JAVASCRIPT

Javascript is a synchronous , blocking, single threaded language.That just means that only one operation can be in progress at a time, That's not the eniter story , through!

To Wrap up, Javascript runs on single threaded. It runs code line by line and must finish executing a piece of code before moving onto the next.

Synchronous JavaScript:

As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed. Let us understand this with the help of an example.

Statement vs Expression

- Statement : JavaScript statements are composed of: Values, Operators, Expressions, Keywords, and Comments. This statement tells the browser to write "Hello Dolly." inside an HTML element with id="demo":
- Expression : JavaScript's expression is a valid set of literals, variables, operators, and expressions that evaluate a single value that is an expression. This single value can be a number, a string, or a logical value depending on the expression.

An expression is any valid unit of code that resolves to a value. A statement is a unit of code that performs an action.

```
# Statements
let x = 0;
function add(a, b) { return a + b; }
 if (true) { console.log('Hi'); }
const isTrue = 12 > 10 ? 'right':'false';

# Expressions
x;          # Resolves to 0
3 + x;      # Resolves to 3 with x variable that computes to value
add(1, 2);  # Resolves to 3
isTrue     # Resovlves to right
```

JavaScript Values

The JavaScript syntax defines two types of values:

1. Fixed values (Number , String)
2. Variable values ()

Fixed values that are used in program without variable keywords are called Literals. Variable values that are used in program with variable keywords are called Variables.

```
Fixed Value
document.getElementById("demo").innerHTML = 10.50; # directly value assigned without carrying variable

Variables
var x = 10.50;
document.getElementById("demo").innerHTML = 10.50 + x; # with carrying variable
```

Syntex

*. Identifiers are JavaScript names. A JavaScript name must begin with:

A letter (A-Z or a-z) A dollar sign ($) Or an underscore (_)

*. All JavaScript identifiers are case sensitive. *. Hyphens are not allowed in JavaScript. They are reserved for subtractions.

JavaScript and Camel Case: Underscore: first_name, last_name, master_card, inter_city.

Upper Camel Case (Pascal Case): FirstName, LastName, MasterCard, InterCity.

Lower Camel Case: JavaScript programmers tend to use camel case that starts with a lowercase letter: firstName, lastName, masterCard, interCity.

Variables

4 Ways to Declare a JavaScript Variable: Using var Using let Using const Using nothing

The general rules for constructing names for variables (unique identifiers) are:

Names can contain letters, digits, underscores, and dollar signs. Names must begin with a letter. Names can also begin with $ and _ (but we will not use it in this tutorial). Names are case sensitive (y and Y are different variables). Reserved words (like JavaScript keywords) cannot be used as names.

Declaring a JavaScript Variable

```
let myname;
let job = null;
let location = undefined;
let myage;
myname = "dulal";
or
let myname = "dulal";
console.log(myage) # undefined becuase of not assigning value.
console.log(location) # value is undefined becuase of assigning undefined value.;

console.log(myage) # value is null;



let myname = 'mkarim', myage = 10, myhome = 'Chittagong' # multi variables called in one statement.

A declaration can span multiple lines:
let person = "John Doe",
carName = "Volvo",
price = 200;
```

Re-Declaring JavaScript Variables

```
let carName = "Volvo";
 let carName = 'green line';  # only var can be used while redeclaring variable, but not using 'var' is better.Let re-declaration is

document.getElementById("demo").innerHTML = carName; # no result is found
```

Scope / Difference among var , let or const

Scope determines the accessibility (visibility) of variables. JavaScript has 3 types of scope:

- Block scope
- Function scope
- Global scope

var: The scope of a var variable is functional scope.

let: The scope of a let variable is block scope.

const: The scope of a const variable is block scope.

```
var num1 = 10;

{
  const num2 = 10;
  let num3 = 25;
  var num1 = 23;
}

let num3 = 45;
num3 or num2 # undefined
console.log(num1) # 23;
console.log(num3) # 45 and it will not return error;
```

var : It can be updated and re-declared into the scope. let : It can be updated but cannot be re-declared into the scope. const : It cannot be updated or re-declared into the scope.

```
var num1 = 10;
var num1 = 12;
  const num2 = 10;
  num2 = 15; # error
  let num3 = 25;
    num3 = 20; # let can't be used again
```

var : It can be declared without initialization.It can be accessed without initialization as its default value is “undefined”. let : It can be declared without initialization.It cannot be accessed without initialization otherwise it will give ‘referenceError’. const : It cannot be declared without initialization.It cannot be accessed without initialization, as it cannot be declared without initialization.

```
 console.log(num1) # undefined
 var num1 = 12; # declared

console.log(num1) # ReferenceError
let num2 = 12; #  declared

console.log(num3) # ReferenceError
const num3 = 12; #  declared
const name; # can\'t used in const but used in var or let
```

Important Differences

var and let is used to redeclare but const is not for redeclare

```
var name = "mahmodul Karim";
name = "dulal sheikh";
let age = 23;
age = 25;
const location = 'Dhaka';
location = "Chittagong" # it will execute error for redeclaring
```

var is declared inside the function. If the user tries to access it outside the function, it will display the error.

```
var b = 12;
        function f() {

                # It can be accessible any
                # where within this function
                var a = 10;
    b = 25;
                console.log(a,b)
        }
        f();

        # A cannot be accessible
        # outside of function
        console.log(a); # referenceError
        console.log(b); # 25 # b is global scope
```

Using Value in Scope

```
# Initialize a global variable
var species = "human";

function transform() {
  # Initialize a local, function-scoped variable
  var species = "werewolf";
  console.log(species);
}

# Log the global and local variable
console.log(species);
transform();
console.log(species);

Output
human
werewolf
human
```

To illustrate the difference between function- and block-scoped variables, we will assign a new variable in an if block using let.

```
var fullMoon = true;

# Initialize a global variable
let species = "human";

if (fullMoon) {
  # Initialize a block-scoped variable
  let species = "werewolf";
  console.log(`It is a full moon. Lupin is currently a ${species}.`);
}

console.log(`It is not a full moon. Lupin is currently a ${species}.`);
Output
It is a full moon. Lupin is currently a werewolf.
It is not a full moon. Lupin is currently a human.
```

In this example, the species variable has one value globally (human), and another value locally (werewolf). If we were to use var, however, there would be a different result.

```
# Use var to initialize a variable
var species = "human";

if (fullMoon) {
  # Attempt to create a new variable in a block
  var species = "werewolf";
  console.log(`It is a full moon. Lupin is currently a ${species}.`);
}

console.log(`It is not a full moon. Lupin is currently a ${species}.`);
Output
It is a full moon. Lupin is currently a werewolf.
It is not a full moon. Lupin is currently a werewolf.
```

JSON

JSON: JSON stands Javascript object notation.It is a text format for storing data and transporting.Data is interchanged between computers by it.It is self-describing techniqe.Data can be send to pure text on server from javascript object and send from javascript object in pure text.

WHY JSON USE

Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.

JavaScript has a built in function for converting JSON strings into JavaScript objects: JSON.parse()

JavaScript also has a built in function for converting an object into a JSON string: JSON.stringify()

JSON Data Types

String: A string is always written in double-quotes. It may consist of numbers, alphanumeric and special characters.Example:"student", "name", "1234", "Ver_1" Number: Number Represents The numeric Characters Like 23,564,3421. Boolean: It can be either True or False.true , false. Null: It is an empty value. like # this is gap for empty value.

JSON Types: It is 2 types.

1. Json Object
2. Json Arrays.

JSON Object

In JSON, objects refer to dictionaries, which are enclosed in curly brackets, i.e., . These objects are written in key/value pairs, where the key has to be a string and values have to be a valid JSON data type such as string, number, object, Boolean or null. Here the key and values are separated by a colon, and a comma separates each key/value pair.

```
{"name" : "Jack", "employeeid" : 001, "present" : false}
```

JSON Arrays

In JSON, arrays can be understood as a list of objects, which are mainly enclosed in square brackets [ ]. An array value can be a string, number, object, array, boolean or null.

```
 "pizza": [{
"PizzaName" : "Country Feast",
"Base" : "Cheese burst",
"Toppings" : ["Jalepenos", "Black Olives", "Extra cheese", "Sausages", "Cherry tomatoes"],
"Spicy" : "yes",
"Veg" : "yes"
},

{
"PizzaName" : "Veggie Paradise",
"Base" : "Thin crust",
"Toppings" : ["Jalepenos", "Black Olives", "Grilled Mushrooms", "Onions", "Cherry tomatoes"],
"Spicy" : "yes",
"Veg" : "yes"
}
]
```

Notes: In the above example, the object "Pizza" is an array. It contains 2 objects, i.e., PizzaName, Base, Toppings, Spicy, and Veg.

JSON Vs XML JSON stands for JavaScript Object Notation, whereas XML stands for Extensive Markup Language. Nowadays, JSON and XML are widely used as data interchange formats, and both have been acquired by applications as a technique to store structured data.

Difference between JSON and XML

JSON is easy to learn. XML is quite more complex to learn than JSON.

JSON is simple to read and write. XML is more complex to read and write than JSON.

JSON is data-oriented. XML is document-oriented.

JSON is less secure in comparison to XML. XML is highly secured.

JSON doesn't provide display capabilities. XML provides the display capability because it is a markup language.

JSON supports the array. XML doesn't support the array Example :

```
JSON:-
[
{
"name" : "Peter",
"employed id" : "E231",
"present" : true,
"numberofdayspresent" : 29
},
{
"name" : "Jhon",
"employed id" : "E331",
"present" : true,
"numberofdayspresent" : 27
}
]
XML:- <name>
<name>Peter</name>
</name>
```

JavaScript Objects

Because JSON syntax is derived from JavaScript object notation, very little extra software is needed to work with JSON within JavaScript.

