........................Pure function ....
A pure function is a function that always returns the same output given a specific input

Expression and Statement:
At its simplest terms, expressions are evaluated to produce a value. On the other hand, statements are executed to make something happen.

....Fetch vs Axios....................................

Fetch's body = Axios' data
Fetch's body has to be stringified, Axios' data contains the object
Fetch has no url in request object, Axios has url in request object
Fetch request function includes the url as parameter, Axios request function does not include the url as parameter.
Fetch request is ok when response object contains the ok property, Axios request is ok when status is 200 and statusText is 'OK'
To get the json object response: in fetch call the json() function on the response object, in Axios get data property of the response object.
While using service worker, you have to use fetch API only if you want to intercept the HTTP request
Ex. While performing caching in PWA using service worker you won't be able to cache if you are using axios API (it works only with fetch API)

......................................Jest vs Enzyme...................
Jest and Enzyme

Both Jest and Enzyme are specifically designed to test React applications, Jest can be used with any other Javascript app but Enzyme only works with React.

Jest can be used without Enzyme to render components and test with snapshots, Enzyme simply adds additional functionality.

Enzyme can be used without Jest, however Enzyme must be paired with another test runner if Jest is not used.

As described, we will use:

Jest as the test runner, assertion library, and mocking library
Enzyme to provide additional testing utilities to interact with elements

....................................Curring....................................

Curring Function: Currying in JavaScript transforms a function with multiple arguments into a nested series of functions, each taking a single argument. Currying helps you avoid passing the same variable multiple times, and it helps you create a higher order function
Exp.
Clouser Function

```
        function sum(a) {
         return (b) => {
                return (c) => {
                        return a + b + c
                }
         }
   }
   console.log(sum(1)(2)(3)) // 6
```

Curring Function

```
        function sum(a) {
         return function(b){
                if(!b) return a;
                return sum(a+b);

         }
   }
   console.log(sum(1)(2)(3)());
```

............................Paritial Application................................
Partial application allows us to fix a function's arguments.
Exp:

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

let partialSum=partialFun.bind(null,10,20);
console.log(partialSum(30));
console.log(partialSum(40));
console.log(partialSum(50));
```

.... Compose..
In Functional Programming, Compose is the mechanism that composes the smaller units (our functions) into something more complex (you guessed it, another function)(right to left)
Exp:

```
function multiply2(price){
   return price*2;
}
function normalizeVal(price){
   return Math.abs(price);
}
function compose(fn1,fn2){
   return function (price){
          return fn1(fn2(price));
   }
}
let cmp=compose(multiply2,normalizeVal);
console.log(cmp(-20));
```

.... Pipe..
A pipe is a function or operator that allows us to pass the output of a function as the input of another or its just reverse of compose(left to right).

Lexical scope: Lexical scope is the ability for a function scope to access variables from the parent scope.

First Order function:-> First Order function is function that does not accept another function as argument and does not return value as a function.

Module In Javascript: JavaScript modules allow you to break up your code into separate files.

Promise: A Promise is a special JavaScript object. It produces a value after an asynchronous operation completes successfully, or an error if it does not complete successfully due to time out, network error, and so on.

Server Sent Event: A server-sent event is when a web page automatically gets updates from a server. This was also possible before, but the web page would have to ask if any updates were available. With server-sent events, the updates come automatically.

Event Bubbling : Propagate the event from inner most element to outer most element.
Event Capturing: Progate the event from outermost element to inner most element .
Event Propagation: Use event efficently

JSON: Javascript Object Notation is used to transmitting data between server and web application.

ECMAScript: ECMAScript (often abbreviated as ES) is a standardized scripting language primarily used for writing client-side and server-side applications. It is the specification that defines the syntax, semantics, and behavior of the scripting language commonly known as JavaScript.

app shell model:The App Shell model, also known as the Application Shell Architecture, is a design pattern used in web development to optimize the performance and user experience of web applications, particularly those that are progressive web applications (PWAs)

*********************Shadow DOM**
Shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree — this shadow DOM tree starts with a shadow root, underneath which you can attach any element, in the same way as the normal DOM.

-From the light DOM, shadow DOM elements are not visible to querySelector. Particularly, Shadow DOM elements can have ids conflicting with the elements in the light DOM.
-Shadow DOM obtains own stylesheets. The style rules from the outer DOM will not be applied
EXP:

```
customElements.define('show-hello', class extends HTMLElement {
  connectedCallback() {
        const shadow = this.attachShadow({mode: 'open'});
        shadow.innerHTML = `<p>
        Hello, ${this.getAttribute('name')}
        </p>`;
  }
});
```

*** Use Strict***
The JavaScript strict mode is used to generates silent errors. It provides "use strict".
- Using a variable, without declaring it, is not allowed.
- Using an object, without declaring it, is not allowed.
- Deleting a variable (or object) is not allowed.
- Deleting a function is not allowed
- Duplicating a parameter name is not allowed.
- Writing to a read-only property is not allowed.

****
Let/var/Const**
Var
- Variables declared with var are in the function scope.
- Hosting allowed
- Reassign the value are allowed
- Redeclaration of the variable are allowed

Let
- Variables declared as let are in the block scope.
- Hosting are not allowed
- Reassign the value are allowed
- Redeclaration of the variable are not allowed

Const
- Variables declared as const are in the block scope
- Hosting are not allowed.
- Reassign the value are not allowed
- Redeclaration of the variable are not allowed

.............................Arrow Function ******************************

Syntax
No arguments (arguments are array-like objects)
No prototype object for the Arrow function
Cannot be invoked with a new keyword (Not a constructor function)
No own this (call, apply & bind won't work as expected)
It cannot be used as a Generator function
Duplicate-named parameters are not allowed

...................................Hoisting...........................
Hoisting is a behavior where a function or variable can be used before declaration.

.... This keyword ....

In javascript this refer to an object.
In object method this refer to an object.
Alone, this refer to an global object.
In function this refer to an object.
In a function, in strict mode this is undefined.
In event,this refer to an element that received the event.
Method like apply,call bind this refer to any object .

.... JS Proxies..
In JavaScript, proxies (proxy object) are used to wrap an object and redefine various operations into the object such as reading, insertion, validation, etc. Proxy allows you to add custom behavior to an object or a function.
Ex.
```
 let target={
          name:'anil',
    email:'anil@gmail.com'
 }
 let handler={
```

```
            get:function(obj,prop){
            return obj[prop]?obj[prop]:'Property does not exit'
      }
  }
  let sethandler={
            set:function(obj,prop,value){
            obj[prop]=value;
              return ;
      }
  }

 let proxy = new Proxy(target,handler);
 let proxy = new Proxy(target,sethandler);
 proxy.age=30;
 console.log(proxy);
```

.... Object destucturing........................

Object Destructuring is the syntax for extracting values from an object property and assigning them to a variable.

Exp.

```
 let obj={
    name:'anil',
    emil:'anil@gmail.com',
    address:{
             country:'India',
             state:'up'
    }
}
let {name,emil,address:{country}}=obj;
console.log(country);
```

.... Array Destructuring.........................

Destructuring the array in JavaScript simply means extracting multiple values from data stored in objects and arrays.

Exp.

```
let arr=[10,20,30,40];
let [a,b,c,d]=arr;
```

..............................Callback function ....

A JavaScript callback is a function which is to be executed after another function has finished execution. A more formal definition would be - Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.

.............................Callback hell..............................
Callback hell refers to the situation where callbacks are nested within other callbacks several levels deep, potentially making it difficult to understand and maintain the code

.... 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. and rejected .

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

.... Aync..............................
The async and await keywords allow asynchronous, promise-based behavior to be written more easily and avoid promise chains.

.... Memory Leak...
A Memory leak can be defined as a piece of memory that is no longer being used or required by an application but for some reason is not returned back to the OS.
"piece of memory that is no longer being used of an application "

Exp.
```
let arr = [];
for(let i = 5; i > 1; i++){
        arr.push(i-1)
}
console.log(arr);
```

The four types of common JavaScript leaks

1.Undeclared/ Global Variables
2.Forgotten timers or callbacks
3.Out of DOM references (Event listeners)
4.Closures

.... Lexical Scope ....
Lexical scope is the ability for a function scope to access variables from the parent scope. We call the child function to be lexically bound by that of the parent function.

Exp:

```
var scope = "I am global";
function whatismyscope(){
  var scope = "I am just a local";
  function func() {return scope;}
  return func;
}
console.log(whatismyscope()());
```

................................Higher Order function ....

Higher order functions are functions that take one or more functions as arguments, or return a function as their result.

Exp.

