What is JavaScript?

JavaScript is a high-level interpreted programming language initially created to make web pages interactive.

In a web page, HTML is used to structure the page, CSS is used to add styling like colors, fonts, etc but a web page with only HTML and CSS is static, meaning a user cannot really interact with the page, here is where Javascript comes in.

Javascript is a cross-platform, interpreted, object-oriented, just-in-time compiled scripting language.

It is used to make web pages interactive(for example if the user clicks a button then the theme of the website changes, or after scrolling the page a pop-up message may be displayed).

It is used to make dynamic web pages.

JavaScript has a standard library of objects, such as Array, Date, and Math, and a fundamental set of language elements such as operators, control structures, and statements.

Javascript can be used for both client and server side programming.

HTML
Heading
Article
Read more
HTML + CSS
Heading
Article
Read more
HTML + CSS + JS
Heading
Article
Read more

Applications of JavaScript

Excellent user interactivity: When you search for something you see that the search suggestions appear, when you fill a form and click submit, the form is submitted all these are because of Javascript, it allows users to interact with the webpage allowing us to build amazing websites and web apps.

Web and Mobile app creation: Javascript based frameworks like React, Angular, Vue allow us to build robust web applications. Example: Facebook, Netflix have been built using React.Upwork, Paypal with Angular, and so on. We can also use a JS based framework called React Native to build mobile applications. Example: Uber eats, Discord, Pinterest, etc.

Server-side development: We can also use Javascript to develop the back end infrastructure of a web application by using Node.js.Example: Netflix, Medium, Trello, etc. use Node.js on the server side.

Game development: Javascript can also be used to develop browser based games some of them are HexGL, CrossCode, etc.

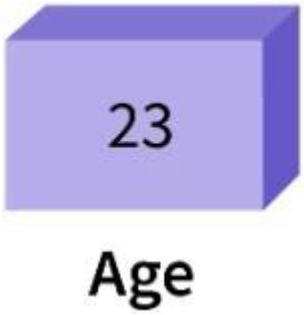
Art: We can also make digital art on the HTML5 canvas by using javascript based libraries.Example.js library

Javascript Variables

In JavaScript, a variable is a name given to a **memory location** that is used to store any *type of data*. As the name suggests, variables mean it can vary, i.e., data stored in a variable can be changed or updated later in the program if required. In short, a variable is a symbolic name for (or reference to) a data or a piece of information.

Highlight: A Javascript variable is a **container**, not a value, this means that variables aren't themselves values; they are a container for values.

Example: Think of a box named **Age** which stores the age of a person. So, the name is the variable name, and the age stored in it is the value.



Rules/Identifiers While Declaring a Javascript Variable

To declare a Javascript variable, one must know the rules to follow, as one can mess up the code while declaring a variable. If one doesn't follow the rules, he/she may end up getting an error.

Rules are as follows:

- **Variables are case-sensitive** in Javascript. This means that schoolName and schoolname are considered different variables.
- We can use letters, digits, symbols like **dollar sign** ($) and **underscore** (_) in a variable name.
- We **cannot** start a variable name with a **digit** (0-9).
- We cannot use any of the reserved keywords (function, return, typeof, break, etc.) of Javascript as a variable name.

Important Points

Javascript allows multiple white spaces and even line breaks in a declaration of a variable.

We can separate different variable declarations using a comma.

In Javascript, we can store any type of value in a variable and also change it any time.

Examples:

1. Declaring a JavaScript variable using line breaks and spaces:

```
var a = 25;
```

2. Declaring two variables in a single line separated by comma:

```
var b=1, c=2;
```

3. Using reserve keywords while declaring a variable:

```
var return = 5; //Will give error, as return is a keyword in Javascript
```

4. Valid variable name examples:

```
var my_school; //using underscore
var one$23; //using digit and dollar sign
```

5. Invalid variable name examples:

```
var 123name ; //variable cannot start with a digit
var variable@ ; //variable cannot contain '@' symbol
var break; //break is a reserved keyword in JS, so we can't use it as a variable
```

Declaring a Variable

One must create a variable before using it. In programming terms, we call this declaring a variable. In Javascript, we can declare a variable using either let, var, or const keywords.

```
var schoolName;
let address;
```

Here we are creating two variables, one using let and another using var.

Currently, these variables **do not store any value**, they are just declared. In other words, we have an empty box named schoolName and address.

If you want to see the value of schoolName, the output will be **undefined**. We will see what is meant by undefined in later part.

Declaring Variables Without the Var Keyword?

We have seen how to declare a variable with var. Now we will see how we can declare a variable without using neither var nor let.

To declare a variable without using any keyword, we have to just write the variable name and must assign a value to that variable. On doing this, the variable becomes a **global variable** (this line means that the scope of the variable is global. The Scope of the variable is described later in this article.).

```
age = 25; //declaring a variable without using var
console.log(age); //output: 25
```

However, it is not recommended to declare a variable without using var as it may alter the value of an already existing global variable.

Undefined vs. Undeclared Variables

Undefined variables

After learning initialization and declaration of variables, you have seen that when we declare a variable without initializing it, then the variable is said to be **undefined**. That is, the variable is declared but not initialized with any value.

Example:

```
var new_variable; //by default its value is undefined.

console.log(new variable); //this will output: undefined
```

Undeclared variables

Suppose you want to use a variable in a statement, but the variable is not declared before. In this case, the code will throw a ReferenceError as the variable is not declared using **var** keyword. In short, if we access any undeclared variable, the code will cause runtime error.

Example:

```
console.log(xyz);
```

Try running the above line in the console, and you will get an error showing that ==ReferenceError: xyz is not defined==. This indicates that **xyz** is not declared earlier you are trying to access an undeclared variable.

JavaScript Variable Hoisting

The most important and unique feature in JavaScript is **Variable Hoisting** or **var Hoisting**. In JavaScript, all variable declarations are processed before any other part of the code is executed. Thus declaring all the variables at the top of the code. This feature is called **hoisting** in JavaScript. The declarations are moved to the top of the global code or to the top of a function, depending on where it is declared. This movement of declarations to the top of the function or global code is done automatically by the JS itself internally. We don't actually see the change. This will be cleared by the examples below. Hoisting also initializes the variable with "**undefined**".

Example:

```
hoistedVariable = 1;
var hoistedVariable;
```

The above code is same as :

```
var hoistedVariable;
hoistedVariable = 1;
```

Note: Hoisting only moves the declaration to the top of the scope(either global or function). It never moves the assignment lines.

Example:

```
console.log(x); //output: undefined
var x = 2;
console.log(x); //output: 2

// The above code is same as:

var x; //default value: undefined
console.log(x); //output: undefined
x = 2;
cosole.log(x); //output: 2
```

Though **hoisting helps to, by default, declare variables at the top, it is highly recommended to declare variables at the top of a function or global manually because one may get confused if he/she uses a variable before declaring it.**

Note: Variables declared using **let** and **const** are also hoisted but they are not initialized with **undefined**. So, if someone tries to use variables before the declaration, then it will throw an exception.

The Difference Between 'var' and 'let'

You might be wondering what is let, as we haven't discussed it in detail. Here we will explain the difference between var and let. First of all, both keywords are used to declare variables in JavaScript.

- The main difference between var and let is that the scope of the variable that is defined by **let** is limited to the block in which it is declared, whereas the variable declared using **var** has the global scope, i.e., can be used throughout the code.
- If we declare a variable using **var** outside of any block (i.e., in the global scope), then the variable gets added to the **window** object, whereas variables declared with **let** will never get added to it.
- We cannot declare the same variable multiple times if one of them is declared using **let**, whereas we can declare the same variable any number of times using **var**.
- Variable hoisting can be done using **var**, but hoisting cannot be done using **let**.

Try the below codes in your browser's console and you will understand it better.

Example 1:

```
let x = 1; //doesn't get added to the window object
var y = 2; //gets added to the window object

console.log(x); // output: undefined
console.log(y); // output: 2
```

Example 2:

```
function scopes(){
var x = 2;
let y = 3;
}
```

```
console.log(x); // will result in a referenceError.
console.log(y); // will result in a referenceError.
```

In this case, the scope of the variables x and y is only limited to the function, that's why we cannot find the reference to the variables x and y outside the function, thus resulting in the reference error. We will describe the Variable Scope in the later part of this article.

Changing the Value/Updating a Variable

Once a variable is declared or initialized, we can change its value anytime and anywhere within its scope. It is similar to re-initializing the variable. We can update/change the value by just typing the variable name followed by an equals sign and then followed by the new value we want it to store.

Example:

```
var variable = 10;// at first the value is 10

variable = 15; // the value gets updated to 15
variable = 20; // now the value is 20 for the variable
```

JavaScript Variable Scope

Scope of a variable means, **the visibility of a variable**. That is, the parts of our program where you can use a variable.

In JavaScript, there are only two types of variable scopes:

1. Local Scope
2. Global Scope

JavaScript local variable

A Local Variable is only visible **inside a function** where it is defined. We cannot use a local variable outside the function where it is defined. On doing so, it will result in reference error.

```
function prints()
{
    var local_var = 2;// a local variable with value 2
    console.log(local_var);
    //the local_var can be used anywhere inside this function
}
console.log(local_variable);//this line will result in ReferenceError,
                    //as local_variable is not visible in this line
```

JavaScript global variable

A Global Variable has a global scope which means that it can be used and viewed anywhere throughout the program.

For example: We can use a globally declared variable inside a function.

```
var global_var = 1;

function prints()
{
    console.log(global_var); //output: 1
}
console.log(global_var); //output: 1
```

Since the global_var is declared globally, it can be used inside the function as well as outside the function (i.e. in the global scope).

Javascript Dynamic Typing

You might have heard that JavaScript is called "a dynamically typed language". This means that the data type of a variable is not fixed. A variable can store any type of data, and the user can achieve this without specifying the type of data he/she wants to store(eg: number, string, objects, arrays, etc).

Also, one can change the stored value to any type of data, anytime freely.

Examples:

```
var variable = "Hello World!";

//lets print the data type of this variable
console.log(typeof(variable)); //this will print 'string'

//Now let's update the value to a number
variable = 2; //By this line, we update the value of variable to 2


//Now let's print the data type of updated variable
console.log(typeof(variable)); //This will print: 'number', since 2 is a
number
```

JavaScript Constants

You have seen that variables are those, whose values could be changed if required. Now, JavaScript also lets you declare constants. We declare a constant using the keyword const;

Syntax:

```
const <variable-name> = <value>;
```

These constants are similar to variables but have some properties which are different from variables, such as:

The constant must be initialized when you declare it, otherwise, it will throw a " SyntaxError: Missing initializer in const declaration ".

Once assigned a value to a constant, you can’t re-assign values afterward. That is, a value is fixed to a constant.

Example 1:

```
var ages;//we can declare a variable without initializing it.
const age; //It will throw an error as we haven't initialized the constant.
```

Example 2:

```
var age = 20;
age = 25; //We can update values of variables like this

const pi = 3.142;
pi = 5; // This line will throw a TypeError: Assignment to constant variable.
```

Difference Between Var, Let, and Const in Javascript

The following table briefs the difference between let and var and const in javascript:

var	let	const
var has the function or global scope.	let's have the block scope.	const variable has the block scope.
It gets hoisted to the top of its scope and initialized undefined.	It also got hoisted to the top of its scope but didn't initialize.	It also got hoisted to the top of its scope but didn't initialize.
It can be updated or re-declared.	It can only be updated and can't be re-declared.	It can't be updated or re-declared.
It's an old way to declare a variable.	It's a new way to declare variables introduced in ES6.	It's also a new way to declare a variable, which introduces in ES6.

Data Types in JavaScript

Data types in JavaScript refer to the attribute associated with the kind of data we are storing or working on. It associates the declared variable (or item) with a particular type so that the compiler can perform operations in an organized manner.

There are mainly two types of data types in JavaScript:

1. Primitive data types.
2. Non-primitive data types.

Primitive Data Types

Primitive data types in javascript refer to those data types that are defined on the most basic level of the language. These Javascript data types cannot be destructured further, nor do they have any pre-defined properties or methods. They are only used to store items of their type.

There are 7 primitive data types in JavaScript:

1. Numbers
2. BigInt
3. String
4. Boolean
5. Undefined
6. Null
7. Symbol

Following is the table for all the primitive data types in javascript with their descriptions and examples.

Data Type	Description	Example
undefined	Represents an undefined value.	let x; // x is undefined
null	Represents an null value.	let y = null; // y is null
boolean	Represents a boolean value, which can be true or false	let a = true; // a is true
number	Represents a numeric value, either integer or floating point.	let b = 5; // b is 5
string	Represents a sequence of characters, enclosed in quotes.	let c = "hello"; // c is "hello"
symbol	Represents a unique identifier, used primarily for object properties.	let d = Symbol("description"); // d is a unique symbol
BigInt	Represents integers with arbitrary precision.	let e = 9007199254740992n; // e is a BigInt

1. Numbers Type

In javascript, the integers, float values (or decimals), and all the other numeric values are represented by the **number datatype**.

Unlike other languages (like C++), javascript doesn't define separate data types for integers, decimals, long integers, etc.

Numbers can be declared by using *var*, *let*, or *const* keyword followed by the *variable name*, and the number is assigned to it.

Example:

```
let   num1 = 10;              //number with integer of value 10
var   num2 = 9753123568;     //number with integer of value 9753123568
const num3 = 3.14;          //number with decimal of value 3.14
```

2. BigInt Type

The BigInt data type in javascript is used to represent numeric values that exceed the *largest safe integer limit*. It is basically used to represent numbers greater than 253−1.

Note: Unlike *number*, the *BigInt* data type doesn't represent decimal values. It only represents whole numbers.

Declaration:

BigInt values can be declared by two methods:

By appending n at the end of the numeric value.