With JavaScript you can create an object and assign data to it, like this:

```
person = {name:"John", age:31, city:"New York"};
```

You can access a JavaScript object like this:

returns John

```
person.name or person["name"];
```

Data can be modify by

```
person.name = "Gilbert";
or
person["name"] = "Gilbert";
```

JSON is Like XML Because

Both JSON and XML are "self describing" (human readable) Both JSON and XML are hierarchical (values within values) Both JSON and XML can be parsed and used by lots of programming languages Both JSON and XML can be fetched with an XMLHttpRequest

For AJAX applications, JSON is faster and easier than XML:

- Using XML
    - Fetch an XML document
    - Use the XML DOM to loop through the document
    - Extract values and store in variables
- Using JSON
    - Fetch a JSON string
    - JSON.Parse the JSON string

Valid Data Types

- In JSON, values must be one of the following data types:
    - a string
    - a number

    - an object (JSON object)
    - an array
    - a boolean
    - null

- JSON Values cannot be one of the following data types

    - a function
    - a date
    - undefined

```
Json String: {"name":"John"}
Json Numbers : {"age":23}
Json Objects : {
"employee":{"name":"John", "age":30, "city":"New York"}
}
JSON Arrays: {
"employees":["John", "Anna", "Peter"]
}
or
myJSON = '["Ford", "BMW", "Fiat"]';
myArray = JSON.Parse(myJSON);
or
JSON Booleans: {"sale":true}
JSON null:  {"middlename":null}
```

- Array using in JSON

```
  var myobj =  '[
       {"name":"mahmodul karim"},
       {},
       {}
   ]';
   myobj[0].name
```

JSON-PHP

---

json data can be transported from web server like(php or laravel) to web page/client side/javascript;

In PHP, Json data array or object will be transported into client side by json_encode() when requested is done from client side by user; then in Client side , The Response data must be converted into object using ajax call -- $.ajax() in jquery, fetch(),axios() or XMLHttpRequest();

Example

```php
<?php
Class Address {
    public $name;
    public $age;
    public $city;
}
$myObj = new Address();
$myObj->name = "John";
$myObj->age = 30;
$myObj->city = "New York";

$myJSON = json_encode($myObj);

echo $myJSON;
?>

OR

<?php
$myArr = array("John", "Mary", "Peter", "Sally");

$myJSON = json_encode($myArr);

echo $myJSON;
?>
```

JSON-PHP WITH DATABASE

AS PHP server side language ,it can be access data from database. On the client, make a JSON object that describes the numbers of rows you want to return.

Before you send the request to the server, convert the JSON object into a string and send it as a parameter to the url of the PHP page:

```php
<?php
header("Content-Type: application/json; charset=UTF-8");
$obj = json_decode($_GET["x"], false);

$conn = new mysqli("myServer", "myUser", "myPassword", "Northwind");
$stmt = $conn->prepare("SELECT name FROM customers LIMIT ?");
$stmt->bind_param("s", $obj->limit);
$stmt->execute();
$result = $stmt->get_result();
$outp = $result->fetch_all(MYSQLI_ASSOC);

echo json_encode($outp);
?>
```

CLIENT SIDE:- using $.ajax() in jQuery

```
$.ajax({
    url:"something-test.php", # server api
    type:"post" , # http request method
    data:{all_data:data},  # client-side user data
    dataType:"json" # Data transporting in json format
    success:(response)=>{
        # data converting and using here in web page/browser
    }
})
```

Synchronous and Asynchronous in JavaScript

Javascript is a synchronous , blocking, single threaded language.That just means that only one operation can be in progress at a time, That's not the eniter story , through!

To Wrap up, Javascript runs on single threaded. It runs code line by line and must finish executing a piece of code before moving onto the next.

Synchronous JavaScript:

As the name suggests synchronous means to be in a sequence, i.e. every statement of the code gets executed one by one. So, basically a statement has to wait for the earlier statement to get executed. Let us understand this with the help of an example.

```
<script>
        document.write("Hi"); # First
        document.write("<br>");

        document.write("Mayukh") ;# Second
        document.write("<br>");

        document.write("How are you"); # Third
</script>

Output
Hi
Mayukh
How are you
```

In the above code snippet, the first line of the code Hi will be logged first then the second line Mayukh will be logged and then after its completion, the third line would be logged How are you.

Asynchronous JavaScript:

Asynchronous code allows the program to be executed immediately where the synchronous code will block further execution of the remaining code until it finishes the current one. This may not look like a big problem but when you see it in a bigger picture you realize that it may lead to delaying the User Interface.

Let us see the example how Asynchronous JavaScript runs.

```
document.write("Hi");
    document.write("<br>");

    setTimeout(() => {
        document.write("Let us see what happens");
    }, 2000);

  # Synchronous way

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

# end of synchronous

# for loop , foreach loop, while, if condition,

    document.write("<br>");
    document.write("End");
    document.write("<br>");

    Output Here
    Hi
    End

The number is 0
The number is 1
The number is 2
The number is 3
The number is 4

    Let us see what happens # this execution doest not block in stack for another log to print

    So, what the code does is first it logs in Hi then rather than executing the setTimeout function it logs in End and then it runs
```

CLIENT-SIDE PROGRAMMING VS SERVER-SIDE PROGRAMMING

SAME AS

AJAX

AJAX = Asynchronous JavaScript and XML.

In short; AJAX is about loading data in the background and display it on the webpage, without reloading the whole page.

Examples of applications using AJAX: Gmail, Google Maps, Youtube, and Facebook tabs.

AJAX allows you to send only important information to the server not the entire page. So only valuable data from the client side is routed to the server side. It makes your application interactive and faster.

```
JSON --- {"name":"mahmodul karim"}
XML --- <name>mahmodul</name>
```

jQuery - AJAX

What About jQuery and AJAX? jQuery provides several methods for AJAX functionality.

With the jQuery AJAX methods, you can request text, HTML, XML, or JSON from a remote server using both HTTP Get and HTTP Post - And you can load the external data directly into the selected HTML elements of your web page!

```
$.ajax({
    url:"something-test.php", # server api
    type:"post" , # http request method
    data:{all_data:data},  # client-side user data
    dataType:"json" # Data transporting in json format
    success:(response)=>{
        # data converting and using here in web page/browser
    }
})

more Methods of jQuery ajax

$.ajax()  #     Performs an async AJAX request
$.get()   #    Loads data from a server using an AJAX HTTP GET request
$.getJSON()      # Loads JSON-encoded data from a server using a HTTP GET request
$.post()         # Loads data from a server using an AJAX HTTP POST request
load()      #  Loads data from a server and puts the returned data into the selected element
serialize()     #  Encodes a set of form elements as a string for submission # {email:"mkarimcu@gmail.com"}
serializeArray() #      Encodes a set of form elements as an array of names and values # ["email"=>"m.karimcu@gmail.com"]
```

AJAX KeyWords

1. Request
2. Response
3. Cross-Origin Response Sharing(CORS)
4. REST API
5. JSON/XML
6. HTTP Client, HTTP Methods
7. FECTH, AXIOS, XMLHttpRequest

HTTP Communication

CORS

CORS (Cross-Origin Resource Sharing) is a mechanism by which data or any other resource of a site could be shared intentionally to a third party website when there is a need. Generally, access to resources that are residing in a third party site is restricted by the browser clients for security purposes.

In response, the server returns a Access-Control-Allow-Origin header with Access-Control-Allow-Origin: *, which means that the resource can be accessed by any origin

```
Access-Control-Allow-Origin: *
```

but no domain other than https:#foo.example can access the resource in a cross-origin manner), they would send:

```
Access-Control-Allow-Origin: https:#foo.example
```

In Details Example , Going to google domain by fetch('http:#facebook.com') to collect the requested resources must be blocked by 'CORS' like

```
Request
---
fetch('https:#facebook.com')
   .then(res=>res.text()).then(data=>console.log(data));

The Response
---

Access to fetch at 'https:#facebook.com/' from origin 'https:#www.google.com' has been blocked by CORS policy: No 'Access-Control-Al
VM813:1
 GET https:#facebook.com/ net::ERR_FAILED 301
```

In above mentioned, Access-Control-Allow-Origin is not allowed. To be allowed , We must use to allow technology in backend.

Cors for PHP

```
/**
 *  An example CORS-compliant method.  It will allow any GET, POST, or OPTIONS requests from any
 *  origin.
 *
 *  In a production environment, you probably want to be more restrictive, but this gives you
 *  the general idea of what is involved.  For the nitty-gritty low-down, read:
 *
 *  - https:#developer.mozilla.org/en/HTTP_access_control
 *  - https:#fetch.spec.whatwg.org/#http-cors-protocol
 *
 */
function cors() {

    # Allow from any origin
    if (isset($_SERVER['HTTP_ORIGIN'])) {
        # Decide if the origin in $_SERVER['HTTP_ORIGIN'] is one
        # you want to allow, and if so:
        header("Access-Control-Allow-Origin: {$_SERVER['HTTP_ORIGIN']}");
        header('Access-Control-Allow-Credentials: true');
        header('Access-Control-Max-Age: 86400');    # cache for 1 day
    }

    # Access-Control headers are received during OPTIONS requests
    if ($_SERVER['REQUEST_METHOD'] == 'OPTIONS') {

        if (isset($_SERVER['HTTP_ACCESS_CONTROL_REQUEST_METHOD']))
            # may also be using PUT, PATCH, HEAD etc
            header("Access-Control-Allow-Methods: GET, POST, OPTIONS");

        if (isset($_SERVER['HTTP_ACCESS_CONTROL_REQUEST_HEADERS']))
            header("Access-Control-Allow-Headers: {$_SERVER['HTTP_ACCESS_CONTROL_REQUEST_HEADERS']}");

        exit(0);
    }

    echo "You have CORS!";
}
```

Fech() , axios(), new XMLHttpRequest()

---

fetch()

The fetch() method in JavaScript is used to request data from a server. The request can be of any type of API that returns the data in JSON or XML. The fetch() method requires one parameter, the URL to request, and returns a promise.

```
fetch('url') #api for the get request
.then(response => response.json())
.then(data => console.log(data));
```

Difference Between fetch() and XMLHttpRequest()

fetch() allows you to make network requests similar to XMLHttpRequest (XHR). The main difference is that the Fetch API uses Promises, which enables a simpler and cleaner API, avoiding callback hell and having to remember the complex API of XMLHttpRequest.

axios()

Axios: Axios is a Javascript library used to make HTTP requests from node.js or XMLHttpRequests from the browser and it supports the Promise API that is native to JS ES6. It can be used intercept HTTP requests and responses and enables client-side protection against XSRF. It also has the ability to cancel requests.

```
axios.get('url')
  .then((response) => {

    # Code for handling the response
  })
  .catch((error) => {

    # Code for handling the error
  })
```

API

What is an API?

Application Programming Interface (API) is a software interface that allows two applications to interact with each other without any user intervention. API is a collection of software functions and procedures. In simple terms, API means a software code that can be accessed or executed. API is defined as a code that helps two different software's to communicate and exchange data with each other.

Why would we need an API?

