Assignment – 5

Q.0) Is JavaScript single threaded or multithreaded ? what does it mean to be any ?

JavaScript is a single threaded language. JavaScript engine runs on a V8 engine that has a one call stack and one memory heap. As expected ,it executes code in order and must finishing a piece code before moving onto the next. It's synchronous, but at times that can be harmful.

For example, if a function takes a while to execute or has to wait on something, it freezes everything up in the meanwhile.

A good example of this happening is the window alert function. Alert("hello world");

You can't interact with the webpage at all until you hit Ok and dismiss the alert.

JavaScript is always synchronous and single threaded. If you are executing a JavaScript block of code on a page then no other JavaScript on that page will currently be executed. JavaScript is only Asynchronous in the sense that it can make , for example ,Ajax calls. The Ajax call will stop executing until the call returns (successfully or otherwise), at which point the callback will run synchronously. No other code will be running at this point . It won't interrupt any other code that's running.

JavaScript timers operate with this same kind of callback. Describing JavaScript as Asynchronous is perhaps misleading. It's more accurate to say that JavaScript is synchronous and single-threaded with various callback mechanisms .

Q.1) What are promises ? Why are they used?

A promise is a value that may produce a value in the future. That value can either be resolved or unresolved (in some error cases, like a network failure). It works like a real-life promise.

```
Let p= new Promise((resolve, reject)=>{
        Let a=1+2
        If(a==2){
            Resolve('success')
        }
    Else{
Reject('failed')
}
})
p.then((message)=>{
console.log('this is in the then'+ message)
}).catch((message)=>{
Console.log('this is in the catch'+ message)
```

}

Promise are used to handle asynchronous operations in JavaScript. They are easy to manage when dealing with multiple asynchronous operations where callbacks can create callback hell leading to unmanageable code. Prior to promises events callback function were used but they had limited functionalities.

Benefits of Promise

- improves code readability
- Better handling of asynchronous operations
- Better flow of control definition in asynchronous logic
- Better error handling

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){
        //do something
});
```

Parameters:

- Promise Constructor takes only one argument, a callback function.
- Callback function takes two arguments, resolve and reject.
- Perform operations inside the callback function and if everything went well then call resolve.
- If desired operations do noy go well then call reject.

Example :

```
var promise=new Promise(function(resolve ,reject){
const x=”I am in”;
const y=”Bangalore”;
if(x==y){
resolve();
```

```
}
else{
reject();
}
});
promise.
then(function(){
  console.log('success, you are in Bangalore');
}).
catch(function(){
console.log('some error has occurred');
});
```

Promise consumers

Promises can be consumed by registering functions using (.then) and (.catch)methods.

Then()

It is invoked when a promise is either resolved or rejected.

Syntax:

```
   .then(function(result){
         //handle success
}, function(error){
  //handle error
});
```

Catch()

It is invoked when a promise is either rejected or some error has occurred in execution.

Syntax:

```
.catch(function(error){
    //handle error
});
```

Q.2) What do async/await do ? Explain it in your own words.

Async/await is a new way to deal with asynchronous operations in JavaScript. It offers you the power to make your code shorter and clearer. Async/await is much more readable and more comfortable to compose while being non blocking. Before async/await callbacks and promises were used to handle asynchronous calls in JavaScript.

The browsers supported by async/await function are listed below :

- Google chrome
- Firefox
- Apple safari
- Opera.

Async means asynchronous. It allows a program to run a function without freezing the entire program. This is done using the async/await keyword. Async/await makes it easier to write promises. The keyword async before a function makes the function return a promise always. And the keyword await is used inside async functions which makes the program wait until the promise method resolves.

Example

```
async function example(){

let promise = new Promise((resolve ,reject)=>{

setTimeout(() =>resolve("done!"), 2000)

});

let result = await promise; //wait until the promise resolves(*)

alert(result); // "done!"

