Callbacks and promises

Created	@May 8, 2022 11:17 AM
Class	
Type	
Materials	
Reviewed	☐

What are callbacks ?

If we try to basically understand core meaning of callback, then it is something that we are expected to do after some time.

In JS , call back function are expected to be passed during the function call, the function which is accepting the call back as argument is going to execute the call back accordingly.

We have a lot of functions that are already present in JS, which accepts callback functions like sort, forEach, map, reduce etc.

Even we can write our own functions that accepts a callback and uses it inside the implementation.

The best part of callback is , when we are calling the functions at that time we can define what implementation of call back we have to use.

The developer who has implemented the function which accepts callback as argument has now given power to the user to change the implementation of callbacks during the function call.

Callback as a concept exist in mostly languages, in JAVA and python it exists as lambda functions, in C++ it exists as function pointer etc.

What is function expression ?

If we are defining a function and the first word of the line is not function, then it is a function expression.

IIFE: Immediately Invoked Function Expression

Wherever we define the function expression at that moment only we call it and later it doesn’t exist.

IIFE is used to avoid naming collisions in our JS files. We generally don't find a lot of IIFE inside production level code, until or unless files are huge.

Nature of JS:

- Javascript is synchronous in nature, that means any piece of code that JS understands that piece of code will for sure execute line by line and if there is a piece of code that is taking a lot of time we have to wait for it.
- Javascript is single threaded.
- But still we have functions like setTimeout working in JS, How ?
- the settimeout function has the capability to wait for some milliseconds and then execute the function that we have passed as a callback after the wait is over.

So, JS is not very powerful, it gets a lot of features from the runtime environment. All the features that the runtime environment provides to JS, these are not native features of JS, that means this can change based on the change in environment.

JS handles the environment based features separately. It doesn't wait for them to get executed. The only thing JS does when it hits a statement which is using a runtime feature is to just notify the runtime to execute the feature and moves forward with the next line of code. It doesn't wait for the runtime to complete the task.

So what happens when runtime has completed the task ?

It waits in a queue called as event queue, whose entries only gets executed when there is no piece of code left in the call stack and no global code also left to be executed. This thing of when the queue is ready to execute is checked by event loop.

Drawbacks of callbacks

- Callback hell
- Inversion of control

Inversion of control

Due to callbacks, there is one part of the code on which we have complete control and some part of code whose control we are passing to someone else.

```
// the function fun is not written by us and is present in some other module
function fun(n, cb) { // fun is a function that takes a call back as argument
  for(let i = 0; i < n; i++) {
    console.log(i);
  }
  cb();
}
// We don't know the implementation of fun, we just know the parameters to call with

fun(10, function gun() {
  console.log("you are executing the callback");
})
```

the callback function gun, was our own function that we solely own and wrote. But the power / authority of executing the function gun, we have passed to fun. We don't know how fun is going to handle our callback and we are not even sure, whether our callback will be executed or not ?

This problem is called as Inversion of control in JS.

NOTE: Inversion of control is a bigger problem than callbacks, to rescue us from these we will introduce Promises.

Promises

Technically promises are readability enhancers which can also deal with the problem of inversion of control in callbacks.

In order to understand promises, we need to understand two foundational concepts:

- How to create a promise ?
- How to consume a promise ?

How to create a promise ?

In order to create a promise, all we have to do is to create a new promise object.

The constructor of this promise takes a function as an argument.