- Application Programming Interface acronym API helps two different software's to communicate and exchange data with each other.
- It helps you to embed content from any site or application more efficiently.
- APIs can access app components. The delivery of services and information is more flexible.
- Content generated can be published automatically.
- It allows the user or a company to customize the content and services which they use the most.
- Software needs to change over time, and APIs help to anticipate changes

Types of API

There are mainly four main types of APIs:

Open APIs: These types of APIs are publicly available to use like OAuth APIs from Google. It has also not given any restriction to use them. So, they are also known as Public APIs.

private APIs:These types of APIs are not publicly available to use

Partner: Specific rights or licenses to access this type of API because they are not available to the public.

Internal APIs: Internal or private. These APIs are developed by companies to use in their internal systems. It helps you to enhance the productivity of your teams. Composite APIs: This type of API combines different data and service APIs.

WEB API

A Web API is an application programming interface for the Web.

Examples of web API:

```
const myElement = document.getElementById("demo");

function getLocation() {
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(showPosition);
  } else {
    myElement.innerHTML = "Geolocation is not supported by this browser.";
  }
}

function showPosition(position) {
  myElement.innerHTML = "Latitude: " + position.coords.latitude +
  "<br>Longitude: " + position.coords.longitude;
}
```

Browser APIs All browsers have a set of built-in Web APIs to support complex operations, and to help accessing data.

For example, the Geolocation API can return the coordinates of where the browser is located.

A Server API can extend the functionality of a web server.

Third Party APIs

Third party APIs are not built into your browser.

To use these APIs, you will have to download the code from the Web.

Examples:

YouTube API - Allows you to display videos on a web site.

Twitter API - Allows you to display Tweets on a web site.

Facebook API - Allows you to display Facebook info on a web site.

Amazon's API -gives developers access to Amazon's product selection.

API Testing tools

```
POSTMAN, Ping API, vREST
```

The Machanism of API

See how API works in:

More WEB APIS

Web History API

The Web History API provides easy methods to access the windows.history object.

The window.history object contains the URLs (Web Sites) visited by the user

- back() # Loads the previous URL in the history list
- forward() # Loads the next URL in the history list
- go() # Loads a specific URL from the history list

```
<button onclick="myFunction()">Go Back</button>

<script>
function myFunction() {

   window.history.back();
   or
   window.history.go(-2);
   or
   history.forward() # this is same as window.history.go(1)

}
</script>
```

Web Storage API

```
<p id="demo"></p>

<script>
localStorage.setItem("name","John Doe");
document.getElementById("demo").innerHTML = localStorage.getItem("name");
</script>
```

Difference LocalStorage ,SessionStorage and Cookie

LocalStorage	SessionStorage	Cookie
The storage capacity of local storage is 5MB/10MB	The storage capacity of session storage is 5MB	The storage capacity of Cookies is 4KB
As it is not session-based, it must be deleted via javascript or manually	It's session-based and works per window or tab. This means that data is stored only for the duration of a session, i.e., until the browser (or tab) is closed	Cookies expire based on the setting and working per tab and window
The client can only read local storage	The client can only read local storage	Both clients and servers can read and write the cookies
There is no transfer of data to the server	There is no transfer of data to the server	Data transfer to the server is exist
There are fewer old browsers that support it	There are fewer old browsers that support it	It is supported by all the browser including older browser

JavaScript Fetch API

The Fetch API interface allows web browser to make HTTP requests to web servers. 😀 No need for XMLHttpRequest anymore.

```
<p id="demo">Fetch a file to change this text.</p>

<script>
getText("fetch_info.txt");

async function getText(file) {
  let myObject = await fetch(file);
  let myText = await myObject.text();
  document.getElementById("demo").innerHTML = myText;
}
</script>
```

RESTFUL API

REST stands for Representational State Transfer. REST defines a set of functions like GET, PUT, DELETE, etc. that clients can use to access server data. Clients and servers exchange data using HTTP.

The main feature of REST API is statelessness. Statelessness means that servers do not save client data between requests. Client requests to the server are similar to URLs you type in your browser to visit a website. The response from the server is plain data, without the typical graphical rendering of a web page.

Communication between Client and Server

In the REST architecture, clients send requests to retrieve or modify resources, and servers send responses to these requests. Let’s take a look at the standard ways to make requests and send responses.

Working:

A request is sent from client to server in the form of a web URL as HTTP GET or POST or PUT or DELETE request. After that, a response comes back from the server in the form of a resource which can be anything like HTML, XML, Image, or JSON. But now JSON is the most popular format being used in Web Services.

Response with code

Conditions to be RESTFUL API

To be considered RESTful, an API (Application Programming Interface) must adhere to a set of architectural constraints and principles. REST, which stands for Representational State Transfer, is an architectural style for designing networked applications. Here are the key conditions to be considered a RESTful API:

Statelessness: The API should be stateless, meaning that each request from a client to the server must contain all the information needed to understand and process the request. The server should not store any client context between requests.

Client-Server Architecture: The API follows a client-server architecture, where the client and server are separate entities. The client is responsible for the user interface and user experience, while the server handles the application's business logic and data storage.

Uniform Interface: The API should have a uniform and consistent interface for interacting with resources. This means using standard HTTP methods (GET, POST, PUT, DELETE, etc.) and adhering to standard HTTP status codes (200 OK, 404 Not Found, 500 Internal Server Error, etc.).

Resource-Based: Resources are the key abstractions in a RESTful API. Each resource should have a unique identifier (URI) and can be manipulated using the standard HTTP methods. Resources can be things like data objects, services, or any other type of information that can be named.

Representation: Resources can have multiple representations, such as JSON, XML, HTML, or others. The client can specify the desired representation in the request, and the server should respond with the requested representation if available.

Stateless Communication: Each request from a client to the server must contain all the information needed to understand and process the request. The server should not store any client context between requests.

Hypermedia as the Engine of Application State (HATEOAS): The API should provide links (hypermedia) in the responses to help clients navigate the API and discover available actions dynamically.

Layered System: The API can be designed in a layered architecture, where different components can interact through a uniform interface without knowing the internal workings of other components.

By adhering to these RESTful principles, an API becomes more scalable, flexible, and easier to maintain, leading to better communication and interactions between clients and servers.

PROMISE

In JavaScript, a promise is a good way to handle asynchronous operations. It is used to find out if the asynchronous operation is successfully completed or not.

A promise may have one of three states.

Pending Fulfilled Rejected

A promise starts in a pending state. That means the process is not complete. If the operation is successful, the process ends in a fulfilled state. And, if an error occurs, the process ends in a rejected state.

For example, when you request data from the server by using a promise, it will be in a pending state. When the data arrives successfully, it will be in a fulfilled state. If an error occurs, then it will be in a rejected state.

The Promise object supports two properties:

state and result.

While a Promise object is "pending" (working), the result is undefined. When a Promise object is "fulfilled", the result is a value. When a Promise object is "rejected", the result is an error object.

A Promise has four states:

fulfilled: Action related to the promise succeeded rejected: Action related to the promise failed pending: Promise is still pending i.e. not fulfilled or rejected yet settled: Promise has fulfilled or rejected

A promise can be created using Promise constructor. Syntax

```
var promise = new Promise(function(resolve, reject) {
const x = "geeksforgeeks";
const y = "geeksforgeeks"
if(x === y) {
        resolve();
} else {
        reject();
}
});

promise.
        then(function () {
                console.log('Success, You are a GEEK');
        }).
        catch(function () {
                console.log('Some error has occurred');
        });


The Promise() constructor takes a function as an argument. The function also accepts two functions resolve() and reject().

If the promise returns successfully, the resolve() function is called. And, if an error occurs, the reject() function is called.

It means either resolve is called or reject is called. Here, then() has taken one argument which will execute, if the promise is res
```

or

```
# returns a promise
let countValue = new Promise(function (resolve, reject) {
   reject('Promise rejected');
});

# executes when promise is resolved successfully
countValue.then(
    function successValue(result) {
        console.log(result);
    },
 )

# executes if there is an error
.catch(
    function errorValue(result) {
        console.log(result);
    }
);

output
Promise rejected
```

In the above program, the promise is rejected. And the catch() method is used with a promise to handle the error.

Advantages of using Promises:

A better option to deal with asynchronous operations.

Provides easy error handling and better code readability.

DIAGRAM OF Promise Object-> How it works:

JAVASCRIPT ASYNC

"async and await make promises easier to write"

async makes a function return a Promise

await makes a function wait for a Promise

Async Syntax

The keyword async before a function makes the function return a promise:

```
async function myFunction() {
  return "Hello";
}

same as

function(){
    return Promise.resolve('hello');
}

full example

<script>
function myDisplayer(some) {
  document.getElementById("demo").innerHTML = some;
}

async function myFunction() {return "Hello";}

myFunction().then(
  function(value) {myDisplayer(value);},
  function(error) {myDisplayer(error);}
);</script>
```

Await Syntax

The await keyword can only be used inside an async function.

The await keyword makes the function pause the execution and wait for a resolved promise before it continues:

```
<scrip>
async function myDisplay() {
  let myPromise = new Promise(function(resolve, reject) {
    resolve("I love You !!");
  });
  document.getElementById("demo").innerHTML = await myPromise;
}

myDisplay();
</script>
```

Hoisting

In most of the examples so far, we've used var to declare a variable, and we have initialized it with a value. After declaring and initializing, we can access or reassign the variable.

If we attempt to use a variable before it has been declared and initialized, it will return undefined.

```
# Attempt to use a variable before declaring it
console.log(x);

# Variable assignment
var x = 100;
but
# Attempt to use a variable before declaring it
console.log(x);

# Variable assignment without var
x = 100;
Output
ReferenceError: x is not defined
```

```
# Initialize x in the global scope
var x = 100;

function hoist() {
  # A condition that should not affect the outcome of the code
  if (false) {
    var x = 200;
  }
  console.log(x); # undefined
}

hoist();
```

The Top Five Errors, According to Google Here they are, listed and explained in reverse order, the five most common HTTP errors. Drumroll, please…

5. HTTP Error 401 (Unauthorized) This error happens when a website visitor tries to access a restricted web page but isn't authorized to do so, usually because of a failed login attempt.

6. HTTP Error 400 (Bad Request) This is basically an error message from the web server telling you that the application you are using (e.g., your web browser) accessed it incorrectly or that the request was somehow corrupted on the way.

7. HTTP Error 404 (Not Found) Most people are bound to recognize this one. A 404 error happens when you try to access a resource on a web server (usually a web page) that doesn't exist. Some reasons for this happening can for example be a broken link, a mistyped URL, or that the webmaster has moved the requested page somewhere else (or deleted it). To counter the ill effect of broken links, some websites set up custom pages for them (and some of those are really cool).

8. HTTP Error 403 (Forbidden) This error is similar to the 401 error, but note the difference between unauthorized and forbidden. In this case no login opportunity was available. This can happen, for example, if you try to access a (forbidden) directory on a website.

