**JAVASCRIPT TUTORIAL**

JavaScript was invented by Brendan Eich in 1995, and became an ECMA standard in 1997.

**The <script> Tag**

In HTML, JavaScript code is inserted between <script> and </script> tags. Scripts can be placed in the <body>, or in the <head> section of an HTML page, or in both. Placing scripts at the **bottom** of the <body> element improves the display speed, because script interpretation slows down the display.

<script>  
document.getElementById("demo").innerHTML = "My First JavaScript";  
</script>

**JavaScript Functions and Events :** A JavaScript function is a block of JavaScript code, that can be executed when "called" for. For example, a function can be called when an **event** occurs, like when the user clicks a button.

**External JavaScript :->** Scripts can also be placed in external files:

External file: myScript.js

function myFunction() {  
  document.getElementById("demo").innerHTML = "Paragraph changed.";  
}

External scripts are practical when the same code is used in many different web pages. JavaScript files have the file extension **.js**. To use an external script, put the name of the script file in the src (source) attribute of a <script> tag:

Example

<script src="myScript.js"></script>

* **External scripts cannot contain <script> tags.**

**To add several script files to one page  - use several script tags:**

Example

<script src="myScript1.js"></script>  
<script src="myScript2.js"></script>

**JavaScript Display Possibilities**

JavaScript can "display" data in different ways:

Writing into an HTML element, using **innerHTML**. : document.getElementById("demo").innerHTML = 5 + 6;

Writing into the HTML output using **document.write()**. document.write() after an HTML document is loaded, will **delete all existing HTML**:

Writing into an alert box, using **window.alert()**.

Writing into the browser console, using **console.log()**.

**Printing Page :** to print the current page from printer.

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

**DECLARING A VARIABLE**

let x,y,z; declaring 3 variables.

There are 3 ways to declare a JavaScript variable:

* Using **var**
* Using **let**
* Using **const**

**One Statement, Many Variables** :-You can declare many variables in one statement.

Start the statement with var and separate the variables by **comma**:

var person = "John Doe", carName = "Volvo", price = 200;

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:

carName = "Volvo"; var carName;

* **A variable declared without a value will have the value undefined.**

**LET :->**

Variables defined with let cannot be Redeclared.

Variables defined with let must be Declared before use.

Variables defined with let have Block Scope.

Using a let variable before it is declared will result in a ReferenceError:

Example

carName = "Saab";  
let carName = "Volvo"; -🡪 referenceError

**CONST**

Variables defined with const cannot be Redeclared.

Variables defined with const cannot be Reassigned.

Variables defined with const have Block Scope.

* JavaScript const variables must be assigned a value when they are declared:
* Using a const variable before it is declared will result in a ReferenceError:

**But CONST CAN:**

* **Change the elements of constant array**
* **Change the properties of constant object**
* JavaScript has dynamic types. This means that the same variable can be used to hold different data types:

**KEYWORDS**

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| 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 |

**COMMENTS**

Code after double slashes // or between /\* and \*/ is treated as a **comment**.

* **Hyphens are not allowed in JavaScript. They are reserved for subtractions.**
* **JavaScript Comparison Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| == | equal to |
| === | equal value and equal type |
| != | not equal |
| !== | not equal value or not equal type |
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| ? | ternary operator |

**TYPE OPERATOR**

typeof 🡪 Returns the type of a variable

instanceof 🡪 Returns true if an object is an instance of an object type

var a = 2

console.log(typeof(a)) // number

var name = "satish"

console.log(typeof(name)) // string

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

**NOTE:**

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 Data Type of typeof**

The typeof operator is not a variable. It is an operator. Operators ( + - \* / ) do not have any data type.

But, the typeof operator always **returns a string** (containing the type of the operand).

**The constructor Property**

The constructor property returns the constructor function for all JavaScript variables.

console.log("========= type of operator ==========")

console.log(typeof "John" )                // Returns "string"

console.log(typeof 3.14  )                 // Returns "number"

console.log(typeof NaN    )                // Returns "number"

console.log(typeof false)               // Returns "boolean"

console.log(typeof [1,2,3,4])              // Returns "object"

console.log(typeof {name:'John', age:34})  // Returns "object"

console.log(typeof new Date())             // Returns "object"

console.log(typeof function () {})         // Returns "function"

console.log(typeof myCar)                  // Returns "undefined" \*

console.log(typeof null)                   // Returns "object"

console.log("John".constructor)                // Returns function String()

console.log((3.14).constructor)                // Returns function Number

console.log(false.constructor)                 // Returns) function Boolean()

console.log([1,2,3,4].constructor)        // Returns) function Array()

console.log({name:')John))',age:34}.constructor)  // Returns function Object()

console.log(new Date().constructor)           // Returns function) Date()

console.log(function () {}.constructor)        // Returns function Function

**TYPE CONVERSION**

**Strings to Numbers** :The global method Number() can convert strings to numbers. Strings containing numbers (like "3.14") convert to numbers (like 3.14).

Empty strings convert to 0. ::: Anything else converts to NaN (Not a Number).

Number("3.14")    // returns 3.14  
Number(" ")       // returns 0  
Number("")        // returns 0  
Number("99 88")   // returns NaN

**Numbers to Strings**

The global method String() can convert numbers 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

**Booleans to Numbers**

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

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

**ARTHIMETIC OPERATOR**

\*\* 🡪 Exponentation

// exponenation

var x = 2;

var y= 3;

x \*\*= y;

console.log(x) // 2^3

**JavaScript variables can hold different data types: numbers, strings, objects and more:**

let length = 16;                               // Number  
let lastName = "Johnson";                      // String  
let x = {firstName:"John", lastName:"Doe"};    // Object

**JS OBJECTS**

In JavaScript, almost "everything" is an object.

