JavaScript

JavaScript is a cross-platform, i.e. , used both on the client-side and server-side, object-oriented scripting language. It is primarily used for enhancing the interaction of a user with the webpage. In other words, one can make their webpage more lively and interactive with the help of JavaScript. JavaScript is also being used widely in Web development, game development etc. It was originally named Mocha, but quickly became known as LiveScript and, later, JavaScript.

History Of JavaScript

- JavaScript was invented by **Brendan Eich** in 1995 and became an ECMA standard in 1997.
- ECMAScript is the official name of the language.
- ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
- Since 2016 new versions are named by year (ECMAScript 2016 / 2017 / 2018).
- Now the latest version of JavaScript, i.e. ES6, is used.

Features of JavaScript

- An interpreted language
- Embedded within HTML
- Minimal Syntax - Easy to learn(C syntax and java OOC)
- Mainly used for client-side scripting because all browsers support it
- Designed for programming user events
- Platform Independence/ Architecture Neutral

What Is JavaScript Used For?

1. **Adding interactive behavior to web pages**

- Add styles to Web Page.
- Add Pop-ups to a Web page.
- Add HTML content.
- Add animations.

- Add click events.

2. **Creating web and mobile apps** - Building web servers and developing server applications
3. **Building web servers and developing server applications** - Beyond websites and apps, JavaScript is also used to build simple web servers and develop the back-end infrastructure using Node.js.

Variables And Data Types

Data Types

There are 7 data types available in JavaScript. You need to remember that JavaScript is loosely typed(or dynamic language), so any value can be assigned to variables in JavaScript.
According to the latest ECMAScript standard, there are 6 primitive data types and 1 non-primitive object.

Primitive	Non-Primitive
Number - represents integer and floating values	Object - represents key-value pair
String - represents textual data	
Boolean - logical entity with values as true or false.	
Undefined - represents a variable whose value is not yet assigned.	
Null - represents the intentional absence of value.	
Symbol - represents a unique value.	

All of them except Object have immutable values, i.e. the values which cannot be changed.

Variables

Variables are named containers for storing data (values). You can place data into these containers and then refer to the data simply by calling the container.
There are 3 ways to declare a JavaScript variable:

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

Variable Declaration

Variable declaration is only declaring a variable with a variable name and not assigning any value to it.
Here temp is the variable name without any value. By default, it is assigned with **undefined.**

Variable Definition

Variable definition is a declaration of a variable with a variable along with assigning a value to it.
Now, the temp variable is assigned with a value of 10

JavaScript Identifiers

JavaScript variables are identified with some unique names.
These unique names are called identifiers.

Rules for assigning unique identifiers to a variable:

- Names can contain letters, digits, underscores, and dollar signs.
- Names must start with a letter (a to z or A to Z), underscore(_), or dollar($) sign.
- Names are case sensitive (temp and Temp are different variables)
- Reserved words (like JavaScript keywords) cannot be used as names

Assigning values to a variable

JavaScript is an untyped language. This means that a JavaScript variable can hold a value of any data type. Unlike many other languages, you don't have to tell JavaScript during variable declaration what value the variable will hold.
In other languages like Java/C++, you need to tell the data type of a variable before assigning a value to it
The value type of a variable can change during the execution of a program, and JavaScript takes care of it automatically.
- - - - - -

Scopes, let & const

Scope in JavaScript
JavaScript has three types of scope:

- Block scope
- Function scope
- Global scope

Variables defined inside a function are in local scope, while variables defined outside of a function are in the global scope. Each function, when invoked, creates a new scope.
Previously before ES6, JavaScript had only Global Scope and Function Scope.

ES6 introduced variable declaration methods like let, const, which now provides block scope also.

Block Scope

Variables declared inside a { } block cannot be accessed from outside the block.
To provide a block scope, you can't use the **var** keyword; either use the let or const keyword.

Function

A JavaScript function is a block of code designed to perform a particular task.
A JavaScript function is executed when it is called.

Local Scope

Variables declared within a JavaScript function becomes **local variables** to the function and have a local scope.
A variable had a local scope to the function, and it can only be accessed within that function.

- Variables with the same name in different functions can also be declared
- Variables made inside a function are moved to the memory when the function is called and deleted from the memory as soon as the function completes its work

Function Scope

- Variables declared inside the function are only accessible within that function.
- Declaring two variables with the same name in different functions is allowed and gives no error.

Variables declared with **var**, **let,** and **const** are quite similar when declared inside a function.

Global scope

- Variable declared outside a function have global scope.
- Global variables can be accessed from anywhere in a JavaScript program.
- Variables declared with **var, let and const** act as same if declared globally.

Block scope vs Function scope

Variables declared with the var keyword are global if declared within a block, but the declaration is made in a function. It can only be accessed within that function.

Hoisting of var

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.
So var variables are hoisted to the top of their scope and initialised with a value of undefined.

Let

let is now preferred for variable declaration. It's no surprise as it comes as an improvement to var declarations. It also solves the problem with var that it is globally available.

Properties of let :

- **let is { } block scoped**
- Declarations of let variables are also hoisted, but we cannot use them before its variable definition. The block of code is aware of the variable, but it cannot be used until it has been declared.
- **Let can be updated but not re-declared**: Just like var, a variable declared with let can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope.
- Updation is allowed but Re-declaration is NOT allowed.

Const

Variables declared with the const maintain constant values. Const declarations share some similarities with let declarations.

Properties of const:

- **const declarations are also block scoped**. Like let declarations, const declarations can only be accessed within the block they were declared.
- **Hoisting of const**: Just like let, const declarations are hoisted to the top but are not initialised.

Differences and Key-Points

- var declarations are globally scoped, or function scoped, while let and const are blocks scoped.
- var variables can be updated and re-declared within its scope; let variables be updated but not re-declared; const variables can neither be updated nor re-declared.
- They are all hoisted to the top of their scope. But while var variables are initialised with undefined, let, and const variables are not initialised.

- While var and let can be declared without being initialised, const must be initialized during declaration.