```
function calculate(operation, initialValue, numbers) {
 let total = initialValue;
 for (const number of numbers) {
         total = operation(total, number);
 }
 return total;
}

function sum(n1, n2) {
 return n1 + n2;
}

function multiply(n1, n2) {
 return n1 * n2;
}

calculate(sum, 0, [1, 2, 4]);     // => 7
calculate(multiply, 1, [1, 2, 4]); // => 8
```

.... Prototype...................................

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

Exp.

```
let employee={
        firstName:'Anil Kumar',
   lastName:'Singh',
   fullName:function(){
        return this.firstName+" "+this.lastName;
   }
}

let student={
        firstName:'Anil Kumar',
    lastName:'Singh',
    __proto__:employee
}
console.log(student.fullName());
```

.............................Prototype and __proto__....................................

-prototype is a property of a Function object. It is the prototype of objects constructed by that function.
__proto__ is an internal property of an object, pointing to its prototype. Current standards provide an equivalent Object.getPrototypeOf(obj) method, though the de facto standard __proto__ is quicker.
exp.

```
let emp={
    email:'anil@gmail.com',
}
function Costing(name){
    this.name=name;
}
Costing.prototype.age=20;  //
let obj= new Costing('Anil');
obj.__proto__=emp;
console.log(obj.email);
```

....Prototypal and classical inheritance..................
Classical model: Object is created from a class.
Prototypal model: Object is created directly from another object.

.... CORS...
Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy (SOP). However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as cross-site request forgery (CSRF).

.... Map Object ....
A map in JavaScript is a data structure that stores key-value pairs, allowing efficient lookup, insertion, and deletion operations based on the keys.

A Map holds key-value pairs where the keys can be any datatype.
A Map remembers the original insertion order of the keys.
A Map has a property that represents the size of the map.

Difference between Map/Object

Object	Map
Not directly iterable	Directly iterable
Do not have a size property	Have a size property
Keys must be Strings (or Symbols)	Keys can be any datatype
Keys are not well ordered	Keys are ordered by insertion
Have default keys	Do not have default keys

.... Set Object ....

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

.................................Memory Heap ....

It is used to store objects and functions in JavaScript. The engine doesn't allocate a fixed amount of memory. Instead, it allocates more space as required.

.... Memory Stack..

Stack: It is a data structure used to store static data. Static data refers to data whose size is known by the engine during compile time. In JavaScript, static data includes primitive values like strings, numbers, boolean, null, and undefined. References that point to objects and functions are also included. A fixed amount of memory is allocated for static data. This process is known as static memory allocation.

.... Memory Heap Vs Memory Stack...

Stack	Heap
Primitive data types and references	Objects and functions
Size is known at compile time	Size is known at run time
Fixed memory allocated	No limit for object memory

.... Call Stack ....

JavaScript is a single-threaded language as it utilizes only one "Call Stack" for the management of the "Global" and "Function" Execution Contexts. Call Stack in JavaScript is a type of data structure that maintains the tracks of invoked and executed functions.

.... CallBack Queqe...

Callback Queue: This is where your asynchronous code gets pushed to, and waits for the execution.

.... Event Loop ....

Event loop is just a guardian who keeps a good communication with Call Stack and Callback Queue. It checks if the call stack is free, then lets know the callback queue. Then Callback queue passes the callback function to Call stack to be executed. When all the callback functions are executed, the call stack is out and global execution context is free.

.... Web APIs ....
Web API works as JS engines assistant. When JS engine have to deal with asynchronous code, it takes the help of Web API. Web API handles the blocking behavior of JavaScript code.
In this case, from our code above, we can say Web API will take the callback function.
Exp.
setTimeout(lunch, 3000);

....Job Queqe..

Job Queue: Apart from Callback Queue, browsers have introduced one more queue which is "Job Queue", reserved only for new Promise() functionality. So when you use promises in your code, you add .then() method, which is a callback method. These `thenable` methods are added to Job Queue once the promise has returned/resolved, and then gets executed.

........................Depandency Injection ....
A dependency is any external resource a program needs to work. These can be external libraries the code literally depends on or services the program functionally needs, like internet APIs and databases.

.... Design Patten..
A design pattern is a term used in software engineering for a general, reusable solution to a commonly occurring problem in software design.

Creational design
  singleton pattern
  fatory pattern
  prototype pattern

Structural
  Proxy Pattern

Behavioural
  Observer Pattern

Singleton Pattern:
Singletons are classes which can be instantiated once, and can be accessed globally. This single instance can be shared throughout our application, which makes Singletons great for managing global state in an application.

Exp:

```
let instance;
let counter=0;
class Counter{
   constructor(){
         if(instance){
               throw new Error("You can only create one instance");
         }
         instance=this;
   }
   getInstance(){
         return this;
   }
   getCount(){
         return counter;
   }
   increment(){
         return ++counter;
   }
   decrement(){
         return ++counter;
   }

}
const singletonCounter = Object.freeze(new Counter());
export default singletonCounter;
```

Proxy Pattern:
Definition1:Intercept and control interactions to target objects
or
Definition2:The Proxy Pattern uses a Proxy intercept and control interactions to target objects. Let's say that we have a person object. We can access properties with either dot or bracket notation, and modify property values in a similar fashion.

```
let proxy= new Proxy(object,handler);

const person = {
  name: "John Doe",
  age: 42,
  email: "john@doe.com",
  country: "Canada",
};

const personProxy = new Proxy(person, {
  get: (target, prop) => {
	console.log(`The value of ${prop} is ${target[prop]}`);
	return target[prop];
  },
  set: (target, prop, value) => {
	console.log(`Changed ${prop} from ${target[prop]} to ${value}`);
	target[prop] = value;
	return true;
  },
});
personProxy.age=30;
console.log(personProxy.age);
```

Reflect
JavaScript provides a built-in object called Reflect, which makes it easier for us to manipulate the target object when working with proxies.

Prototype Pattern:
Definition 1: Share properties among many objects of the same type.
Exp:

```
class Dog {
  constructor(name) {
	this.name = name;
  }
  bark() {
	return `Woof!`;
  }
}

const dog1 = new Dog("Daisy");
const dog2 = new Dog("Max");
const dog3 = new Dog("Spot");
Dog.prototype.play = () => console.log("Playing now!");
dog1.play();
```

Observer Pattern:
Sort Definition: Use observables to notify subscribers when an event occurs

Definition: With the observer pattern, we can subscribe certain objects, the observers, to another object, called the observable. Whenever an event occurs, the observable notifies all its observers!

Exp:
Comp1

```
class Observable {
  constructor() {
          this.observers = [];
  }

  subscribe(f) {
          this.observers.push(f);
  }

  unsubscribe(f) {
          this.observers = this.observers.filter(subscriber => subscriber !== f);
  }

  notify(data) {
          this.observers.forEach(observer => observer(data));
  }
}

export default new Observable();
```

Comp 2:

```
import React from "react";
import { Button, Switch, FormControlLabel } from "@material-ui/core";
import { ToastContainer, toast } from "react-toastify";
import observable from "./Observable";

function handleClick() {
  observable.notify("User clicked button!");
}

function handleToggle() {
  observable.notify("User toggled switch!");
}

function logger(data) {
  console.log(`${Date.now()} ${data}`);
}
```

```
function toastify(data) {
  toast(data, {
        position: toast.POSITION.BOTTOM_RIGHT,
        closeButton: false,
        autoClose: 2000
  });
}

observable.subscribe(logger);
observable.subscribe(toastify);

export default function App() {
  return (
        <div className="App">
        <Button variant="contained" onClick={handleClick}>
        Click me!
        </Button>
        <FormControlLabel
        control={<Switch name="" onChange={handleToggle} />}
        label="Toggle me!"
        />
        <ToastContainer />
        </div>
  );
}
```

Factory Pattern:
Sort Definition: Use a factory function in order to create objects.
Definition: With the factory pattern we can use factory functions in order to create new objects. A function is a factory function when it returns a new object without the use of the new keyword!

Exp:

```
const createUser = ({ firstName, lastName, email }) => ({
  firstName,
  lastName,
  email,
  fullName() {
        return `${this.firstName} ${this.lastName}`;
  }
});
```