* Booleans can be objects (if defined with the new keyword)
* Numbers can be objects (if defined with the new keyword)
* Strings can be objects (if defined with the new keyword)
* Dates are always objects
* Maths are always objects
* Regular expressions are always objects
* Arrays are always objects
* Functions are always objects
* Objects are always objects

JavaScript objects are written with curly braces {}. A JavaScript object is a collection of **named values**. Object properties are written as name:value pairs, separated by commas.

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

**Accessing Object Properties**

You can access object properties in two ways:

*objectName.propertyName* or *objectName["propertyName"]*

Objects can also have **methods**.

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

Comparing two JavaScript objects **always** returns **false**.

* If you access a method **without** the () parentheses, it will return the **function definition**:

**DELETING PROPERTIES OF OBJECT**

The delete keyword deletes a property from an object: delete person.id; // delete id property from person object.

**Nested Objects**

Values in an object can be another object:

Example

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"];

**Do Not Declare Strings, Numbers, and Booleans as Objects!**

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

* **Any variable can be emptied, by setting the value to undefined. The type will also be undefined**.

car = undefined;    // Value is undefined, type is undefined

**JS FUNCTIONS**

function myFunction(p1, p2) {  
  return p1 \* p2;   // The function returns the product of p1 and p2  
}

**CREATING JS OBJECTS USING NEW KEYWORD**

const person = new Object();  
person.firstName = "John";  
person.lastName = "Doe";  
person.age = 50;  
person.eyeColor = "blue";

**SYNTAX :**

A JavaScript function is defined with the function keyword, followed by a **name**, followed by parentheses **()**.

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas:  
**(*parameter1, parameter2, ...*)**

The code to be executed, by the function, is placed inside curly brackets: **{}**

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

**JS EVENTS**

When JavaScript is used in HTML pages, JavaScript can **"react"** on these events. JavaScript lets you execute code when events are detected.

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

In the next example, the code changes the content of its own element (using **this**.innerHTML):

Example : <button onclick="this.innerHTML = Date()">The time is?</button>

**Common HTML Events**

Here is a list of some 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 |

**STRING**

Length of string : varName.length

Do not create Strings objects. The new keyword complicates the code and slows down execution speed.

String objects can produce unexpected results:

* **JS Objects cannot be compared and so the result is always false.**

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

console.log(x==y) // gives false as both x and y are objects and can’t be compared.

methods and properties are also available **to primitive values**, because JavaScript treats primitive values as objects when executing methods and properties.

**STRING METHODS**

Extracting String Parts

There are 3 methods for extracting a part of a string:

* slice(*start*, *end*)
* substring(*start*, *end*)
* substr(*start*, *length*)

If a parameter is negative, the position is counted from the end of the string.

If you omit the second parameter, the method will slice out the rest of the string:

**The String substring() Method**

substring() is similar to slice(). The difference is that substring() cannot accept negative indexes.

**substr() is similar to slice().**

The difference is that the second parameter specifies the **length** of the extracted part.

**Replacing String Content**

The replace() method replaces a specified value with another value in a string:

The replace() method does not change the string it is called on. It returns a new string. replace() method is case sensitive

var myName = "satish kumar singh"

console.log(myName.slice(0,6))

console.log(myName.substring(3))

console.log(myName.replace('s', 't'))

console.log(myName.toUpperCase())

**Converting to Upper and Lower Case**

A string is converted to upper case with **toUpperCase**() and lower using **toLowerCase**():

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

let text3 = text1.toLowerCase(); // to lower case

**concat**() joins two or more strings:

trim() : to remove blank spaces

charAt(n) : returns character at specified index

charCodeAt(n) : returns unicode of the character at a specified index in a string

**Converting a String to an Array**

A string can be converted to an array with the split() method:

Example: the split produces array on the basis of delimiter specified.

console.log(myName.split(" "))  // [ 'satish', 'kumar', 'singh' ]

text.split(",")    // Split on commas  
text.split(" ")    // Split on spaces  
text.split("|")    // Split on pipe

**JavaScript methods for searching strings:**

* String indexOf()
* String lastIndexOf()
* String startsWith()
* String endsWith()

Both indexOf(), and lastIndexOf() return -1 if the text is not found:

console.log(myName.indexOf('t'))  //2

console.log(myName.lastIndexOf('s'))  //13

console.log(myName.startsWith('s')) // true

console.log(myName.startsWith('S')) // false

**String search()**

The search() method searches a string for a specified value and returns the position of the match:

**String match()**

The match() method searches a string for a match against a regular expression, and returns the matches, as an Array object.

**String includes()**

The includes() method returns true if a string contains a specified value.

**Quotes Inside Strings**

With **template literals**, you can use both single and double quotes inside a string:

Example : let text = `He's often called "Johnny"`;

Template literals provide an easy way to interpolate variables and expressions into strings.

The method is called **string interpolation**.

The syntax is: ${...}

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

Automatic replacing of variables with real values is called **string interpolation**.

**JS NUMBERS**

JavaScript has only one type of number. Numbers can be written with or without decimals. JavaScript Numbers are Always 64-bit Floating Point. Unlike many other programming languages, JavaScript does not define different types of numbers, like integers, short, long, floating-point etc.

