Basics JS APIs

In JavaScript, APIs refer to interfaces provided by browsers or third-party libraries that allow developers to access functionalities and perform operations within the environment where JavaScript is executed

String Manipulation

split and **slice** will be used mainly

Auxiliary methods in string

1. str.length

```
// take something as input and give output

const str = "Nishant thapa"
//getlength
function getLength(str) {
    console.log("Original String:", str);
    console.log("Length:", str.length);
    // current length of string
  }
  getLength(str);
```

2. str.indexOf and lastIndexOf

```
// indexOf
function findIndexOf(str, target) {
    console.log("Original String:", str);
    console.log("Index:", str.indexOf(target));
  }
  findIndexOf("Nishant Thapa", "Thapa");
//  8th index
//  what is the index of second word
//  It gives  first occurrence of the target

// to find the last index of the target
```

```
// lastIndexOf
function findLastIndexOf(str, target) {
    console.log("Original String:", str);
    console.log("Index:", str.lastIndexOf(target));
  }
  findLastIndexOf("Hello World World", "World");
```

Output:

Original String: Nishant Thapa

Index: 8

Original String: Hello World World

Index: 12

3. str.slice

```
// slice
function getSlice(str, start, end) {
    console.log("Original String:", str);
    console.log("After slice:", str.slice(start, end));
  }
  getSlice("Hello World", 0, 5);
//   everything between 0th and 5th index(not included) is printed
let fname="Nishant Thapa".slice(0, 8)
console.log(fname)
```

We can write this code on our own

```
function cutIt(str,startIndex,endIndex){
    let newStr = "";
    for(let i = 0; i < str.length; i++){
        if(i >= startIndex && i <= endIndex){
            newStr += str[i];
        }
    }
    console.log(newStr);
}
```

```
cutIt("Hello World", 0, 5);
```

4. substr

   There is only a minute difference between slice and substr

```
function getSubstr(str, start, end) {
    console.log("Original String:", str);
    console.log("After substring:", str.substr(start, end));
  }
  getSubstr("Hello World", 0, 5);
// starting from 0 th index and i need 5 characters
getSubstr("Hello World", 4, 5);
```

5. substring

   Similar to slice

```
// substring
function getSubstring(str, start, end) {
  console.log("Original String:", str);
  console.log("After substring:", str.substring(start, end));
}
getSubstring("Hello World", 0, 5);
```

6. str.replace(where str is string variable/constant)

```
// replace
function replaceString(str, target, replacement) {
    console.log("Original String:", str);
    console.log("After replace:", str.replace(target, replacement));
  }
  replaceString("Hello World", "World", "JavaScript");

  let str = "Hello World";
  str.replace("World", "JavaScript");
  console.log(str);
```

```
//   The reason you're seeing "Hello World" instead of "Hello JavaScript"
in the output is because replace() method in JavaScript doesn't modify the
original string. Instead, it returns a new string with the replacement
made. Strings in JavaScript are immutable, which means that methods like
replace() do not change the original string; they create and return a new
string with the modifications.

// To fix this, you can assign the returned string to a new variable, or
to the same variable you used to store the original string. Here's an
example:

str = str.replace("World", "JavaScript");
console.log(str);
```

7. split

```
const value = "Good evening everyone, my name is Nishant Thapa"
const words = value.split(" ")
const words1 = value.split(",")
console.log(words)
console.log(words1)
```

Output

In the form of an array

[
  'Good',   'evening',
  'everyone,', 'my',
  'name',   'is',
  'Nishant',   'Thapa'
]

['Good evening everyone', ' my name is Nishant Thapa']

8. trim,toUpperCase,toLowerCase

```
// trim
const value = "    Nishant     Thapa  ";
```

```
console.log(value.trim());

// toLowerCase
console.log(value.toLowerCase());

// toUpperCase
console.log(value.toUpperCase());
```

Number

parseInt,parseFloat

```
  // Example Usage for parseInt
//   parseInt is a global function we can call directly
  console.log(parseInt("42"));//42
  console.log(parseInt("42px"));//42
  console.log(parseInt("3.14"));//3
//   it(String etc) should start with number

    // Example Usage for parseFloat
//   parseFloat is global function we can call it directly
    console.log(parseFloat("3.14"));//3.14
    console.log(parseFloat("42"));//42
    console.log(parseFloat("42px"));//42
```

Array

push, pop, shift, unshift

```
const initialArray = [1,2,3]
initialArray.push(4)
console.log("After push:",initialArray)
initialArray.pop()
console.log("After pop:",initialArray)
//What if I want to pop from the front
initialArray.shift()
console.log("After shift:",initialArray)
//What if I want to add to the front
```

```
initialArray.unshift(7)
console.log("After unshift:", initialArray);
```

Output:

[1, 2, 3, 4]

[1, 2, 3]

[2, 3]

[7, 2, 3]

concat

```
const initialArray = [1,2,3]
const secondArray = [4,5,6]
console.log(initialArray.concat(secondArray))
// other way
for(let i=0; i<secondArray.length; i++){
    initialArray.push(secondArray[i])
}
console.log(initialArray)
```