And the most common HTTP error of all is…….

1. HTTP Error 500 (Internal Server Error) The description of this error pretty much says it all. It's a general-purpose error message for when a web server encounters some form of internal error. For example, the web server could be overloaded and therefore unable to handle requests properly.

Judging by Google's search statistics, this problem is a lot more common than 404 errors.

JS DATATYPE

In JavaScript, there are eight data types, which can be categorized into two main groups: primitive data types and reference data types. Here's a list of all the data types in JavaScript:

Primitive Data Types :

Number: Represents numeric values, e.g., 42, 3.14. *String*: Represents textual data, e.g., "Hello, World!". *Boolean*: Represents true or false values. *Undefined*: Represents a variable that has been declared but not assigned a value. *Null*: Represents an intentional absence of any object value. *Symbol (ES6)*: Represents a unique and immutable value used as an identifier for object properties.

Reference Data Types :

Object: Represents a collection of key-value pairs. *Array*: Represents a list-like collection of elements. *Function*: Represents executable code. *Date*: Represents a date and time. *RegExp*: Represents regular expressions. *Map (ES6)*: Represents a collection of key-value pairs with ordered keys. *Set (ES6)*: Represents a collection of unique values with no duplicates.

```
# Primitive Data Types
let num = 42; # Number
let str = "Hello, World!"; # String
let isTrue = true; # Boolean
let undef; # Undefined
let noValue = null; # Null
let sym = Symbol("unique"); # Symbol (ES6)

# Reference Data Types
let obj = { key: "value" }; # Object
let arr = [1, 2, 3]; # Array
let func = function () { return "I am a function."; }; # Function
let today = new Date(); # Date
let regex = /pattern/; # RegExp
let map = new Map(); # Map (ES6)
let set = new Set(); # Set (ES6)
```

Symbol Data Type :

n JavaScript, the Symbol data type was introduced in ECMAScript 2015 (ES6) and is used to create unique and immutable values. Symbols are often used as *property keys in objects* to ensure that they are unique and not accidentally overridden or accessed by other parts of the code. Symbols are guaranteed to be unique, which means two symbols with the same description will not be equal.

To create a symbol, you can use the Symbol() function, optionally passing a string as its description:

```
# Creating symbols
const symbol1 = Symbol();
const symbol2 = Symbol("description");
const symbol3 = Symbol("description"); # Even with the same description, symbol3 !== symbol2

console.log(typeof symbol1); # Output: "symbol"
```

Here are some important points about symbols:

Unique: Every symbol value is unique, even if they have the same description.

Cannot be Coerced: Symbols do not have an automatic conversion to other data types. Attempting to add a symbol to a string, for example, will not concatenate them; they remain distinct.

Hidden from *for...in Loops*: Symbols are not enumerable in *for...in loops*, meaning they won't be iterated over unintentionally.

Object.getOwnPropertySymbols: You can use Object.getOwnPropertySymbols() to retrieve an object's own symbol properties.

Global Symbol Registry: You can use Symbol.for() and Symbol.keyFor() to create and retrieve symbols from a global symbol registry.

```
# Using symbols as object properties
const id = Symbol("id");
const user = {
  [id]: "12345",
  name: "John Doe",
  age: 30
};

console.log(user[id]); # Output: "12345"

# Symbol not enumerable in for...in loop
for (const key in user) {
  console.log(key);# Output: "name" and "age" (Symbol "id" is not shown)
}

# Retrieving an object's own symbol properties
const symbols = Object.getOwnPropertySymbols(user);
console.log(symbols); # Output: [Symbol(id)]

# Global symbol registry
const globalSymbol = Symbol.for("global");
console.log(Symbol.keyFor(globalSymbol)); # Output: "global"
```

what is the difference between primitive and reference data types in javascript with definitions and examples

In JavaScript, data types can be broadly categorized into two main groups: primitive data types and reference data types.

1. **Primitive Data Types**: Primitive data types are immutable data types that hold simple values. They are stored directly in the variable's location and are copied by value when assigned to another variable. There are six primitive data types in JavaScript:

a. Number: Represents numeric values, including integers and floating-point numbers.

```
let age = 25; // Integer
let pi = 3.14; // Floating-point number
```

b. String: Represents a sequence of characters (text).

```
let name = "John Doe";
let message = 'Hello, world!';
```

c. Boolean: Represents a logical value, either `true` or `false`.

```
let isStudent = true;
let isAdmin = false;
```

d. Undefined: Represents a variable that has been declared but not assigned a value.

```
let undefinedVar;
console.log(undefinedVar); // Output: undefined
```

e. Null: Represents the intentional absence of any value.

```
let nullValue = null;
```

f. Symbol (added in ECMAScript 6): Represents a unique, immutable value used as an identifier for object properties.

```
const sym = Symbol('mySymbol');
```

2.**Reference Data Types**: Reference data types are objects and are more complex than primitive data types. They store references (memory addresses) to the actual values rather than the values themselves. When you assign a reference data type to a variable or pass it as an argument to a function, you are working with a reference to the original object in memory, not a copy of it. There are four main reference data types in JavaScript:

a. Object: Represents a collection of key-value pairs and is a fundamental data structure in JavaScript.

```
let person = { name: "John", age: 30 };
let car = { make: "Toyota", model: "Camry" };
```

b. Array: Represents an ordered list of elements, which can be of any data type.

```
let numbers = [1, 2, 3, 4, 5];
let fruits = ["apple", "banana", "orange"];
```

c. Function: Represents a reusable block of code that can be invoked using its name.

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

d. Date: Represents a specific moment in time.

```
let currentDate = new Date();
let anothertDate = new Date();
# the both variable not equal.
```

Write in detail how primitive data type work with immutable and reference data type work with memory and mutable in javascript

1. **Primitive Data Types**:

a. *Immutable Nature*: Primitive data types in JavaScript are immutable, meaning once they are assigned a value, that value cannot be changed. Any operation that appears to modify a primitive data type actually creates a new value rather than modifying the original one.

Let's take the example of a string:

```
let name = "John";
```

If you try to change the string value like this:

```
name = "Jane";
```

In reality, the original value "John" remains unchanged, and a new value "Jane" is created and assigned to the variable `name`. This immutability ensures that the original value is preserved and helps avoid unexpected side effects in the code.

b. *Memory Allocation*: Primitive data types are stored directly in memory locations known as the stack. When you declare a variable of a primitive data type, the actual value is directly stored in the memory location associated with that variable.

For example, if you declare two variables holding primitive data types:

```
let age = 25; // A memory location is reserved for `age` to hold the value 25.
let pi = 3.14; // A separate memory location is reserved for `pi` to hold the value 3.14.
```

The variables `age` and `pi` store their respective values directly in their allocated memory locations.

c.*Immutable Behavior*: As mentioned earlier, primitive data types are immutable. When you assign a new value to a variable holding a primitive data type, the old value is discarded, and the new value is stored in the same memory location.

For example:

```
let x = 10;
x = 20; // The value 10 is replaced by 20 in the same memory location for variable `x`.
```

```
function primitive() {
    let car = "volvo";
let bus = "volvo";
if(car === bus){
  return true; # true
}else{
  return false;
}
};

# Another Example let yourage = 50;
let yourage = 50;
let anotherage = yourage;
yourage = 60;
console.log(yourage)  # 60
console.log(anotherage)  # 50
```

In this case, the value 10 is replaced by 20 in the memory location for the variable `x`.

2. **Reference Data Types**:

a. *Memory Allocation*: Reference data types, such as objects and arrays, are more complex than primitive data types. They are stored in memory locations known as the heap. When you create an object or an array, the variable holds a reference (memory address) to the location where the object or array is stored, rather than directly storing the value itself.

For example:

```
let person = { name: "John", age: 30 };
```

The `person` variable holds a reference to the memory location where the object `{ name: "John", age: 30 }` is stored in the heap.

```
function obj() {
    let car = {name:"Volvo",company:"China"};
let bus = {name:"Volvo",company:"China"};
if(car === bus){
  return true;
}else{
  return false;
}
}
obj();

 /* For Function Reference  */

 var func = function(){return "mahmodul Karim"};

console.log(func); // output ƒ (){return "mahmodul Karim"} because it is used as refernce

var getFunc = func;
console.log(getFunc); // output ƒ (){return "mahmodul Karim"} because it is used as refernce

console.log(func()) // output 'mahmodul Karim'

console.log(getFunc()) // output 'mahmodul Karim'

 /* it will return false becuase their location is difference although carrying same property value  */
```

b. *Mutable Nature*: Reference data types are mutable, which means you can change their properties or elements without creating a new object. Since you're working with references, any change you make to the object through one reference will affect all other references pointing to the same object.

```
let person = { name: "John", age: 30 };
let anotherReference = person; // both `person` and `anotherReference` now point to the same object in memory.

person.age = 31; // This change will affect both `person` and `anotherReference`.

console.log(person.age); // Output: 31
console.log(anotherReference.age); // Output: 31
```

c. *Mutability Behavior*: Since reference data types are mutable, you can modify their properties or elements in place, which can be a powerful feature. However, it also requires careful consideration to avoid unintentional side effects when multiple references point to the same object.

JAVASCRIPT Accessing Properties

In JavaScript, you can access properties of an object using dot notation or square bracket notation. Here's how you can access object properties using both methods:

1. Dot Notation:

Dot notation is the most common way to access object properties. It involves using a dot (`.`) followed by the property name.

```
const person = {
  firstName: "John",
  age: 30
};

console.log(person.firstName); // Output: John
```

2. Square Bracket Notation:

Square bracket notation involves using square brackets `[]` and providing the property name as a string inside the brackets.

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

Square bracket notation is especially useful when the property name contains special characters or spaces, or when the property name is stored in a variable:

```
const propertyName = "firstName";
console.log(person[propertyName]); // Output: John

const specialProperty = "first name";
console.log(person[specialProperty]); // Output: John
```

Both dot notation and square bracket notation achieve the same result of accessing object properties. Choose the one that best fits your needs and use cases.

JavaScript Functions

As of my last update in September 2021, JavaScript is a versatile programming language that allows developers to define their own functions. The number of user-defined functions in a JavaScript codebase can vary widely depending on the complexity of the application. There is no specific limit to the number of user-defined functions you can have in JavaScript.