Extra large or extra small numbers can be written with scientific (exponent) notation:

let x = 123e5;    // 12300000  
let y = 123e-5;   // 0.00123

**Numeric Strings** : JavaScript strings can have numeric content:

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

This will work:

let x = "100";  
let y = "10";  
let z = x / y; // 10

let z = x \* y; // 1000

let z = x - y; // 90

let z = x + y; // 10010 // concatenate happens here

**NaN - Not a Number**

**NaN** is a JavaScript reserved word indicating that a number is not a legal number. Trying to do arithmetic with a non-numeric string will result in NaN (Not a Number):

let x = 100 / "Apple"; // NaN

You can use the global JavaScript function isNaN() to find out if a value is a not a number:

NaN is a number: typeof NaN returns number:

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

Division by 0 (zero) also generates Infinity:

Example

let x =  2 / 0; // Infinity  
let y = -2 / 0; // -Infinity

Infinity is a number: typeof Infinity returns number.

**NUMBER METHODS**

toString() method returns a number as a string.

x = 123

x.toString()

**toExponential()** returns a string, with a number rounded and written using exponential notation

let x = 9.656;  
x.toExponential(2); 9.66e+0  
x.toExponential(4); 9.6560e+0

**toFixed()** returns a string, with the number written with a specified number of decimals:

let x = 9.656;  
x.toFixed(0); // 10  
x.toFixed(2); // 9.66  
x.toFixed(4); // 9.6560

All JavaScript data types have a valueOf() and a toString() method.

**Converting Variables to Numbers**

There are 3 JavaScript methods that can be used to convert variables to numbers:

* The Number() method -- Returns a number, converted from its argument.
* The parseInt() -- Parses its argument and returns a floating point number
* The parseFloat() -- Parses its argument and returns an integer

These methods are not **number** methods, but **global** JavaScript methods.

console.log("=================== NUMBER METHOD ============")

console.log(Number(true)); // 1

console.log(Number(false)); //0

console.log(Number("10")); // 10

console.log(Number("  10")); // 10

console.log(Number("10  ")); //10

console.log(Number("10.33")); // 10.33

console.log(Number("10,33")); // NaN

console.log(Number("10 33")); // NaN

console.log(Number("John")); //NaN

parseInt() 🡪 parses a string and returns a whole number. Spaces are allowed. Only the first number is returned:

console.log(parseInt("-10")); // -10

console.log(parseInt("-10.33")); // -10

console.log(parseInt("10")); // 10

console.log(parseInt("10.33")); //10

console.log(parseInt("10 20 30"));// 10

console.log(parseInt("10 years"));//10

console.log(parseInt("years 10")); //NaN

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

console.log(parseFloat("10")); // 10

console.log(parseFloat("10.33")); //10.33

console.log(parseFloat("10 20 30"));// 10

console.log(parseFloat("10 years")); //10

console.log(parseFloat("years 10")); //NaN

**JS ARRAYS**

it is a common practice to declare arrays with the **const** keyword.

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

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

Example

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

creating array with new keyword : const cars = new Array("Saab", "Volvo", "BMW");

**Access the Full Array** :->With JavaScript, the full array can be accessed by referring to the array name: Example

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

typeof(cars) // objects

Arrays are a special type of objects.  Arrays use **numbers** to access its "elements". person[0] returns 0th item. Whereas Objects use **names** to access its "members". person.firstName returns firstName propery of object person.

**Printing array elements on WebPage:**

<!-- ACCESSING ARRAY ELEMENTS -->

<br>

<h3> Priniting Array Elements </h3>

<p id="my\_array"></p>

<script>

    const my\_cars = ['bmw', 'volovl', 'mercedes']

    text = "<ul>"

    for (let i=0;i<my\_cars.length;i++){

        text += "<li>" + my\_cars[i] + "</li>";

    }

    text += "</ul>"

    document.getElementById("my\_array").innerHTML = text;

</script>

**USING FOR EACH FUNCTION**

**<!-- using for each function -->**

**<p id="fruits\_place"></p>**

**<script>**

**const fruits = ["apple", "banana", "Guava"]**

**text = "<ul>"**

**fruits.forEach(my\_function)**

**text+= "</ul>"**

**function my\_function(fruit){**

**text+= "<li>" + fruit +"</li>";**

**}**

**document.getElementById("fruits\_place").innerHTML = text;**

**</script>**

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

**Adding Array Elements**

The easiest way to add a new element to an array is using the **push()** method:

Example

const fruits = ["Banana", "Orange", "Apple"];  
fruits.push("Lemon");  // Adds a new element (Lemon) to fruits

New element can also be added to an array using the length property:

Example

const fruits = ["Banana", "Orange", "Apple"];  
fruits[fruits.length] = "Lemon";  // Adds "Lemon" to fruits

Adding elements with high indexes can create undefined "holes" in an array: HOLES MEANS VACANT INDEXES(UNDEFINED)

**IMPORTANT**

const points = [40]; is not the same as: const points = new Array(40);

const points = [40]; // Create an array with one element:

const points = new Array(40);  // Create an array with 40 undefined elements:  
to check if variable is array or not : Array.isArray(fruits);

**JS ARRAY METHODS**

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

**join()** method also joins all array elements into a string. It behaves just like toString(), but in addition you can specify the separator:

**pop()** method removes the last element from an array: returns the value that was "popped out":

**push()** method adds a new element to an array (at the end): returns the new array length:

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

**unshift()** method adds a new element to an array (at the beginning), and "unshifts" older elements: unshift() method returns the new array length.