Methods

- **tostring( ) :** Converts from Number to String
- **toFixed( ) :** returns a string, with the number written with a specified number of decimals

```
let num = 10.126;
 num.toFixed(0); // 10
 num.toFixed(3) ; //10.126
```

- **toPrecision( ) :** returns a string, with a number written with a specified length

```
let num = 10.58 ;
num.toPrecision(3) ; // 10.6
num.toPrecision(2) ; // 11
num.toPrecision(5) ; //10.580
```

- **Number( ) :** Converts from a variable to number

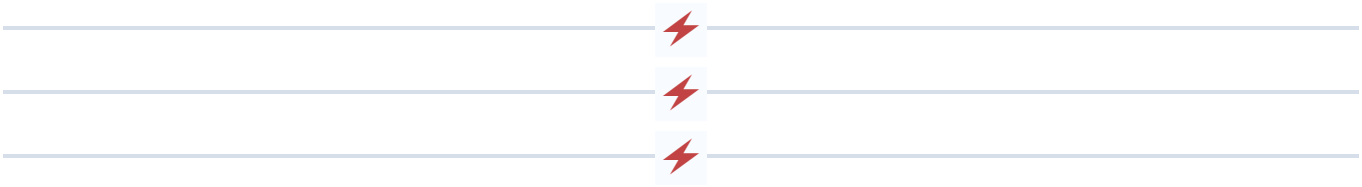
```
Number(“10”); // Returns 10 , string → number
Number(“10.5”); // Returns 10.5
Number(“Coding Ninjas ”); // Returns NaN
```

- **parseInt( ) :** Parses its argument and returns an integer

```
parseInt("-5"); // returns -5
parseInt("-5.25"); //returns -5 , whole numbers only
parseInt("10 20 30"); // returns 10
parseInt("10 Coding Ninjas"); // returns 10
parseInt("Coding Ninjas 10"); // returns NaN
```

parses a string and returns a whole number, does not contains decimals.
Spaces are allowed.
Only the first number is returned.

- **parseFloat( ) :** Parses its argument and returns a floating point number



Conditionals

Conditional statements are used to run different sets of statements according to the conditions. This means that based upon various conditions, we can perform different actions.

if-else Statements

The **if** statement specifies a block of statements (JavaScript code) that gets executed
when the **condition** specified in the if statement evaluates to true.
Else, if the condition is false, then the statements inside the **else** block are executed.

```
if (condition) {
        //code to be executed if condition1 is true
} else {
        //code to be executed if the condition is false
}
```

Switch Statement

The **switch** statements are similar to **else-if** statements but **work on a single expression**. This expression evaluates different values which decide which block of
code needs to be executed.

```
Syntax :
    switch(expression) {
         case x:
                 // code block
```

```
        break;
        case y:
                // code block
        break;
        default:
                // code block
    }
```

Work Flow of Switch Statements :

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

break Keyword

We have provided a **break;** statement after each case is completed to stop the execution of the **switch** block. If the **break** is not present, then the cases after the one that matches also get executed until a **break;** is found or the **switch** block does not end.

> **Tip**
>
> If you omit the break statement, the following case will be executed even if the evaluation does not match the case

default Keyword

The default keyword specifies the code to run if there is no case match
default case does not have to be the last case in a switch statements

Loops

Loops are used to do something repeatedly. You need to print 'n' numbers, then you can use a loop to do so. There are many different kinds of loops, but they almost do the same thing. Their use varies depending on the type of situation, where one loop

will be easy to implement over another.
They are used to run the same code repeatedly, each time with a different value.

Types of loops

- **for**
- **for/in**
- **for/off**
- **while**
- **do-while**

for loop

A for loop is used to repeat something until the condition evaluates to false.

```
for ([initializationStatement]; [condition]; [updateStatement])
    {
            //code to be executed multiple times
    }
```

- The *initialization statement* is used to initialize loop counters.
- The *condition* is the expression that is evaluated to Boolean value **true** or **false.**
- The *update statement* is used to update the loop counters.

Initialization statement

- The initialization statement is optional, and you can initialize these statements before the **for** loop.

> **Tip**
>
> - You need to provide a semicolon for each part of the loop, even if the options are missing.
> - You can provide more than one *initialisation statement* using a comma(",") as a separator.
> - When using multiple initialisation variables , it is not mandatory to mention conditions for each variable, but it is compulsory to mention an

update statement for each of them

```
for (let i = 1, j=3; i ⩽ 3; i++, j--) {
     console.log( i+j );
     }
     Output : 4 4 4
```

For/in and For/of Loops

- **for/in** - loops through the properties of an object
- **for/of** - loops through the values of an iterable object

while loop

The while statement executes the statements until the condition is not **false**. First, the condition is evaluated, and if it is true, then the statements are executed, and the condition is tested again. The execution stops when the condition returns false.

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

do while loop

The do-while loop is similar to the while loop, except that the statements are executed at least once

```
do {
        // code to be executed multiple times
        // updation
 }
 while (condition);
```

Jump Statements

In JavaScript there are two types of jump statements :

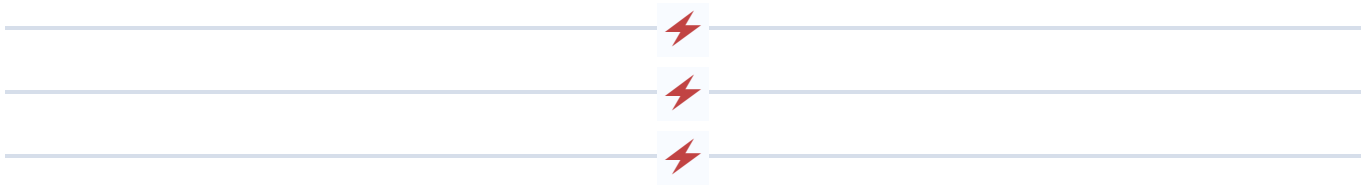
- break
- continue

break Statement

We talked about this in the for loop when there is no condition passed then it is used to **jump out** of a switch( ) statement. The break statement can also be used to jump out of a loop.

continue Statement

The **continue** statement breaks one iteration (in the loop) if a specified condition occurs and continues with the next iteration.



Functions

JavaScript is based on **functional programming.** Therefore, functions are fundamental building blocks of JavaScript.
The function contains a set of statements that perform some task.
The function execution stops either when all the statements have been executed or a return statement is found. The return statement stops the execution of the function and returns the value written after the return keyword. . A function may or may not return some value after its execution.

Function Arguments

JavaScript is a dynamic language, and therefore it allows passing different numbers of arguments to the function and does not give error in this condition.

