**Name: gunjal gupta**

**Student Id- 014560715**

**Hosted URL :** [**http://18.220.254.24/**](http://18.220.254.24/)

**Youtube link URL:** [**https://youtu.be/GHRQksxaLpM**](https://youtu.be/GHRQksxaLpM)

**GITHUB URL : https://github.com/gunjalgupta/appointment\_booking**

**Javascript**

Please use strict mode and make full use of arrow functions.

1. Use of arrow functions, split, slice, includes, typeof, JSON.stringify, JSON.parse in your program.

**Arrow functions**

* While an arrow function expression is a more compact alternative to a regular function expression, it has limitations and cannot be utilized in all cases.
* Some limitations include the inability to utilize call, apply, and bind methods as constructors, as they require on defining a scope.
* In ES6, the arrow functions were introduced.
* const bookinginfo = () => {
* document.getElementById("result").innerHTML = "Your appointments for today " + sessionStorage.getItem("data");
* }

**Split function**

The split() method splits a String into an ordered list of substrings, places them in an array, and returns the array. The division is carried out by looking for a pattern, which is passed in as the method's first parameter.

**includes function**

The includes() method returns true or false depending on if an array contains a specific value among its elements.

var firstname,lastname;

if(fname.includes(" ")){ // Use of INCLUDES OPERATOR

let arr=fname.split(" "); //Use of SPLIT OPERATION

firstname= arr[0];

lastname=arr[1];

}

else{

firstname= fname;

lastname= " ";

}

**Slice function**

The slice() function creates a new array object with a shallow copy of a piece of an array from start to end (end not included), where start and end denote the index of elements in that array. There will be no changes to the original array.

console.log("\nThe two major symptoms the patient facing are:"+imp1+" and "+imp2);

console.log("\nThe two less major symptoms the patient facing are:"+ syms.slice(2,4)); //Use of SLICE OPERATOR

**Typeof function**

The typeof operator returns a string that specifies the type of the undeclared operand.

The  return type of the typeof operator can be these primitive types:

* string
* number
* boolean
* undefined

if(typeof(fname)!== "string") { //Use of TYPEOF OPERATOR

alert(" not valid name");

}

**JSON.stringfy function**

When a replacer function or a replacer array is supplied, the JSON.stringify() method turns a JavaScript object or value to a JSON string, possibly changing values and including just the specified attributes.

**JSON.parse function**

The JSON.parse() method parses a JSON string and creates the JavaScript value or object that the string describes. Before returning the object, an optional reviver function can be used to execute a transformation on it.

let obj = {name: fname, email: email, birthdate: birth};

let data = [];

if(JSON.parse(sessionStorage.getItem('data'))) // Use of JSON.PARSE

{

data = JSON.parse(sessionStorage.getItem('data'))

}

data.push(obj)

sessionStorage.setItem('data', JSON.stringify(data)); // Use of JSON.STRINGFY

1. Destructuring object and array, spread and rest operator, closures.

**Destructuring object and array**

The destructuring assignment syntax is a JavaScript expression that allows you to extract values from arrays or properties from objects and store them in separate variables. The object and array literal expressions make it simple to construct ad hoc data packages. Destructuring can be used to assign a value to a variable that isn't part of the variable's declaration.

let obj = {name: fname, email: email, birthdate: birth};

let {name: fullname, email:mailid}= obj; //Use of OBJECT DESTRUCTURING

document.getElementById("result").innerHTML = "Appointment is booked by "+fullname+" with email "+mailid+" with age "+agee;

let [imp1,imp2,,]=syms; //Use of ARRAY DESTRUCTURING

console.log("\nThe two major symptoms the patient facing are:"+imp1+" and "+imp2);

**Spread and rest operator**

Spread syntax (...) allows an iterable, such as an array expression or string, to be extended when zero or more parameters (for function calls) or elements (for array literals) are anticipated, or an object expression to be expanded where zero or more key-value pairs (for object literals) are requested.