Output:

[1,2,3,4,5,6]

[1,2,3,4,5,6]

More example:

```
// concat()
function concatExample(arr1, arr2) {
  console.log("Original Arrays:", arr1, arr2);

  let arr3 = arr1.concat(arr2);
  console.log("After concat:", arr3);
}
concatExample([1, 2, 3], [7, 9, 23]);
```

forEach:

```
const initialArray = [1,2,3]
console.log("Original Array simple way parsing:")
for(let i=0;i<initialArray.length;i++){
    console.log(initialArray[i])
}

console.log("Original Array forEach way parsing:")
function logThing(str){
    console.log(str)
}
initialArray.forEach(logThing)
// call the function inside the forEach each time for all the element
inside the array
// passed  a function inside the forEach (callbacks)

// OR
console.log("Another way")
logThing(1)
logThing(2)
logThing(3)

// OR
console.log("Another way")
console.log("Items and Indexes")
initialArray.forEach(function(item,index){
    console.log(item,index)
})
```

(**Callbacks:**

Example:

```
function log1(){
    console.log("Hello World 1");
}
function log2(){
```

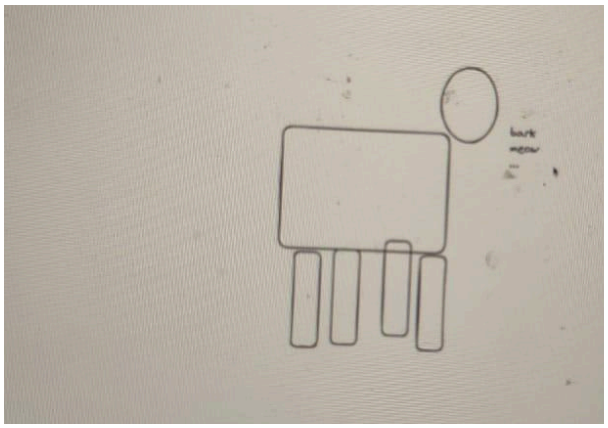
```
    console.log("Hello World 2");
}
function logWhatsPresent(callback){
    callback();
    //We can name it anything
}
logWhatsPresent(log1);
logWhatsPresent(log2);
```

)

//sort, map, filter will be done late

Class

example: Defining animal



```
const dog = {
    name: 'duggu',
    legCount: 4,
    speaks: 'bark'
}
const cat = {
    name: 'puggu',
    legCount: 4,
    speaks: 'meow'
```

```
}
console.log("animal ", dog['name']+" "+dog['speaks']);
console.log("animal ", cat['name']+" "+cat['speaks']);
```

We can do in this way for different animals but if we look closer we can see that there is repetitive logic.

One way can be like:

```
function printStr(animal){
    console.log("animal " + animal["name"] +" "+animal["speaks"]);
}
printStr(dog);
printStr(cat);
```

What if there are more functions

```
class Animal {
    constructor(name,legCount,speaks) {
    this.name = name
    this.legCount = legCount
    this.speaks = speaks
    }
    speak() {
       console.log(this.name + " says " + this.speaks)
    }
}
// class is like a blueprint, it has attributes and function
// object is like house, real thing
let dog = new Animal("Dog",4,"Bark Bark")
let cat = new Animal("Cat",4,"Meow Meow")
// classes are good for the common functionality
dog.speak()
cat.speak()
// DONT DO THIS
let doggg = {
    name: "Dog",
```

```
    legCount: 4,
    speaks: "Bark"
}
```

Output

Dog says Bark Bark

Cat says Meow Meow

Other examples

```
class Aliens {
    constructor(name,legCount,speaks) {
    this.name = name
    this.legCount = legCount
    this.speaks = speaks
    }
    static myType() {
        console.log("I am an alien")
    }
    // can be directly called without creating an object
    speak() {
        console.log(this.name + " says " + this.speaks)
    }
}
Aliens.myType()
```

Date

Date is not primitive, it is like class, made for our usage

```
// Date functions can be used as counter or to see how much time it takes
to run certain function.

const currentDate = new Date();
```

```
console.log("Current Date:", currentDate);
// Current Date: 2023-12-15T19:18:48.760Z
console.log("Date:", currentDate.getDate());
// Date: 16
console.log("Month:", currentDate.getMonth() + 1); // Months are
zero-indexed, so adding 1
// Month: 12
console.log("Year:", currentDate.getYear());
// Year: 123, it basically returns the year since 1900
console.log("Years:", currentDate.getFullYear());
// Years: 2023
console.log("Hours:", currentDate.getHours());
// Hours: 0
console.log("Minutes:", currentDate.getMinutes());
// Minutes: 51
console.log("Seconds:", currentDate.getSeconds());
// Seconds: 9
```

Date has more usages like as a **counter**, or to **calculate how much time it takes to run a certain function**

Application 1: To calculate the time to run a certain function we use

```
const currentDate = new Date();
console.log("Time in milliseconds since 1970:", currentDate.getTime());
// epoch timestamp
// 1702668310002
```

.getTime()