- **Passing fewer arguments** - In this case, when few arguments are passed, the other parameters that do not get any value assigned to them get value **undefined** by default.

- **Passing more arguments -** In this case, when more arguments are passed, the extra arguments are not considered.

```
function add(a, b, c){
         return a+b+c;
  }
  console.log(add(10, 20)) ; // Prints - NaN
  console.log(add(10, 20, 30, 40)); // Prints - 60
```

Default Parameters

If you pass fewer arguments than in the function, the remaining parameters default to **undefined in JavaScript**. But you can also set your default values to them.

```
function findInterest(p,r=5,t=1) {
     console.log("Interest over", t, "years is:",(p*r*t)/100);
  }
findInterest(1000); // Prints - Interest over 1 years is: 50
findInterest(1000, 7); // Prints - Interest over 1 years is: 70
findInterest(1000, 8, 2); // Prints - Interest over 2 years is:
160
```

Rest Parameters

The rest parameter syntax **(...variableName)** is used to represent an indefinite number of arguments. It is defined in an array-like structure.

```
`If we want to add at least 3 number`
function addAtLeastThree(a, b, c,  ... numbers){
     var sum = a+b+c;
      for(var i=0; i<numbers.length; ++i) {
          sum += numbers[i];
          }
      return sum;
  }
  console.log(addAtLeastThree(10, 20, 30, 40, 50)); // Prints -
150
```

Hoisting

JavaScript provides an exciting feature called hoisting. This means that you can use a variable or function even **before it's a declaration.**
Hoisting is a mechanism in JavaScript where variables and function declarations are moved to the top of their scope before code execution.

> **Danger**
>
> If you use a variable or function and do not declare them somewhere in the code, it will give an error.

Variable Hoisting

Variable hoisting means that you can use a variable even before it has been declared.

```
Console.log(a); // Prints - undefined
//Other JavaScript Statements
var a = 10;
`The above prints 'undefined' because only the variable
declaration is moved to the top, not the definition.`
```

Function Hoisting

Function hoisting means that you can use function even before function declaration is done.

```
console.log(cube(3)); // Prints - 27 //Other JavaScript
Statements function cube(n) {
     return n*n*n;
  }
```

> **Danger**
>
> You cannot use function hoisting when you have used function expression. If you use function expression and use function hoisting, then it will result in an

error.

Function within Function

You can define a function within a function which we can also call a **nested function.** The nested function can be called inside the **parent** function only.
The nested function can access the variables of the parent function and as well as the global variables.
But the parent function cannot access the variables of the inner function.
This is useful when you want to create a variable that needs to be passed to a function. But using a global variable is not good, as other functions can modify it. So, using **nested functions will prevent the other functions from using this variable.**

```
`Using a count variable that can only be accessed and modified
by the inner function`
function totalCount(){
     var count = 0;
      function increaseCount(){
          count++;
        }
    increaseCount();
    increaseCount();
    increaseCount();
    return count;
}
console.log(totalCount( )); // Prints - 3
```

Scope Chain

This tells how do interpreters look for a variable. This is decided according to the Scope Chain.
When a variable is used **inside a function**, the JavaScript interpreter looks for that variable inside the function. If the variable is not found there, it looks for it in the outer scope, i.e., the parent function. If it is still not found there, it will move to the outer scope of the parent and check it; and do the same until the variable is not

found. The search goes on until the global scope, and if the variable is not present even in the global scope, the interpreter will throw an error.

```
function a(){
 var i = 20;
    function b(){
     function c(){
      console.log(i); // Prints - 20
        }
    c();
    }
    b();
    }
    a();
```

Function Definition and Function Expression

Functions in JavaScript can be defined in two ways –

- **Function Definition -** Creating a function using function keyword and function name.
- **Function Expression -** Creating a function as an expression and storing it in a variable.

Function Definition

A function definition is creating a function in a normal way, which we have read until now.
Here, the parameters take values differently for different types of variables. We can either pass primitive value or non-primitive value.

- If the **value passed as an argument is primitive**, then it gets passed by value. This means that the changes to the argument does not reflect the changes globally and only remains local.

```
function abc(b){
b = 20;
console.log(b); // Prints - 20
```

```
}
var a = 10;
abc(a);
console.log(a); // Prints - 10
```

- If the **value passed as an argument is non-primitive**, then it gets passed by reference. This means that change is visible outside the function.

```
function abc(arr2){
     arr2[1] = 50;
    console.log(arr2); // Prints - Array [10, 50, 30]
}
var arr1 = [10, 20, 30];
 abc(arr1);
 console.log(arr1); // Prints - Array [10, 50, 30]
```

Function Expression

Variables can take primitive and non-primitive values. So the function is one of the possible values that a variable can have. A function expression is **used to assign the function to a variable.**

```
var add= function sum(a , b){
 return a + b;
  }
  console.log(add( 2 , 4 )); // Prints – 6
```

However, note that the name **sum** that this function has can be used only inside the function to refer to itself; it can't be used outside the function.

```
var factorial = function fac(n){
 var ans = 1;
  for(var i = 2; i ⩽ n; i++){
   ans *= i;
}
return ans;
```

```
}
 console.log(factorial(3)) ; // Prints - 6
```

Passing Function As Argument (CallBack Function)

Functions in JavaScript are objects. So we can also pass a function as an argument to another function.

- Pass function as argument like - function abc(arg1, functionName);
- Define function as an argument like -
  function abc(arg1, function( ) { });

```
function abc(a, b, compute){
 compute( a , b );
  }
function multiple(a, b){
console.log(a*b);
}
function add(a, b){
console.log(a+b);
}
abc(5, 2, multiple); // Prints - 10
abc(5, 2, add); // Prints - 3
```

The function passed is also called the **callback function**. A callback is a code that is passed as an argument to another code, which is expected to execute the argument(function) at some convenient time.

> **Tip**
>
> JavaScript is an event-driven language, meaning that instead of waiting for a response from a function, it keeps executing the code in sequence. So if you want to execute something after some line of code, then callbacks are very useful.