By passing the number as an argument to the BigInt() constructor.

```
    var num = 9007199254740991n;
    var num = BigInt(9007199254740991);
```

Example:

```
const num1 = 112233445566778899233445566778894556646n; //BigInt number
declared by appendening n.
const num2 = BigInt(112233445566778899233445566778894556646); //Bignint
number declared by calling constructor
const num3 = BigInt('112233445566778899233445566778894556646'); //BigInt
number declared by calling constructor where argument passed is a string.
```

3. String Type

String type is used to store elements in a textual form. The elements are stored in *16-bit integer form.*

Declaration:

Strings are declared by assigning the text to the declared literals. The literal can be declared by either *const*, *var*, or *let*. In Javascript, a string can be declared in the following three ways:

a. **Single tick declaration:** *Strings are declared by enclosing the text between two single ticks ('')*

```
    var str = 'Javascript';
```

b. **Double tick declaration:** *Strings are declared by enclosing the text between two double ticks ("")*

```
    var str = "Javascript";
```

c. **Backticks declaration:** *Strings are declared by enclosing the text between two backticks ticks (``)*

```
    var str = `Javascript`;
```

Example:

```
const str1 = 'This is String';        // single tick declaration
const str2 = "Javascript";           // double tick declaration
const str3 = `This is an example`; //backtick declaration
```

4. Boolean Type

In javascript, the *boolean* data type is used to check the truthy or falsy condition. It is also known as *logical data type*.

```
bool flag = false;
```

5. Undefined Type

In javascript, undefined is a primitive data type in javascript that gets assigned to *variables* or *function arguments* when their values aren't initialized.

Declaration:

undefined data types in JavaScipt can be declared by simply declaring a literal by using *var*, *let*, or *const* keyword. *We don't assign anything to the literal.*

Syntax:

```
var undefinedValue;
```

Example:

```
function example(x){
    console.log('The user has entered: ' + x);
}

var a;
var b = 10;
var c = 'Javascript';

example(a);
//output: The user has entered: undefined

example(b);
//output: The user has entered: 10

example(c);
//output: The user has entered: Javascript
```

6. Null Type

The *null* type in javascript refers to an empty object pointer. It holds only one value. It is used to represent intentional absence **of any object.**

Declaration: **It is declared by declaring a variable using *var*, *let* or *const* keyword and it is assigned the value null.**

Syntax:

```
var a = null;
```

Example:

```
function example(x){
    console.log(x);
}

var a = 10;
var b = null;

example(a);
//output: 10

example(b);
//output: null
```

7. Symbol Type

The symbol is a primitive data type in javascript that returns a unique symbol upon being called. Symbols do not have a literal form and are unique.

Declaration:

The symbol is declared by calling the **Symbol()** function. The parameter field doesn't default; thus it can be called either by passing an argument or empty.

Note: the Symbol() isn't a constructor thus, if it is called with the **new** keyword, it will throw an error.

Syntax:

```
let s1 = Symbol();
let s2 = Symbol('javascript');
```

The Symbol() creates a new symbol everytime it is called and the symbol has nothing to do with the *key* passed.

Example:

```
let s1 = Symbol('x');
let s2 = Symbol('x');

console.log(s1===s2);
// output: false
```

In the above example, even though the key passed is same still the symbols *s1* and *s2* will be different.

Non-primitive Data Types in JavaScript

Non-primitive data types are those data types that aren't defined at a basic level but are complex data types formed upon operations with the primitive data types. The non-primitive data types in javascript refer to objects and can perform multiple functions using their methods.

There are mainly 3 types of non-primitive (or complex) data types in JavaScript:

- **Object**
- **Array**
- **RegExp**

Following is the table for all the non-primitive data types in javascript with their descriptions and examples.

Data Type	**Description**	**Example**
object	Represents a collection of key-value pairs, or properties, where each key is a string and each value can be any data type, including other objects.	let obj = {name: "John", age: 30}; // obj is an objectd
array	Represents a collection of elements, where each element can be any data type, including other arrays and objects. Arrays are also objects in JavaScript.	let arr = [1, 2, "three"]; // arr is an array
RegExp	Represents a regular expression, which is a pattern used to match character combinations in strings.	let pattern = /[a-z]+/; // pattern is a regular expression

Object Type

Object in javascript is a collection of data that is kept in a *key-value* pair manner.

Declaration:

Objects in javascript can be created by two methods:

1. **Literal method**
2. **By calling the constructor**

Example:

```
//By literal method
let car = {
    name: "Maruti",
    model: "Swift",
    color: "red",
    price: 550000
}

//By constructor method
let bike = new Object();
bike.name = "Pulsar";
bike.model = "Dominar";
bike.color = "red";
bike.price = 200000;
```

In a literal declaration, the key-value pair is separated by commas ,

Array Type

Arrays in javascript are abstract data types that are used to store a set of elements. These can be multiple elements of the same or different types. Arrays are basically containers in javascript.

Declaration:

Javascript arrays can be declared by three methods:

1. **By array literal**
2. **Constructor method**

Example:

```
// By literal
let X = ['Red', 'Yellow', 'Orange'];

//By constructor
let Y = new Array('Apple', 'Orange', 'Grapes', 'Banana');

console.log(X);
// 'Red', 'Yellow', 'Orange'

console.log(Y);
// 'Apple', 'Orange', 'Grapes', 'Banana'
```

Note: In javascript, the elements of an array can be of different types

Example:

```
let ar = [1, 'Pen', false, 'Text'];
```

RegExp Type

We all have seen detective movies; remember how words are decoded and read using patterns? Even in real life, we encounter many instances where we need to validate the texts with particular patterns.

The **RegExp** in javascript is an object which is used to match a string with a particular pattern.

Declaration:

Regular Expressions (RegExp) can be created by two methods:

1. **By using forward slashes**
2. **By Constructor**

Example:

```
//forward slash declaration
let p1 = /javascript/i;

//constructor declaration
let p2 = new RegExp('/javascript/i');
```

Here p1, and p2 are regular expressions for javascript.

Here the **/javascript/** part is the RegExp and **i** is the modifier.

Conditional Flow in JavaScript

Ternary Operator in Javascript is a special operator which has **three operands**. In JavaScript, there is only one such operator, and that is the **Conditional Operator** or the **Question Mark Operator**(**?:**)

This operator is used to write a conditional block in a simpler form and also can be written inside an expression or a statement; that's why this operator is also known as the **inline-if** operator.

Syntax

```
condition ? expression_if_true : expression_if_false
```

The ternary operator is basically a combination of a question mark and a colon. To use this operator, you need to write a **conditional expression**, followed by a question mark (?).

Then you need to provide two statements or expressions separated by a colon. The first expression (**expression_if_true**) evaluates when the condition results in a **true** value, and the second expression (**expression_if_false**) evaluates when the condition results in a **false** value.

Ternary Operator Examples

Using the Ternary Operator, we can write the above block in just one line.

```
var marks = 50;
console.log(( marks >= 40 ) ? "Passed" : "Failed" );
```

The output is:

```
Passed
```

JavaScript if, else, and else if Statement

The if Statement in JavaScript

The if statement in javascript executes the statements inside it only if the condition that is mentioned is *true*.

Note: if statement can be executed *independently*, i.e. without any else statement.

Syntax:

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

Example 1:

```
function example(x){
    if(x){
        console.log('If statement is executed');
    }
}

//Example 1: when condition statement is a boolean value
bool flag1 = true;
bool flag2 = false;

example(flag1);
//output: If statement is executed

example(flag2);
//This won't produce any output

//Example 2: when condition statement is a comparison
let a = 10;
let b = 100;
let c = 10;

example(a==b);
//This won't produce any output

example(a==c);
//output: If statement is executed
```

Example 2:

```
function checkEligibility(age){
    if(age<18){
        console.log('Sorry! You are not eligible for voting');
    }

    if(age>=18){
        console.log('You are eligible for voting');
    }
}

checkEligibility(11);
// Sorry! You are not eligible for voting

checkEligibility(34);
// You are eligible for voting
```

The if...else Statement in JavaScript

We have already discussed how the if statement executes only when the condition is true. But what if the if statement fails? In such cases, we can use the ***else*** *statement.*

else statement in javascript is used to execute a block when all if conditions fail. In order to write an else block, it is mandatory to declare an if statement, although the vice versa is not true.

Note: In if else in Javascript, the else statement can **not** be executed *independently*, i.e. it is mandatory to write an if statement before an else statement.

Syntax:

```
if(condition){
    //code to be executed
}else{
    //code to be executed
}
```

Example:

```
function example(x){
    if(x){
        console.log('If statement is executed');
    }else{
        consoel.log('Else statement is executed');
    }
}

//Example 1: when condition statement is a boolean value
bool flag1 = true;
bool flag2 = false;

example(flag1);
//output: If statement is executed

example(flag2);
//output: Else statement is executed

//Example 2: when condition statement is a comparison
let a = 10;
let b = 100;
let c = 10;

example(a==b);
//output: Else statement is executed

example(a==c);
//output: If statement is executed
```

Example 2:

```
function checkEligibility(age){
    if(age<18){
        console.log('Sorry! You are not eligible for voting');
    }else{
        console.log('You are eligible for voting');
    }
}

checkEligibility(11);
// Sorry! You are not eligible for voting
```

```
checkEligibility(34);
// You are eligible for voting
```

Switch Case in JavaScript

In javascript, situations, where an expression produces different outputs with respect to different cases, can be handled by switch case.

Switch case in javascript can be defined as an equivalent of an if-else statement. The switch statement takes an expression and evaluates it with respect to other cases, and then outputs the values associated with the case that matches the expression.

If no case passes the expression, then the switch statement will execute the default code.

When to Use The Switch Statement

Switch case in javascript can be used at places where our output needs to be dependent on the value an expression holds.

Syntax

```
switch (expression) {
  case caseValue1:
    //code a
    break;
  case caseValue2:
    //code b
    break;
  default:
    //code c
    break;
}
```

Examples

```
function groceryPrice(exp){
 switch (exp) {
  case 'Cookies':
    console.log('Cookies cost 100 rupees');
    break;
  case 'Milk':
    console.log('Milk cost 60 rupees');
    break;
  case 'Fruits':
    console.log('Fruits cost 300 rupees');
    break;
  case 'Corn Flakes':
    console.log('Corn Flakes cost 150 rupees');
    break;
  default:
    console.log(exp + ' is not available right now');
  }
}


groceryPrice('Cookies');
//output: Cookies cost 100 rupees

groceryPrice('Fruits');
//output: Fruits cost 300 rupees

groceryPrice('Peanut');
//output: Peanut is not available right now
```

Break Statement

In the switch case, the break keyword is used associated with the *cases* to terminate the switch case once the expression matches that particular case.

Switch vs if…else

If else	**Switch case**
Its input is an expression that can be a specific condition or a range	It's input is an expression that can be an enumerated value of a string object
It has less readable syntax	It has more readable syntax
Multiple if blocks can not be processed unless each condition is true	Multiple cases can be processed if we remove the break statement
It tests equality as well as logical expressions	It tests only equality.
It is relatively slow for large values	It works fast for large values because the compiler creates a *jump table*

Loops in JavaScript

While loops in JavaScript

A **while loop** executes the respective statements as long as the conditions are met. As soon as the condition becomes false, the loop is terminated, and the control is passed on to the statement outside the loop. It is important to note that **while** is an entry-controlled loop in JavaScript.

Here is the basic syntax of a while loop

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

Example of while loops in JavaScript

Let us print the numbers 1 to 10 using the while loop in JavaScript.

```
var i = 1;
while (i <= 10) {
 console.log(i);
 i++;
}
```

Do...while loops in JavaScript

A **do...while loop** is an exit-controlled loop. This means that the do...while loop executes the block of code once without checking the condition. After the code has been executed once, the condition is checked. If the condition is true, the code is executed further. If the condition is false, the loop is terminated.

The do...while statement is used when you want to run a loop at least one time.

Here is the basic syntax for a typical do...while loop.

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

Example of Do...while loops in JavaScript

Let us write a simple code in JavaScript to print numbers from 1 to 10.

```
let i = 1;
do {
    console.log(i);
    i++;
} while(i <= 10);
```

For loops in JavaScript

For loops in JavaScript are one of the most commonly used loop constructs.

A **for loop** is an entry-controlled loop that repeats a block of code if the specified condition is met. A for loop in JavaScript is typically used to loop through a block of code multiple times.

Syntax

```
for (initialising statement; conditional statement; updation statement) {
  // code block to be executed
}
```

Example of For Loops in JavaScript

Let us write a simple program to print the numbers 1 to 10 in JavaScript using for loops.

```
for (var i=1;i<=10;i++){
console.log(i)
}
```

For...in Statements

A for...in the statement is used to iterate over those properties of objects that have been keyed. We can say in crude terms that they have been 'indexed.' This means that each time the loop iterates, a key is assigned to a variable named *key*. This key is used to display the properties of the object it is linked to. This is like a for loop, but it only iterates over enumerable properties keyed by strings or arrays.

Here is the basic syntax for a *for...in* statement:

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

Example of for...in in JavaScript

```
const patient = {
    name: 'Isabel',
    height: 164, //in cm
    weight: 60,  //in kg
    disease: 'hypertension'

}
for ( let key in patient) {
    console.log(`${key} => ${patient[key]}`);
}
```

The a bove code is used to print the details of the patient Isabel by viewing her health record. The for...in statement iterates over the object *patient* and displays the records in it. The object key is assigned the variable 'key'. The value of the key could be accessed by the **patient[key]**

For...of Statement in JavaScript

A for...of statement is like a counterpart of the *for...in* statement. Except iterating through the properties of the enumerable object, it loops through its values.

Here is the basic syntax for a *for...of statement*

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

Example of for...of in JavaScript

```
const patients = ['Isabel', 'Marie', 'Skylar'];

for ( let the element of patients ) {
    // print
    console.log(element);
}
```

The above code is used to print the patients linked to a hospital's health records. Here, *patients* is the iterable object (array, in our case). *Element* is an item in the iterable object. For every element in the iterable object '*patients*', the body of the loop is executed.

Function declarations

In JavaScript, functions are blocks of reusable code that can be defined and invoked for performing specific tasks. There are two common ways to declare functions in JavaScript: function declarations and function expressions.

A function declaration has the following syntax:

```
function functionName(parameters) {
  // Function body: code to be executed
  // ...
  return something; // optional
}
```

Here's an example of a function declaration:

```
// Function declaration
function greet(name) {
  return "Hello, " + name + "!";
}
// Function invocation
var message = greet("Javascript");
console.log(message); // Output: Hello, Javascript!
```

In this example, the greet function takes a name parameter and returns a greeting message containing the provided name. The function is then invoked with the argument “Javascript” and the result is stored in the message variable, which is subsequently printed to the console.

Function declarations are hoisted in JavaScript, which means they can be called before they are defined in the code. This behavior can be helpful, but it's still a good practice to define your functions before invoking them to improve code readability.

