Siddharth Singha Roy

Winter 19-20

Asynchronous JavaScript

Async JavaScript refers to anything in JavaScript that happens in the background. Synchronous JavaScript refers to anything that makes our program execution sequential. This means that our code is executed in the order they appear.

```
const second = () => {
    setTimeout(() => {
        console.log("Async Second called");
    }, 2000); // Execute this call back function after 2 seconds
}

const first = () => {
    console.log("Hi there");
    second();
    console.log("Done calling second");

}

first();
```

Synchronous, Asynchronous and the Event Loop

Synchronous JavaScript means all the code that we wrote previously up until this point. All the codes were executed sequentially in the order they were written in a single thread.

For example let's say we want to create a image processor in JS. We know that it takes time to process the image. So instead of waiting for the result we can pass a callback function that will be executed when the image is processed.

We do not wait for a function to finish its work and then do something with the result. Instead we let the function do its job on the background and then let the code execution continue. We also pass a callback function along with our async function. This callback function will be executed once we have completed execution of that async function.

So when our code starts executing, we create new execution stacks and execute each line of codes sequentially. However we may encounter some methods that do no live inside the JS engine. They live outside in something known as **Web API**. Other methods in the **WEB API** are DOM Manipulations, HttpRequests, Geolocation, local storage etc. We can access them only because they are in the **JavaScript runtime**. So when we call the setTimeout function a timer is created in the **web API** along with our callback function. Then the setTimeout method will be popped from execution stack and normal execution will continue. After that all the code will be executed and the execution stack becomes empty. Meantime the timer also ends. Now the callback function will be passed to the message queue. This is where **event loop** comes in. The event loop is responsible for consistently monitoring the message queue and pushing the callback function onto the execution stack if the stack is empty. The **event loop** will execute the callback functions in the same order they were passed into the message queue by the Web API.

The setTimeout function takes three arguments

1. Callback function
2. Timeout period in milliseconds
3. Argument to the callback function

Callback

```
function getRecipeID() {
    setTimeout(() => {
        const arr = [15,2,64,34];
        console.log(arr);
        setTimeout((id) => {
            console.log(id);
        }, 1500, arr[1]);
    }, 2000);
};

getRecipeID();
```

This is how we use callbacks to get async data from the server. However it results into something known as callback hell which makes our code unmanageable after some time.

We cannot access arr inside the second callback function. The reason being they have different scope chain. Also we cannot use this because if we use this then it will refer to the setTimeout.

Promises

Promises are objects that keeps track about whether a certain event has happened already or not. If it did happen then Promises determine what to do next. Implements a concept of future value that we're expecting.

So promises are finally resolved into 2 states a success state(Fulfilled) or failure state (Rejected) .

Promises are an object and hence are created with the new keyword.

Promises have 2 phases :

1. Creation phase
2. Consumption phase

The result of the then function will be a callback function that has the arguments of the resolve.

The catch method allows us to handle the case when the promise gets rejected, for example an error happened.

So if the promise gets accepted then we pass arguments to the resolve callback function. These arguments become arguments for the callback function of then.

The reject arguments become arguments for the catch method.

For multiple promises which are related to each other we use something known as chaining of promises.

So after we resolve one promise return the next promise. This allows chaining of promises.

Also one catch block is enough for all the promises in case of promise chaining.\

```
var promise1 = Promise.resolve(3);
var promise2 = 42;
var promise3 = new Promise(function(resolve, reject) {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then(function(values) {
  console.log(values);
```

```
});
```

This is the use of promises.all(). This will return a single promise if all the promises are resolved and passes the resolve argument of all the arguments as an array to the resultant promise. If any one of the promise gets rejected then the reject function will have the arguments of that rejected promise only.

Async Await

Async Await gives us a way to consume promises, not create promises. If we still want to create promises we need to do it using new Promise(…..)

Async is a special function that runs in the background. With async function, we have another keyword known as await. Await is a keyword that blocks the code execution till a promise has been resolved or rejected.

```
async function recipe() {
    const ids = await getRecipeIDs;
    const recipe = await getRecipe(ids[2]);
    console.log(recipe);
}

recipe();
```

However by default we do not have any way of handling rejects. So if a rejection occurs an error is thrown. We can use a try catch block for error handling.

```
    try{
        const ids = await getRecipeIDs;
        const recipe = await getRecipe(ids[2]);
        console.log(recipe);
    } catch(error) {
        console.log(error);
    }
```

The await keyword can only be used inside async functions. Await is used for blocking execution, Async means that the function will run in the background.

An Async function returns a promise (and not a value even if we use return 5). If we simply try to get the return in a variable and then log it, we get a Promise[<Pending>].

Instead we can use the .then method to resolve that promise.

```
recipe().then(res => console.log(res));
```