Arrow Function

Arrow functions allow us to write shorter function syntax :

```
`Before :`
print = function(){
return “Coding Ninjas”;
 }
```

```
`After :`
print = () ⇒ {
return “Coding Ninjas”;
 }
```

If the function has only one statement, and the statement returns a value, you can remove the brackets and the return keyword

Arrow Function With Parameters

```
sum = ( a, b )⇒{
console.log(a+2);
console.log(b+2);
 }
sum(2,3);
Output : 4 5
```

Arrays

The array is an **ordered collection of data**(can be primitive or non-primitive) used to store multiple values. This helps in storing an indefinite number of values. Each item/value has an **index** attached to it, using which we can access the values. In JavaScript index starts at 0.
The array also contains a property **length** that stores the number of elements present inside the array. It changes its value dynamically as the size of the array changes.

Creating an Array

There are two ways to create an array -

- **Using square brackets
- **Using Array Object
- **Using Array indexes

1.Using square brackets

- Create an empty array as

```
var arrayName = [ ];
```

- Create an array with initial values as

```
var arrayName = [value1, value2, …, valueN] ;
```

2. Using Array Object

Create an empty array using a **new** keyword

```
var arrayName = new Array( ) ;
```

Create an array with some length as

```
var arrayName = new Array(N) ;
`where 'N' is length of array.`
```

Providing the values

```
var arrayName = new Array(value1, value2, …, valueN) ;
```

```
Examples :var arr = [ 1 , 2 , 3 , 4 ] ;
    var arr = new Array(1 ,2 , 3 ,4 ) ;
`Both the statements do the same work`
```

3. **Using Array indexes

Create an empty array, and then provide the elements.

```
Example :var arr = [ ];
arr[0]= 1 ;
arr[1]= 2 ;
arr[2]= 3 ;
Creates an array [ 1 , 2 , 3  ]
```

Placing Elements at Outside Array Range

When a value is assigned to a positive index outside the range of the array, then the array stores this value at the specified index and all other indices before it are filled with **undefined.** The length of the array also changes to **index+1.**
When a negative value is used, the array stores the element as a key-value pair, where the negative index is the key and the element to be inserted is the value.

```
Example :var arr = [ 1 , 2 , 3 ] ;
        arr[5] = 10 ;

`Initially, the array has a size of 3; now, the 10 is pushed at
the 5th index, size becomes 6, and the remaining index is filled
with undefined values`

arr becomes : [ 1 , 2 , 3 , undefined , undefined , 10 ] ;
```

Accessing Element in Array

You can access the individual elements of the array using the **square bracket notation.**

```
Example:var arr = [ 1 , 2 , 3 ] ;
console.log( arr[1] ) ;  // Output 2
```

Modify the value

```
Example :var arr = [ 1 , 2 , 3 ] ;
    arr[1] = 20 ;
array now becomes -  [ 1 , 20 , 3 ]
```

Tip

When using an array inside an array, you can access the value of the inner array directly
If you access the array outside its range, i.e., pass an invalid index(whether negative or greater than the array's length), then **undefined** is returned.

```
Example :   var arr = [ 1 , 2 , [ 3 , 4 ] ];
    arr[2][1]; //  return 3
    arr[5];     // return undefined
    arr[-1];   //return undefined
```

Heterogeneous in Nature

It can contain different types of values at the same time. Also, the array can store primitive and non-primitive values.

```
Examples :var arr = [ 1 , 2 , "Coding Ninjas" ] ;
`arr contains both Number and String type values`
var arr = [ 1 , "Coding Ninjas" , [3,4] ] ;
`arr contains Number, String and Array type values`
```

Functions on Arrays

push Method :

The push( ) method **adds one or more elements to the end of the array and returns the new length of the array**. It uses the length property to add elements.
If the array is empty, the push( ) method will create the length property and add an

element.
In case you are adding multiple elements, separate them using a comma(,).

```
Example : var arr = [ 1 , 2 , 3 ] ;
arr.push(4) ;
arr becomes [1 , 2 , 3 , 4 ]

arr.push(5,6) ;
arr becomes [1 , 2 , 3 , 4 , 5 , 6]
```

pop Method :

The pop() method is used to **remove the last element from the array and return that element**. It also decreases the length of the array by 1.
Using pop on an empty array returns 'undefined'.

```
Example :var arr = [ 1 , 2 , 3 ] ;
arr.pop( ) ;  // returns 3
```

shift Method :

The shift( ) method is used to **remove the first element from an array and return that element.**
If the length property is 0, i.e. empty array, then **undefined** is returned.

```
Example :var arr = [1, 2 , 3 ] ;
arr.shift( ) ;    // returns 1

var arr = [ ] ;
arr.shift( ) ;    // returns undefined
```

unshift Method :

The unshift( ) method is used to **add one or more elements to the beginning of the array and returns the new length of array.**
In case, you are adding multiple elements, separate them using a comma(,).

```
Example :var arr = [ 1 , 2 , 3 ] ;
arr.unshift( 0 ) ;
arr becomes [0 , 1 , 2 , 3 ]

arr.unshift( 10 ) ;
arr becomes [10 , 0 , 1 , 2 , 3 ]
```

indexOf Method :

The indexOf( ) method is used to **return the first index at which the given element is found in the array**. If the element is not found, then -1 is returned. By default, the whole array is searched, but you can provide the start index from which the search should begin. It is optional. If the index provided is negative, the offset is set from the end of the array, and a search in the opposite direction is done.

```
Example :var arr = [ 1 , 2 , 3 , 2 , 5 ] ;
arr.indexOf( 2 );      // return 1
arr.indexOf(2 , 2);  // return 3
```

splice Method :

The splice( ) method is used to **remove or replace or add elements to an array**. If an element is removed, it returns the array of deleted elements. If no elements are removed, an empty array is returned.

```
Syntax :arr.splice(start ,  deleteCount , item1, …, itemN ) ;
```

```
Example : var arr =  [ 1, 2 , 3 , 4 ] ;
          arr.splice( 1 , 1 ) ;
arr becomes - [ 1 , 3 , 4 ]
```

reverse Method :

The reverse method is used to **reverse the content of the array** and return the new reversed array. This means that the first element becomes last and vice-versa.

```
Example :var arr = [ 1 , 2 , 3 , 4] ;
         arr.reverse( ) ;
arr becomes - [4 , 3 , 2 , 1 ]
```

sort Method :

The sort( ) method is used to **sort the elements of an array and return the sorted array**. The sort is done by converting the elements to string and then comparing them.

```
Example :var arr = [ 1 , -2 , 13 , 4] ;
         arr.sort( ) ;
arr becomes - [ -2 , 1 , 4 , 13 ]
```

join Method

The join( ) method is used to **concatenate all the elements in an array and return a new string.**
They are separated by comma(,) by default, but you can also provide your separator as a string. It is optional to provide a separator.

```
Example : var arr =  [ 1, 2 , 3 , “Coding Ninjas”]  ;
arr.join( ' ,' ) ;  // returns 1,2,3,Coding Ninjas as a String

var arr = [ 1, 2 , null , 3 ] ;
arr.join( ' ,' ) ;  // returns 1,2,,3 as a String
```

Tip

If an element in the array is **undefined** or **null**, it is converted into an empty string.

toString Method

The toString( ) method is used to **return the array in the form of a string**. The string contains all the elements separated by a comma.