The rest parameter syntax enables a function to receive an endless number of arguments as an array, allowing for the representation of variadic functions in JavaScript.

let arr1=[],arr2=[];

if(sym1!==null)

{

arr1.push(sym1);

}

if(sym2!==null)

{

arr1.push(sym2);

}

if(sym3!==null)

{

arr2.push(sym3);

}

if(sym4!==null)

{

arr2.push(sym4);

}

var syms=[...arr1,...arr2]; //Use of SPREAD OPERATOR

static getCost = (...syms) =>{ // USE OF REST OPERATOR

return syms.length\*150;

}

**Closures**

A closure is a function that has been packaged (contained) with references to its surrounding state (the lexical environment). In other words, a closure allows you to access the scope of an outside function from an inside function. Closures are produced in JavaScript every time a function is created, during function creation time.

1. function age(birth) {
2. var today = new Date();
3. birth = new Date(birth);
4. const b = async () => {
5. return (today.getFullYear() - birth.getFullYear());
6. console.log(today)
7. console.log(birth)
8. }
9. return b
10. }
11. exports =age()

3.Use of require and exports, class static method, regular expressions and default arguments

**Require and exports**

When building JavaScript modules, the export statement is used to export live bindings to functions, objects, or primitive values from the module so that they may be utilized by other applications with the require statement. An imported binding's value is susceptible to change in the module that exports it. When a module changes the value of a binding that it exports, the change is reflected in the imported value.

1. function age(birth) {
2. var today = new Date();
3. birth = new Date(birth);
4. const b = async () => {
5. return (today.getFullYear() - birth.getFullYear());
6. console.log(today)
7. console.log(birth)
8. }
9. return b
10. }
11. exports =age()

const age= require('./age.js');

**Class static method**

A static method or attribute for a class is defined using the static keyword. On instances of the class, neither static methods nor static properties may be invoked. Instead, they are summoned by the class itself.

Static methods are frequently utility functions, such as those used to create or clone objects, whereas static properties are helpful for caches, fixed-configuration data, or any other data that does not need to be copied between instances.

static getCost = (...syms) =>{ // USE OF CLASS STATIC METHOD

return syms.length\*150;

}

**Regular Expression**

Regular expressions are patterns that are used to match characters in strings. Regular expressions are also objects in JavaScript. These patterns are utilized with RegExp's exec() and test() methods, as well as String's match(), matchAll(), replace(), replaceAll(), search(), and split() methods.

var emailPattern = /^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/; // Use of REGULAR EXPRESSION

if(emailPattern.test(email)==false){

alert("Not a valid email id")

} ;

**Default arguments**

If no value or undefined is supplied, default function parameters allow named parameters to be populated with default values. In JavaScript, function arguments are undefined by default. It is, however, frequently useful to provide a different default value. This is when the default settings come in handy.

static getCost(n=1) { // Use of DEFAULT ARGUMENTS

return n\*120;

}

1. Object and Classes with Object.assign demonstrating inheritance using subclasses in JavaScript with method overriding

**Inheritance in classes**

Class inheritance allows one class to extend another. In JavaScript, inheritance is allowed through the use of a prototype object refered as "Prototypal Inheritance." In JavaScript, there is just one inheritance construct: objects. Each object has a private property that contains a reference to another object known as its prototype. That prototype object has its own prototype, and so on until an object with null as its prototype is reached.

**Method overriding**

Method overriding is an OOPs concept that is closely related to inheritance. Method overriding occurs when a child class method overrides a parent class method with the same name, parameters, and return type.

// Use of CLASSES with INHERITANCE and METHOD OVERRIDING

class Fees {

static description = 'Calculates fees for the symptoms provided';

static getCost = (...syms) =>{ // USE OF REST OPERATOR and CLASS STATIC METHOD

return syms.length\*150;

}

}

class RecurringFees extends Fees {

static description = 'Calculates fees for recurring patients';

static getCost(n=1) { // Use of DEFAULT ARGUMENTS and METHOD OVERRIDING

return n\*120;

}

}

let fee

if(recurr=='first'){

fee = Fees.getCost(...syms);

console.log("\nVisiting fees for first-time patients: "+fee);

}

else{

fee= RecurringFees.getCost();

console.log("\nVisiting fees for recurring visits: "+fee);

}

**Object.assign**

Object.assign() transfers all enumerable own properties from one or more source objects to a destination object. It returns the target object as it has been changed.

let obj = {name: fname, email: email, birthdate: birth};

let src = {date: date, time:time};

Object.assign(obj,src); // Use of OBJECT.ASSIGN

1. Make external API call, you can use npm package like axios

**Fetch() for external API call**

The Fetch API exposes a JavaScript interface for accessing and altering HTTP pipeline components such as requests and replies. It also has a global fetch() function for retrieving resources asynchronously via the network in an easy and logical manner.