```
const user1 = createUser({
  firstName: "John",
  lastName: "Doe",
  email: "john@doe.com"
});

const user2 = createUser({
  firstName: "Jane",
  lastName: "Doe",
  email: "jane@doe.com"
});

console.log(user1);
console.log(user2);
```

............................Require vs Import........................

require is a function used to import modules in Node. js, while import is a new keyword that is used to import modules in ECMAScript 6 (ES6). require is a synchronous operation and will block the execution of the script until the module is loaded and ready to be used.

............................obj.freeze vs obj.seal..........................

freeze makes an object completely immutable, while Object. seal allows existing properties to be modified, but prevents the addition and deletion of new properties.

Exp1:

```
const originalStudent = {
        name: 'xyz',
        percentage: 85
};
const frozenStudent = Object.freeze(originalStudent);
frozenStudent.percentage = 90; // Property cannot be altered
frozenStudent.age = 16;        // No new property can be added
delete frozenObject.name;      // Returns false. Property cannot be deleted
```

Exp2:

```
const student = { name: 'xyz', age: 16, address: { street: '136 street' } };
```

```
Object.freeze(student);
obj.name = 'abc'; // will be ignored
obj.address.street = '456 street'; // will be successful
```

Exp 21:

```
const person = {
  name: 'xyz',
  designation: 'develper'
};

Object.seal(person);
person.name = 'abc'; // changes the value of the name property
console.log(person.name); // 'abc'
person.age = 16; // No new property can be added
console.log(person.age); // undefined
delete person.designation; // returns false.because the object is sealed
console.log(person.designation); // developer
```

Real-life examples where we might use Object.freeze() and Object.seal():
Object.freeze(): When you're working with configuration objects that should not be modified at runtime, you can use Object.freeze() to make sure that these objects are immutable and cannot be modified by mistake.
Object.seal(): To secure the user input, like form data, by preventing the addition of new properties while still allowing modifications to existing properties such as changing the form fields value.

.........................Way to create Object..................................
1. Object constructor

```
        var person = new Object();
  person.name = "Anand",
  person.getName = function(){
    return this.name ;
  };
```

2. Literal constructor

```
  var person = {
    name : "Anand",
    getName : function (){
        return this.name
    }
```

```
   }
```

3. function Constructor

```
        function Person(name){
    this.name = name
    this.getName = function(){
          return this.name
    }
   }
```

4. Prototype

```
  function Person(){};
  Person.prototype.name = "Anand";
```

5. Function/Prototype combination.

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

6. Singleton

```
  var person = new function(){
    this.name = "Anand"
   }
```

................Error Handling Strategies for Async-Await...................

Using try-catch:The most common way to handle errors in async-await code is by using try-catch blocks, similar to how you would handle errors in synchronous code.

Exp:

```
async function fetchData() {
  try {
        const response = await fetch('https://api.example.com/data');
        const data = await response.json();
        console.log(data);
  }catch (error){
        console.error('Error:', error);
  }
}
```

```
fetchData();
```

Handling Rejections at the Call Site: Another strategy for handling errors in async-await code is to handle the rejection of the Promise returned by the async function at the call site. This can be done using the catch() method of the Promise, similar to how you would handle errors in Promise-based code.

Exp:

```
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
}
fetchData().catch((error) => {
  console.error('Error:', error);
});
```

.... String interpolation ....

String interpolation is a great programming language feature that allows injecting variables, function calls, arithmetic expressions directly into a string. String interpolation was absent in JavaScript before ES6. String interpolation is a new feature of ES6, that can make multi-line strings without the need for an escape character. We can use apostrophes and quotes easily that they can make our strings and therefore our code easier to read as well. These are some of the reasons to use string interpolation over string concatenation.

Exp:

```
function myInfo(fname, lname, country) {
        return `My name is ${fname} ${lname}. ${country} is my favorite country`;
}
console.log(myInfo("john", "doe", "India"));
```

.... What is the second value in parseInt.......

Radix

The parseInt function accepts a second parameter known as radix . This parameter specifies what number system to use. If the radix is omitted then 10 is taken as the default. This is usually an integer between 2 and 36.

......................Dependency injection...

Dependency injection: Allows an application to load its dependencies when needed.