We do not need to provide the separator as a parameter as we did in the join method.

```
Example :var arr =  [ 1, 2 , 3 , “Coding Ninjas”]  ;
arr.toString(  ) ;  // returns 1,2,3,Coding Ninjas as a String
```

Iterating over arrays

Iterating over the array in accessing the values and manipulating each one of them individually, using a lesser line of code. We have used two methods to iterate over the arrays.
for loop: It is used commonly to iterate over all the values of the array.

```
Example :var arr = [10, 20, 30] ;
 for(var i=0; i<arr; ++i) {
       console.log( arr[i]*2 ) ;
 }
Output : 20 40 60
```

forEach Method :

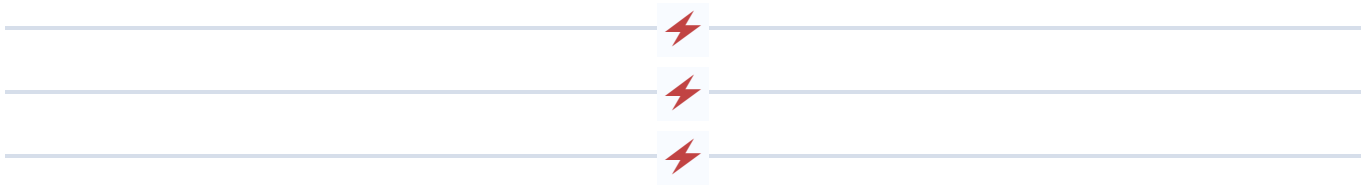
The forEach( ) method calls a function once for each array element.

```
Syntax :arr.forEach( function callback(currentValue, index,
array) {
 /* Function Statements */
}, thisArg);
```

You can either provide a function definition as shown in the syntax above. Or you can pass the function name to it.

```
Example :var items = [ 1 , 2 , 3 ] ;
 items.forEach( function(item) {
        console.log ( item * 10  );
  }) ;
Output :10
```

```
20
30
```



String

The String object is used to represent and manipulate a sequence of characters.
It could be written with both single and double quotes.

Length

The length of the string could be found using built-in property.

```
Example:
let text = "Coding Ninjas";
 console.log( text.length) ;     // Output 13
```

String Methods

substring( ) :

It is used to find a contiguous sequence of characters within a string using indxes

```
Example :
let str = “Coding Ninjas” ;
str.substring( 2 ) ;  // ding Ninjas
str.substring(3 , 8 ) ; // ing N
```

substr( ) :

It is same as substring method , the difference is that the second parameter specifies the length of the extracted part.

```
Example :
let str = “Coding Ninjas” ;
```

```
str.substr( 2 , 4 ) ;  // ding
```

replace( ) :

It replaces a specified value with another value in a string . But it does not affect the original string rather it makes another string the return the result

```
Example :
let str = “Hello World” ;
let str2 = str.replace("Hello", "Bye") ;   // Bye world
```

toUpperCase( ) & toLowerCase( ) :

toUpperCase( ) Converts the characters of string to uppercase characters and vice-versa for toLowerCase( )

```
Example :
let str = “Coding Ninjas” ;
let str2 = str.toUpperCase( ) ;    // CODING NINJAS
let str3 = str.toLowerCase( ) ;    // coding ninjas
```

trim( ) :

This method removes whitespace from both sides of a string

```
Example :
let str = “     Coding Ninjas     ”;
let str2 = str.trim(); // Coding Ninjas
```

Searching Methods in Strings

indexOf( ) :

This method returns the index of (the position of) the ***first*** occurrence of a specified character/text in a string

```
Example :
let str = “Coding Ninjas” ;
```

```
str.indexOf(“Ninjas” ) ; // 7
str.indexOf(“N” ) ; // 7
str.indexOf(“n” ) ; // 4
str.indexOf(“ninjas” ) ; // -1
```

lastIndexOf( ) :

This method returns the index of the last occurrence of a specified character/text in a string

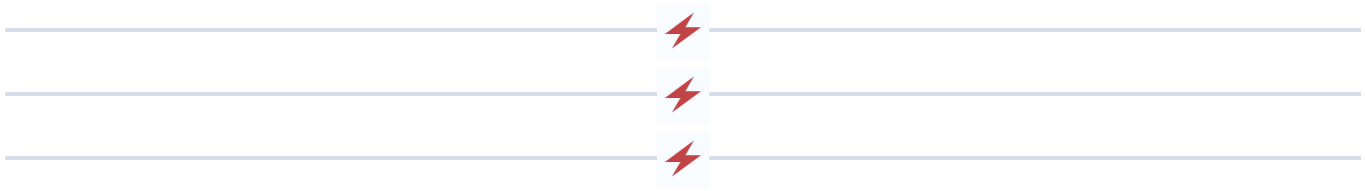
```
Example :
let str = “Coding Ninjas” ;
str.lastIndexOf(“N” ) ; // 7
str.lastIndexOf(“n” ) ; // 9
```

includes( ) :

This method returns true if a string contains a specified text

```
Example :
let str = “Coding Ninjas” ;
str.includes(“Ninjas” ) ; // true
```

> **Tip**
>
> These methods are case sensitive.



Objects

JavaScript objects are a collection of properties in a **key-value pair**. These objects can be understood with real-life objects, like similar objects have the same type of properties, but they differ from each other.

Some important points about objects are -

- Object contains properties separated with a comma(,).
- Each property is represented in a **key-value pair.**
- Key and value are separated using a colon(:).
- The key can be a string or variable name that does not contain special characters, except underscore(_).
- The value can contain any type of data - primitive, non-primitive and even a function.
- The objects are **passed by reference** to a function.