Remember that in addition to function declarations, you can also use function expressions, arrow functions, and various other approaches to define functions in JavaScript. Each has its own use cases and syntactic differences.

Function parameters

Function parameters are placeholders for values that you can pass to a function when you invoke it. Parameters allow you to make your functions more flexible and reusable by allowing them to work with different inputs. Here's how you can use function parameters with examples:

```
// Function declaration with parameters
function addNumbers(a, b) {
  return a + b;
}
// Function invocation with arguments
var sum = addNumbers(5, 3);
console.log(sum); // Output: 8

var anotherSum = addNumbers(10, 7);
console.log(anotherSum); // Output: 17
```

In this example, the addNumbers function takes two parameters, a and b. When the function is invoked, you provide actual values for these parameters (arguments) inside the parentheses. The function then adds the provided numbers and returns the result.

You can have any number of parameters in a function declaration, separated by commas:

```
function multiply(a, b, c) {
  return a * b * c;
}
var product = multiply(2, 3, 4);
console.log (product); // Output: 24
```

It's worth noting that JavaScript is a dynamically typed language, so you don't need to specify the types of parameters when declaring functions. You simply use the parameter names in the function body, and their values will be automatically assigned when the function is called.

Default Parameters:

You can also provide default values for parameters, which are used when no value is passed for that parameter during function invocation:

```
function greet(name = "Guest") {
  return "Hello, " + name + "!";
}

console.log(greet());       // Output: Hello, Guest!
console.log(greet("Bhavana")); // Output: Hello, Bhavana!
```

In this example, the greet function has a default parameter value of "Guest". If no argument is provided, the default value is used. If an argument is provided, it overrides the default value.

Rest Parameters:

Rest parameters allow you to pass a variable number of arguments to a function as an array:

```
function calculateSum(...numbers) {
  let sum = 0;
  for (const number of numbers) {
    sum += number;
  }
  return sum;
}
console.log(calculateSum(1, 2, 3));       // Output: 6
console.log(calculateSum(10, 20, 30));    // Output: 60
```

In this example, the calculateSum function uses the rest parameter ...numbers to capture all the provided arguments as an array. It then calculates and returns the sum of those numbers.

These are some common ways to use function parameters in JavaScript. Parameters make your functions more flexible and adaptable to different inputs, improving code reuse and organization.

Functions as Objects

In JavaScript, functions are first-class objects, which mean they can be treated like any other value or object. This property enables you to assign functions to variables, pass them as arguments to other functions, and even store them in data structures like arrays and objects. Here's an example of using functions as objects:

```
// Assigning a function to a variable
var greet = function(name) {
  return "Hello, " + name + "!";
};
console.log(greet("Dhatrika")); // Output: Hello, Dhatrika!
```

In this example, the function greet is assigned to the variable greet. You can then use greet as a regular function by invoking it with an argument.

Functions can also be passed as arguments to other functions, allowing for powerful patterns like callbacks:

```
function calculate(a, b, operation) {
  return operation(a, b);
}
function add(x, y) {
  return x + y;
}
function subtract(x, y) {
  return x - y;
}
console.log(calculate(5, 3, add));       // Output: 8
console.log(calculate(10, 7, subtract)); // Output: 3
```

In this example, the calculate function takes two numbers and an operation (which is another function) as arguments. It then uses the provided operation function to perform the calculation.

Functions can also be stored in arrays or objects:

```
var mathFunctions = [add, subtract];
console.log(mathFunctions[0](3, 2)); // Output: 5
var operations = {
  addition: add,
  subtraction: subtract
};

console.log(operations.addition(8,  4));       // Output: 12
console.log(operations.subtraction(15,  7));   // Output: 8
```

Here, an array mathFunctions and object operations are created to store function references. You can access and invoke the stored functions using array index or object property syntax.

Functions can also be returned from other functions, creating closures:

```
function createMultiplier(factor) {
  return function(number) {
    return number * factor;
  };
}
var double = createMultiplier(2);
console.log(double(5)); // Output: 10
```

In this example, the createMultiplier function returns a new function that multiplies a given number by the provided factor. The returned function captures the factor value in a closure, allowing you to create custom multiplication functions.

In summary, JavaScript's treatment of functions as first-class objects is a powerful feature that enables you to write more flexible and modular code. You can assign functions to variables, pass them as arguments, store them in data structures, and even return them from other functions.

Anonymous functions

Anonymous functions, also known as "function expressions," are functions that are defined without a name. They are often used when you need a function for a specific task but don't necessarily need to reuse it elsewhere. Here's an example of using anonymous functions in JavaScript:

```
// Anonymous function assigned to a variable
var greet = function(name) {
  return "Hello, " + name + "!";
};
```

```
console.log(greet("Nirmith")); // Output: Hello, Nirmith!
```

In this example, the anonymous function is assigned to the variable greet. You can then invoke the function using the greet variable.

Anonymous functions are commonly used when passing functions as arguments to other functions. For instance, with the Array.prototype.map function:

```
var numbers = [1, 2, 3, 4, 5];
var squaredNumbers = numbers.map(function(number) {
  return number * number;
});
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

In this example, the anonymous function is used as an argument to the map function. The anonymous function calculates the square of each number in the numbers array, and the result is stored in the squaredNumbers array.

You can also use arrow functions as a more concise syntax for anonymous functions:

```
var numbers = [1, 2, 3, 4, 5];
var squaredNumbers = numbers.map(number => number * number);
console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

Arrow functions are a shorthand for writing anonymous functions, and they automatically capture the surrounding this value.

Anonymous functions are useful when you have a simple task that you want to define inline, without giving the function a permanent name. They're often used for callback functions, array methods, and other situations where a short-lived function is needed.

Arrow functions

Arrow functions are a concise way to write anonymous functions in JavaScript. They provide a shorter syntax and automatically capture the value of this from the surrounding context. Here are some examples of using arrow functions:

Basic Arrow Function:

```
// Regular function
function add(a, b) {
  return a + b;
}

// Equivalent arrow function
const addArrow = (a, b) => a + b;
console.log(addArrow(3, 5)); // Output: 8
Arrow Function with One Parameter:
// Regular function
function square(x) {
  return x * x;
}

// Equivalent arrow function
const squareArrow = x => x * x;
console.log(squareArrow(4)); // Output: 16
```

Arrow Function with No Parameters:

```
// Regular function
function greet() {
  return "Hello!";
}

// Equivalent arrow function
const greetArrow = () => "Hello!";
console.log(greetArrow()); // Output: Hello!
Arrow Function as Callback
const numbers = [1, 2, 3, 4, 5];

// Using regular function
const squaredNumbers = numbers.map(function(number) {
  return number * number;
});

// Using arrow function
const squaredNumbersArrow = numbers.map(number => number * number);
console.log(squaredNumbers);        // Output: [1, 4, 9, 16, 25]
console.log(squaredNumbersArrow);  // Output: [1, 4, 9, 16, 25]
```

Arrow Function with this Context

```
// Regular function using a traditional approach
const person = {
  firstName: "Alice",
  lastName: "Johnson",
  getFullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
console.log(person.getFullName()); // Output: Alice Johnson

// Arrow function capturing 'this' from the surrounding context
const personArrow = {
  firstName: "Alice",
  lastName: "Johnson",
  getFullName: () => this.firstName + " " + this.lastName
};
console.log(personArrow.getFullName()); // Output: undefined undefined
```

Keep in mind that arrow functions have a few differences from regular functions, such as their handling of the this keyword, absence of an arguments object, and lack of hoisting. They are particularly useful for concise one-liner functions, especially in situations where you don't need to create a new this context within the function.

Asynchronous Functions

JavaScript supports asynchronous programming through callbacks, promises, and async/await syntax, enabling non-blocking operations.

The **async** keyword is added to functions to tell them to return a promise. The await keyword is used to pause async function execution until a promise is settled.

```
async function fetchData(url) {
	const response = await fetch(url);
	const data = await response.json();
	console.log(data);
 }
```

Variable scopes

In JavaScript, variable scopes determine where a variable is accessible and where it's not. There are two main types of variable scopes: global scope and local scope (also known as function scope or block scope). Let's explore these concepts with examples:

Global Scope:

Variables declared outside of any function or block have global scope. They can be accessed from any part of the code.

```
var globalVariable = "I'm global";
function printGlobal() {
  console.log(globalVariable);
```

```
}
printGlobal(); // Output: I'm global
```

In this example, the variable globalVariable is accessible both inside the function printGlobal and outside it, because it's declared in the global scope.

Local Scope (Function Scope):

Variables declared inside a function are limited to that function's scope. They are not accessible from outside the function.

```
function localScopeExample() {
  var localVariable = "I'm local";
  console.log(localVariable);
}
localScopeExample(); // Output: I'm local
// Trying to access localVariable outside its scope
console.log(localVariable); // Output: ReferenceError: localVariable is
not defined
```

In this example, the variable localVariable is declared inside the localScopeExample function, so it's accessible only within that function.

Block Scope (Introduced by ES6):

Variables declared using let and const have block scope, which means they are limited to the block in which they are defined (e.g., within loops or conditional statements).

```
if (true) {
  var x = 10; // Not block-scoped
  let y = 20; // Block-scoped
  const z = 30; // Block-scoped
}
console.log(x); // Output: 10
console.log(y); // Output: ReferenceError: y is not defined
console.log(z); // Output: ReferenceError: z is not defined
```

In this example, the variable x is accessible outside the block, while y and z are only accessible within the block.

Nested Scopes:

Scopes can be nested; meaning a variable declared in an outer scope is accessible in inner scopes, but not the other way around.

```
function outer() {
  var outerVariable = "I'm outer";
  function inner() {
    var innerVariable = "I'm inner";
    console.log(outerVariable); // Accessible
  }
  inner();
  console.log(innerVariable); // Not accessible
}
outer();
```

In this example, the outerVariable declared in the outer function is accessible within the inner function, but the innerVariable declared in the inner function is not accessible outside its scope.

Understanding variable scopes is crucial for writing organized and maintainable code in JavaScript. It helps you manage variable visibility, prevent accidental name collisions, and control access to data.

Built in functions

JavaScript provides a variety of built-in functions that offer useful functionality without needing to create them from scratch. These functions cover a wide range of tasks, from manipulating strings and arrays to performing mathematical calculations and working with dates. Here are some examples of built-in functions in JavaScript:

String Functions:

```
var text = "Hello, world!";
console.log(text.length);          // Output: 13
console.log(text.toUpperCase()); // Output: HELLO, WORLD!
console.log(text.indexOf("world")); // Output: 7
console.log(text.substring(0, 5));  // Output: Hello
```

Array Functions:

```
var numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3];
console.log(numbers.length);      // Output: 10
console.log(numbers.sort());      // Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]
console.log(numbers.indexOf(5)); // Output: 4
console.log(numbers.reduce((a, b) => a + b, 0)); // Output: 39
```

Math Functions:

```
console.log(Math.sqrt(25));       // Output: 5
console.log(Math.random());       // Output: Random decimal between 0 and 1
console.log(Math.floor(5.7));     // Output: 5
console.log(Math.ceil(5.2));      // Output: 6
console.log(Math.round(5.5));     // Output: 6
```

Date Functions:

```
var currentDate = new Date();
console.log(currentDate.getFullYear());// Output: Current year
console.log(currentDate.getMonth());    // Output: Current month (0-11)
console.log(currentDate.getDate());     // Output: Current day of the month
console.log(currentDate.getDay());      // Output: Current day of the week
(0-6, where 0 is Sunday)
```

Regular Expression Functions

```
var regex = /\d+/;
console.log(regex.test("123abc"));    // Output: true (matches digits)
console.log("Hello 123 World".match(regex)); // Output: ["123"] (array of
matches)
console.log("Hello".replace("e", "a"));      // Output: Hallo
```

These are just a few examples of the many built-in functions available in JavaScript. The built-in functions save you time and effort by providing pre-built solutions to common programming tasks.

Arrays in JavaScript

In JavaScript, an array is a user-defined or non-primitive data type used to store a collection of elements in a single variable.

Creating Arrays :-

Different ways to create arrays are

1. Array literals
2. The ... spread operator on an iterable object
3. The Array() constructor
4. The Array.of() and Array.from() factory methods

1. Array literals :-

Using an array literal is the easiest way to create a JavaScript Array. which is simply acomma-separated list of array elements within square brackets.

Syntax:

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

For example:

```
let empty = []; // An array with no elements
let primes = [2, 3, 5, 7, 11]; // An array with 5 numeric elements
let misc = [ 1.1, true, "a", ]; // 3 elements of various types + trailing
comma
```

The values in an array literal need not be constants; they may be arbitraryexpressions:

```
let base = 1024;
let table = [base, base+1, base+2, base+3];
```

2. Spread Operator

Without spread syntax, to create a new array using an existing array as one part of it, the array literal syntax is no longer sufficient , With spread syntax this becomes much more easy

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

```
const parts = ["shoulders", "knees"];
const lyrics = ["head", ...parts, "and", "toes"];
//  ["head", "shoulders", "knees", "and", "toes"]
```

The spread operator works on any iterable object.

3. The Array() Constructor

Another way to create an array is with the Array() constructor. You can invoke this constructor in three distinct ways:

• Call it with no arguments:

```
let a = new Array();
```

This method creates an empty array with no elements and is equivalent to the array literal [].

