  }
}
```

- Skips to the next iteration of the outer loop

```
outerLoop: for (let i = 0; i < 3; i++) {
  for (let j = 0; j < 3; j++) {
    if (i === 1 && j === 1) {
      continue outerLoop;  // Skips to the next iteration of the outer loop
    }
    console.log(i, j);
  }
}
```

JS Iterables

JavaScript Iterators

The iterator protocol defines how to produce a sequence of values from an object. An object becomes an iterator when it implements a **next()** method. The next() method must return an object with two properties:

- **value** (the next value)
- **done** (true or false)

Example: Custom Iterable

```
const myIterable = {
  data: [1, 2, 3, 4],
  [Symbol.iterator]() {
    let index = 0;
    return {
      next: () => {
        if (index < this.data.length) {
          return { value: this.data[index++], done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for (const value of myIterable) {
  console.log(value); // Outputs: 1, 2, 3, 4
}
```

JS Sets

- A JavaScript Set is a collection of unique values.
- Each value can only occur once in a Set.
- The values can be of any type, primitive values or objects.
- Sets are Objects

How to Create a Set

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

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

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

JavaScript Set Methods

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

- The **has()** method returns true if a specified value exists in a set.
  answer = letters.has("d");

- The **forEach()** method invokes a function for each Set element.
  let text = "";
  letters.forEach (function(value) {
  text += value;
  })

- The **values()** method returns an Iterator object with the values in a Set:
  const myIterator = letters.values();

- The **keys()** method returns an Iterator object with the values in a Set:
  const myIterator = letters.keys();

- The **entries()** method returns an Iterator with [value,value] pairs from a Set. The entries() method is supposed to return a [key,value] pair from an object. A Set has no keys, so the entries() method returns [value,value].

JS Maps

- A Map holds key-value pairs where the keys can be any datatype.
- Maps are Objects

How to Create a Map

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

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

The set() method can also be used to change existing Map values:

fruits.set("apples", 200);

JavaScript Objects vs Maps

Object	**Map**
Not directly iterable	Directly iterable
Do not have a size property	Have a size property
Keys must be Strings (or Symbols)	Keys can be any datatype
Keys are not well ordered	Keys are ordered by insertion
Have default keys	Do not have default keys

JavaScript Map Methods

- The **get()** method gets the value of a key in a Map:

  fruits.get("apples");

- The size property returns the number of elements in a map:

  fruits.size;

- The **delete()** method removes a map element:

  fruits.delete("apples");

- The **clear()** method removes all the elements from a map:

- The **has()** method returns true if a key exists in a map:

  fruits.has("apples");

- The **forEach()** method invokes a callback for each key/value pair in a map:
- The **entries()** method returns an iterator object with the [key,values] in a map:
- The **keys()** method returns an iterator object with the keys in a map:
- The **values()** method returns an iterator object with the values in a map:
- The **Map.groupBy()** method groups elements of an object according to string values returned from a callback function. The Map.groupBy() method does not change the original object.

JS Destructing

The Rest Property

You can end a destructuring syntax with a rest property. This syntax will store all remaining values into a new array.

```
// Create an Array
const numbers = [10, 20, 30, 40, 50, 60, 70];

// Destructuring
const [a,b, ...rest] = numbers
```

JS Destructing

Regular Expression Modifiers

- **i** Perform case-insensitive matching
- **g** Perform a global match (find all)
- **m** Perform multiline matching
- **d** Perform start and end matching (New in ES2022)

Regular Expression Patterns

- **[abc]** Find any of the characters between the brackets
- **[0-9]** Find any of the digits between the brackets
- **(x|y)** Find any of the alternatives separated with |
  text.match(/(red|green)/g);

JS Errors

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.

JavaScript try and catch

- The try statement allows you to define a block of code to be tested for errors while it is being executed.
- The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

```
try {
  Block of code to try
```

}
catch(err) {
  Block of code to handle errors
}

JavaScript Throws Errors

- The throw statement allows you to create a custom error.

```
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;
 }
```

The finally Statement

- The finally statement lets you execute code, after try and catch, regardless of the result.

```
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
}
```

JS Scope

- Scope determines the accessibility (visibility) of variables.
- JavaScript variables have 3 types of scope.
  Block scope
  Function scope
  Global scope

Block Scope

- let and const keywords provide Block Scope in JavaScript.
- Variables declared inside a { } block cannot be accessed from outside the block.

Local Scope

- Variables declared within a JavaScript function, are LOCAL to the function:
- Local variables have Function Scope.
- They can only be accessed from within the function.
- Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.
- Local variables are created when a function starts, and deleted when the function is completed.