Creating an Object

Using curly brackets -

- Create empty object as - var obj = { } ;
- Object with some initial properties as -
  var obj = { key1: value1, … , keyN: valueN }
  Using new operator -
- Create empty object as - var obj = new Object( );
- Object with properties as -
  var obj = new Object({ key1: value1, ..., keyN: valueN })
  The properties can be created at the time of creating an object and also after that. **Both**

creating and accessing the properties share similar syntax.

Creating and Accessing Properties

The properties are created in a key-value pair, but some restrictions exist in the way some keys are created. There are two ways to create and access properties -

1. **Using a dot operator -** You can use dot operator only when the property name starts with a character. Property can be **accessed like** - obj.propertyName. Similarly, you can **create property** like - obj.propertyName = value

2. **Using a square bracket -** You need to use a square bracket when the key name starts with a number. If the name contains a special character, then it will be stored as a string. Property is **accessed like** - obj["propertyName"].

Similarly, you **create property** like - obj["propertyName"] = value

> **Tip**
>
> If you access a property that has not been defined, then **undefined** is returned.

You can also **set function as the value to the key.** So the key then becomes the method name and **need parentheses to execute**. So you can execute methods like - obj.methodName( ) and obj"methodName".

```
var ball = {
            sport : "Cricket",
            colour : "Yellow",
            radius : 3,
            print : function( ){
                 console.log("Coding Ninjas");
             }
        }
```

How are Objects Stored

There are two things that are very important in objects -

- Objects are **stored in a heap.**
- Objects are **reference types.**
  These two are important in regard that **object variables point to the location** where they are stored. This means that **more than one variable can point to the same location.**
  Until now, you are **creating new objects** every time like -

```
var item1 = { name: “Coding Ninjas” } ;
var item2 = { name: “Coding Ninjas” } ;
```

```
item1 == item2;    // Returns - false
item1 === item2;  // Returns - false
```

The **above two lines will create two different objects** are not therefore equal
But, if you assign **one object to another,** then the value of **'item1' gets assigned to 'item2',** and therefore, they both will point to the same location -

```
var item1 = { name: "Coding Ninjas" } ;
var item2 = { name: "Coding Ninjas" } ;
item1 = item2;
console.log(item1 == item2) ; // Returns true
console.log(item1 === item2) ; // Returns true
```

Iterating Objects

JavaScript provides a special form of loop to traverse all the keys of an object. This loop is called **'for...in'** loop.

```
Syntax : for (variable in object)
{ // Statements
}
```

Here the **'variable' gets assigned the property name** on each iteration, and 'object' is the object you want to iterate. Use the **square bracket notation with variables to access the property values.**

The **iteration may not be in a similar order as to how you see properties in objects** or how you have added them because the objects are ordered specially.
The **property names as integers are iterated first** in ascending order. Then the other names are iterated in the order they were added.

ARRAY AS OBJECT

Arrays are actually objects. If you use the **typeOf( )** method on an array, you will see that it will return an **object**. If you see an array on a console, they are **key-**

value pairs, with the **positive integers as the keys.**
Arrays can also store properties just like objects.

Arrays vs Object

- Arrays have a **length** property that objects does not have.
- You can access the values of the arrays like - array[0]; or array["0"]; whereas in objects, you have to use **double quotes ("")** only.
- Only when you use an integer as a key, it will change the 'length' property.
- Adding a non-integer key will not have any effect on the length' property.

> **Tip**
>
> Length property will be set according to the maximum integer key of the array.

Using for...in loop to Iterate

Since **arrays are also objects**, you can use the 'for-in' loop to traverse it.
Traversing the array using 'for-in' loop is the same like traversing an object.

```
var arr = [10, 20, 30];
arr["four"] = 40 ;
console.log(arr);
Output :
        Array(3) [ 10, 20, 30 ]

`But, it also contains the property "four: 40", but it does not
show in the array. But if you use the for-in loop to traverse
it, you can traverse all the properties.`

for(var i in arr) {
    console.log( i, ":", arr[i]);
    }
Output :
    0 : 10
    1 : 20
    2 : 30
    four: 40
```

this keyword

```
var person = {
    firstName: "Tony",
    lastName : "Stark",
    age : 40 ,
    getname: function( ) {
         return this.firstName + " " + this.lastName;
          }
    };
    console.log(person.getname( )) ; //Tony Stark
```

- In a function definition, this refers to the "owner" of the function.
- In the example above, **this** is the person object that owns the getname function.
- In other words, **this.firstName** means the firstName property of this object and **this.lastName** means the firstName property of this object

Class and Constructors

You can create objects in JavaScript using curly braces { … } syntax. But what if you need to create multiple similar objects. You can either write the same syntax for every object, or you can use the **constructor** to create objects.
Using { … } syntax to create multiple objects can create certain inconsistencies in code-

- There can be spelling mistakes.
- The code can become difficult to maintain
- Changes to all the objects will be difficult.
  To overcome all the above inconsistencies, JavaScript provides a **function constructor**. The constructor provides a **blueprint/structure for objects.** You use this same structure to create multiple objects.

Constructors technically are regular functions. There is one convention to constructors -

- The first letter of the function is capital.

  Objects can be created by calling the constructor function with the **new** keyword. Using a constructor means that -
- All of these objects will be created with the same basic structure.
- We are less likely to accidentally make a mistake than if we were creating a whole bunch of generic objects by hand.

  It is important always to use the **new keyword** when invoking the constructor. If new is not used, the constructor may clobber the 'this', which was accidentally passed in most cases as the global object (window in the browser or global in Node.). Without the **new** function will not work as a constructor.

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

var stu1 = new Student("John", "Doe", 50);
var stu2 = new Student("Sally", "Rally", 48);
```

The **new keyword is the one that is converting the function call into constructor call,** and the following things happen -

1. A brand new empty object gets created.
2. The empty object gets bound as this keyword for the execution context of that function call.
3. If that function does not return anything, then it implicitly returns this object.

> **Tip**
>
> **this** referred in the constructor bound to the new object being constructed.

Classes

Classes are introduced in ECMAScript 2015. These are **special functions that allow one to define prototype-based classes** with a clean, nice-looking

syntax. It also introduces great new features which are useful for object-oriented programming.

```
Example :class Person {
    constructor(first, last) {
        this.firstName = first;
        this.lastName = last;
        }
     }
```

The way we have defined the class above is known as the **class declaration** in order to create a new instance of class Person by using the new keyword.