- Call it with a single numeric argument, which specifies a length:

```
let a = new Array(10);
```

- Explicitly specify two or more array elements or a single non-numeric element

for the array:

```
let a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

4. **Array.of()**

The Array.of() function addresses this problem of Array() constructor ,it is a factory method that

creates and returns a new array, using its argument values (regardless of how many ofthem there are) as the array elements:

```
Array.of() // => []; returns empty array with no arguments
Array.of(10) // => [10]; can create arrays with a single numeric argument
Array.of(1,2,3) // => [1, 2, 3]
```

5. **Array.from()**

The Array.from() static method creates a new, shallow-copied Array instance from an iterable or array-like object. The JavascriptArray.from() method is used to create a new array instance from a given array.

It is also a simple way to make a copy of anarray:

let copy = Array.from(original);

Example :-

```
let text = "web technology"
const myArr = Array.from(text);
console.log(myArr);
//  output: w,e,b, ,t,e,c,h,n,o,l,o,g,y
```

Array: length Property

The length data property of an Array instance represents the number of elements in that array.

Example :-

```
const clothing = ['shoes', 'shirts', 'socks', 'sweaters'];
console.log(clothing.length);
//  output: 4
```

Reading and Writing Array Elements

An item in a JavaScript array is accessed by referring to the index number of the item in square brackets. A reference to the array should appear to the left of the brackets. An arbitrary expression that has a non negative integer value should be inside the brackets. Same syntax can be used for both read and write the value of an element of an array. Thus, the following are all legal JavaScript statements:

```
let a = ["world"]; // Start with a one-element array
let value = a[0]; // Read element 0
a[1] = 3.14; // Write element 1
let i = 2;
a[i] = 3; // Write element 2
a[i + 1] = "hello"; // Write element 3
a[a[i]] = a[0]; // Read elements 0 and 2, write element 3
```

Example :-

```
var myArray = new Array(4);
myArray[0] = "A";
myArray[1] = "c";
myArray[2] = "s";
myArray[3] = "d";
var i = 3;
var theYear = myArray[i];
console.log("It is the year of the " + theYear + ".");

//  output: It is the year of the d.
```

For loop in array javascript :-

Using the basic for loop :-

The default for loop can be used to iterate through the array and each element can be accessed by its respective index.

```
const scores = [22, 54, 76, 92, 43, 33];
for (let i = 0; i<scores.length; i++) {
    console.log(scores[i]);
}

// output:
22
54
76
92
43
33
```

Using for…in

The for…in loop is an easier way to loop through arrays as it gives us the key which we can now use to get the values from our array this way:

```
for (i in scores) {
    console.log(scores[i]);
}
// output:
22
54
76
92
43
33
```

Using the for…of statement

The for…of statement can be used to loop through iterable objects and perform the required functions. Iterable objects include arrays, strings, and other array-like objects.

```
for (score of scores) {
    console.log(score);
}
// output:
22
54
76
92
43
33
```

Array Destructing :-

Destructuring the array in JavaScript simply means extracting multiple values from data stored in objects and arrays. The destructing assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

Example 1 :

```
// we have an array with the name and surname
let arr = ["John", "Smith"]
//  destructuring  assignment
let [firstName, surname] = arr;
console.log(firstName); // John
console.log(surname); // Smith
```

Example 2 :

```
let [firstName, surname] = "John Smith".split(' ');
console.log(firstName); // John
console.log(surname); // Smith
```

Ignoring Some values

We can also ignore some parts of an array that we are not interested in

Unwanted elements of the array can also be thrown away via an extra comma:

Example:-

```
var a, b;
[a, , b] = ["Person_1", "Person_2", "Person_3"];
console.log(a); //Output: Person_1
console.log(b); //Output: Person_2

// output:
Person_1
Person_3
```

The rest '…'

In array destructuringjavascript, the Rest parameters are generally used for creating a function that can be able to accept many numbers of arguments

An (...) operator" is used as The rest parameter. if it appears on the left-hand side while destructuring. This rest parameter is used to gather all the remaining elements in the array that are not mapped to the rest variable yet. This rest variable should always be used in the last of the code

Example :-

```
var working_days = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
"Saturday"];
var [first, , third, ...others] = working_days;
console.log(first); //Output: Monday
console.log(third); //Output: Wednesday
console.log(others); //Output: ["Thursday", "Friday", "Saturday"]

//  output:
Monday
Wednesday
[‘Thrusday’, ‘Friday’, ‘Saturday’]
```

If the array is shorter than the list of variables at the left, there’ll be no errors. Absent values are considered undefined:

Ex:

```
let [firstName, surname] = [];
console.log(firstName); // undefined
console.log(surname); // undefined
```

Destructuring using function:-

A function can return an array of values. It is always possible to return an array from a function, but array destructuring makes it more concise to parse the returned array from functions.

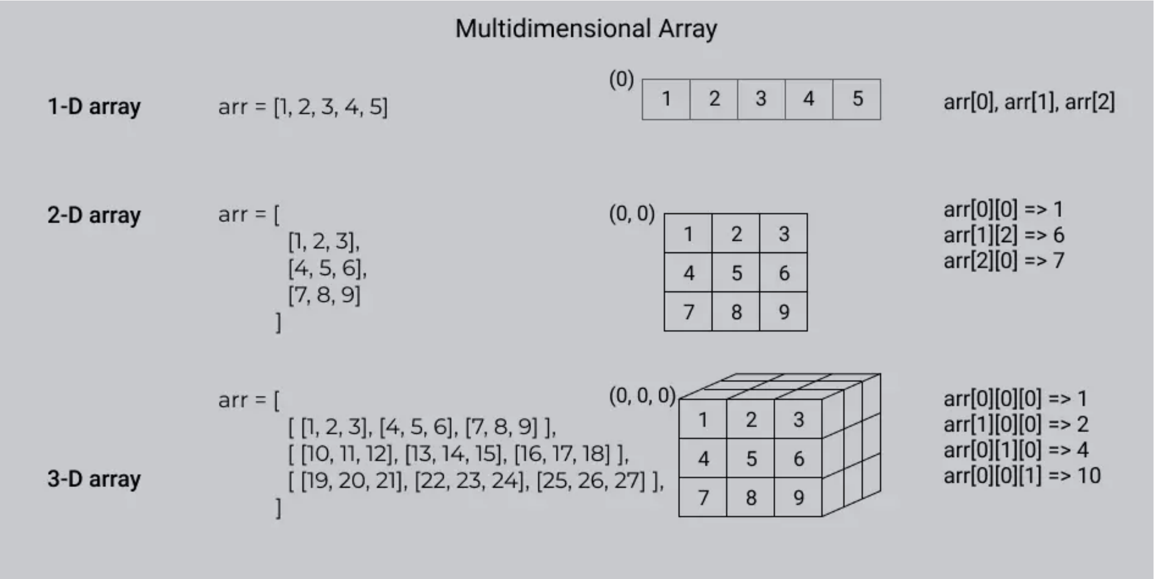
```
function array() {
    return [100, 200, 300];
}
var [x, y, z] = array();
console.log(x);
console.log(y);
console.log(z);

// output:
100
200
300
```

We used array destructuring to destructure the above array elements into specific elements x, y, and z in a single line of code.

Multidimensional array

A multidimensional array is an array that has one or more nested arrays. This increases the number of dimensions of the array.

a normal array is a 1-Dimensional array and can be accessed using a single index. But when there is one nested array, it becomes a 2-Dimensional array. In this case, we can access the elements of the nested array using two indices. Similarly, when there is an array that has an array inside it which have another array inside that then it becomes a 3-Dimensional array. And so on.



```
arr = [1, 2, 3, 4, 5]
```

arr[0], arr[1], arr[2]

```
arr = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]
      ]
```

arr[0][0] => 1
arr[1][2] => 6
arr[2][0] => 7

```
arr = [
        [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ],
        [ [10, 11, 12], [13, 14, 15], [16, 17, 18] ],
        [ [19, 20, 21], [22, 23, 24], [25, 26, 27] ],
      ]
```

arr[0][0][0] => 1
arr[1][0][0] => 2
arr[0][1][0] => 4
arr[0][0][1] => 10

JavaScript does not provide any built-in support for multidimensional arrays. There is no direct way to create a multidimensional array in JavaScript. Instead, we can create a multidimensional array by using a nested array

The simplest way to create a JavaScript multidimensional array is using array literals. For this, you can manually create the array using square brackets and nest the other arrays inside the square brackets.

Example :-

```
var arr = [
  [1, 2, 3],
  [4, 5, 6]
];
```

The above code creates a multidimensional array with two nested arrays.

Accessing elements in JavaScript multidimensional array

A multidimensional array is accessed in the same fashion as a normal array. The only difference is that we need to use two or more indices to access the elements of the nested array.

For example, to access the second element of the first nested array, we need to use arr[0][1].

For example :-

```
var arr = [
  [1, 2, 3],
  [4, 5, 6]
];
console.log(arr[1][1]); //  output: 5
```

Adding elements in JavaScript multidimensional array

The Push or Splice method :-

The push method will add an element at the end of the array. You can push an entire array at the end of the array or can add a new element to any internal array.

Example :-

```
var arr = [
  ['a', 'b', 'c'],
  ['d', 'e', 'f']
];
console.log(arr);
arr.push(['g', 'h', 'i']);
console.log(arr);

//  output:
[ [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ] ]
[ [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ], [ 'g', 'h', 'i' ] ]
```

Another method to add elements to the array is the **splice** method. It can add as well as remove elements at any index of the array.

Example :-

```
var arr = [
  ['a', 'b', 'c'],
  ['d', 'e', 'f']
];
console.log(arr);
arr.splice(2, 0, ['g', 'h', 'i']);
console.log(arr);
arr.splice(1, 0, [1, 2]);
console.log(arr);

//  output:
[ [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ] ]
[ [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ], [ 'g', 'h', 'i' ] ]
[ [ 'a', 'b', 'c' ], [ 1, 2 ], [ 'd', 'e', 'f' ], [ 'g', 'h', 'i' ] ]
```

Removing elements from multidimensional array in JavaScript

To remove elements from a multidimensional array you can use the **pop** or **splice** method.

The **pop** method will always remove the last element of the array and return it.

The **splice** method can remove any element from the array.

Example :-

```
var arr = [
  ['a', 'b', 'c'],
  ['d', 'e', 'f'],
  ['g', 'h', 'i']
];
console.log(arr);
arr.pop();
console.log(arr);
arr[0].splice(1, 1);
console.log(arr);

//  output:
[ [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ], [ 'g', 'h', 'i' ] ]
[ [ 'a', 'b', 'c' ], [ 'd', 'e', 'f' ] ]
[ [ 'a', 'c' ], [ 'd', 'e', 'f' ] ]
```

Shallow copy

Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.

Example:-

```
constfirst_person = {
name: "Jack",
age: 24,
}
constsecond_person = first_person;
second_person.age = 25;
console.log(first_person.age); // output: 25
console.log(second_person.age); // output: 25
```

Deep copy

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

Example :-

```
constfirst_person = {
  name: "Jack",
  age: 24
};
let second_person = first_person;
second_person = {
  name: "Jack",
  age: 23
};

console.log(first_person.age); // Output: 24
console.log(second_person.age); // Output: 23
```

Array Methods and Properties

at() :-	Returns an indexed element of an array concat() :- Joins arrays and returns an array with the joined arrays

constructor :- Returns the function that created the Array object's prototype

copyWithin() :- Copies array elements within the array, to and from specified positions

entries() :- Returns a key/value pair Array Iteration Object

every() :-Checks if every element in an array pass a test

fill() :- Fill the elements in an array with a static value

filter():- Creates a new array with every element in an array that pass a test

find() :- Returns the value of the first element in an array that pass a test

findIndex() :- Returns the index of the first element in an array that pass a test

flat():- Concatenates sub-array elements

flatMap() :- Maps all array elements and creates a new flat array

forEach():- Calls a function for each array element

from() :-Creates an array from an object

includes() :- Check if an array contains the specified element

indexOf() :- Search the array for an element and returns its position

isArray() :- Checks whether an object is an array

join() :- Joins all elements of an array into a string

keys() :- Returns a Array Iteration Object, containing the keys of the original array

lastIndexOf() :- Search the array for an element, starting at the end, and returns its position

length :- Sets or returns the number of elements in an array

map():- Creates a new array with the result of calling a function for each array element

pop():- Removes the last element of an array, and returns that element

prototype :- Allows you to add properties and methods to an Array object

push():- Adds new elements to the end of an array, and returns the new length

reduce() :- Reduce the values of an array to a single value (going left-to-right)

reduceRight():- Reduce the values of an array to a single value (going right-to-left)

reverse():- Reverses the order of the elements in an array

shift() :- Removes the first element of an array, and returns that element

slice():- Selects a part of an array, and returns the new array

some() :-Checks if any of the elements in an array pass a test sort():- Sorts the elements of an array

splice() :-Adds/Removes elements from an array

toString() :-Converts an array to a string, and returns the result

unshift():-Adds new elements to the beginning of an array, and returns the new length

valueOf():-Returns the primitive value of an array

Destructuring is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

a. Array Destructuring

- Basic Syntax:

  javascript

  ```
  let [a, b] = [1, 2];
  console.log(a); // Outputs: 1
  console.log(b); // Outputs: 2
  ```

- Skipping Items:

  javascript

  ```
  let [a, , b] = [1, 2, 3];
  console.log(a); // Outputs: 1
  console.log(b); // Outputs: 3
  ```

- Rest Operator:

  javascript

  ```
  let [a, ...rest] = [1, 2, 3, 4];
  console.log(a); // Outputs: 1
  console.log(rest); // Outputs: [2, 3, 4]
  ```

- Default Values:

  javascript

  ```
  let [a, b, c = 3] = [1, 2];
  console.log(c); // Outputs: 3
  ```

b. Object Destructuring

- Basic Syntax:

  javascript

  ```
  let {a, b} = {a: 1, b: 2};
  console.log(a); // Outputs: 1
  ```

```
console.log(b); // Outputs: 2
```

- New Variable Names:

  javascript

  ```
  let {a: alpha, b: beta} = {a: 1, b: 2};
  console.log(alpha); // Outputs: 1
  console.log(beta); // Outputs: 2
  ```

- Default Values:

  javascript

  ```
  let {a, b = 2} = {a: 1};
  console.log(b); // Outputs: 2
  ```

- Nested Destructuring:

  javascript

  ```
  let {a: {b, c}} = {a: {b: 1, c: 2}};
  console.log(b); // Outputs: 1
  console.log(c); // Outputs: 2
  ```

2. Arrays in JavaScript

Arrays are used to store multiple values in a single variable, accessible by indexed positions.

a. Creating Arrays

- Literal Syntax:

  javascript

  ```
  let array = [1, 2, 3];
  ```

- Constructor Syntax:

  javascript

  ```
  let array = new Array(1, 2, 3);
  ```

b. Accessing Array Elements

javascript

```
let array = [1, 2, 3];
console.log(array[0]); // Outputs: 1
```

c. Array Methods

- Adding/Removing Elements:
  - `push()`, `pop()` for end of array

  - `shift()`, `unshift()` for beginning of array
- Finding Elements:
  - `indexOf()`, `find()`, `findIndex()`
- Iterating Over Arrays:
  - `forEach()`, `map()`, `filter()`, `reduce()`
- Others:
  - `slice()`, `splice()`, `concat()`, `join()`

d. Spread Operator

The spread operator `...` allows an array to expand in places where zero or more arguments (for function calls) or elements (for array literals) are expected.

javascript