Here are some common types of user-defined functions in JavaScript along with explanations for each:

1. **Function Declarations**:

   ```
   console.log(add(12,33)); // it is hoisted

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

   Function declarations are created using the `function` keyword followed by the function name, a list of parameters in parentheses, and the function body enclosed in curly braces. They are hoisted to the top of their scope, meaning you can call the function even before its declaration in the code.

2. **Function Expressions**:

```
console.log(multiply(10,15)); //output Uncaught ReferenceError: multiply is not defined
const multiply = function(a, b) {
  return a * b;
};
```

Function expressions involve assigning an anonymous function to a variable. The `const` keyword here is used to declare a constant variable holding the function. Function expressions are not hoisted, so you can only use them after they have been declared.

3. **Arrow Functions**:

```
const divide = (a, b) =>  a / b; // {} and return are optional when single line statement
 const divide = name =>  name; // () it is optional to use with single parameter

  Note : Single Line arrow function can't be used return keyword and not used to elclose {} curely brackets.these are optional.
  In arrow function , () can't be used while only one parameter here. () is optional.
```

Arrow functions provide a shorter syntax for creating functions, especially when the function body consists of a single expression. They automatically capture the `this` value of the enclosing context. Like function expressions, arrow functions are not hoisted.

4. **Generator Functions**: *Generator functions* return a generator object. Generator objects are used either by calling the next method on the generator object or using the generator object in a “for of” loop (as shown in the above program) The Generator object is returned by a generating function and it conforms to both the iterable protocol and the iterator protocol.

```
// Generate Function generates an
// infinite series of Natural Numbers
function* nextNatural() {
        let naturalNumber = 1;

        // Infinite Generation
        while (true) {
                yield naturalNumber++;
        }
}

// Calling the Generate Function
let gen = nextNatural();

// Loop to print the first
// 10 Generated number
for (let i = 0; i < 10; i++) {

        // Generating Next Number
        console.log(gen.next().value);
}
// Output
1
2
3
4
5
6
7
8
9
10
```

- Another Example of Generator Function.

```
function* countToFive() {
   for (let i = 1; i <= 5; i++) {
     yield i;
   }
}
```

Generator functions use the `function*` syntax and contain one or more `yield` statements. They can be paused and resumed during execution, allowing you to generate a sequence of values over time.

5. **Constructor Functions**:

```
function Person(name, age) {
  this.myname = name;
  this.myage = age;
  this.address = "Chitagong";



}

# as an example of class constructor to differentiate from function constructor

class Person {
 constructor(){
  this.myname = name;
  this.myage = age;
  this.address = "Chitagong";
 }

}

const person1 = new Person('John', 30); // it's instantiat of both function cnstructor and class constructor
 console.log(person1);
 // output Person {myname: 'John', myage: 30, address: 'Chittagong'}

const person2 = new Person('Jane', 25);
 console.log(person2);
 // output Person {myname: 'Jane', myage: 25, address: 'Chittagong'}
```

Constructor functions are used to create objects using the `new` keyword. They typically initialize object properties and methods using the `this` keyword.

6. **Method Functions**:

```
const car = {
  brand: 'Toyota',
  model: 'Camry',
  startEngine: function() {
    return 'Engine started!';
  //  or
    return "brand Name is " + this.brand + "Model name is " + this.model;
  },
};
```

Method functions are functions defined as properties of objects. They can access the object's properties and are often used to encapsulate logic relevant to the object.

7. **Immediately Invoked Function Expressions (IIFE)**:

```
(function() {
  // Your code here
})();

// Example with Paremeter
  (function(num2,num3) {
  num2 = 45; num3 = 50; return num2 * num3;
})(); // output 2250 // not invoked outside.
```

IIFEs are anonymous functions that are executed immediately after being defined. They are commonly used to create a private scope and avoid polluting the global namespace.

8. **Callback Functions**:

```
function fetchData(url, callback) {
  // Asynchronous operation to fetch data
  // Once data is retrieved, call the callback function
  fetch(url)
  .then(res=>res.json())
  .then(data=> callback(data));
  callback(data); // {name:'mahmodul Karim',age:40};
}

function processData(mydata) {

  // Process the data here
  // mydata = {name:'mahmodul Karim',age:40}

  let name = mydata.name;
  let age = mydata.age;
}

fetchData('https://example.com/data', processData);
```

Callback functions are functions that are passed as arguments to other functions. They are called once the asynchronous operation is complete.

9. **Recursive Functions**:

```
function factorial(n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1); // 5 *
  }                              // 5 * 4
                                 // 5 * 4 * 3
                                 // 5 * 4 * 3 * 2
                                 // 5 * 4 * 3 * 2 * 1  = 120
}

factorial(5); // 120
```

Recursive functions are functions that call themselves during their execution. They are often used for tasks that can be broken down into smaller, repetitive subtasks.

These are just a few examples of the many types of user-defined functions that can be created in JavaScript. Each function type serves a specific purpose and offers flexibility in how you structure your code. Understanding these function types and their use cases will help you become more proficient in writing JavaScript code.

JS OBJECTS

A JavaScript object is a fundamental data structure that allows you to store and organize data using key-value pairs. Objects are used extensively in JavaScript to represent complex data structures, and they can hold various types of values, including other objects, functions, arrays, and primitive values like strings and numbers.

Here's a basic example of a JavaScript object:

```
// Creating an object using object literal notation
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  isStudent: false,
  sayHello: function() {
    console.log(`Hello, my name is ${this.firstName} ${this.lastName}.`);
  }
};
```

In this example, `person` is an object that has several properties:

- `firstName`, `lastName`, `age`, and `isStudent` are properties with corresponding values.
- `sayHello` is a property that holds a function, which can be invoked using the `person.sayHello()` syntax.

JavaScript objects can be classified into several categories based on their properties and behavior:

1. **Plain Objects**: These are generic objects created using object literal notation or the `Object` constructor. They can have various properties and methods.

```
const person = {
  firstName: "Alice",
  lastName: "Johnson",
  age: 25
};
```

2. **Function Objects / Constructor**: Objects created using constructor functions. These are used to create instances of user-defined classes or functions.

```
function Car(make, model) {
  this.make = make;
  this.model = model;
}

const myCar = new Car("Toyota", "Camry");



function Book(title, author) {
  this.title = title;
  this.author = author;
  this.getInfo = function() {
    return `${this.title} by ${this.author}`;
  };
}

const book1 = new Book("The Great Gatsby", "F. Scott Fitzgerald");
const book2 = new Book("To Kill a Mockingbird", "Harper Lee");

console.log(book1.getInfo());
console.log(book2.getInfo());
```

3. **Built-in Objects**: These are objects provided by the JavaScript environment, such as `Array`, `String`, `Math`, and `Date`.

```
const numbers = [1, 2, 3, 4]; or numbers = new Array(); numbers[0] or numbers['name']
const text = new String("Hello, world!");
const currentDate = new Date(); // currentDate.getFullYear()

const x1 = new String();   // A new String object
const x2 = new Number();   // A new Number object
const x3 = new Boolean();  // A new Boolean object
const x4 = new Object();   // A new Object object
const x5 = new Array();    // A new Array object
const x6 = new RegExp();   // A new RegExp object
const x7 = new Function(); // A new Function object
const x8 = new Date();     // A new Date object
```

4. **Custom Objects/User-defined object/class constructor**: Objects created from custom classes using modern JavaScript class syntax.

// Class constructor simple example.

```
class Animal {

  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(`${this.name} makes a sound.`);
  }
}

const cat = new Animal("Whiskers");
cat.speak(); // Output: Whiskers makes a sound.
```

Another Example of user defined or custom object.

```
var person = {
  firstName: "John",
  lastName: "Doe",
  age: 30,
  isStudent: false,
  address: {
    street: "123 Main St",
    city: "Anytown",
    country: "USA"
  },
  hobbies: ["reading", "swimming", "gardening"],
  sayHello: function() {
    console.log("Hello, I'm " + this.firstName);
  }
};
```

Primitive Properties: firstName: A string property representing the first name of the person. lastName: A string property representing the last name of the person. age: A number property representing the age of the person. isStudent: A boolean property indicating whether the person is a student. Nested Object:

address: An object property representing the address of the person. It contains nested properties like street, city, and country. Array Property:

hobbies: An array property containing a list of hobbies. Method:

sayHello: A method (a function within an object) that logs a greeting message using the person's firstName.

5. **DOM Objects**: Objects that represent elements in the Document Object Model (DOM), used to manipulate HTML and XML documents in the browser.

```
const myButton = document.getElementById("myButton");
myButton.addEventListener("click", () => {
  console.log("Button clicked!");
});
```

6. **Host Objects**: Objects provided by the hosting environment (like the browser or Node.js) rather than by the JavaScript language itself. Examples include `window` in the browser or `process` in Node.js.

```
console.log(window.innerWidth); // Browser-specific object
console.log(process.env.NODE_ENV); // Node.js-specific object
```

7. **Object Prototypes**: Object prototypes are shared properties and methods that can be used by multiple instances of objects, reducing memory usage.

Example: Adding a shared method to the Person prototype.

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

or
class Person{

  constructor(name,age){
  this.name = name;
  this.age = age;

  }

}

Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.name} and I'm ${this.age} years old.`);
};
const person1 = new Person('dullal',30);
const person2 = new Person('mkarim',29);
person1.sayHello();
person2.sayHello();
```

8. **Object Literal Notation**:Object literal notation allows you to create objects directly without using constructors.

Example: Creating an object using literal notation.

```
const car = {
  make: "Toyota",
  model: "Corolla",
  year: 2023,
  getInfo: function() {
    return `${this.year} ${this.make} ${this.model}`;
  }
};

console.log(car.getInfo());
```

9. **Singleton Objects**:Singleton objects are objects that are instantiated only once. Subsequent requests for the object return the same instance.

Example: Creating a singleton object using a closure.

```
const Singleton = (function() {
  let instance;

  function createInstance() {
    return {
      data: "This is a singleton object."
    };
  }

  return {
    getInstance: function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

const singleton1 = Singleton.getInstance(); // "This is a singleton object."
const singleton2 = Singleton.getInstance();  // "This is a singleton object."

console.log(singleton1 === singleton2); // true
```

10. **`call()`, `bind()` and `apply()`** : are methods provided by JavaScript's Function prototype. They allow you to manipulate the `this` context of a function and pass arguments to it. While they are commonly used with functions, you can also use them with object methods since methods in JavaScript are essentially functions that are properties of objects.

Here's how you can use `call()`, `bind()`, and `apply()` with an object and its methods:

Let's say we have an object representing a person:

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

1. **Using `call()`:**

The `call()` method allows you to call a function and explicitly set the value of `this` within that function. It also accepts arguments as separate parameters.

```
const fullName = person.fullName.call(person);
console.log(fullName); // Output: "John Doe"
```

In this example, we're using `call()` to invoke the `fullName` function of the `person` object, while explicitly setting the value of `this` to `person`.