**length** property provides an easy way to append a new element to an array:

Array elements can be **deleted** using the JavaScript operator delete. Using delete leaves undefined holes in the array.

Use pop() or shift() instead.

**concat()** method creates a new array by merging (concatenating) existing arrays:

const my\_merged\_array = arr1.concat(arr2);

The concat() method can take any number of array arguments: const myChildren = arr1.concat(arr2, arr3);

**splice()** method can be used to add new items to an array:

const fruits = ["Banana", "Orange", "Apple", "Mango"];  
fruits.splice(2, 0, "Lemon", "Kiwi");

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 rest of the parameters ("Lemon" , "Kiwi") define the new elements to be **added**.

The splice() method returns an array with the deleted items:

**slice()** method slices out a piece of an array into a new array. The slice() method creates a new array. The slice() method does not remove any elements from the source array.

**sort**() method sorts an array alphabetically:

**reverse**() method reverses the elements in an array.

console.log("========= JS ARRAY METHODS ==========")

// toString() methods

console.log(cars\_new\_array.toString()) // Saab,Volvo,BMW

// join method

console.log(cars\_new\_array.join("\*")) // Saab\*Volvo\*BMW

// pop

console.log(cars\_new\_array.pop()) // BMW

// push

console.log(cars\_new\_array.push("BMW")) // 3

//shifting

console.log(cars\_new\_array.shift()) // saab

// unshift

console.log(cars\_new\_array.unshift('maruti')) //3

// delete

delete cars\_new\_array[0];

console.log(cars\_new\_array) // [ <1 empty item>, 'Volvo', 'BMW' ]

//indexOf()

console.log("INDEX OF BMW : ",cars\_new\_array.indexOf('BMW'))

// concat

my\_name = ['satish']

my\_last\_name = ['kumar', 'singh']

merged\_array = my\_name.concat(my\_last\_name)

console.log(merged\_array) // [ 'satish', 'kumar', 'singh' ]

// splice

const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.splice(2, 0, "Lemon", "Kiwi");

console.log(fruits) // [ 'Banana', 'Orange', 'Lemon', 'Kiwi', 'Apple', 'Mango' ]

// slice

const citrus = fruits.slice(1);

console.log(citrus) //['Orange', 'Lemon', 'Kiwi', 'Apple', 'Mango' ]]

// sorting

console.log(fruits.sort()) // [ 'Apple', 'Banana', 'Kiwi', 'Lemon', 'Mango', 'Orange’]

**Find the Highest (or Lowest) Array Value**

There are no built-in functions for finding the max or min value in an array.

However, after you have sorted an array, you can use the index to obtain the highest and lowest values.

Sorting ascending:

const points = [40, 100, 1, 5, 25, 10];  
points.sort(function(a, b){return a - b});  
// now points[0] contains the lowest value  
// and points[points.length-1] contains the highest value

**JavaScript Array map()**

The map() method creates a new array by performing a function on each array element.

The map() method does not execute the function for array elements without values.

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

// map()

const numbers1 = [45, 4, 9, 16, 25];

const numbers2 = numbers1.map(myFunction);

console.log(numbers2) // [ 90, 8, 18, 32, 50 ]

function myFunction(value, index, array) {

  return value \* 2;

}

**filter()** method creates a new array with array elements that passes a test.

filtered\_number = numbers1.filter(filter\_function);

console.log(filtered\_number) //[ 45, 16, 25 ]

function filter\_function(value){

  return value > 10 // return only value which are greater than 18

}

**JavaScript Array reduce()**

The reduce() method runs a function on each array element to produce (reduce it to) a single value.

The reduce() method works from **left-to-right in the array**. The reduce() method does not reduce the original array.

Note that the function takes 4 arguments:

* The total (the initial value / previously returned value)
* The item value
* The item index
* The array itself

// reduce

sum = numbers1.reduce(myFunction);

function myFunction(total, value, index, array) {

  return total + value;

}

console.log(sum) // 99

**reduceRight()** method runs a function on each array element to produce (reduce it to) a single value.

The reduceRight() works from right-to-left in the array.

**JavaScript Array every()**

The every() method check if all array values pass a test. Entire items of the array should satisfy the condition then the every method will **return True else false**.

**JavaScript Array some(some\_function)**

The some() method check if some array values pass a test. This example check if some array values are larger than 18:

**indexOf()** method searches an array for an element value and returns its position. Array.indexOf() returns -1 if the item is not found.

**JavaScript Array lastIndexOf()**

Array.lastIndexOf() is the same as Array.indexOf(), but returns the position of the last occurrence of the specified element.

**find()**

The find() method returns the value of the first array element that passes a test function.

**findIndex()** method returns the index of the first array element that passes a test function

**Array.from()** method returns an Array object from any object with a length property or any iterable object.

console.log(Array.from("ABCDE")) // [ 'A', 'B', 'C', 'D', 'E' ]

**Array.keys**() method returns an Array Iterator object with the keys of an array.

const keys = cars\_new\_array.keys()

console.log(keys)   // Object [Array Iterator] {}

for (let x of keys){

  console.log(x)  // 0   1   2

}

**Array.includes()** to arrays. This allows us to check if an element is present in an array (including NaN, unlike indexOf).