```
let parts = ['shoulders', 'knees'];
let lyrics = ['head', ...parts, 'and', 'toes'];
// lyrics = ['head', 'shoulders', 'knees', 'and', 'toes']
```

DOM Manipulations

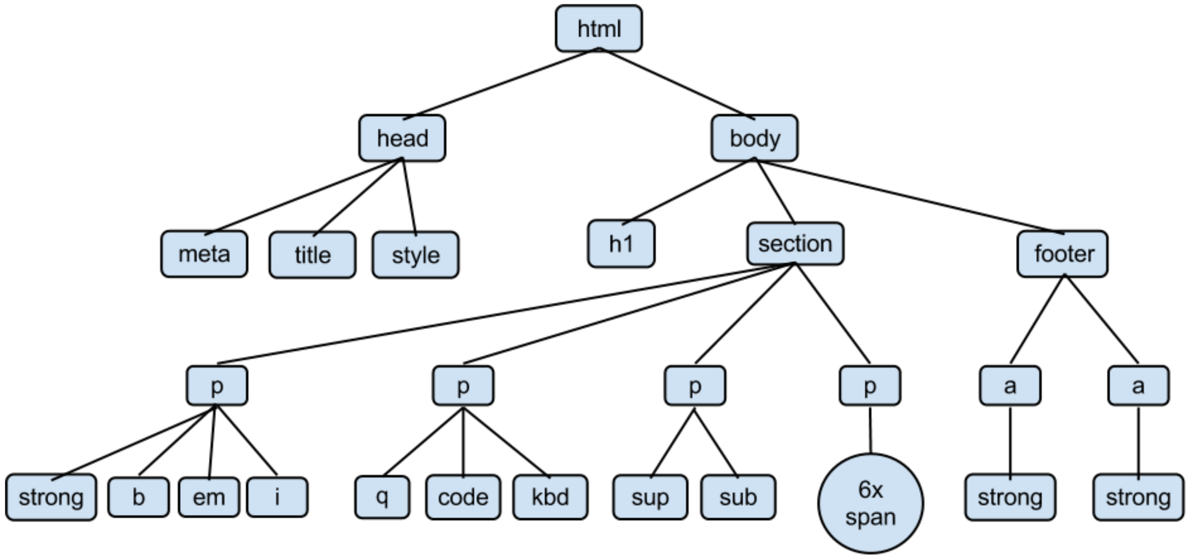
DOM and its importance:

The Document Object Model (DOM) is a programming interface for web documents. It represents the structure of a web page and allows you to interact with and manipulate its elements using JavaScript.

DOM manipulation is crucial for creating dynamic web applications. It allows to update the content, structure, and style of a web page in response to user actions, making your website interactive and responsive.

DOM Tree Structure:

The DOM is structured as a tree, with the Document Object at the top. Each HTML element becomes a node in this tree, and you can traverse, access, and modify these nodes.



Accessing Elements:

Selecting Elements by ID:

You can select elements with a specific ID using document.getElementById().

Selecting Elements by Class:

To select elements by class name, use document.getElementsByClassName().

Selecting Elements by Tag Name:

To select elements by their HTML tag, use document.getElementsByTagName().

Selecting Elements by CSS Selectors:

Use document.querySelectorAll() to select elements using CSS selectors.

```
// Selecting an element by ID
const elementById = document.getElementById('myElementId');

// Selecting elements by class
```

```
const elementsByClass = document.getElementsByClassName('myClass');

// Selecting elements by tag name
const elementsByTag = document.getElementsByTagName('div');

// Selecting elements using CSS selectors
const elementsBySelector = document.querySelectorAll('.myClass p');
```

Modifying Elements:

Changing HTML Content:

You can change the content of an element using the innerHTML property.

Changing Element Attributes:

Use the setAttribute() method to modify element attributes.

Modifying Element Styles:

Alter the styles of elements by accessing their style property.

```
// Changing HTML content
elementById.innerHTML = 'New Content';

// Changing element attributes
elementById.setAttribute('src', 'new-image.jpg');

// Modifying element styles
elementById.style.backgroundColor = 'blue';
```

Creating and Removing Elements:

Creating New Elements:

You can create new elements using document.createElement().

Appending and Removing Elements:

Add or remove elements from the DOM using appendChild() and removeChild().

Cloning Elements:

Use cloneNode() to create copies of elements.

```
// Creating a new element
const newElement = document.createElement('div');

// Appending the new element
elementById.appendChild(newElement);

// Removing an element
elementById.removeChild(newElement);

// Cloning an element
const cloneElement = elementById.cloneNode(true);
```

Event Handling:

Adding Event Listeners:

Attach event listeners to elements using addEventListener().

Event Object and Event Types:

Event listeners receive an event object with details about the event, including its type.

Event Bubbling and Event Delegation:

Understand how events propagate through the DOM and use event delegation for efficiency.

```
// Adding an event listener
elementById.addEventListener('click', (event) => {
    alert('Element clicked!');
});
```

Do refer the table provided towards the end of the document for more such events available in javascipt.

DOM Traversal:

Navigating the DOM Tree:

You can navigate up and down the DOM tree using properties like parentElement, childNodes, etc.

Parent, Child, and Sibling Elements:

Access parent, child, and sibling elements with parentNode, children, nextSibling, and previousSibling.

Finding Elements within a Parent:

Use querySelectorAll() on a parent element to find specific child elements.

```
// Accessing parent element
const parentElement = elementById.parentElement;

// Accessing child elements
const childElements = elementById.childNodes;

// Finding elements within a parent
const childDivs = parentElement.querySelectorAll('div');
```

Here is a list of available events:

Event	**Occurs When**
abort	The loading of a media is aborted
afterprint	A page has started printing
animationend	A CSS animation has completed
animationiteration	A CSS animation is repeated
animationstart	A CSS animation has started
beforeprint	A page is about to be printed
beforeunload	Before a document is about to be unloaded
blur	An element loses focus
canplay	The browser can start playing a media (has buffered enough to begin)
canplaythrough	The browser can play through a media without stopping for buffering
change	The content of a form element has changed
click	An element is clicked on
contextmenu	An element is right-clicked to open a context menu
copy	The content of an element is copied
cut	The content of an element is cut
dblclick	An element is double-clicked
drag	An element is being dragged
dragend	Dragging of an element has ended
dragenter	A dragged element enters the drop target
dragleave	A dragged element leaves the drop target
dragover	A dragged element is over the drop target
dragstart	Dragging of an element has started
drop	A dragged element is dropped on the target
durationchange	The duration of a media is changed
ended	A media has reach the end ("thanks for listening")
error	An error has occurred while loading a file
focus	An element gets focus
focusin	An element is about to get focus
focusout	An element is about to lose focus
fullscreenchange	An element is displayed in fullscreen mode
fullscreenerror	An element can not be displayed in fullscreen mode
hashchange	There has been changes to the anchor part of a URL
input	An element gets user input
invalid	An element is invalid
keydown	A key is down
keypress	A key is pressed
keyup	A key is released
load	An object has loaded
loadeddata	Media data is loaded
loadedmetadata	Meta data (like dimensions and duration) are loaded
loadstart	The browser starts looking for the specified media
message	A message is received through the event source
mousedown	The mouse button is pressed over an element
mouseenter	The pointer is moved onto an element
mouseleave	The pointer is moved out of an element
mousemove	The pointer is moved over an element
mouseover	The pointer is moved onto an element

mouseout	The pointer is moved out of an element
mouseup	A user releases a mouse button over an element
mousewheel	Deprecated. Use the wheel event instead
offline	The browser starts working offline
online	The browser starts working online
open	A connection with the event source is opened
pagehide	User navigates away from a webpage
pageshow	User navigates to a webpage
paste	Some content is pasted in an element
pause	A media is paused
play	The media has started or is no longer paused
playing	The media is playing after being paused or buffered
popstate	The window's history changes
progress	The browser is downloading media data
ratechange	The playing speed of a media is changed
resize	The document view is resized
reset	A form is reset
scroll	An scrollbar is being scrolled
search	Something is written in a search field
seeked	Skipping to a media position is finished
seeking	Skipping to a media position is started
select	User selects some text
show	A <menu> element is shown as a context menu
stalled	The browser is trying to get unavailable media data
storage	A Web Storage area is updated
submit	A form is submitted
suspend	The browser is intentionally not getting media data
timeupdate	The playing position has changed (the user moves to a different point in the media)
toggle	The user opens or closes the <details> element
touchcancel	The touch is interrupted
touchend	A finger is removed from a touch screen
touchmove	A finger is dragged across the screen
touchstart	A finger is placed on a touch screen
transitionend	A CSS transition has completed
unload	A page has unloaded
volumechange	The volume of a media is changed (includes muting)
waiting	A media is paused but is expected to resume (e.g. buffering)
wheel	The mouse wheel rolls up or down over an element

JavaScript Strings

JavaScript Strings are useful for holding data that can be represented in text form.

A JavaScript string is zero or more characters written inside quotes.

Example:

```
 let text= “Keshav Memorial Institute of Technology.”
```

text→ variable name

Keshav Memorial Institute of Technology. → String literal encoded with double quotes

You can use single or double quotes:

```
let carName1= "BMWXC99";  //Doublequotes
let carName2= 'BMWXC98';  // Single quotes
```

String primitives and String objects:

Note that JavaScript distinguishes between String objects and **primitive string** values. (The same is true of **Boolean** and **Numbers**.)

String literals (denoted by double or single quotes) and strings returned from String calls in a non-constructor context (that is, called without using the **new** keyword) are primitive strings. In contexts where a method is to be invoked on a primitive string or a property lookup occurs, JavaScript will automatically wrap the string primitive and call the method or perform the property lookup on the wrapper object instead.

```
conststr = "foo"; // A literal is a string primitive
conststr2 = String(1); // Coerced into the string primitive "1"
conststr3 = String(true); // Coerced into the string primitive "true"
conststrObj = new String(str); // String with new returns a string wrapper
object.

console.log(typeofstr); // "string"
console.log(typeofstr2); // "string"
console.log(typeofstr3); // "string"
console.log(typeofstrObj); // "object"
```

String primitives and String objects also give different results when using eval(). Primitives passed to eval are treated as source code; String objects are treated as all other objects are, by returning the object. For example:

```
const s1 = "2 + 2"; // creates a string primitive
const s2 = new String("2 + 2"); // creates a String object
console.log(eval(s1)); // returns the number 4
console.log(eval(s2)); // returns the string "2 + 2"
               note: use node to run this
```

Strings are immutable:

It’s important to know that in JavaScript, strings are immutable. This means that once a string is created, its contents cannot be changed.

Instead, you must create a new string representing the modified version when you want to modify a string.

For example, if you have a string assigned to a variable, you cannot modify it. Instead, you will create a new string and assign the new string to the same variable like this:

```
let name ="John Doe";
name ="Jane";
```

This means that the original string "John Doe" still exists in memory, but the variable name now refers to the new string "Jane".

JavaScript Template Literals:

Back-Tics Syntax

Template Literals use back-ticks (``) rather than the quotes ("") to define a string:

Example:

```
let text = `Hello World!`;
```

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"`;
```

Multiline Strings

Template literals allows multiline strings:

Example

```
let text=`The
quick
brown
fox
jumps
over
the lazy dog`;
```

Expression Substitution

Template literals allow expressions in strings:

Example

```
let price= 10;
let VAT= 0.25;
let total = `Total: ${(price * (1 + VAT)).toFixed(2)}`;       //Total: 12.50
```

JavaScript String Methods

String length
String slice()
String substring()
String substr()
String replace()
String replaceAll()
String toUpperCase()
String toLowerCase()
String concat()
String trim()
String trimStart()
String trimEnd()
String padStart()
String padEnd()
String charAt()
String charCodeAt()
String split()

String Length

The **length** property returns the length of a string:

Example

```
let text= "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
let length = text.length;
console.log(length);    // 26
```

Extracting String Parts
There are 3 methods for extracting a part of a string:
slice(start, end)
substring(start, end)
substr(start, length)

String slice()

slice() extracts a part of a string and returns the extracted part in a new string.

The method takes 2 parameters: start position, and end position (end not included).

Example

Slice out a portion of a string from position 7 to position 13:

```
let text= "Apple,Banana,Kiwi";
let part = text.slice(7, 13);
```

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

```
let text= "Apple,Banana,Kiwi";
let part = text.slice(7);
```

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

```
let text= "Apple,Banana,Kiwi";
let part = text.slice(-12);
```

This example slices out a portion of a string from position -12 to position -6:

```
let text= "Apple,Banana,Kiwi";
let part = text.slice(-12, -6);
```

String substring()

substring() is similar to **slice().**

The difference is that start and end values less than 0 are treated as 0 in **substring()**.

Example:

```
let str= "Apple,Banana,Kiwi";
let part = str.substring(7, 13);
```

If you omit the second parameter, **substring()** will slice out the rest of the string.

Replacing String Content

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

Example

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

Note

- The replace() method does not change the string it is called on.
- The replace() method returns a new string.
- The replace() method replaces only the first match

If you want to replace all matches, use a regular expression with the /g flag set. See examples below.

String ReplaceAll()

In 2021, JavaScript introduced the string method replaceAll():

Example

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

The replaceAll() method allows you to specify a regular expression instead of a string to be replaced. If the parameter is a regular expression, the global flag (g) must be set, otherwise a TypeError is thrown.

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

String toUpperCase()

```
let text1 = "Hello World!";
let text2 = text1.toUpperCase();     // text2 is text1 converted to Upper
HELLO WORLD!
```

String toLowerCase()

```
let text1 = "Hello World!";         // String
let text2 = text1.toLowerCase();      // text2 is text1 converted to
lowerhello world!
```

String concat()

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

The concat() method can be used instead of the plus operator. These two lines do the same:

```
text = "Hello" + " " + "World!";
text = "Hello".concat(" ", "World!");// both will give same result  Hello
World!
```

String split()

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

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

If the separator is omitted, the returned array will contain the whole string in index [0].