2. **Using `apply()`:**

The `apply()` method is similar to `call()`, but it takes arguments as an array.

```
const fullName = person.fullName.apply(person);
console.log(fullName); // Output: "John Doe"
```

Like `call()`, `apply()` sets the value of `this` to the `person` object while invoking the `fullName` function.

3. **Using `bind()`:**

The `bind()` method returns a new function that, when invoked, has its `this` context set to the provided value, and optionally, it can "bind" specific arguments to the function.

```
const fullNameFunction = person.fullName.bind(person);
const fullName = fullNameFunction();
console.log(fullName); // Output: "John Doe"
```

In this example, `bind()` creates a new function (`fullNameFunction`) that is bound to the `person` object. When `fullNameFunction` is called, it has the same behavior as the original `fullName` method of the `person` object.

These methods can be particularly useful when you want to borrow methods from one object to use with another or when you need to control the `this` context explicitly. They provide flexibility in how you manage function execution and context within your JavaScript objects.

JavaScript objects and their classifications play a crucial role in creating structured and maintainable code for a wide range of applications.

OOP-JAVASCRIPT And Class Constructor

Certainly! Let's break down the core principles of Object-Oriented Programming (OOP) in JavaScript: class constructor, encapsulation, inheritance, polymorphism, and abstraction, along with code examples for each concept.

Class Constructor:

A class constructor is a blueprint for creating objects of a certain type. It defines the structure and behavior of objects that belong to that class. In JavaScript, you can create a class using the `class` keyword.

```
class Animal {

  constructor(name, sound) {
    this.animalName = name;
    this.animalsound = sound;
    this.default = "all are four-footed animals";
  }

  makeSound() {
    console.log(`${this.animalName} makes a ${this.animalsound} sound. Moreover animal Default Behaviour is ${this.default}`);
  }

}

const cat = new Animal("Cat", "meow");

cat.default = "Cat is four footed animal"; // Cat is four footed animal

const dog = new Animal('Dog','geo geo');

cat.makeSound(); // Output: Cat makes a meow sound.
dog.makeSound(); // Output: Dog makes a geo sound.
```

Encapsulation:

Encapsulation refers to the practice of bundling data (attributes) and methods (functions) that operate on that data into a single unit, usually a class. This restricts direct access to the internal details of an object and enforces controlled interaction. For example, if you have a class that has one or more private fields that you use to store the data, then you are in encapsulation.

OOPs restrict direct access to its methods and variables by encapsulating the code and data together. In javascript, the data binds with the functions acting on that data in the process of achieving encapsulation in order to control the data and validate it.

```
class BankAccount {

  constructor(accountNumber, balance) {
    this.accountNumber = accountNumber;
    this.balance = balance;
  }

  deposit(amount) {
    this.balance += amount;
    console.log(`Deposited ${amount}. New balance: ${this.balance}`);
  }

  withdraw(amount) {
    if (amount <= this.balance) {
      this.balance -= amount;
      console.log(`Withdrawn ${amount}. New balance: ${this.balance}`);
    } else {
      console.log("Insufficient funds.");
    }
  }
}

const account = new BankAccount("12345", 1000);
account.deposit(500); // Output: Deposited 500. New balance: 1500
account.withdraw(200); // Output: Withdrawn 200. New balance: 1300

// Use var keyword to make data members private.
// Use setter methods to set the data and getter methods to get that data.

class Student
  {
    constructor()
    {
       var name;
       var marks;
    }
        getName()
        {
          return this.name;
        }
      setName(name)
      {
        this.name=name;
      }

      getMarks()
      {
        return this.marks;
      }
    setMarks(marks)
    {
        if(marks<0||marks>100) //  this validated is an encapsulation
        {
          alert("Invalid Marks");
        }
      else
        {
          this.marks=marks;
        }
    }
    }
    var stud=new Student();
     stud.setName("John");
```

```
    stud.setMarks(110);//alert() invokes
    document.writeln(stud.getName()+" "+stud.getMarks()); // jhon undefined
```

Inheritance:

Inheritance allows a class to inherit properties and methods from another class, creating a parent-child relationship. The child class can extend the functionality of the parent class while also adding new features.

```
class Bird extends Animal {
  constructor(name, sound, flying) {
    super(name, sound); // it is a symptom that we have borrowed properties from parent class
    this.Bflying = flying;
    this.Bsound = sound;
  }

  fly() {
    if (this.Bflying) {
      console.log(`${this.name} is flying.`);
    } else {
      console.log(`${this.name} cannot fly.`);
    }
  }
}

const eagle = new Bird("Eagle", "screech", true);
eagle.makeSound(); // Output: Eagle makes a screech sound.
eagle.fly(); // Output: Eagle is flying.
```

Polymorphism:

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables you to write code that can work with objects of various types in a consistent way.

```
class Shape {
  area() {
    // Placeholder for calculating area
  }
}

class Circle extends Shape {
  constructor(radius) {
    super();
    this.radius = radius;
  }

  area() {
    return Math.PI * this.radius ** 2; // new Math() ; Math.PI = 3.14
  }
}

class Rectangle extends Shape {
  constructor(width, height) {
    super();
    this.width = width;
    this.height = height;
  }

  area() {
    return this.width * this.height;
  }
}

const shapes = [new Circle(5), new Rectangle(4, 6)];
shapes.forEach(shape => {
  console.log(`Area: ${shape.area()}`);
});
// Output: Area: 78.53981633974483
//         Area: 24
```

Abstraction:

The concept of Abstraction in JavaScript is to hide the implementation details and highlight an object's essential features to the users. That's how embedding Abstraction in a JavaScript program can enhance the readability of the code and avoid duplication. By providing only important details to the users, it also improves the security of an application.

```
class Car {
  constructor(make, model) {
    this.make = make;
    this.model = model;
  }

  start() {
    console.log(`${this.make} ${this.model} engine started.`);
  }
}

const myCar = new Car("Toyota", "Camry");
myCar.start(); // Output: Toyota Camry engine started.


function Shape() {
    if (this.constructor === Shape) {
        throw new Error("Cannot instantiate abstract class Shape");
    }
    this.draw = function() {
        throw new Error("Cannot call abstract method draw from Shape");
    }
}

function Circle() {
    Shape.call(this); // function Shape (as a cnstructor) , call method used to get object accessibility from Shape Constructor.
    this.draw = function() {
        console.log("Drawing a Circle");
    }
}
Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;

let circle = new Circle();
circle.draw(); // "Drawing a Circle"
let shape = new Shape(); // Error: Cannot instantiate abstract class Shape
```

In this example, the `Car` class abstracts away the complex details of an actual car and provides a simple interface for starting the engine.

These concepts form the foundation of object-oriented programming in JavaScript. They allow you to create modular, maintainable, and extensible code by modeling real-world entities and interactions in a structured manner.

what is proto type in easy definition with code explanation ? how is javascript is proto type based opp language with code example? what is diffrence between proto and proto type?

Prototype in Easy Definition:

In JavaScript, a prototype is a mechanism that allows objects to inherit properties and methods from other objects. It serves as a blueprint for creating new objects with shared functionality.

Code Explanation:

In JavaScript, every object has an associated prototype. When you access a property or method on an object, JavaScript first checks if the object itself has that property or method. If not, it looks up the prototype chain to find the property or method in the prototype of the object.

Here's a simple example to illustrate prototypes in JavaScript:

```
// Create a prototype object
const animalPrototype = {
  makeSound() {
    console.log("Animal makes a sound.");
  }
};

// Create a new object using the prototype
const dog = Object.create(animalPrototype); // new animalPrototype();
dog.makeSound(); // Output: Animal makes a sound.

// Add a new property to the prototype
animalPrototype.legs = 4;

// Access property from prototype
console.log(dog.legs); // Output: 4
```

In this example, we create an `animalPrototype` object with a `makeSound` method. Then, we create a `dog` object using `Object.create(animalPrototype)`, which sets the prototype of `dog` to be `animalPrototype`. This means that `dog` inherits the `makeSound` method from `animalPrototype`.

We also add a `legs` property to the `animalPrototype`. Since `dog` inherits from `animalPrototype`, it can access the `legs` property as well.

Difference between `__proto__` and `prototype`:

1. **`__proto__` (also known as `__proto__`):** It's a property that exists on every object in JavaScript and points to the object's prototype. It's used to establish the prototype chain, allowing inheritance of properties and methods.
2. **`prototype`:** This is a property of constructor functions. Constructor functions are used to create objects with shared properties and methods. The `prototype` property is an object that becomes the prototype of objects created using the constructor function. It is not directly accessible from instances; rather, it is used as a template for the instances' prototypes through the `__proto__` property.

Here's an example to demonstrate the difference:

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

Person.prototype.sayHello = function() {
  console.log(`Hello, my name is ${this.name}`);
};

const person1 = new Person("Alice"); // person1.__proto__

console.log(person1.__proto__ === Person.prototype); // Output: true
console.log(person1.hasOwnProperty("name")); // Output: true
console.log(person1.hasOwnProperty("sayHello")); // Output: false
```

In this example, `Person.prototype` is the prototype of objects created by the `Person` constructor. The `__proto__` property of `person1` points to `Person.prototype`, allowing it to access the `sayHello` method even though `person1` does not directly have a `sayHello` property.

HTML EVENTS HANDERLS FOR JAVASCRIPT

Certainly! HTML event handlers are attributes or properties that allow you to specify JavaScript code to be executed when a particular event occurs on an HTML element. Here are some important HTML event handlers explained with definitions and code examples:

1. **onclick:**
   - **Definition:** The `onclick` event handler triggers when the user clicks the element.
   - **Example:**

```
<button onclick="alert('Button clicked')">Click me</button>
```

2. **onmouseover:**
   - **Definition:** The `onmouseover` event handler is triggered when the user's mouse pointer enters the element.
   - **Example:**

```
<div onmouseover="this.style.backgroundColor = 'yellow'">Hover over me</div>
```

3. **onmouseout:**
   - **Definition:** The `onmouseout` event handler fires when the user's mouse pointer leaves the element.
   - **Example:**

```
<div onmouseout="this.style.backgroundColor = 'white'">Hover over me</div>
```

4. **onchange:**
   - **Definition:** The `onchange` event handler is triggered when the value of an input element changes and focus is lost.
   - **Example:**

```
<input type="text" onchange="alert('Value changed')" />
```

5. **onsubmit:**
   - **Definition:** The `onsubmit` event handler is used in forms and is triggered when the form is submitted.
   - **Example:**