This function takes two arguments, `resolve` and `reject` which are also functions.

```
function fetch(url) {
    return new Promise(function (resolve, reject) {
      // some code
    });
}

function fetch1(url) {
    return new Promise((resolve, reject) => {
      // some code
    });
}
```

Any promise is in one of the following given states:

- pending → initial state, when the work is still in progress
- fulfilled → operation was completed successfully
- rejected → operation was not completed successfully

```
function fetch(url) {
    return new Promise(function (resolve, reject) {
        console.log("Started fetching from", url);
        // let's mimic the functionality, say it takes 5 sec to fetch the data
        setTimeout(function () {
            // let's create a dummy data
            let data = { // assuming this data was fetched
                message: 'completed fetching',
                success: true
            }
            console.log("Successfully fetched the data");
            if(data.success) {
                resolve(data); // resolve is an indication that promise resolved successfully
            } else {
                reject(data); // reject is an indication that promise did not resolve successfully
            }
        }, 5000);
    });
}
```

`resolve` → this is acting as an indication that change the state of promise from pending to fulfilled. Whatever data we will pass inside resolve will be returned.

`reject` → this is acting as an indication that change the state of promise from pending to rejected. Whatever data we will pass inside reject will be returned.

```
function fetch1(url) {
    return new Promise((resolve, reject) => {
        for(let i = 0; i < 10000000000; i++) {
            // holding
        }
        console.log("completed");
        resolve(true);
    });
}
```

How to consume a promise ?

The consumption of promise is the main beauty of this object which helps us to avoid inversion of control.

Whenever we call a function returning a promise, it will actually return a promise object and this is like any other JS object so we can store it inside a variable.

Now the question is will JS wait for the code to complete ? NO, because inside the logic of promise we might some async task, that JS doesn't support natively. But JS understands promise, so the moment you call the function returning promise it will immediately return the promise object and moves forward. If you're doing everything sync, then JS will wait.

This promise object is in `pending` state initially because the async task is still executing in the runtime.

When the task will be completed the state of the promise object returned back will change to `fullfilled` / `rejected` based on the outcome.

Now after the promise has resolved/rejected we can do some operations. We can bind a few functions to this promise object which will be executed when and only when the promise is completed.

How to bind these functions ?

We can use `.then` on the promise object to bind the functions. This `.then` takes function as an argument that you want to execute after promise is resolved, and this function takes an argument data which is the same data that we return while resolving the promise.

```
const response = fetch("www.xyz.com");
response.then(function (data) {
    console.log("inside the then of promise", data);
})
console.log("hurray");
```

Now if you carefully see, we are not passing the control of our function as a callback to other function. We have proper complete control on how our function is expected to be executed after promise is completed. Hence, Inversion of control is resolved.

How the promise object works behind the scenes ?

The promise object that we return has 4 major properties:

- status: status shows the current promise status I.e. pending, fulfilled, rejected
- Values: when the status variable is pending, value is undefined, the moment promise is resolved, this value is updated with the returned value. So this value property acts like a placeholder till the time promise finishes.
- OnFulfillment: This is an array, which contains the functions that we bind with `.then` . When the value parameter is updated from undefined, to the new value, and JS gives the chance for these functions to execute after call stack and global code is completed, all these functions start executing one by one with a parameter as the value.
- OnReject: This is also an array that works same like onFulfillment, with a only difference that it gets executed when promise is rejected. To insert functions in onReject, we use `.catch`

NOTE: While chaining `.then` with each other, we should keep on thing in mind, every `.then` returns another promise only, so if we don't return anything the value parameter stays undefined for the next `.then`

```
const response = fetch("www.xyz.com");
response
.then(function (data) {
    // this then also returns a promise
    console.log("inside the then of promise", data);
    return data;
})
.then(function (data) {
    console.log("inside the next then of the promise", data);
    return "Sanket"
})
.then(function (data) {
    console.log("hey hi", data);
})
console.log("hurray");
```

NOTE: In the code below, we have used the `response` variable promise only for everytime binding a `.then` so we have never used the new promise returned by `.then` so the data properties of the functions inside the `.then` will take `value` property of the `response` promise variable only.

```
const response = fetch("www.xyz.com");
response.then(function (data) {
    // this then also returns a promise
    console.log("inside the then of promise", data);
    return 100;
});
response.then(function (data) {
    console.log("inside the next then of the promise", data);
    return "Sanket"
});
response.then(function (data) {
    console.log("hey hi", data);
});
console.log("hurray");
```