// Use of external API call to find users of the app.

console.log("\n Users of the app are:")

const URL = "https://jsonplaceholder.typicode.com/users";

fetch(URL)

.then((response) => response.json())

.then((data) => {

for (let i=0;i<data.length;i++)

{

let users=data[i];

console.log(users.name)

}

}).catch((error) => {

console.log(error)

});

1. Demonstrate the difference between: a. call, apply, bind b. var, let, const c. callbacks, promises, async and await

**Call apply bind**

The call() method calls a function with a given 'this' value and one-by-one parameters. Apply( ): Invokes the function and allows you to pass arguments as an array. Bind() returns a new function that accepts an array and any number of parameters.

//Call

var employee1 = {firstName: 'Jon', lastName: 'Kuperman'};

var employee2 = {firstName: 'Kelly', lastName: 'King'};

function say(greeting) {

console.log(greeting + ' ' + this.firstName + ' ' + this.lastName);

}

console.log("Calls using call()");

say.call(employee1, 'Hello'); // Hello Jon Kuperman

say.call(employee2, 'Hello'); // Hello Kelly King

//Apply

console.log("Calls using apply()");

say.apply(employee1, ['Hi']); // Hello Jon Kuperman

say.apply(employee2, ['Hi']); // Hello Kelly King

//Bind

console.log("Calls using bind()");

var sayHelloJon = say.bind(employee1);

var sayHelloKelly = say.bind(employee2);

sayHelloJon("Hello"); // Hello Jon Kuperman

sayHelloKelly("Hello"); // Hello Kelly King

**OUTPUT:**

**Text

Description automatically generated**

**Var Let and Const**

* var variables can be renamed and changed. The let function can be changed but not re-declared. Const cannot be changed or redefined.
* Var declarations are globally or function scoped, whereas let and const declarations are block scoped.
* They are all hoisted to the top of their scope. However, although var variables are initialized with undefined, let and const variables are not.
* Unlike var and let, which may be declared without being initialized, const must be initialized during the declaration process.

console.log("Use of var to indicate it can be redeclared and redefined");

var greeting = "hey hi";

var greeting = "say Hello instead";

console.log(greeting);

console.log("Use of let to indicate it can be redefined but not redeclared");

let greeter = "say Hi";

greeter = "say Hello instead"; // This will work

console.log(greeter);

let greeters = "say Hi";

//let greeters = "say Hello instead"; // error: Identifier 'greeter' has already been declared

console.log(greeters);

console.log("Use of const to indicate it can neither be redefined but not redeclared");

const greetings = "say Hi";

//greetings = "say Hello instead";// error: Assignment to constant variable.

console.log(greetings);

**OUTPUT**

**Text

Description automatically generated**

**Callbacks**

A function that is given to another function is referred to as a callback. When the first function is completed, the second function is executed. The issue with callbacks is that they cause something known as "Callback Hell." When you start layering functions within functions within functions, it becomes quite difficult to comprehend the code.

//Using Callbacks

function concatString(previous, current, callback){

setTimeout(

() => {

callback((previous + ' ' + current))

},

Math.floor(Math.random() \* 100) + 1

)

}

function concatAll(){

concatString('', 'Output', result => {

concatString(result, 'Using', result => {

concatString(result, 'Callbacks', result => {

console.log(result)

})

})

})

}

concatAll()

**Promise**

You encapsulate the whole function in a Promise and use resolve instead of the callback (or reject if there is an error). This Promise object is returned by the function. This is called a Promise Chain.

// Using promise

function concatString(previous, current){

return new Promise((resolve, reject) => {

setTimeout(

() => {

resolve(previous + ' ' + current)

},

Math.floor(Math.random() \* 100) + 1

)

})

}

function concatAll(){

concatString('', 'Output ')

.then(result => concatString(result, 'using'))

.then(result => concatString(result, 'Promise'))

.then(result => {

console.log(result) // Prints out " A B C"

})

}

concatAll()

**Async-await**

Await serves as syntactic sugar for Promises. It transforms your asynchronous code into synchronous/procedural code, making it easier for people to understand. In an async function, the await keyword ensures that all promises delivered by the async function are synchronized, i.e. they wait for each other. Await eliminates the need for callbacks in.then() and.catch() expressions ().