Example : fruits.includes("Mango"); // is true

IMP ARRAY DECLARATION : <https://www.w3schools.com/js/js_array_const.asp>

**JS DATES**

Const d = new Date() //2021-12-23T06:21:22.212Z

There are **4 ways** to create a new date object:

new Date()  
new Date(*year, month, day, hours, minutes, seconds, milliseconds*)  
new Date(*milliseconds*)  
new Date(*date string*)

Date objects are static. The computer time is ticking, but date objects are not.

**Note:** JavaScript counts months from **0** to **11**: **January = 0. December = 11**.

Specifying a month higher than 11, will not result in an error but add the overflow to the next year:

Specifying a day higher than max, will not result in an error but add the overflow to the next month:

JavaScript stores dates as number of milliseconds since January 01, 1970, 00:00:00 UTC (Universal Time Coordinated).

**toUTCString()** method converts a date to a UTC string (a date display standard).

const d = new Date()

console.log(d) //2021-12-23T06:27:09.270Z

console.log("toUTCString() : "+d.toUTCString()) // Thu, 23 Dec 2021 06:27:09 GMT

console.log("toDateString(): ", d.toDateString()) // Thu Dec 23 2021

**Date Input - Parsing Dates**

If you have a valid date string, you can use the Date.parse() method to convert it to milliseconds.

**Date.parse()** returns the number of milliseconds between the date and January 1, 1970:

can then use the number of milliseconds to **convert it to a date** object:

let msec = Date.parse("March 21, 2012");

console.log("MiliiSecods : "+msec) // 1332268200000

const millisec\_to\_date = new Date(msec);

console.log("miliseconds parsed to date : "+millisec\_to\_date)

**Date Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| getFullYear() | Get the **year** as a four digit number (yyyy) |
| getMonth() | Get the **month** as a number (0-11) |
| getDate() | Get the **day** as a number (1-31) |
| getHours() | Get the **hour** (0-23) |
| getMinutes() | Get the **minute** (0-59) |
| getSeconds() | Get the **second** (0-59) |
| getMilliseconds() | Get the **millisecond** (0-999) |
| getTime() | Get the time (milliseconds since January 1, 1970) |
| getDay() | Get the weekday as a number (0-6) |
| Date.now() | Get the time. ECMAScript 5. |

**UTC DATE METHODS**

|  |  |
| --- | --- |
| **Method** | **Description** |
| getUTCDate() | Same as getDate(), but returns the UTC date |
| getUTCDay() | Same as getDay(), but returns the UTC day |
| getUTCFullYear() | Same as getFullYear(), but returns the UTC year |
| getUTCHours() | Same as getHours(), but returns the UTC hour |
| getUTCMilliseconds() | Same as getMilliseconds(), but returns the UTC milliseconds |
| getUTCMinutes() | Same as getMinutes(), but returns the UTC minutes |
| getUTCMonth() | Same as getMonth(), but returns the UTC month |
| getUTCSeconds() | Same as getSeconds(), but returns the UTC seconds |

**JS SET DATE METHODS**

|  |  |
| --- | --- |
| 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) |

// SET DATE METHODS

d.setFullYear(2020)

console.log("Setting year to 2020 : "+d) // Wed Dec 23 2020 12:22:45 GMT+0530 (India Standard Time)

d.setMonth(6)

console.log("Setting Month to 6: "+d)  //  Thu Jul 23 2020 12:23:47 GMT+0530 (India Standard Time)

d.setDate(15)

console.log("Setting Date to 15: "+d) // Wed Jul 15 2020 12:23:47 GMT+0530 (India Standard Time)

// setDate() method can also be used to add days to a date:

d.setDate(d.getDate() + 50);

console.log("Adding 50 to days : "+d) // Thu Sep 03 2020 12:24:45 GMT+0530 (India Standard Time)

// comparing dates

let date\_result = "";

const today = new Date();

const someday = new Date();

someday.setFullYear(2100, 0, 14);

if (someday > today) {

  date\_result = "Today is before January 14, 2100.";

} else {

  date\_result = "Today is after January 14, 2100.";

}

console.log("Date Result is : "+date\_result)

**BOOLEAN VALUES**

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

Example : Boolean(10 > 9) // True

**JS COMPARISIONS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Description** | **Comparing** | **Returns** | **Try it** |
| == | equal to | x == 8 | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison1) |
| x == 5 | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison2) |
| x == "5" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison12) |
| === | equal value and equal type | x === 5 | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison4) |
| x === "5" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison3) |
| != | not equal | x != 8 | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison5) |
| !== | not equal value or not equal type | x !== 5 | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison7) |
| x !== "5" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison6) |
| x !== 8 | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison13) |
| > | greater than | x > 8 | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison8) |
| < | less than | x < 8 | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison9) |
| >= | greater than or equal to | x >= 8 | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison10) |
| <= | less than or equal to | x <= 8 | true |  |

**CONDITIONAL OR TERNARY OPERATOR**

*variablename*= (*condition*) ?*value1*:*value2*

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

**Comparing Different Types**

Comparing data of different types may give unexpected results.When comparing a string with a number, JavaScript will convert the string to a number when doing the comparison. An empty string converts to 0. A non-numeric string converts to NaN which is always false.

|  |  |  |
| --- | --- | --- |
| **Case** | **Value** | **Try** |
| 2 < 12 | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_20) |
| 2 < "12" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_21) |
| 2 < "John" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_23) |
| 2 > "John" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_24) |
| 2 == "John" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_25) |
| "2" < "12" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_26) |
| "2" > "12" | true | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_27) |
| "2" == "12" | false | [Try it »](https://www.w3schools.com/js/tryit.asp?filename=tryjs_comparison_28) |

When comparing two strings, "2" will be greater than "12", because (alphabetically) 1 is less than 2.