Here we are calculating the time required to run the calculateSum() function for i=0 to i =1000000000

```
function calculateSum(){
    let a=0;
    for(let i=0;i<1000000000;i++){
        a=a+i;
```

```
    }
    return a;
}
const beforeDate = new Date();
const beforeTimeInMs = beforeDate.getTime();
calculateSum();
const afterDate = new Date();
const afterTimeInMs = afterDate.getTime();
console.log("Time taken to execute calculateSum in milliseconds:",
afterTimeInMs - beforeTimeInMs);
// Time taken to execute calculateSum in milliseconds: 1658
// for i =1000000000
```

Application 2: Now to show timer or counter

```
function currentTimePrint() {
    console.log(new Date().getTime());
}
setInterval(currentTimePrint, 1000);
//This displays the epoch time
```

JSON(Javascript Object Notation)

```
const users = {
    name: 'John',
    age: 25,
    gender: 'male'
}
console.log(users['name']);
//Now what if we want to send this data somewhere, How will we send it
//  or in which format we will send it?
//We will send it in the form of a string
const user = '{"name":"Thapa","age":22,"gender":"male"}'
//is no longer is object but a string
//Now we can't access it using
// user['name'] or user.name
```

```
// We have to convert many times Javascript object to string and
string to object
//So this is where JSON class comes in
// JSON.parse() and JSON.stringify()
```

parse() and stringify()

```
const users = '{"name":"Thapa","age":22,"gender":"male"}'

const user = JSON.parse(users)
console.log(user['gender'])

const user1 = {
    name:'Nishu',
    gender:'male',
    age:30
}
const finalString = JSON.stringify(user1)
//We send out a string, because it may be possible that the place where we
sending data may not understand the object
console.log(finalString)
// console.log(finalString['name'])
```

More examples:

```
function jsonMethods(jsonString) {
  console.log("Original JSON String:", jsonString);

  // Parsing JSON string to JavaScript object
  let parsedObject = JSON.parse(jsonString);
  console.log("After JSON.parse():", parsedObject);

  // Stringifying JavaScript object to JSON string
  let jsonStringified = JSON.stringify(parsedObject);
  console.log("After JSON.stringify():", jsonStringified);
}
```

```
// Example Usage for JSON Methods
const sampleJSONString =
  '{"key": "value", "number": 42, "nested": {"nestedKey":
"nestedValue"}}';

jsonMethods(sampleJSONString);
```

Output:

Original JSON String: {"key": "value", "number": 42, "nested": {"nestedKey": "nestedValue"}}

After JSON.parse(): { key: 'value', number: 42, nested: { nestedKey: 'nestedValue' } }

After JSON.stringify(): {"key":"value","number":42,"nested":{"nestedKey":"nestedValue"}}

Maths

Mainly this is used

Math.random()

The most of the Math functions are

```
function mathMethods(value) {
  console.log("Original Value:", value);

  let rounded = Math.round(value);
  console.log("After round():", rounded);

  let ceiling = Math.ceil(value);
  console.log("After ceil():", ceiling);

  let flooring = Math.floor(value);
  console.log("After floor():", flooring);

  let randomValue = Math.random();
  console.log("After random():", randomValue);

  let maxValue = Math.max(5, 10, 15);
```

```
  console.log("After max():", maxValue);

  let minValue = Math.min(5, 10, 15);
  console.log("After min():", minValue);

  let powerOfTwo = Math.pow(value, 2);
  console.log("After pow():", powerOfTwo);

  let squareRoot = Math.sqrt(value);
  console.log("After sqrt():", squareRoot);
}

// Example Usage for Math Methods
mathMethods(4.56);
mathMethods(9);
mathMethods(25);
```

Object

Object Method Explanation:

```
// Object Methods Explanation
function objectMethods(obj) {
  console.log("Original Object:", obj);

  let keys = Object.keys(obj);
  console.log("After Object.keys():", keys);
// Static method(.keys()) on class, an array containing keys of object

  let values = Object.values(obj);
  console.log("After Object.values():", values);

  let entries = Object.entries(obj);
  console.log("After Object.entries():", entries);
//Every entry is in array format stored inside array
// Basically converting object into an array, no data is lost (All data is
present)
```

```
  let hasProp = obj.hasOwnProperty("property");
  // does obj object has "property"
  console.log("After hasOwnProperty():", hasProp);

  let newObj = Object.assign({}, obj, { newProperty: "newValue" });
  // assign can be used to merge two objects
  console.log("After Object.assign():", newObj);


}

// Example Usage for Object Methods
const sampleObject = {
  key1: "value1",
  key2: "value2",
  key3: "value3",
};

objectMethods(sampleObject);
```

Output:

Original Object: { key1: 'value1', key2: 'value2', key3: 'value3' }
After Object.keys(): ['key1', 'key2', 'key3']
After Object.values(): ['value1', 'value2', 'value3']
After Object.entries(): [['key1', 'value1'], ['key2', 'value2'], ['key3', 'value3']]
After hasOwnProperty(): false
After Object.assign(): {
  key1: 'value1',
  key2: 'value2',
  key3: 'value3',
  newProperty: 'newValue'
}