If the separator is "", the returned array will be an array of single characters:

```
text.split("")
```

String charCodeAt()

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

The method returns a UTF-16 code (an integer between 0 and 65535).

```
let text = "HELLO WORLD";
let char = text.charCodeAt(0);
```

Property Access

ECMAScript 5 (2009) allows property access [ ] on strings:

Example

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

Note

Property access might be a little unpredictable:

It makes strings look like arrays (but they are not)

If no character is found, [ ] returns undefined, while charAt() returns an empty string.

It is read only. str[0] = "A" gives no error (but does not work!)

Example

let text = "HELLO WORLD";

text[0] = "A";   // Gives no error, but does not work

Search Methods

- String indexOf()
- String lastIndexOf()
- String search()
- String match()
- String matchAll()
- String includes()
- String startsWith()
- String endsWith()

String indexOf()

The indexOf() method returns the index (position) the first occurrence of a string in a string:

```
let text = "Please locate where 'locate' occurs!";
let index = text.indexOf("locate");
```

Note

JavaScript counts positions from zero.

0 is the first position in a string, 1 is the second, 2 is the third, ..

String search()

The search() method searches a string for a string (or a regular expression) and returns the position of the match:

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

JavaScript String match()

The match() method returns an array containing the results of matching a string against a string (or a regular expression).

Perform a search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match("ain");  // array size =1
```

Perform a search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/);  //array size =1
```

Perform a global search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/g);    //array size =3
```

Perform a global, case-insensitive search for "ain":

```
let text = "The rain in SPAIN stays mainly in the plain";
text.match(/ain/gi); // array size =4
```

Regular Expression:

Regular expressions are patterns used to match character combinations in strings. In JavaScript, regular expressions are also objects. These patterns are used with the exec() and test() methods of RegExp, and with the match(), matchAll(), replace(), replaceAll(), search(), and split() methods of String.

Using String Methods

In JavaScript, regular expressions are often used with the two string methods: search() and replace().

1. The **search()** method uses an expression to search for a match, and returns the position of the match.
2. The **replace()** method returns a modified string where the pattern is replaced.

Example-1:

```
//input:
let data="Hi Welcome!!! to kmit. kmit locates in HYD"
console.log(data.search("kmit"))
//output: 17
```

Example-2:

```
//input:
let data="Hi Welcome!!! to kmit. kmit locates in HYD''
console.log(data.replace("kmit","KMIT"))
console.log(data.replaceAll("kmit","KMIT"))
//output:
 Hi Welcome!!! to KMIT. kmit locates in HYD
 Hi Welcome!!! to KMIT. KMIT locates in HYD
```

Example 3:

```
//input:
let data="Hi Welcome!!! to kmit. kmit locates in HYD"
console.log(data.split(" "))
console.log(data.match("kmit"))
//output:
[ 'Hi', 'Welcome!!!', 'to', 'kmit.', 'kmit', 'locates', 'in', 'HYD' ]
[   'kmit',     index: 17,     input: 'Hi Welcome!!! to kmit. kmit locates in
HYD',
  groups: undefined ]
```

Note: In Example1 , While we search **kmit** only its printing one time, actually we have two kmit in one statement , to resolve such problems we are using Regular Expression Techniques.

Create a RegEx :

There are two ways you can create a regular expression in JavaScript.

1. **Using a regular expression literal:**
   The regular expression consists of a pattern enclosed between slashes /.
   For example: **cost regularExp = /abc/;**
   Here, /abc/ is a regular expression.
2. **Using the RegExp() constructor function:**
   You can also create a regular expression by calling the RegExp() constructor function. For example: const reguarExp = new RegExp('abc');

Example-1:

```
Input:
const data1="Hi Welcome!!! to kmit. kmit locates in HYD"
const rex=/kmit/g
let result=data1.matchAll(rex)
console.log(Array.from(result));
Output:
        0: ['kmit', index: 17, input: 'Hi Welcome!!! to kmit. kmit locates in 
HYD']
1: ['kmit', index: 23, input: 'Hi Welcome!!! to kmit. kmit locates in HYD']
length: 2
```

Example-2:

Write a Regular Expression that displays all words which should contains second letter must be an Vowel using function

```
function RegularExpression()
{
  let data1="Hi Welcome!!! to kmit. kmit locates in HYD"
  let regex = new RegExp(/.[a-e]\w/g);
  let res=data1.matchAll(regex);
  console.log(Array.from(res));
}
RegularExpression()
```

Regular Expression Patterns

Pattern Matching or String Matching:

Expression	**Description**
[abc]	Should match any single of character
[^abc]	Should not match any single character
[0-9]	should match any decimal digit from 0 through 9
[a-z]	should match any character from lowercase a through lowercase z.
[A-Z]	should match any character from uppercase A through uppercase Z.
[a-Z]	should match any character from lowercase a through uppercase Z.
[a-zA-Z0-9]	Characters range lowercase a-z, uppercase A-Z and numbers.
[a-z-._]	Match against character range lowercase a-z and ._ refers special character
(.*?)	Capture everything enclosed with brackets
(com|info)	Input should be "com" or "info"
{2}	Exactly two characters
{2,3}	Minimum 2 characters and Maximum 3 characters
{2,}	More than 2 characters

Metacharacters are characters with a special meaning:

Expression	**Description**
\d	Find a digit
\s	Find a whitespace character
\b	Find a match at the beginning of a word like this: \bWORD, or at the end of a word like this: WORD\b
\uxxxx	Find the Unicode character specified by the hexadecimal number xxxx

Quantifiers define quantities:

Expression	**Description**
n+	Matches any string that contains at least one n

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

Symbols:

Expression	**Description**
^	Start of string
$	End of string
.	Any single character
+	One or more character
\	Escape Special characters
?	Zero or more characters

Regular Expression Flags:

Expression	**Description**
g	g is used for global search which means the search will not return after the first match.
i	i is used for case-insensitive search meaning that a match can occur regardless of the casing.
m	m is used for multiline search.
u	u is used for Unicode search

Regular Expression Object Methods:

Methods	**Description**	**Example:**

compile()	● Used to compile the regular expression while executing the script.
	● The compile() method is used to compile a regular expression during execution of a script.
	● The compile() method can also be used to change and recompile a regular expression.
		`var str="Every man in the world! Every woman on earth!";`
		`var patt=/man/g;`
		`var str2=str.replace(patt,"person");`
		`console.log(str2)`
		
		`patt=/(wo)?man/g;`
		`patt.compile(patt);`
		`str2=str.replace(patt,"person");`
		`console.log(str2)`

exec()	● Used to test for the match in a string.
	● If it finds a match, it returns a result array, otherwise it returns null.
		`let data = "Hi Welcome!!! to kmit. kmit locates in HYD";`
		`let result = /Hi/.exec(data);`
		`console.log(result)`
		`o/p: [ 'Hi', index: 0,`
		`  input: 'Hi Welcome!!! to kmit. kmit locates in HYD']`

test()	● Used to test for a match in a string
	● If it finds a match, it returns true, otherwise it returns false.
		`let data = "Hi Welcome!!! to kmit. kmit locates in HYD";`
		`let pattern = /Hi/g;`
		`let result = pattern.test(data);`
		`console.log(result)`
		`o/p: true`

toString()	- Return the string value of the regular expression
	- For RegExp objects, the toString() method returns a string representation of the regular expression
		`console.log(new RegExp('a+b+c'));`
		`// Expected output: /a+b+c/`
		
		`console.log(new RegExp('a+b+c').toString());`
		`// Expected output: "/a+b+c/"`
		
		`console.log(new RegExp('bar', 'g').toString());`
		`// Expected output: "/bar/g"`
		
		`console.log(new RegExp('\n', 'g').toString());`
		`// Expected output (if your browser supports escaping): "/\n/g"`
		
		`console.log(new RegExp('\\n', 'g').toString());`
		`// Expected output: "/\n/g"`

Regex Cheat Sheet

Quantifiers

a|b	Match either "a" or "b"
?	Match either "a" or "b"
+	One or more
*	Zero or more
*?	Zero or more, but stop after first match
{N}	Exactly N number of times (Where N is number)
{N, M}	From N to M number of times (Where N and M are numbers)

General Tokens

.	Any character
\n	Newline character
\t	Tab character
\s	Any whitespace character (Including \t, \n, etc)
\S	Any non-whitespace character
\w	Any word character (Upper/lowercase letters, 0-9, _)
\W	Any non-word character
\b	Word boundary (Matches between characters)
\B	Non-word boundary
^	The start of a line
$	The end of a line
\\	The literal character "\"

CoderPad

For a full Regex guide:

https://bit.ly/regexblog

Pattern Collections

[A-Z]	Match any uppercase character from "A" to "Z"
[a-z]	Match any lowercase character from "a" to "z"
[0-9]	Match any number
[asdf]	Match any character that's either "a", "s", "d", or "f"
[^asdf]	Match any character that's not any of the following: "a", "s", "d", or "f"

Flags

g	Global, match more than once
m	Force $ and ^ to match each newline individually
i	Make the regex case-insensitive

Groups

(...)	Capture group (Matches any 3 characters)
(?: ...)	Non-capture group (Matches any 3 characters)
(?<name> ...)	Named capture group Group is called "name"

Named Back Reference

\k<name>	Reference named capture group "name" in query

Lookahead and Lookbehind

(?!)	Negative lookahead
(?=)	Positive lookahead
(?<!)	Negative lookbehind
(?<=)	Positive lookbehind

Sets

Set is a built-in data structure that allows you to store unique values of any type, whether they are primitive values or object references.

Sets ensure that each value can only appear once in the collection.

Creating a Set:

You can create a Set using the **Set** constructor.

You can pass an iterable (such as an array) to initialize the Set.

Example:

```
const mySet = new Set();
const mySetFromArray = new Set([1, 2, 3, 2, 1]); // Duplicates are automatically removed
console.log(mySetFromArray)
```

Set methods and properties:

Some of the commonly used methods of the JavaScript Set object:

1. **add(value):** Adds a new element with the given value to the Set.

```
const mySet = new Set();
mySet.add(42);
mySet.add('hello');
console.log(mySet)
```

2. **delete(value):** Removes the element with the specified value from the Set.

```
const mySet = new Set([1, 2, 3]);
mySet.delete(2);
```

3. **has(value):** Checks if the Set contains an element with the specified value. Returns **true** if found, otherwise **false**.

```
const mySet = newSet(['apple','banana','cherry']);
console.log(mySet.has('banana')); // true
console.log(mySet.has('grape')); // false
```

4. **clear():** Removes all elements from the Set.

```
const mySet = new Set([1, 2, 3]); mySet.clear();
```

5. **forEach(callbackFn, thisArg):** Executes the given callback function for each element in the Set.

```
const mySet = new Set([10, 20, 30]);
mySet.forEach((value) => { console.log(value); });
```

6. **keys():** Returns an iterator of all the keys in the Set. (This is the same as the values, as Sets only store unique values).

```
const mySet = new Set([1, 2, 3]);
const keyIterator = mySet.keys();
for (const key of keyIterator) { console.log(key); }
```

7. **values():** Returns an iterator of all the values in the Set.

```
const mySet = new Set(['a', 'b', 'c']);
const valueIterator = mySet.values();
for (const value of valueIterator) { console.log(value); }
```

8. **entries():** Returns an iterator of all entries (key-value pairs) in the Set, where the key is the same as the value.

```
const mySet = new Set(['x', 'y', 'z']);
const entryIterator = mySet.entries();
for (const [key, value] of entryIterator) { console.log(key, value); }
```

9. **mySet.clear() :** removes all elements from the set

Properties:

size: Returns the number of set elements

```
const mySet = new Set(['red', 'green', 'blue']);
console.log(mySet.size); // 3
```

Remember that Sets are iterable, so you can use the **for...of** loop to iterate through their values, keys, or entries.

Also, note that Sets only store unique values, so duplicate values are automatically eliminated when adding elements.

Maps

Map is a built-in data structure that allows you to store key-value pairs. Unlike Sets, which store only values,

Maps associate values with unique keys.

Maps can use any data type as keys and values, including objects and functions.

Creating a Map:

You can create a Map using the **Map** constructor.

You can also initialize a Map with an array of key-value pairs.

Example:

```
const myMap = new Map();
const  myMapWithInitialData  =  new  Map([  ['key1',  'value1'],  ['key2',
'value2'] ]); //Map(2) { 'key1' => 'value1', 'key2' => 'value2' }
```

Map methods and properties:

1. **set(key, value):** Adds a new key-value pair to the Map.

```
const myMap = new Map();
myMap.set('name', 'Alice');
myMap.set('age', 30);
```

2. **get(key):** Returns the value associated with the specified key.

```
console.log(myMap.get('name')); // 'Alice'
```

3. **has(key):** Checks if the Map contains a key. Returns **true** if found, otherwise **false**.

```
console.log(myMap.has('age')); // true
console.log(myMap.has('gender'));// false
```

4. **delete(key):** Removes the key-value pair with the specified key from the Map.

```
myMap.delete('age');
```

5. **clear():** Removes all key-value pairs from the Map.

```
myMap.clear();
```

6. **forEach(callbackFn, thisArg):** Executes the given callback function for each key-value pair in the Map.

```
myMap.forEach((value, key) => { console.log(`${key}: ${value}`); });
```

7. **keys():** Returns an iterator of all keys in the Map.

```
const keyIterator = myMap.keys();
for (const key of keyIterator) { console.log(key); }
```

8. **values():** Returns an iterator of all values in the Map.

```
const valueIterator = myMap.values();
for (const value of valueIterator) { console.log(value); }
```

9. **entries():** Returns an iterator of all key-value pairs in the Map.

```
const entryIterator = myMap.entries();
for (const [key, value] of entryIterator) { console.log(`${key}:
${value}`); }
```

Properties:

size: Returns the number of key-value pairs in the Map.