//Using async await

function concatString(previous, current){

return new Promise((resolve, reject) => {

setTimeout(

() => {

resolve(previous + ' ' + current)

},

Math.floor(Math.random() \* 100) + 1

)

})

}

async function concatAll(){

let toPrint = await concatString('', 'Output ')

toPrint = await concatString(toPrint, 'using ')

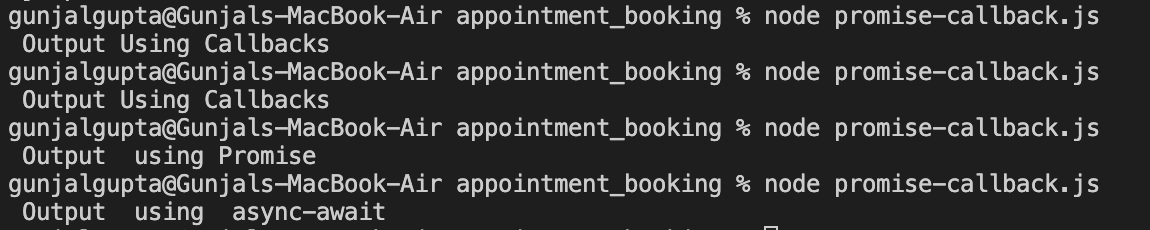
toPrint = await concatString(toPrint, 'async-await')

console.log(toPrint);

}

concatAll()

Output:



HTML5:

1.Demonstrate LocalStorage and SessionStorage using different HTML input types along with input validations (Make use of different inputvalidationsin HTML5 like patterns, autofocus, required, email etc. Describe all the concepts used in your Introduction to Topicsection).

**SessionStrorage**

The read-only sessionStorage attribute gets access to the current origin's session Storage object. The distinction between sessionStorage and localStorage is that data in localStorage does not expire, but data in sessionStorage is erased when the page session expires.

LocalStorage is similar to sessionStorage in that it has no expiration date, but sessionStorage data is removed when the page session expires — that is, when the page is closed. (When the last "private" tab is closed, localStorage data for a page loaded in a "private browsing" or "incognito" session is deleted.)

let obj = {name: fname, email: email, birthdate: birth};

let data = [];

if(JSON.parse(sessionStorage.getItem('data'))) // Use of JSON.PARSE

{

data = JSON.parse(sessionStorage.getItem('data'))

}

data.push(obj)

sessionStorage.setItem('data', JSON.stringify(data)); // Use of JSON.STRINGFY

**LocalStorage**

The window interface's localStorage read-only feature allows you to access a Storage object for the Document's origin; the stored data is preserved throughout browser sessions.

<script>

function clickCounter() {

if (localStorage.clickcount) {

localStorage.clickcount = Number(localStorage.clickcount)+1;

} else {

localStorage.clickcount = 1;

}

document.getElementById("result").innerHTML = "Your number of appointments for today " + localStorage.clickcount + ".";

}

</script>

2.GeolocationandEvents

**Geolocation**

If the user desires, the Geolocation API allows them to give their location to web apps. The user is requested for permission to report location information for privacy concerns.

function getLocation() {

navigator.geolocation.getCurrentPosition(successful, console.log);

}

const successful = async (position) => {

const {latitude, longitude} = position.coords;

let response = await fetch(`https://api.opencagedata.com/geocode/v1/json?q=${latitude}+${longitude}&key=bdcde0d9ad2b471188fe440c59b542f3`)

let data = await response.json();

console.log(data);

document.getElementById("location").innerHTML = data.results[0].formatted;

};

**Events**

Events are triggered to alert code to "interesting changes" that may impact code execution. These might result from user behaviors such as using a mouse or expanding a window, changes in the status of the underlying environment (e.g. low battery or operating system media events), and other reasons.

<p><button onclick="myFunction()" type="button">Confirm Booking</button></p>

<p><button onclick="clickCounter()" type="button">Total Bookings</button></p>

<p><button onclick="bookinginfo()" type="button">Booking Information</button></p>

<p><button onclick="getLocation()" type="button" id="location">Get Location</button></p>