**SWITCH DETAILS**

switch statement to select one of many code blocks to be executed.

**Syntax**

switch(*expression*) {  
  case *x*:  
*// code block*    break;  
  case *y*:  
*// code block*    break;  
  default:  
    // *code block*  
}

**This is how it works:**

* The switch expression is evaluated once.
* The value of the expression is compared with the values of each case.
* If there is a match, the associated block of code is executed.
* If there is no match, the default code block is executed.

**Strict Comparison**

Switch cases use **strict** comparison (===). The values must be of the same type to match.

**LOOPS**

JavaScript supports different kinds of loops:

* for - loops through a block of code a number of times
* for/in - loops through the properties of an object
* for/of - loops through the values of an iterable object
* while - loops through a block of code while a specified condition is true
* do/while - also loops through a block of code while a specified condition is true

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

**Statement 1** is executed (one time) before the execution of the code block.

**Statement 2** defines the condition for executing the code block.

**Statement 3** is executed (every time) after the code block has been executed.

Example

for (let i = 0; i < 5; i++) {  
  text += "The number is " + i + "<br>";  
}

**For In Loop**

The JavaScript for in statement loops through the properties of an Object:

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

const person = {fname:"John", lname:"Doe", age:25};  
let text = "";  
for (let x in person) {  
  text += person[x];  
}

const person\_loop = {name : "satish", title:"Kumar", lastName : "Singh"}

for (x in person\_loop){

  console.log("Key Name : ", x)

  console.log("Key Values: ", person\_loop[x])

}

o/p:

Key Name : name

Key Values: satish

Key Name : title

Key Values: Kumar

Key Name : lastName

Key Values: Singh

For in on loop :

const numbers\_for\_in = [45, 4, 9, 16, 25];

let txt\_for\_in = "";

for (let x in numbers\_for\_in) {

  txt\_for\_in += numbers\_for\_in[x];

}

console.log(txt\_for\_in)

**For Of Loop**

The JavaScript for of statement loops through the values of an iterable object.

for\_of\_text = "Javascript"

for (let x of for\_of\_text){

  console.log(x) // prints each letter in new line

}

**JS SETS**

A JavaScript Set is a collection of unique values. Each value can only occur once in a Set.

|  |  |
| --- | --- |
| **Method** | **Description** |
| new Set() | Creates a new Set |
| add() | Adds a new element to the Set |
| delete() | Removes an element from a Set |
| has() | Returns true if a value exists in the Set |
| forEach() | Invokes a callback for each element in the Set |
| values() | Returns an iterator with all the values in a Set |
| **Property** | **Description** |
| size | Returns the number of elements in a Set |

const a\_set = new Set(["a", "b", "c"])

console.log("the set is : ", a\_set)  // Set(3) { 'a', 'b', 'c' }

// add set

a\_set.add('d')

console.log("The new set is : ", a\_set) //  Set(4) { 'a', 'b', 'c', 'd' }

// values method

console.log(a\_set.values())  // [Set Iterator] { 'a', 'b', 'c', 'd' }

**JS MAPS**

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

|  |  |
| --- | --- |
| **Method** | **Description** |
| new Map() | Creates a new Map |
| set() | Sets the value for a key in a Map |
| get() | Gets the value for a key in a Map |
| delete() | Removes a Map element specified by the key |
| has() | Returns true if a key exists in a Map |
| forEach() | Calls a function for each key/value pair in a Map |
| entries() | Returns an iterator with the [key, value] pairs in a Map |
| **Property** | **Description** |
| size | Returns the number of elements in a Map |

**entries() Method**

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

Example

// List all entries  
let text = "";  
for (const x of fruits.entries()) {  
  text += x;  
}

// new map

const fruits\_map = new Map([

  ["apples", 500],

  ["bananas", 300],

  ["oranges", 200]

]);

console.log(" MAP IS : ", fruits\_map) // Map(3) { 'apples' => 500, 'bananas' => 300, 'oranges' => 200 }

// add element using set()

fruits\_map.set("guavaa", 100)

console.log(fruits\_map) //Map(4) { 'apples' => 500, 'bananas' => 300, 'oranges' => 200, 'guavaa' => 100 }

// get()

console.log(fruits\_map.get("apples")) // 500

// size property

console.log("Size property : ", fruits\_map.size)

// set

console.log(fruits\_map.set("apples", 1000))

console.log("New fruits map is : ", fruits\_map) // apple value is set to 100

// delete

console.log(fruits\_map.delete("appless")) // returns true if deleted or false if not exists

**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.

The JavaScript statements try and catch come in pairs:

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*}

finally statement lets you execute code, after try and catch, regardless of the result:

**JavaScript Throws Errors**

When an error occurs, JavaScript will normally stop and generate an error message. The technical term for this is: JavaScript will **throw an exception (throw an error)**. JavaScript will actually create an **Error object** with two Properties: **name** and **message**.

**The throw Statement**

The throw statement allows you to create a custom error.

Technically 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

**JS SCOPE**

Scope determines the accessibility (visibility) of variables.

JavaScript has 3 types of scope:

* Block scope
* Function scope
* Global scope

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

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

Variables declared with the **var** keyword can NOT have block scope.

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**.

Example

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

**Automatically Global**

If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.