```
// code here can NOT use carName

function myFunction() {
  let carName = "Volvo";
  // code here CAN use carName
}

// code here can NOT use carName
```

Function Scope

- JavaScript has function scope. Each function creates a new scope.
- Variables defined inside a function are not accessible (visible) from outside the function.
- Variables declared with var, let and const are quite similar when declared inside a function.

Global JavaScript Variables

A variable declared outside a function, becomes GLOBAL.

```
let carName = "Volvo";
// code here can use carName

function myFunction() {
// code here can also use carName
}
```

Global Scope

- Variables declared Globally (outside any function) have Global Scope.
- Global variables can be accessed from anywhere in a JavaScript program.
- Variables declared with var, let and const are quite similar when declared outside a block.

Automatically Global

If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable. In "Strict Mode", undeclared variables are not automatically global.

```
myFunction();

// code here can use carName

function myFunction() {
  carName = "Volvo";
}
```

The Lifetime of JavaScript Variables

- The lifetime of a JavaScript variable starts when it is declared.
- Function (local) variables are deleted when the function is completed.
- In a web browser, global variables are deleted when you close the browser window (or tab).

JS Hoisting

Hoisting is JavaScript's default behavior of moving declarations to the top. Hoisting is JavaScript's default behavior of moving all declarations to the top of the current scope (to the top of the current script or the current function).

JavaScript Declarations are Hoisted

```
x = 5; // Assign 5 to x
elem = document.getElementById("demo"); // Find an element
elem.innerHTML = x;                     // Display x in the element
var x; // Declare x
```

- 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 Initializations are Not Hoisted

- JavaScript only hoists declarations, not initializations.
- Declare Your Variables At the Top
- JavaScript in strict mode does not allow variables to be used if they are not declared.

JS Strict Mode

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

Why Strict Mode?

- Strict mode makes it easier to write "secure" JavaScript.
- Strict mode changes previously accepted "bad syntax" into real errors.
- As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.
- In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.
- In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

Not Allowed in Strict Mode

- Using a variable, without declaring it, is not allowed:

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

- Deleting a variable (or object) is not allowed. Deleting a function is not allowed.

```
delete x;          // This will cause an error
```

- Duplicating a parameter name is not allowed:

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

JS Arrow Function

- 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!";
```

- If you have parameters, you pass them inside the parentheses.

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

- In fact, if you have only one parameter, you can skip the parentheses as well.

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

JS Classes

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

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

- When you have a class, you can use the class to create objects.

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

The Constructor Method

- 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

JS Modules

- JavaScript modules allow you to break up your code into separate files.
- This makes it easier to maintain a code-base.
- 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

There are two types of exports: **Named Exports** and **Default Exports**.

1. **Named Exports**

   You can create named exports two ways. In-line individually, or all at once at the bottom.

   In-line individually:

   person.js
   ```
   export const name = "Jesse";
   export const age = 40;
   ```

   All at once at the bottom:

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

2. **Default Exports**

   message.js
   ```
   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";
  ```

- **Import from default exports**
  ```
  import message from "./message.js";
  ```

JS 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.

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

Converting a JSON Text to a JavaScript Object

- 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);
```

- Finally, use the new JavaScript object in your page.

```
obj.employees[1].firstName
```

JS Debugging

- If your browser supports debugging, you can use **console.log()** to display JavaScript values in the debugger window:
- The **debugger** keyword stops the execution of JavaScript, and calls (if available) the debugging function. If no debugging is available, the debugger statement has no effect.

Major Browsers' Debugging Tools

Normally, you activate debugging in your browser with F12, and select "Console" in the debugger menu.

1. **Chrome**
    - Open the browser.
    - From the menu, select "More tools".
    - From tools, choose "Developer tools".
    - Finally, select Console.
2. **Firefox**
    - Open the browser.
    - From the menu, select "Web Developer".
    - Finally, select "Web Console".
3. Edge
    - Open the browser.
    - From the menu, select "Developer Tools".
    - Finally, select "Console".

JS Mistakes

- You must use a "backslash" if you must break a statement in a string:

```
let x = "Hello \
World!";
```