```
let p1 = new Person("Vishal", "Mishra") ;
```

Some points you need to remember -

- You define class members inside the class, such as methods or constructors.
- By default, the body of the class is executed in strict mode.
- The constructor method is a special method for creating and initializing an object.
- You cannot use the constructor method more than once; else, Syntax Error is thrown.
- Just like the constructor function, a **new** keyword is required to create a new object.

> **Tip**
>
> The type of the class is a **function**

Getter and Setter

You can also have a **getter/setter** to **get the property value** or **set the property values** respectively. You have to use the **get** and **set** methods to create a getter and setter, respectively.

```
class Vehicle {

constructor(variant) {
this.model = variant;
}
get model( ) {
return this.model;
 }
set model(value) {
this.model = value;
}
 }

let v = new Vehicle("dummy");
console.log(v.model) ; // get is invoked
v.model = "demo" ;       // set is invoked
```

Class Expression

A class expression is another way to define a class, which is similar to a function expression. They can be **named and unnamed both**, like -

```
let Person = class { };
                OR
let Person = class Person2 { };
```

> **Tip**
>
> The name given to the class expression is local to the class's body.

Hoisting

Class declarations are **not hoisted**. If you try to use hoisting, it will return **not defined** error.

Instance

It is made using the blueprint of the class having its behaviors and properties.

```
`Example: Let's say Human is a class, and we are all its
instances`

class Human{
constructor(name , height , weight){
          this.name = name;
          this.height=height;
          this.weight=weight;
    }
}

let Sam = new Human("Sam" , "6" , "80");

`Sam is an instance of Human class.`
```

Encapsulation

The process of wrapping property and function within a single unit is known as encapsulation.

```
class Person{
   constructor(name,age){
       this.name = name;
       this.age = age;
   }
   add_Address(add){
       this.add = add;
   }
   getDetails(){
    console.log("Name is",this.name,"Address is:",this.add);
   }
}

let person1 = new Person('Sam',22);
person1.add_Address('Delhi');
person1.getDetails();
```

In the above example, we simply created a person instance using the constructor and Initialized its properties.
Encapsulation refers to the **hiding of data** or data Abstraction which means representing essential features hiding the background detail. Most of the OOP languages provide access modifiers to restrict the scope of a variable, but there are no such access modifiers in JavaScript but there are certain ways by which we can restrict the scope of variable within the Class/Object.

```
function person(firstname,lastname){

   let firstname = firstname ;
   let lastname = lastname ;
    let getDetails_noaccess = function( ){
         return (“Name is:”, this.firstname , this.lastname) ;
   }
    this.getDetails_access = function( ){
        return (“Name is:”, this.firstname , this.lastname) ;
   }
}

let person1 = new person(‘Sam’, ‘Cumberbatch’);

console.log( person1.firstname);  //Undefined
console.log( person1.getDetails_noaccess); // Undefined
console.log( person1.getDetails_access( ) );   // Name is Sam
Cumberbatch
```

In the above example we try to access some property **person1.firstname** and functions **person1.getDetails_noaccess** but it returns undefined while their is a method which we can access from the person
object **person1.getDetails_access( )** and by changing the way to define a function we can restrict its scope.

Inheritance

It is a concept in which another Object is using some property and methods of an object. Unlike most OOP languages where classes inherit classes, JavaScript Object

inherits Object other Objects can reuse, i.e. certain features (property and methods)of one object.

```
lass Vehicle {
constructor(make, model, color) {
this.make = make;
this.model = model;
this.color = color;
}
getName( ) {
return this.make + " " + this.model;
 }
}

class Car extends Vehicle{
getName( ){
return this.make + " " + this.model +" in child class.";
}
}

let car = new Car("Honda", "Accord", "Purple");
car.getName( ) ;  // "Honda Accord in child class."
```

In the above example, we define a Vehicle class with certain properties and methods. Then we inherit the Vehicle class in the Car class, use all the property and method of Vehicle class, and define certain property and methods for Car.

Method Overriding:

It allows a method in a child class to have the same name and method signature as that of a parent class.

> **Tip**
>
> **super** keyword is used to refer to the immediate parent class instance variable.

Polymorphism

Polymorphism is a core concept of an object-oriented paradigm that provides a way to perform a single action in different forms. It provides the ability to call the same method on different JavaScript objects. As JavaScript is not a type-safe language, we can pass any type of data member with the methods.

Polymorphism is the OOPs principle that provides the facility to perform one task in many ways.

```
class parent
  {
     display ( ) {
      console.log("parent is invoked");
    }
  }

class child extends parent
  {
    display( ) {
      console.log("child is invoked");
    }
  }

var a= new A() ;
var b = new B();

a.display( ) ;
b.display( ) ;

OUTPUT:  parent is invoked
         child is invoked
```

In the above example display( ) function is used multiple times for two different classes. A is parent class, and B is the child class to A.

No.	Method Overloading	Method Overriding
1)	Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method already provided by its super class.
2)	Method overloading is performed within the class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
3)	In the case of method overloading, the parameters must be different.	In the case of method overriding, the parameter must be the same.
4)	Method overloading is an example of compile-time polymorphism.	Method overriding is an example of run time polymorphism.
5)	Method overloading can't be performed by changing the return type of the method only. The return type can be the same or different in method overloading. But you must have to change the parameter.	The return type must be the same or covariant in method overriding.

Exception Handling

An exception is a strange code that breaks the normal flow of the code. Such exceptions require specialized programming constructs for its execution.

Exception handling is a process or method used to handle the abnormal statements in the code and execute them. It also enables handling the flow control of the code/program. For handling the code, various handlers are used that process the exception and execute the code.

```
Example:  5/0 =  `infinity always, and it is an exception.
Thus, with the help of exception handling, it can be executed
and handled.``
```

Method

A **throw statement** is used to raise an exception. It means when an abnormal condition occurs, an exception is thrown using throw.
The thrown exception is handled by wrapping the code into the **try...catch block**. Statements in which there is a doubt that an error can occur, those statements are kept in the try block, and when the error occurs, the statements that need to be run are kept into the catch block.

Types of Errors

1. **Compile Time Error:** When a user makes a mistake in the pre-defined syntax of a programming language, a syntax error may appear.
2. **Runtime Error:** When an error occurs during the execution of the program, such an error is known as a Runtime error. **The codes which create runtime errors are known as Exceptions.** Thus, exception handlers are used for handling runtime errors.
3. **Logical Error** An error occurs when there is any logical mistake in the program that may not produce the desired output and may terminate abnormally. Example : Division by zero.

Exception Handling Statements

- try…catch statements
- throw statement
- try…catch…finally statements

try catch

The **try statement** is used to define a block of code to be tested for errors while it is being executed.

The **catch statement** is used to define a block of code to be executed if an error occurs in the try block.