This code example will declare a global variable carName, even if the value is assigned inside a function.

Example

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

**JavaScript Hoisting**

Hoisting is JavaScript's default behavior of moving declarations to the top. In JavaScript, a variable can be declared after it has been used. In other words; a variable can be used before it has been declared. Variables defined with let and const are hoisted to the top of the block, but not *initialized*.

**JavaScript Use Strict**

"use strict"; Defines that JavaScript code should be executed in "strict mode". The purpose of "use strict" is to indicate that the code should be executed in "strict mode". With strict mode, you can not, for example, use undeclared variables.

**Declaring Strict Mode**

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

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

Ex:

"use strict";  
myFunction();  
function myFunction() {  
  y = 3.14;   // This will also cause an error because y is not declared  
}

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  
}

The "use strict" directive is only recognized at the **beginning** of a script or a function.

**this keyword**

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

this keyword refers to the object it belongs to.

**Object Method Binding**

In these examples, this is the **person** object (The person object is the "owner" of the function):

const person\_this = {

  firstName: "John",

  lastName : "Doe",

  id       : 5566,

  fullName : function() {

    return this.firstName + " " + this.lastName;

  },

  myFunction : function() {

    return this;

  }

};

console.log(person\_this.fullName()) // John Doe

console.log(person\_this.myFunction()) /\* OP : {

  firstName: 'John',

  lastName: 'Doe',

  id: 5566,

  fullName: [Function: fullName],

  myFunction: [Function: myFunction]

}\*/

**ARRROW FUNCTION**

Arrow functions allow us to write shorter function syntax:

Before:

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

With Arrow Function:

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

**What About this in arrow function?**

with arrow functions there are no binding of this. 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.

**JS CALLBACKS**

A callback is a function passed as an argument to another function. When you nest a function inside another function as an argument, that's called a callback.

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 the name of a function. It is passed to myCalculator() as an argument.

When you pass a function as an argument, remember not to use parenthesis.

console.log("========= js callbacks ==========")

function displayer(){

  console.log("DISPLAYER FUNCTION CALLED")

  console.log("THE SUM IS : "+sum)

}

function calculator(a, b, callbackFunction){

  console.log("CALCULATOR FUNCTION CALLED")

  sum = a + b;

  displayer(sum)

}

calculator(2,3, displayer)

**Why do we use callbacks?**

When doing a complex task, we break that task down into smaller steps. To help us establish a relationship between these steps according to time (optional) and order, we use callbacks.

**Asynchronous JavaScript**

Functions running in parallel with other functions are called asynchronous. A good example is JavaScript setTimeout().

In the real world, callbacks are most often used with asynchronous functions.

When using the JavaScript function **setTimeout()**, you can specify a callback function to be executed on time-out:

Example

setTimeout(myFunction, 3000);  
function myFunction() {  
  document.getElementById("demo").innerHTML = "I love You !!"; // I love you appears after 3 sec automatically  
}

* So basically setTimeout() is calling myFunction as callback after 3 secs. 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.

**Waiting for Intervals:**

When using the JavaScript function **setInterval(),** you can specify a callback function to be executed for each interval:

Example

<!-- live working clock on webpage -->

<h3> live working clock </h3>

<p id="working\_time"></p>

<script>

    setInterval(getTimeFunction, 1000)   // calls the function after each 1 sec

    function getTimeFunction(){

        let d = new Date();

        document.getElementById("working\_time").innerHTML

        = d.getHours()+" : "+d.getMinutes()+":"+d.getSeconds();

    }

</script>

**JS PROMISE**

<https://www.youtube.com/watch?v=2IPw-mWe10U>

A JavaScript Promise object contains both the producing code and calls to the consuming code. A JS promise object guarantees us to do something : if the task is success then it calls the resolve() if the task is failed it calls the reject().

Syntax:

// syntax of promise

let myPromise = new Promise(function(myResolve, myReject) {

    // "Producing Code" (May take some time)

      myResolve(); // when successful

      myReject();  // when error

    });

// "Consuming Code" (Must wait for a fulfilled Promise)

myPromise.then(

    function(value) { /\* code if successful \*/ },

    function(error) { /\* code if some error \*/ }

);

**A JavaScript Promise object can be:**

* **Pending**  Promise is still pending i.e. not fulfilled or rejected yet
* **Fulfilled :** Action related to the promise succeeded
* **Rejected :** Action related to the promise failed
* **Settled:** Promise has fulfilled or rejected

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

While a Promise object is "pending" (working), the result is undefined.

When a Promise object is "fulfilled", the result is a value.

When a Promise object is "rejected", the result is an error object





<https://www.geeksforgeeks.org/javascript-promises/>

**Promise creation using constructor**

var promise = new Promise(function(resolve, reject){

//do something

});

**Parameters**

* + Promise constructor takes only one argument which is a callback function (and that callback function is also referred as anonymous function too).
  + Callback function takes two arguments, *resolve* and *reject*
  + Perform operations inside the callback function and if everything went well then call resolve.
  + If desired operations do not go well then call reject.

**Promises can be consumed by registering functions using *.then* and *.catch* methods.**

**1. then()**   
*then()* is invoked when a promise is either resolved or rejected. It may also be defined as a career which takes data from promise and further executes it successfully.

**Parameters:**  
*then()* method takes two functions as parameters.

1. First function is executed if promise is resolved and a result is received.
2. Second function is executed if promise is rejected and an error is received. (It is optional and there is a better way to handle error using *.catch() method*

**Syntax:**

.then(function(result){

//handle success

}, function(error){

//handle error

})

You cannot access the Promise properties **state** and **result**. You must use a Promise method to handle promises.