}

example();
```

Async/await is the extension of promises which we get as a support in the language. Async/await is a syntax that was added in ES8(ECMAScript 2017) of the JavaScript version. When we use async/await we rarely need(.then) because await handles the waiting for us . And we can use a regularly try…catch instead of (.catch). That's usually (but not always) more convenient. But at the top level of the code , when we are outside any async function we are syntactically unable to use await() so it's normal practice to add (.then/catch) to handle the final result or falling or falling through error .

Q.4) What do async function return?

Async function always returns promise. If the return value of an async function is not explicitly a promise , it will be implicitly wrapped in a promise. Since axios returns a promise the async/await can be omitted for the getData function like so your data getData will return a promise. Await the function as well to get the result . However , to be able to use await , you need to be in async function , so you need to wrap this.

```
const axios = requires('axios');

async function getData(){

const data = await axios .get('https://jsonplaceholder.typicode.com/posts');

return data ;

}

console.log(getData());
```

A function always returns a promise. Other values are wrapped in a resolved promise automatically. So , async ensures that the function returns a promise and wraps non promises in it. **Async methods can have the following return types :**

- Task<TResult>, for an async method that returns a value.
- Task, for an async method that performs an operations but returns no value .
- Void for an event handler.
- Starting with C# 7.0 , any type that as an accessible GetAwaiter() method

Q7): What are the states a promise can be in?

- Pending, if it has been resolved to a thenable which will call neither handler back as soon as possible, or resolved to another promise that is pending. This is the initial state of a promise.

- Fulfilled, if it has been resolved to a non-promise value, or resolved to a thenable which will call any passed fulfilment handler back as soon as possible, or resolved to another promise that is fulfilled. This is the state of a promise representing a successful operation.

- Rejected, if it has been rejected directly, or resolved to a thenable which will call any passed rejection handler back as soon as possible, or resolved to another promise that is rejected. This is the state of a promise representing a successful a failed operation

Q.10) What does the finally() method on promise do? Provide your explanation.

A finally() method is used for registering a callback to be invoked when a promise is settle or rejected . A finally() callback will not receive any argument since there's no reliable means of determining if the promise was fulfilled or rejected once the promise has been dealt with . The finally() method can be useful if you want to do some processing or clean up once the promise is settled, regardless of its outcome. Returns a Promise whose finally handler is set to the specified function, onFinally. This helps to avoid duplicating code in both the promise's then() and catch() handlers.

Q.15) **what is the difference between the following two lines of codes: promise.then(f1).catch(f2) and promise.then(f1,f2);**

Assuming that f1 and f2 represent asynchronous operations that return promises , yes there is a significant difference .

Option1:

- It serializes f1 and f2 so that f2 is not called until after the promise returned by f1 has been resolved
- The .catch() applies to an error in either f1 or f2 or if PromiseObj rejects.
- F2 will not be called if f1 rejects

Option 2:

- F2 does not wait for f1 to resolve . f2 is called as soon as f1 returns similar to f1() this means the async operations started by f1 and f2 will both be in flight at the same time(sometimes referred to running in parallel instead of running serially)
- The .catch() does not apply to either because it is not on the promise that is created by either of the .then() calls . then .catch() only applies to if PromiseObj rejects not f1() 0r f2().
- Both f1() and f2() will be called regardless of an error in either.

Q.5) What do then() consumers in promises return?

The then() method returns a promise which allows for method chaining . If the function passed as handler to then return a promise an equivalent promise will be exposed to the subsequent then in the method chain. A then call will return a rejected promise if the function throws an error or returns a rejected promise if onFulfilled returns a promise the return value of then will be resolved/rejected by the promise . When you return something from a then() callback, it's a big magic .If you return a value the next then() is called with that value . However if you return something promise like the next then() waits on it and is only called when that promise settles(succeeds/fails). Promises don't return values they pass them to callback (which you supply with .then()). Its probably trying to say that you are supposed to do resolve(someObjects);

Q.11) What are microtasks in JS?

A microtasks is short function which is executed after the function or program which created it exits and only if the JavaScript execution stack is empty, but before returning control to the event loop being used by the user agent to drive the scripts execution. JavaScript promises and the mutation observer API both use the microtask queue to run their callbacks. This event loop may be either the browser's main event loop or the event loop driving the web worker. However , whereas the event loop runs only the tasks present on the queue when the iteration began , one after the another , it handles the microtasks queue very differently.