```
console.log(myMap.size);
```

Maps are versatile data structures for managing key-value pairs with various data types as keys and values.

They maintain the insertion order, making them useful for cases where order matters.

Keep in mind that the keys in a Map can be any data type, unlike objects where keys are always converted to strings.

JavaScript: Iterators and Generators

Iterators

Iterators are objects in JavaScript that provide a mechanism to traverse through a collection (like arrays, strings, or other iterable objects) one element at a time. An iterator implements a specific interface and allows programmers to use a uniform approach to explore various types of data collections.

Key Concepts:

- Iterable Protocol: This protocol is implemented by objects that can return an iterator using the Symbol.iterator property.
- Iterator Protocol: This protocol defines a standard way to produce a sequence of values (either finite or infinite). The object must implement a next() method that returns an object with two properties:
    - value: the data representing the next element in the sequence.
    - done: a boolean value indicating if the sequence is finished.

Example of an Iterator:

javascript

```
const array1 = [1, 2, 3];
const iterator = array1[Symbol.iterator]();
console.log(iterator.next().value); // 1
console.log(iterator.next().value); // 2
console.log(iterator.next().value); // 3
console.log(iterator.next().done); // true
```

Generators

Generators are a special class of functions that simplify the creation of iterators. A generator function is declared with the function syntax. When called, generator functions do not initially execute their code. Instead, they return a special type of iterator, called a "Generator object".

Key Features:

- Yield: Generators can yield multiple values over time, each time retaining their internal states, unlike regular functions that start over.
- Function Syntax: Declares a generator function.
- Yield Keyword: Pauses the generator and returns a value.

Example of a Generator:

javascript

```
function numberGen() {
  yield 1;
  yield 2;
  yield 3;
}
const gen = numberGen();
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
console.log(gen.next().value); // 3
console.log(gen.next().done); // true
```

Practical Uses of Iterators and Generators

- Handling Asynchronous Operations: Generators can be used in conjunction with promises to handle asynchronous operations more smoothly.
- Implementing Custom Iterators: Both iterators and generators provide a way to implement custom iterables that are not possible with standard JavaScript objects.
- Efficient Data Processing: Generators are particularly useful when dealing with data streams or large datasets that should not be loaded entirely in memory.

Differences Between Iterators and Generators

- Creation Complexity: Creating an iterator manually can be verbose compared to a generator which is more succinct and handles the iterator protocol automatically.
- Use Cases: While both can be used for similar purposes, generators offer additional benefits like better readability and integration with asynchronous operations.

Conclusion

Understanding iterators and generators enhances your ability to manage sequences of data efficiently and leverage lazy evaluation in JavaScript, which can be crucial for performance-intensive applications

Objects in JavaScript

In JavaScript, an object is a fundamental data structure that allows you to store and organize data in a key-value pair format.

Objects are a powerful way to organize and manipulate data in your code, and they are extensively used in JavaScript programming for various purposes, including creating classes, managing data structures, and representing real-world entities.

In JavaScript, most things are objects, from core JavaScript features like arrays to the browser APIs built on top of JavaScript. You can even create your own objects to encapsulate related functions and variables into efficient packages and act as handy data containers.

```
//input:
let data="Hi Welcome!!! to kmit. kmit locates in HYD"
console.log(data.search("kmit"))
//output: 17
```

Object creation using Object Literal:

The simplest way to create an object is by using object literals. An object literal is a comma-separated list of key-value pairs enclosed in curly braces `{}`.

```
const person = {
     name: "John Doe",
     age: 30,
     profession: "Software Developer"
   };
```

Object Creation using Constructor Functions:

You can create multiple objects with similar structures using constructor functions or ES6 classes.

```
   function Person(name, age) {
     this.name = name;
this.age = age;
   }

const person1 = new Person("Alice", 25);
const person2 = new Person("Bob", 30);
```

Object creation using ES6 Class:

ES6 introduced class syntax, which provides a more structured way to define and create objects.

```
class Person {
        constructor(firstName, lastName, age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
  }
}
const person = new Person("John", "Doe", 30);
```

Accessing Object Properties:

You can access object properties using dot notation or square brackets notation.

```
 console.log(person.name); // Output: John Doe
   console.log(person["age"]); // Output: 30
```

Adding and Modifying Properties:

You can add or modify properties of an object even after it's created.

```
person.location = "New York";
person.age = 31;
```

Object Iteration:

You can loop through an object's properties using various methods:

```
for (const key in person) {
  console.log(key, person[key]);
}
Object.keys(person).forEach(key => {
  console.log(key, person[key]);
});
```

Deleting a property from an object

There isn't any method in an Object itself to delete its own properties.

To do so, one must use the **delete** operator.

The delete operator removes a property from an object. If the property's value is an object and there are no more references to the object, the object held by that property is eventually released automatically.

Syntax:

```
delete object.property
```

OR

```
delete object[property]
```

Example.

```
const Employee = {
firstname: 'John',
lastname: 'Doe',
};
console.log(Employee.firstname);
// Expected output: "John"
delete Employee.firstname;
console.log(Employee.firstname);
// Expected output: undefined
```

Nested Objects:

Objects can contain other objects as properties, creating a nested structure.

```
const car = {
     make: "Toyota",
     model: "Camry",
     year: 2023,
     owner: {
       name: "Alice",
       age: 28
     }
   };
```

Methods in Objects:

Objects can also have methods, which are functions defined as properties.

```
const calculator = {
     add: function(a, b) {
       return a + b;
     },
     subtract: function(a, b) {
       return a - b;
     }
   };
```

Important Object Methods:

Objects come with built-in methods to manipulate and work with them.

Object.keys(obj): Returns an array of object property names.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};

console.log(Object.keys(object1));
// Expected output: Array ["a", "b", "c"]
```

Object.values(obj): Returns an array of object property values.

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};
console.log(Object.values(object1));
// Expected output: Array ["somestring", 42, false]
```

Object.entries(obj): Returns an array of arrays, each containing a key-value pair.

```
const object1 = {
  a: 'somestring',
  b: 42,
};
```

```
for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}

// Expected output:
// "a: somestring"
// "b: 42"
```

Object.assign(target, source): Copies properties from one or more source objects to a target object.

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

constreturnedTarget = Object.assign(target, source);

console.log(target);
// Expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget === target);
// Expected output: true
```

Object.defineProperties()

The Object.defineProperties() static method defines new or modifies existing properties directly on an object, returning the object.

```
const object1 = {};
Object.defineProperties(object1, {
  property1: {
    value: 42,
    writable: true,
  },
  property2: {},
});

console.log(object1.property1);
// Expected output: 42
```

Object.defineProperty()

The Object.defineProperty() static method defines a new property directly on an object, or modifies an existing property on an object, and returns the object.

```
const object1 = {};

Object.defineProperty(object1, 'property1', {
  value: 42,
  writable: false,
});

object1.property1 = 77;
// Throws an error in strict mode

console.log(object1.property1);
// Expected output: 42
```

Object.freeze()

Freezes an object. Other code cannot delete or change its properties. A frozen object can no longer be changed: new properties cannot be added, existing properties cannot be removed

```
constobj = {
  prop: 42,
};
Object.freeze(obj);
obj.prop = 33;
// Throws an error in strict mode
console.log(obj.prop);
// Expected output: 42
```

Prototypes and Inheritance:

JavaScript uses prototypal inheritance. Objects can inherit properties and methods from other objects.

```
function Animal(name) {
    this.name = name;
}
Animal.prototype.speak = function() {
     console.log(`${this.name} makes a sound.`);
   };
function Dog(name) {
        Animal.call(this, name);
   }
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;
const dog = new Dog("Buddy");
dog.speak(); // Output: Buddy makes a sound.
```

Working with JSON data

JavaScript Object Notation (JSON) is a standard text-based format for representing structured data based on JavaScript object syntax, which is commonly used for representing and transmitting data on the web (i.e., sending some data from the server to the client, so it can be displayed on a web page).

What is JSON?

JSON is a text-based data format following JavaScript object syntax, which was popularized by Douglas Crockford. Even though it closely resembles JavaScript object literal syntax, it can be used independently from JavaScript, and many programming environments feature the ability to read (parse) and generate JSON.

JSON exists as a string — useful when you want to transmit data across a network. It needs to be converted to a native JavaScript object when you want to access the data. This is not a big issue — JavaScript provides a global JSON object that has methods available for converting between the two.

A JSON string can be stored in its own file, which is basically just a text file with an extension of .json

JSON is purely a string with a specified data format — it contains only properties, no methods.

JSON requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string.

Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated JSON is less likely to include errors, as long as the generator program is working correctly). You can validate JSON using an application like JSONLint.

JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be valid JSON.

Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties.

JSON structure

As described above, JSON is a string whose format very much resembles JavaScript object literal format. You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals.

For example:

```
{
  "squadName": "Super hero squad",
  "homeTown": "Metro City",
  "formed": 2016,
  "secretBase": "Super tower",
  "active": true,
  "members": [
    {
      "name": "Molecule Man",
      "age": 29,
      "secretIdentity": "Dan Jukes",
      "powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]
    },
    {
      "name": "Madame Uppercut",
      "age": 39,
      "secretIdentity": "Jane Wilson",
      "powers": [
        "Million tonne punch",
        "Damage resistance",
        "Superhuman reflexes"
      ]
    },
    {
      "name": "Eternal Flame",
      "age": 1000000,
      "secretIdentity": "Unknown",
      "powers": [
        "Immortality",
        "Heat Immunity",
        "Inferno",
        "Teleportation",
        "Interdimensional travel"
      ]
    }
  ]
}
```

We could then access the data inside it using the same dot/bracket notation we looked at in the JavaScript object basics article.

For example:

```
superHeroes.homeTown;
superHeroes["active"];
```

To access data further down the hierarchy, you have to chain the required property names and array indexes together. For example, to access the third superpower of the second hero listed in the members list, you'd do this:

For example:

```
superHeroes["members"][1]["powers"][2];
```

- First, we have the variable name — superHeroes.
- Inside that, we want to access the members property, so we use ["members"].
- members contains an array populated by objects. We want to access the second object inside the array, so we use [1].
- Inside this object, we want to access the powers property, so we use ["powers"].
- Inside the powers property is an array containing the selected hero's superpowers. We want the third one, so we use [2].

Arrays as JSON

Above we mentioned that JSON text basically looks like a JavaScript object inside a string. We can also convert arrays to/from JSON. Below is also valid JSON,

for example:

```
[
  {
    "name": "Molecule Man",
    "age": 29,
    "secretIdentity": "Dan Jukes",
    "powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]
  },
  {
    "name": "Madame Uppercut",
    "age": 39,
    "secretIdentity": "Jane Wilson",
    "powers": [
      "Million tonne punch",
      "Damage resistance",
      "Superhuman reflexes"
    ]
  }
]
```

The above is perfectly valid JSON. You'd just have to access array items (in its parsed version) by starting with an array index, for example [0]["powers"][0].

Fetching the JSON data

To be able to display this data in our HTML file, we first need to fetch the data with JavaScript.

We will fetch this data by using the fetch API. We use the fetch API in the following way:

```
fetch(url)
.then(function (response) {
    // The JSON data will arrive here
  })
.catch(function (err) {
    // If an error occured, you will catch it here
  });
```

The url parameter used in the fetch function is where we get the JSON data. This is often an http address. In our case it is just the filename people.json. We don't have to drill down to any directory since the json file is in the same directory as our index.html.

The fetch function will return a promise. When the JSON data is fetched from the file, the then function will run with the JSON data in the response.

If anything goes wrong (like the JSON file cannot be found), the catch function will run.

JSON Stringify

The JSON.stringify() method converts a *JSON*- to a string.

Syntax

JSON.stringify(value [, replacer [, space]])

In its simplest and most used form:

JSON.stringify (`value` **)**

Parameters

value : The JavaScript value to be 'stringified'.
replacer : (Optional) A function or an array which serves as a filter for properties of the value object to be included in the JSON string.
space : (Optional) A numeric or string value to provide indentation to the JSON string. If a numeric value is provided, that many spaces (upto 10) act as indentaion at each level. If a string value is provided, that string (upto first 10 chracters) acts as indentation at each level.

Return type

The return type of the method is: string.

```
const newJoke = {
  categories: ['dev'],
  value: "Chuck Norris's keyboard is made up entirely of Cmd keys because
Chuck Norris is always in command."
};

console.log(JSON.stringify(newJoke));
// {"categories":["dev"],"value":"Chuck Norris's keyboard is made up
entirely of Cmd keys because Chuck Norris is always in command."}

console.log(typeof JSON.stringify(newJoke)); // string
const user = {
  id: 101010,
  name: "Derek",
  email: "derek@awesome.com"
};

function replacer(key, value) {
  if (typeof value === "number") {
    return undefined;
  }
  if (key === "email") {
    return "Removed for privacy";
  }
  return value;
}

console.log(JSON.stringify(user));
// result: {"id":101010,"name":"Derek","email":"derek@awesome.com"}

console.log(JSON.stringify(user, replacer));
// {"name":"Derek","email":"Removed for privacy"}
```

JSON.parse

JSON is used to exchange data from a web server. Data is always received in a string form from a web server.

JSON.parse(), method helps to convert string data form into a JavaScript object.

```
var myObj = '{ "name": "Black Widow", "age": 32, "city": "New York" }';
var result = JSON.parse(myObj);
// Output: { name: "Black Widow", age: 32, city: "New York"}
```

Asynchronous Operations

Asynchronous operations provide a response after being processed using Web APIs. The purpose of writing an asynchronous function is to utilize the function's output for subsequent operations.

JavaScript is single-threaded. This means that it carries out asynchronous operations via the callback queue and event loop.

In synchronous JavaScript, each function is performed in turn, waiting for the previous one to complete before executing the subsequent one. Synchronous code is written from top to bottom.

JavaScript Event Loop

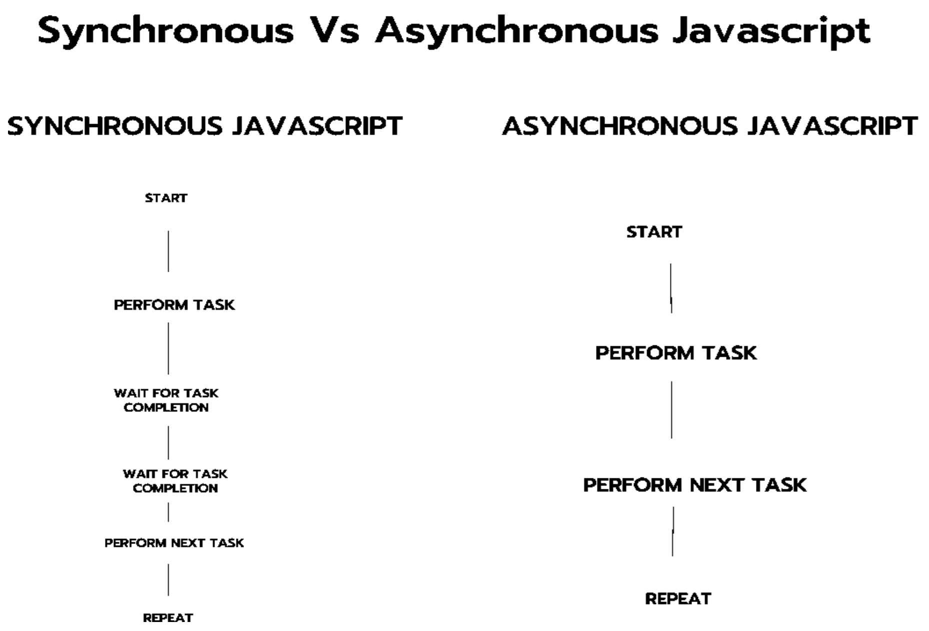
The JavaScript runtime uses an event loop, which works with the call stack, web APIs, the callback queue, and the micro-task queue to handle asynchronous operations. Here's how it works:

-Call Stack: Where your JavaScript code is executed using the Last In, First Out (LIFO) principle.