The two arguments (resolve and reject) are pre-defined by JavaScript.

/ FLOW OF PROMISE

/\*

1. A NEW PROMISE OBJECT IS CREATED USING new Promise(function(resovle, reject)) constructor

2. the resolve and reject are callback methods, the method that will be called on success and failure

3. when success occur, resolve is called -- the resolve will then look for promise object then part and it will

    call the first method

4. when error/reject occur, reject is called -- the reject will look for 2nd method in promise object and it will

   call the second method

5. below ex when x = 0 then resolve() is called which calls the myDisplayer() with OK value

6. when x != 0 then reject() is called which calls the myDisplayer() with Not Okay! value.

\*/

let my\_promise = new Promise(function(resolve, reject){

    let x = 10;

    if (x == 0){

        resolve("OK")

    }

    else{

        reject("Not Okay!")

    }

});

my\_promise.then(

    function(value){myDisplayer(value);},

    function(error){myDisplayer(error);}

);

function myDisplayer(value){

    console.log("My Displayer Called")

    console.log(value)

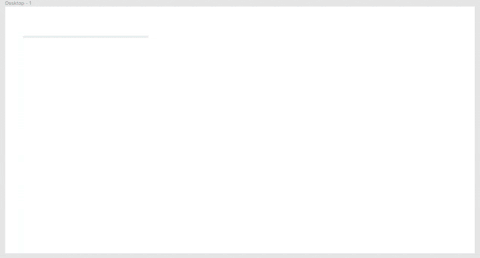
}

We all know that Javascript is a **Synchronous** which means that it has an event loop that allows you to queue up an action that won’t take place until the loop is available sometime after the code that queued the action has finished executing.

**SYNCHRONOUS vs ASYNCHRONOUS**

In a synchronous system, tasks are completed one after another.

Think of this as if you have just one hand to accomplish 10 tasks. So, you have to complete one task at a time.



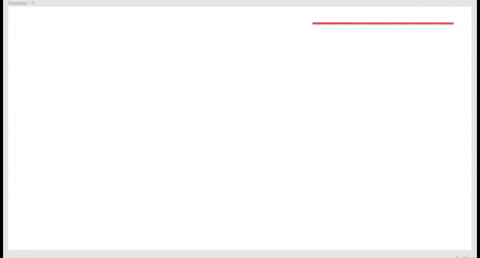
You'll see that until the first image is loaded completely, the second image doesn't start loading.

**What is an Asynchronous System?**

In this system, tasks are completed independently.

Here, imagine that for 10 tasks, you have 10 hands. So, each hand can do each task independently and at the same time.

Take a look at the GIF 👇 – you can see that each image loads at the same time.



**ASYNC**

The keyword async before a function makes the function return a promise: It simply allows us to write promises based code as if it was synchronous and it checks that we are not breaking the execution thread. It operates asynchronously via the event-loop. Async functions will always return a value. It makes sure that a promise is returned and if it is not returned then javascript automatically wraps it in a promise which is resolved with its value.

**AWAIT**

The keyword await before a function makes the function wait for a promise: The await keyword can only be used inside an async function. The keyword await makes JavaScript wait until a promise settles and returns its result.

**ASYNC AND AWAIT** are  better way to write promises and it helps us keep our code simple and clean. All you have to do is write the word async before any regular function and it becomes a promise.

**Promises vs Async/Await in JavaScript**

* Before async/await, to make a promise we wrote this:

function order(){

return new Promise( (resolve, reject) =>{

// Write code here

} )

}

* Now using async/await, we write one like this:

//👇 the magical keyword

async function order() {

// Write code here

}

* **Async / Await in JS -> try, catch**

When we're using async/await, we use this format:

//👇 Magical keyword

async function kitchen(){

try{

// Let's create a fake problem

await abc;

}

catch(error){

console.log("abc does not exist", error)

}

finally{

console.log("Runs code anyways")

}

}

kitchen() // run the code

**Small note**

When using Async/ Await, you can also use the .then, .catch, and .finally  handlers as well which are a core part of promises.

<https://www.freecodecamp.org/news/javascript-async-await-tutorial-learn-callbacks-promises-async-await-by-making-icecream/>

// ice cream shop using async and await

let is\_shop\_open = true;

function time(ms) {

   return new Promise( (resolve, reject) => {

      if(is\_shop\_open){

          // console.log("SHOP IS OPEN AND PROMISE RESOLVED")

         setTimeout(resolve,ms);

      }

      else{

         reject(console.log("Shop is closed"))

      }

    });

}

async function kitchen(){

    try{

    await time(2000) // calling L103 time() function with 2000 as argument

    console.log(`${stocks.Fruits[0]} was selected`)

    await time(0000)

    console.log("production has started")

    await time(2000)

    console.log("fruit has been chopped")

    await time(1000)

    console.log(`${stocks.liquid[0]} and ${stocks.liquid[1]} added`)

    await time(1000)

    console.log("start the machine")

    await time(2000)

    console.log(`ice cream placed on ${stocks.holder[1]}`)

    await time(3000)

    console.log(`${stocks.toppings[0]} as toppings`)

    await time(2000)

    console.log("Serve Ice Cream")

    }

    catch(error){

     console.log("customer left")

    }

}

 // Trigger

 kitchen();

sdafs