```
Syntax :try {
      //  Block of code to try
}
```

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

```
Example :var a = [1,2,3];

try{
    console.log(b[4]);
}catch(e){
    console.log("Error");
}
Output : Error
Because b array was not defined anywhere
```

throw Statement

Throw statements are used for throwing user-defined errors. Users can **define and throw their own custom errors.** When a throw statement is executed, the statements present after it will not execute. The control will directly pass to the catch block.

It does the same work of console.log( ), but the output shown on the console is like an error i.e. in red color

```
Syntax : throw Exception ;
```

```
`throw “Custom Error”`
Example :var a = [1,2,3];
try{
    console.log(b[4]);
}catch(e){
    throw "Error in code";
}
Output : `Uncaught Error in code`
```

try catch finally

Finally is an optional block of statements which is executed after the execution of try and catch statements. Finally, the ..block does not hold for the exception to be thrown. Any exception is thrown or not, finally block code, if present, will execute. It does not care for the output too.

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

- It is like the **default** in switch case

```
Example :var password = 1234 ;
    try{
      if(password == 1234)
        console.log("Correct password");
    }catch(err){
        throw "Incorrect password";
   }finally{
         console.log("Welcome to Coding Ninjas");
   }
Output :  Correct password
      Welcome to Coding Ninjas
```

Tip

The **err** parameter passed into the catch statement is called the error Object.

Error Object

JavaScript has a built-in error object that provides error information when an error occurs.
The error object provides two useful properties: name and message.

Error Name	Description
EvalError	An error has occurred in the eval( ) function
RangeError	A number "out of range" has occurred
ReferenceError	An illegal reference has occurred
SyntaxError	A syntax error has occurred
TypeError	A type error has occurred
URIError	An error in encodeURI() has occurred

Strict mode

The **strict mode** was introduced in ECMAScript 5. It is a way to **add a strict checking in JavaScript** that would make fewer mistakes.
JavaScript **allows strict mode code and non-strict mode code to coexist**. So you can add your new JavaScript code in strict mode in old JavaScript files.
Strict mode introduces several restrictions to the JavaScript code, like eliminating some silent errors by throwing errors.
You can introduce a strict mode in your JavaScript code by writing this simple statement.

```
Syntax :  'use strict' ;
                    OR
          "use strict" ;
```

You can apply strict mode to an entire script or individual function -

- Write this at the top of the whole script to **apply strict mode globally.**
- Or write it inside functions to **apply it to a particular function only.**

Using strict mode globally

Example :

```
x = 10 ;
console.log(x) ; // Outputs 10
```

```
"use strict"
x = 10 ;
console.log(x) ; // ReferenceError: x is not defined
```

- Creating variables without a literal is **not allowed in strict mode**

Using strict mode within a function

Example :

```
function abc(a, a) {
console.log(a + a);
}
abc(10, 20) ;  // Output 40
```

```
function abc(a, a) {
'use strict' ;
console.log(a + a);
}
abc(10, 20);  // Duplicate parameter name not allowed in this
context
```

debugger keyword

In debugging, generally, we set breakpoints to examine each line of code step by step. There is no requirement to perform this task manually in JavaScript. JavaScript provides the **debugger keyword** to set the breakpoint through the code itself. The debugger stops the execution of the program at the position it is

applied. Now, we can start the flow of execution manually. If an exception occurs, the execution will stop again on that particular line.

```
Example :var a = 10 ;
         var b = 2 ;
         var c = a/b ;
         debugger ;
         console.log( c ) ;
```

The flow of control of the above code will stop at the debugger keyword and take us to the debugging console, from where we can execute the code line by line manually.

JSON

The JavaScript JSON is an acronym for **JavaScript Object Notation.** It provides a format for storing and transporting data. It is a lightweight, human-readable collection of data that can be accessed logically.

- It generates and stores the data from user input.
- It can transport the data from the server to the client, client to server, and server to server.
- It can also build and verifying the data.
- JSON is often used when data is sent from a server to a web page.

JSON Object Example

The **{ (curly brace)** represents the JSON object.

```
{
 "student": {
         "name":  "Sam",
         "fees": 56000,
         "institution": “Coding Ninjas”
     }
}
```

JSON Array Example

The [(square bracket)] represents the JSON array. A JSON array can have values and objects.

```
[
        {"name":"Sam", "email":"Sam@gmail.com"},
        {"name":"Yash", "email":"Yash@gmail.com"}
]
```

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

Purpose of using JSON

The JSON format is syntactically similar to the JavaScript objects. Because of this, a JavaScript program can **easily convert JSON data into JavaScript objects.**
Since the format is text only, JSON data can easily be sent between computers and used by any programming language.

Methods	Description
JSON.parse( )	This method takes a JSON string and converts it into a JavaScript object.
JSON.stringify( )	This method converts a JavaScript Object to a JSON string.

JSON.parse( )

JSON is used to **exchange data to/from a web server.**
When receiving data from a web server, the data is always a string.

Data is parsed with JSON.parse( ), and the data becomes a JavaScript object.

```
Syntax :   JSON.parse( text )  ;
```

```
Example :var text = ‘ { "name":  "Sam", "fees": 56000,
"institution": “Coding Ninjas” } ’
        var json = JSON.parse( text ) ;

        json.name ; // Returns Sam
        json.fees ;   //Returns 56000
```

JSON.stringify( )

JavaScript JSON.stringify( ) method converts a JavaScript Object to a JSON string.

```
Syntax :   Json.stringify( value ) ;
```

```
Example :     var obj ={
 "name":  "Sam",
 "fees": 56000,
 "institution": "Coding Ninjas"
  }

  var json = JSON.stringify( obj ) ;
  console.log(json) ;

Output : {"name":"Sam","fees":56000,"institution":"Coding
Ninjas"}
```

- json is now a string and ready to be sent to a server