-Web APIs: Browser provided APIs like DOM, AJAX, and Timers that can handle asynchronous tasks outside the JavaScript engine.

-Callback Queue: Where your callbacks are pushed to wait for execution.

-Micro-task Queue: Monitors the call stack and pushes the first event from the callback queue when the stack is empty



Asynchronous is a non-blocking architecture, so the execution of one task isn't dependent on another. Tasks can run simultaneously.

Synchronous is a blocking architecture, so the execution of each operation is dependent on the completion of the one before it. Each task requires an answer before moving on to the next iteration.

- **Async** is multi-thread, which means operations or programs can run in parallel.
- **Sync** is single-thread, so only one operation or program will run at a time.
- **Async** is non-blocking, which means it will send multiple requests to a server.
- **Sync** is blocking — it will only send the server one request at a time and will wait for that request to be answered by the server.
- **Async** increases throughput because multiple operations can run at the same time.
- **Sync** is slower and more methodical.

Why Asynchronous JS Programming is required:

JavaScript is synchronous and has a single call stack. Code will be executed in a last-in, first-out (LIFO) order in which it is called.

```
constmySet = new Set();
constmySetFromArray = new Set([1, 2, 3, 2, 1]); // Duplicates are
automatically removed
console.log(mySetFromArray)
```

Example for synchronous:

```
console.log("synchronous.");
console.log("synchronous javascript!");
console.log("Be good do good");

//Output
synchronous.
synchronous javascript!
Be good do good
```

Synchronous JavaScript runs from top to bottom, as we can observe the console running the function linearly from top to bottom.

Async programming is vital because it allows multiple processes to run concurrently without interfering with the main thread.

This is significant because the **main thread is in charge** of managing the call stack, which is a **data structure** that holds the current sequence of function calls.

Blockage of the main thread leads to decreased performance because async programming permits the main thread to stay unblocked, and additional tasks can be completed while the asynchronous task is ongoing.

A JavaScript function call called the **setTimeout()** function, which allows us to run javascript code after a certain amount of time.

After the specified time, the setTimeout() method executes a block of code. The method only runs the code once.

Example:

```
console.log("asynchronous.");
setTimeout(() => console.log("asynchronous javascript!"), 3000));
console.log("asynchronous again!");

//Output
asynchronous.
asynchronous again!
asynchronous javascript!
```

Step 1: The first line of code will be executed, logging the string asynchronous to the console.

Step 2: The setTimeout method is invoked, which will execute the anonymous function after 3 seconds (3,000 milliseconds). The anonymous function will log asynchronous javascript! to the console.

Step 3: The third line of code will be executed, logging the string asynchronous again! to the console.

Step 4: After 3 seconds, the anonymous function from the setTimeout method will be executed, logging asynchronous javascript! to the console.

Techniques for writing asynchronous:

Callbacks: this allows for asynchronous code to be written in a synchronous fashion.

Promises: writing asynchronous JavaScript code is made easy with promises. They enable you to create code that runs after a predetermined period of time or when a predetermined condition is satisfied.

Async/await: async/await is a more recent technique for creating asynchronous JavaScript code. It helps you write more succinct and readable code that will execute after a set period of time or when a set condition is met.

Callbacks

A callback is a function that executes after the outer code call has finished running. It is supplied as an input to another function. When a function has completed its purpose, it is utilized to enable it to invoke another function.

Callbacks are used in arrays, timer functions, promises, event handlers, and much more.

Example:

```
function incrementDigits(callback) {
        callback();
}
```

The incrementDigits( ) function takes another function as a parameter and calls it inside.

a function that is passed to another function as a parameter is a callback function.

In order to carry out asynchronous actions in JavaScript, such as executing an AJAX request or anticipating a user click, callbacks are widely utilized.

Example:

```
setTimeout(() => {
        console.log("output is initiated after 5 seconds");
}, 5000)
```

The setTimeout() method is being used in this code to postpone the execution of a function. It will log the string output is begun after 5 seconds to the console using the function, which in this case is an arrow function. The provided delay is 5,000 milliseconds (or 5 seconds).

Example for callback hell:

```
function incrementDigits(num, callback) {
setTimeout(function() {
        num++;
        console.log(num);
        if (num< 10) {
                incrementDigits(num, callback);
         } else {
                callback();
         }
    }, 1000);
}
incrementDigits(0, function() {
    console.log('done!');
});
```

The setTimeout() function is used to delay the execution of the code in order to simulate a longer process. The function will print out each number, starting at 0, until it reaches 10, and then the done! message will be printed out.

As a result, the code can run asynchronously, which means that the functions can run concurrently rather than waiting for one another to complete.

Handling callback hell

The way to create a callback function is to pass it as a parameter to another function and then call it back after the task is completed.

There are ways to handle callbacks.

1. Use of promise: a promise can be generated for each callback. When the callback is successful, the promise is resolved, and if the callback fails, the promise is rejected.

2. Use of async-await: asynchronous functions are executed sequentially with the use of await; execution stops until the promise is revolved and function execution is successful.

Synchronous callbacks

listing few of callback methods in java script they are:

- array.map(callback),
- array.forEach(callback),
- array.find(callback),
- array.filter(callback),
- array.reduce(callback, init)

The asynchronous callback

The asynchronous callback is executed after the execution of the higher-order function.

Example1:

```
function a() {
 b();
}
function b() {
 setTimeout(() => {
  console.log("After 5 secs");
 }, 5000);
}
function c() {
 console.log("Welcome to KMIT");
}
a();
c();
```

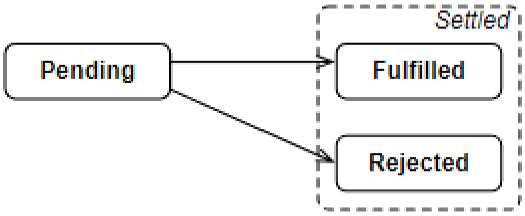
Output:

Welcome to KMIT

After 5 secs

Promise

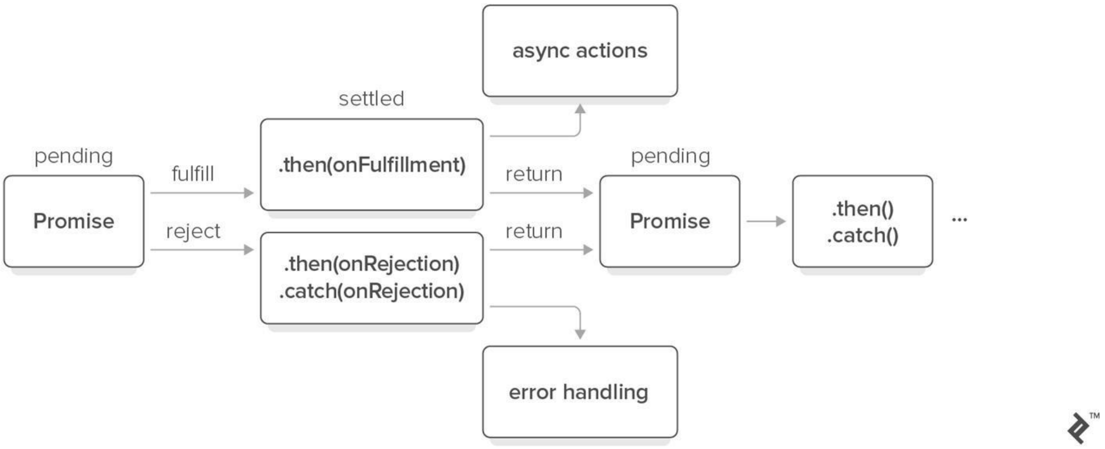
An async promise operation’s eventual success or failure is represented as a JavaScript object. It enables the creation of asynchronous code that works and appears synchronous. A promise, in our context, is something that will take some time to do. A promise has three possible states: pending, fulfilled, or rejected.



Pending: the promise has been created, and the asynchronous function it's associated with has not succeeded or failed yet. This is the state your promise is in when it's returned from a call to fetch(), and the request is still being made.

Fulfilled: the asynchronous function has succeeded. When a promise is fulfilled, its then() handler is called.

Rejected: the asynchronous function has failed. When a promise is rejected, its catch() handler is called.



```
let promise =newPromise(function(resolve, reject){
// create a new promise to resolve or reject
});
```

The constructor function takes a function as an argument. This function is called the executor function.

```
// Promise constructor as an argument

function(resolve, reject){
// doSomethingHere
}
```

The executor function takes two arguments, resolve and reject.

which runs automatically when a new promise is created and processes the code as follows:

Step 1: A new promise is created using the promise constructor and two arguments, resolve and reject.

Step 2: The `.then()“ method is called on the promise, which takes two callback functions, one for if the promise is resolved and one for if the promise is rejected.

Step 3: The first callback function will be executed if the promise is successfully resolved, while the second callback function will be executed if the promise is rejected.

```
function myDisplayer(some) {
  console.log(some);
}

let myPromise = new Promise(function(myResolve, myReject) {
  let x = 0;

// The producing code (this may take some time)

  if (x == 0) {
    myResolve("OK");
  } else {
    myReject("Error");
  }
});

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

Creating a Promise

A promise is created using the Promise constructor which takes a function called the executor. This function is executed immediately by the JavaScript engine and is given two functions as parameters: resolve and reject.

Example of Creating a Promise:

```
const myPromise = new Promise((resolve, reject) => {
  const condition = true; // A condition for demonstration
  if (condition) {
    resolve('Promise is resolved successfully.');
  } else {
    reject('Promise is rejected.');
  }
});
```

Consuming Promises

To consume values fulfilled by a promise, you use the then() method. To handle errors, you use the catch() method. There is also finally() which is executed no matter the outcome.

Example of Consuming a Promise:

```
myPromise.then((value) => {
  console.log(value); // Logs if resolved
}).catch((error) => {
  console.error(error); // Logs if rejected
}).finally(() => {
  console.log('This is executed regardless of the promise fate.');
});
```

Chaining Promises

Promises can be chained, meaning that the output of one promise can be used as the input for another promise. This is a powerful feature that keeps code clean and avoids the "callback hell" scenario.

Example of Chaining Promises:

```
const cleanRoom = () => {
  return new Promise((resolve) => {
    resolve('Cleaned the room');
  });
};
const removeGarbage = (message) => {
  return new Promise((resolve) => {
    resolve(`${message}, then removed garbage`);
  });
};
const getIceCream = (message) => {
  return new Promise((resolve) => {
    resolve(`${message}, then got ice cream`);
  });
};
cleanRoom()
  .then(result => removeGarbage(result))
  .then(result => getIceCream(result))
  .then(result => console.log('Finished:', result));
```

Error Handling

Handling errors in promises is crucial to write robust applications. The catch() method is specifically designed for catching any errors that occur during the promise execution or in the previous then() handlers.

Example of Error Handling:

```
new Promise((resolve, reject) => {
  throw new Error('Something went wrong!');
}).catch(error => {
  console.error('Error:', error.message);
});
```

Promise Utilities

Promise.all([promises]): Takes an array of promises and returns a single Promise that resolves when all of the promises have resolved or when any one of them is rejected.

Promise.race([promises]): Returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects, with the value or reason from that promise.

Example of Promise.all:

```
Promise.all([Promise.resolve('hello'), Promise.resolve('world')])
  .then(results => console.log(results.join(' '))); // "hello world"
```

Example of Promise.race:

javascript

```
Promise.race([
  new Promise((resolve) => setTimeout(() => resolve('slow'), 500)),
  new Promise((resolve) => setTimeout(() => resolve('fast'), 100))
]).then(value => console.log(value)); // "fast"
```

Async/await

The use of await, which halts execution until the promise is fulfilled, allows us to write asynchronous functions as though they were synchronous (executed sequentially). Async/await is a technique for building asynchronous code that looks and behaves like synchronous code. It enables you to construct code that appears to run in sequence but operates asynchronously.

Example:

```
//async/await

const result =asyncFunc();
const asyncFunc = async()=>{
        const result =await asyncFunc();
        //doSomethingNow
}
```

To perform an asynchronous operation, the incrementDigits() function can be written using asynchronous functions and the async/await idiom.

```
// Example using async/await
const incrementDigits =a sync num=>{
num++;
    console.log(num);
if(num<10){
 await incrementDigits(num);
}else{
 return'done!';
}
};

(async()=>{
const res =await incrementDigits(0);
    console.log(res);
})();
```

The async/await syntax is used in this code fragment to increment a number until it reaches 10. The async keyword, which comes before the declaration of the incrementDigits() function, denotes that a promise will be returned by the function.

Error handling in asynchronous JavaScript

Error handling in asynchronous JavaScript involves try/catch blocks, which are used in JavaScript to catch any mistakes that might happen throughout the asynchronous process.

```
try{
 alert("This is an error function!")
}catch(e){
 alert(e)
}
```

A catch block can have parameters that will give you error information. Generally, the catch block is used to log an error or display specific messages to the user. Now that the “incrementDigits()“ asynchronous function has an error handler function

Example

```
const incrementDigits = async num=>{
try{
 num++;
 console.log(num);
 if(num<10){
  await incrementDigits(num);
 }else{
  return'done!';
 }
}catch(err){
  console.log(err);
}
};

(async()=>{
const res =await incrementDigits(0);
    console.log(res);
})();
```