```
<form onsubmit="alert('Form submitted'); return false;">
  <input type="text" />
  <input type="submit" value="Submit" />
</form>
```

6. **onkeydown, onkeyup, onkeypress:**
   - **Definition:** These event handlers trigger when a key is pressed down, released, or pressed respectively.
   - **Example:**

```
<input type="text" onkeydown="console.log('Key down: ' + event.key)" />
```

7. **onload:**
   - **Definition:** The `onload` event handler is triggered when the element (usually an image or a webpage) has finished loading.
   - **Example:**

```
<img src="image.jpg" onload="console.log('Image loaded')" />
```

8. **onerror:**
   - **Definition:** The `onerror` event handler is triggered when an error occurs while loading an element, like an image that fails to load.
   - **Example:**

```
<img src="nonexistent.jpg" onerror="console.log('Image failed to load')" />
```

9. **onfocus, onblur:**
   - **Definition:** These event handlers trigger when an element gains or loses focus.
   - **Example:**

```
<input type="text" onfocus="console.log('Input focused')" />
```

10. **onmousemove, onmousedown, onmouseup:**
    - **Definition:** These event handlers trigger when the mouse pointer moves over the element, when a mouse button is pressed down, and when a mouse button is released, respectively.
    - **Example:**

```
<div onmousemove="console.log('Mouse moved')" onmousedown="console.log('Mouse down')" onmouseup="console.log('Mouse up')">
  Hover over and interact with me
</div>
```

1. **onclick:**

   - Definition: Triggered when an element is clicked by the user.
   - Example:

```
<button onclick="alert('Button clicked!')">Click me</button>
```

2. **onmouseover:**

   - Definition: Fired when the mouse cursor moves over an element.
   - Example:

```
<div onmouseover="this.style.backgroundColor='lightgray'">Hover over me</div>
```

3. **onmouseout:**

   - Definition: Triggered when the mouse cursor leaves an element.
   - Example:

```
<div onmouseout="this.style.backgroundColor='white'">Move the mouse away</div>
```

4. **onkeydown:**

   - Definition: Fired when a key is pressed down.
   - Example:

```
<input type="text" onkeydown="console.log('Key pressed:', event.key)">
```

5. **onkeyup:**

   - Definition: Triggered when a key is released.
   - Example:

```
<input type="text" onkeyup="console.log('Key released:', event.key)">
```

6. **onsubmit:**

   - Definition: Fired when a form is submitted.
   - Example:

```
<form onsubmit="alert('Form submitted!'); return false;">
  <input type="text">
  <button type="submit">Submit</button>
</form>
```

7. **onchange:**

   - Definition: Triggered when the value of an input element (like text input, select) changes and the element loses focus.
   - Example:

```
<select onchange="alert('Value changed: ' + this.value)">
  <option value="option1">Option 1</option>
  <option value="option2">Option 2</option>
</select>
```

8. **onload:**

   - Definition: Fired when a page or an image finishes loading.
   - Example:

```
<body onload="alert('Page loaded!')">
  <!-- Page content -->
</body>
```

9. **onresize:**

   - Definition: Triggered when the browser window is resized.
   - Example:

```
<body onresize="console.log('Window resized:', window.innerWidth, window.innerHeight)">
  <!-- Page content -->
</body>
```

10. **onfocus:**
    - Definition: Fired when an element receives focus (e.g., through a click or tab navigation).
    - Example:

```
<input type="text" onfocus="console.log('Input focused')">
```

These event handlers are just a subset of what's available. They can be used inline in HTML or assigned through JavaScript for more organized code. Keep in mind that modern web development often recommends separating JavaScript from HTML for better maintainability.

STRINGS IN JAVASCRIPT

Certainly! JavaScript provides a variety of string methods that are essential for advanced-level learners. Here's a list of important string methods along with their definitions and code examples:

1. **charAt(index):** Returns the character at the specified index in a string.

```
const str = "Hello, World!";
const char = str.charAt(7); // Returns 'W'
```

2. **charCodeAt(index):** Returns the Unicode value of the character at the specified index in a string.

```
const str = "Hello";
const unicode = str.charCodeAt(0); // Returns 72
```

3. **concat(str1, str2, ...):** Concatenates two or more strings and returns the result.

```
const str1 = "Hello, ";
const str2 = "World!";
const result = str1.concat(str2); // Returns "Hello, World!"
```

4. **indexOf(searchValue, startIndex):** Returns the index of the first occurrence of the specified search value in a string, starting from the given index.

```
const str = "Hello, World!";
const index = str.indexOf("World"); // Returns 7
```

5. **lastIndexOf(searchValue, startIndex):** Returns the index of the last occurrence of the specified search value in a string, starting from the given index.

```
const str = "Hello, World!";
const index = str.lastIndexOf("o"); // Returns 8
```

6. **startsWith(searchString, startIndex):** Checks if a string starts with the specified substring and returns a boolean.

```
const str = "Hello, World!";
const startsWithHello = str.startsWith("Hello"); // Returns true
```

7. **endsWith(searchString, endIndex):** Checks if a string ends with the specified substring and returns a boolean.

```
const str = "Hello, World!";
const endsWithWorld = str.endsWith("World!"); // Returns true
```

8. **includes(searchString, startIndex):** Checks if a string contains the specified substring and returns a boolean.

```
const str = "Hello, World!";
const containsHello = str.includes("Hello"); // Returns true
```

9. **slice(startIndex, endIndex):** Extracts a portion of a string from the specified start index to the end index (exclusive).

```
const str = "Hello, World!";
const sliced = str.slice(7, 12); // Returns "World"
```

10. **substring(startIndex, endIndex):** Similar to `slice()`, but doesn't allow negative indices and can swap indices.

```
const str = "Hello, World!";
const sub = str.substring(7, 12); // Returns "World"
```

11. **substr(startIndex, length):** Extracts a substring of a specified length from the given starting index.

```
const str = "Hello, World!";
const substr = str.substr(7, 5); // Returns "World"
```

12. **toUpperCase():** Converts a string to uppercase.

```
const str = "Hello";
const uppercase = str.toUpperCase(); // Returns "HELLO"
```

13. **toLowerCase():** Converts a string to lowercase.

```
const str = "World";
const lowercase = str.toLowerCase(); // Returns "world"
```

14. **trim():** Removes leading and trailing whitespace from a string.

```
const str = "   Hello, World!   ";
const trimmed = str.trim(); // Returns "Hello, World!"
```

15. **replace(searchValue, newValue):** Replaces occurrences of a substring with a new value.

```
const str = "Hello, World!";
const newStr = str.replace("World", "Universe"); // Returns "Hello, Universe!"
```

16. **split(separator, limit):** Splits a string into an array of substrings based on a specified separator.

```
const str = "apple,banana,grape";
const fruits = str.split(","); // Returns ["apple", "banana", "grape"]
```

17. **match(regexp):** Searches a string using a regular expression and returns an array of matched results.

```
const str = "Hello, 123!";
const matches = str.match(/\d+/g); // Returns ["123"]

const str_match = "mkarim 123.dulal 456345.mdsf434asfjsd";
str_match.match(/\d+/g); // output (3) ['123', '456345', '434']
str_match.match(/\d+/); // ['123', index: 7, input: 'mkarim 123.dulal 456345.mdsf434asfjsd', groups: undefined]
```

18. **search(regexp):** Searches for a pattern using a regular expression and returns the index of the first match.

```
const str = "Hello, World!";
const index = str.search(/W\w+/); // Returns 7
```

19. **localeCompare(compareString, locales, options):** Compares two strings and returns a number indicating their relative order according to the current locale.

```
const string1 = "apple";
const string2 = "banana";
const comparison = string1.localeCompare(string2); // -1 // Returns a value based on locale comparison
```

const comparison2 = string2.localeCompare(string1); // 1

```
if (comparison < 0) {
```

console.log(`${string1} comes before ${string2}`); } else if (result > 0) { console.log(`${string1} comes after ${string2}`); } else { console.log(`${string1} and ${string2} are equal`); } ```

20. **padStart(targetLength, padString):** Pads the current string with another string until the resulting string reaches the given length.

```
const str = "5";
const padded = str.padStart(3, "0"); // Returns "005"
```

21. **padEnd(targetLength, padString):** Pads the current string with another string until the resulting string reaches the given length, but adds padding to the end.

```
const str = "5";
const padded = str.padEnd(3, "0"); // Returns "500"
```

These are some of the most important and commonly used string methods in JavaScript. Experiment with them in your code to become more proficient in manipulating strings.

ARRAY METHODS OF JAVASCRIPT

Certainly! JavaScript provides a rich set of array methods that allow you to manipulate and work with arrays in various ways. Here's a list of important array methods along with their definitions and code examples:

1. `forEach(callbackFn, thisArg?):` Executes a provided function once for each array element.

```
const numbers = [1, 2, 3];
numbers.forEach(number => {
  console.log(number);
});
// 1
// 2
// 3
```

2. `map(callbackFn, thisArg?):` Creates a new array with the results of calling a provided function on every element.

```
const numbers = [1, 2, 3];
const squared = numbers.map(number => number * number); // (3) [1, 4, 9]
```

3. `filter(callbackFn, thisArg?):` Creates a new array with all elements that pass the test implemented by the provided function.

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(number => number % 2 === 0); // 2,4
```

4. `reduce(callbackFn, initialValue):` Applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue, 0); // 0+1 = 1 // 1+2=3 // 3 + 4 == 7
```

5. `find(callbackFn, thisArg?):` Returns the first element in the array that satisfies the provided testing function.

```
const numbers = [1, 2, 3, 4, 5];
const evenNumber = numbers.find(number => number % 2 === 0); // 2
```

6. `some(callbackFn, thisArg?):` Checks if at least one element in the array satisfies the provided testing function.

```
const numbers = [1, 2, 3, 4, 5];
const hasEvenNumber = numbers.some(number => number % 2 === 0); // true
```

7. `every(callbackFn, thisArg?):` Checks if all elements in the array satisfy the provided testing function.

```
const numbers = [2, 4, 6, 8, 10,7];
const allEven = numbers.every(number => number % 2 === 0); // false
```

8. `sort(compareFn?):` Sorts the elements of an array in place and returns the sorted array.

```
const numbers = [3, 1, 4, 1, 5, 9, 2];
numbers.sort((a, b) => a - b);   // (7) [1, 1, 2, 3, 4, 5, 9]
numbers.sort((a, b) => b - a);   // (7) [9, 5, 4, 3, 2, 1, 1]
```

9. `concat(...arrays)`: Returns a new array that is a shallow copy of the original arrays concatenated together.

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const combinedArray = array1.concat(array2); // [1,2,3,4,5,6]
```

10. `slice(startIndex, endIndex)`: Returns a shallow copy of a portion of an array into a new array.

```
const numbers = [1, 2, 3, 4, 5];
const slicedArray = numbers.slice(1, 4); // [2,3,4]
console.log(numbers);  // (5) [1, 2, 3, 4, 5] after slice
```

11. `splice(startIndex, deleteCount, ...items)`: Changes the contents of an array by removing or replacing existing elements and/or adding new elements.

```
const numbers = [1, 2, 3, 4, 5];
numbers.splice(2, 2, 6, 7); // (2) [3, 4]  // Removes 2 elements starting from index 2 and inserts 6 and 7

console.log(numbers);  // (5) [1, 2, 3, 4, 5] after splice
```

12. `indexOf(searchElement, fromIndex?)`: Returns the first index at which a given element can be found in the array, or -1 if it is not present.

```
const numbers = [1, 2, 3, 4, 5];
const index = numbers.indexOf(3);
```

13. `lastIndexOf(searchElement, fromIndex?)`: Returns the last index at which a given element can be found in the array, or -1 if it is not present.

```
const numbers = [1, 2, 3, 4, 3, 5];
const lastIndex = numbers.lastIndexOf(3);
```

14. `includes(searchElement, fromIndex?)`: Checks if an array contains a certain element, returning `true` or `false`.

```
const numbers = [1, 2, 3, 4, 5];
const includes3 = numbers.includes(3);
```

15. `isArray(array)`: Checks if a given value is an array.

```
const arr = [1, 2, 3];
const isArr = Array.isArray(arr);
```

16. `push(...elements)`: Adds one or more elements to the end of an array and returns the new length of the array.

```
const numbers = [1, 2, 3];
const newLength = numbers.push(4, 5);
```

17. `pop()`: Removes the last element from an array and returns that element.

```
const numbers = [1, 2, 3];
const lastElement = numbers.pop();
```

18. `shift()`: Removes the first element from an array and returns that element.

```
const numbers = [1, 2, 3];
const firstElement = numbers.shift();
```

19. `unshift(...elements)`: Adds one or more elements to the beginning of an array and returns the new length of the array.

```
const numbers = [2, 3, 4];
const newLength = numbers.unshift(0, 1);
```

20. `reverse()`: Reverses the order of the elements in an array in place.

```
const numbers = [1, 2, 3, 4, 5];
numbers.reverse();
```

These are some of the important array methods in JavaScript. They can be powerful tools for working with arrays efficiently and elegantly. Remember that mastering these methods and understanding their nuances will greatly enhance your ability to work with arrays in advanced JavaScript programming.

Sure, I can provide you with an extensive list of JavaScript array methods along with their definitions and code examples. Array methods are an essential part of working with arrays in JavaScript, allowing you to manipulate, transform, and manipulate array data efficiently.

Here are the commonly used array methods and some important ones for advanced-level learners:

1. **push(item1, item2, ...)**: Adds one or more items to the end of an array and returns the new length of the array.

```
const fruits = ['apple', 'banana'];
fruits.push('orange', 'grape');
// fruits: ['apple', 'banana', 'orange', 'grape']
```

2. **pop()**: Removes the last item from an array and returns that item.

```
const colors = ['red', 'blue', 'green'];
const removedColor = colors.pop();
// colors: ['red', 'blue'], removedColor: 'green'
```

3. **unshift(item1, item2, ...)**: Adds one or more items to the beginning of an array and returns the new length of the array.

```
const numbers = [2, 3];
numbers.unshift(0, 1);
// numbers: [0, 1, 2, 3]
```

4. **shift()**: Removes the first item from an array and returns that item.

```
const letters = ['b', 'c', 'd'];
const removedLetter = letters.shift();
// letters: ['c', 'd'], removedLetter: 'b'
```

5. **concat(array1, array2, ...)**: Combines two or more arrays and returns a new array.

```
const arr1 = [1, 2];
const arr2 = [3, 4];
const combined = arr1.concat(arr2);
// combined: [1, 2, 3, 4]
```

6. **slice(start, end)**: Returns a new array containing elements from the original array within the specified range.

```
const animals = ['elephant', 'lion', 'tiger', 'giraffe'];
const selectedAnimals = animals.slice(1, 3);
// selectedAnimals: ['lion', 'tiger']
```

7. **splice(start, deleteCount, item1, item2, ...)**: Changes the contents of an array by removing, replacing, or adding items in place.

```
const months = ['January', 'March', 'April'];
months.splice(1, 0, 'February');
// months: ['January', 'February', 'March', 'April']
```

8. **forEach(callback(item, index, array))**: Executes a provided function once for each array element.

```
const numbers = [1, 2, 3];
numbers.forEach(num => {
  console.log(num * 2);
});
// Output: 2, 4, 6
```

9. **map(callback(item, index, array))**: Creates a new array by calling the provided function on each element of the original array.

```
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);
// doubled: [2, 4, 6]
```

10. **filter(callback(item, index, array))**: Creates a new array with all elements that pass the test implemented by the provided function.

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);
// evenNumbers: [2, 4]
```

11. **reduce(callback(accumulator, item, index, array), initialValue)**: Applies a function against an accumulator and each element in the array to reduce it to a single value.

```
const numbers = [1, 2, 3, 4];
const sum = numbers.reduce((acc, num) => acc + num, 0);
// sum: 10
```

12. **some(callback(item, index, array))**: Checks if at least one element in the array passes the test implemented by the provided function.

```
const numbers = [1, 2, 3, 4, 5];
const hasEvenNumber = numbers.some(num => num % 2 === 0);
// hasEvenNumber: true
```

13. **every(callback(item, index, array))**: Checks if all elements in the array pass the test implemented by the provided function.

```
const numbers = [2, 4, 6, 8, 10];
const allEven = numbers.every(num => num % 2 === 0);
// allEven: true
```

14. **find(callback(item, index, array))**: Returns the first element in the array that passes the test implemented by the provided function.

```
const fruits = ['apple', 'banana', 'orange'];
const foundFruit = fruits.find(fruit => fruit.length > 5);
// foundFruit: 'banana'
```

15. **findIndex(callback(item, index, array))**: Returns the index of the first element in the array that passes the test implemented by the provided function.

```
const animals = ['cat', 'dog', 'elephant', 'lion'];
const index = animals.findIndex(animal => animal === 'elephant');
// index: 2
```

16. **sort(compareFunction(a, b))**: Sorts the elements of an array in place and returns the sorted array. You can provide a custom sorting function.

```
const numbers = [3, 1, 2, 4];
numbers.sort((a, b) => a - b);
// numbers: [1, 2, 3, 4]
```

17. **reverse()**: Reverses the order of elements in an array in place.

```
const letters = ['a', 'b', 'c'];
letters.reverse();
// letters: ['c', 'b', 'a']
```

18. **includes(item, fromIndex)**: Checks if an array contains a certain element, returning `true` or `false`.

```
const colors = ['red', 'green', 'blue'];
const hasGreen = colors.includes('green');
// hasGreen: true
```

19. **indexOf(item, fromIndex)**: Returns the first index at which a given element can be found in the array, or -1 if it is not present.

```
const fruits = ['apple', 'banana', 'orange'];
const indexOfBanana = fruits.indexOf('banana');
// indexOfBanana: 1
```

20. **lastIndexOf(item, fromIndex)**: Returns the last index at which a given element can be found in the array, or -1 if it is not present.

```
const numbers = [1, 2, 3, 2, 1];
const lastIndexOf2 = numbers.lastIndexOf(2);
// lastIndexOf2: 3
```

21. **join(separator)**: Combines all elements of an array into a string, separated by the provided separator.

```
const words = ['Hello', 'world'];
const sentence = words.join(' ');
// sentence: 'Hello world'
```

22. **toString()**: Returns a string representing the

array and its elements.

```
const numbers = [1, 2, 3];
const numbersString = numbers.toString();
// numbersString: '1,2,3'
```

23. **isArray(obj)**: Checks if a given object is an array.

```
const array = [1, 2, 3];
const isAnArray = Array.isArray(array);
// isAnArray: true
```

24. **flat(depth)**: Creates a new array with all sub-array elements concatenated into it recursively up to the specified depth.

```
const nestedArray = [1, [2, [3, 4]]];
const flattenedArray = nestedArray.flat(2);
// flattenedArray: [1, 2, 3, 4]
```

25. **flatMap(callback(item, index, array))**: Maps each element using a mapping function, then flattens the result into a new array.

```
const numbers = [1, 2, 3];
const doubledAndFlattened = numbers.flatMap(num => [num * 2]);
// doubledAndFlattened: [2, 4, 6]
```

26. **reduceRight(callback(accumulator, item, index, array), initialValue)**: Applies a function against an accumulator and each element in the array (from right to left) to reduce it to a single value.

```
const numbers = [1, 2, 3, 4];
const reversedSum = numbers.reduceRight((acc, num) => acc + num, 0);
// reversedSum: 10
```

27. **copyWithin(target, start, end)**: Copies a portion of an array to another location within the same array.

```
const colors = ['red', 'green', 'blue', 'yellow'];
colors.copyWithin(1, 0, 2);
// colors: ['red', 'red', 'green', 'yellow']
```

28. **fill(value, start, end)**: Fills all or part of an array with a static value.

```
const numbers = [1, 2, 3, 4];
numbers.fill(0, 1, 3);
// numbers: [1, 0, 0, 4]
```

29. **keys()**: Returns an array iterator with the keys of the array.

```
const fruits = ['apple', 'banana', 'orange'];
const iterator = fruits.keys();
for (const key of iterator) {
  console.log(key);
}
// Output: 0, 1, 2
```

30. **values()**: Returns an array iterator with the values of the array.

```
const colors = ['red', 'green', 'blue'];
const iterator = colors.values();
for (const value of iterator) {
  console.log(value);
}
// Output: 'red', 'green', 'blue'
```

31. **entries()**: Returns an array iterator with key/value pairs of the array.

```
const animals = ['cat', 'dog', 'elephant'];
const iterator = animals.entries();
for (const [index, animal] of iterator) {
  console.log(`Index: ${index}, Animal: ${animal}`);
}
// Output: Index: 0, Animal: 'cat'
//         Index: 1, Animal: 'dog'
//         Index: 2, Animal: 'elephant'
```

These are some of the most commonly used array methods in JavaScript. They provide powerful tools for manipulating and working with arrays efficiently. Understanding these methods will greatly enhance your ability to work with array data in your advanced-level JavaScript projects.