Java Script
Object-Oriented JavaScript

FSMVÜ Computer Eng. Dr. Öğr Üyesi Ali NİZAM

Callback

- Simply put: A callback is a function that is to be executed after another function has finished executing—hence the name ‘call back’.

Callback requirement


```
function first(){
  console.log(1);
}
function second(){
  console.log(2);
}
first();
second();
```

Callbacks

- In implementing timers or callbacks, you need to call the handler asynchronously, mostly at a later point in time.
- Due to the asynchronous calls, we need to access variables from outside the scope in such functions.

Simulate a code delay


```
function first(){
  // Simulate a code delay
   setTimeout( function(){
    console.log(1);
  }, 500 );
}
function second(){
  console.log(2);
}
first();
second();
```

Creating callback


```
function doHomework(subject, callback) {
  alert(`Starting my ${subject} homework.`);
  callback();
}

doHomework('math', function() {
  alert('Finished my homework');
});
```

https://codeburst.io/javascript-what-the-heck-is-a-callback-aba4da2deced

Promise

- A promise represents the eventual result of an asynchronous operation.

Real-life analogy

- A “producing code” that does something and takes time. For instance, the code loads a remote script. That’s a “singer”.
- A “consuming code” that wants the result of the “producing code” once it’s ready. Many functions may need that result. These are the “fans”.
- A promise is a special JavaScript object that links the “producing code” and the “consuming code” together. In terms of our analogy: this is the “subscription list”. The “producing code” takes whatever time it needs to produce the promised result, and the “promise” makes that result available to all of the subscribed code when it’s ready.

https://javascript.info/promise-basics

The constructor syntax for a promise object


```
let promise = new Promise(function(resolve, reject)
{
  // executor (the producing code, "singer")
});
```

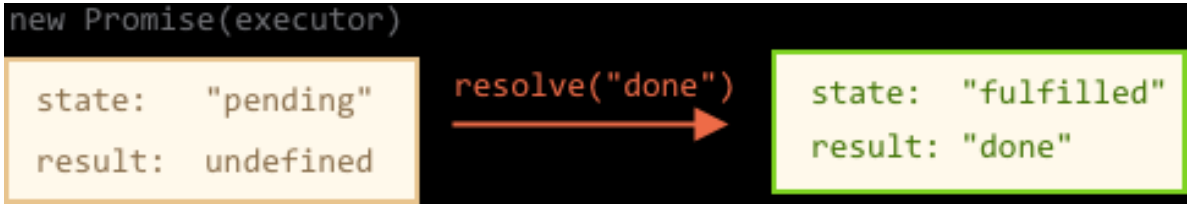
The resulting promise object has internal properties

- state — initially “pending”, then changes to either “fulfilled” or “rejected”,
- result — an arbitrary value of your choosing, initially undefined.

Example

- let promise = new Promise(function(resolve, reject) {
- // the function is executed automatically when the promise is constructed

- // after 1 second signal that the job is done with the result "done"
- setTimeout(() => resolve("done"), 1000);
- });



Consumers

- Consumers: then, catch, finally
- A Promise object serves as a link between the executor (the “producing code” or “singer”) and the consuming functions (the “fans”), which will receive the result or error. Consuming functions can be registered (subscribed) using methods .then, .catch and .finally.

- promise.then(
- function(result) { /* handle a successful result */ },
- function(error) { /* handle an error */ }
-);

Example


```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

Callback and Promise

- callback
- function loadScript(src, callback) {
- let script = document.createElement('script');
- script.src = src;

- script.onload = () => callback(null, script);
- script.onerror = () => callback(new Error(`Script load error ` + src));

- document.head.append(script);
- }

Promise


```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;


    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error("Script load error: " + src));


    document.head.append(script);
  });
}
```

Promise states

- Promises have states and at any point in time, can be in one of the following:
  - Pending : The promise's value is not yet determined and its state may transition to either fulfilled or rejected.
  - Fulfilled : The promise was fulfilled with success and now has a value that must not change. Additionally, it must not transition to any other state from the fulfilled state.
  - Rejected : The promise is returned from a failed operation and must have a reason for failure. This reason must not change and the promise must not transition to any other state from this state.

Consumer syntax

- From an API perspective, a promise is defined as an object that has a function as the value for the property then .
- The promise object has a primary then method that returns a new promise object. Its syntax will look like the following:
- then(onFulfilled, onRejected);

Promise usage


```
promise.then(function (value){
                    var result = JSON.parse(data).value;
                },
                function (reason) {
                    alert(error.message);
                });
```

Promise Example (JQuery)


```
$.getJSON('example.json').
then(JSON.parse).
then(function(response) {
alert("Hello There: ", response);
});
```

- the previous code sample does is chain the promise returned from the first then() call to the second then() call.
- Hence, the getJSON method will return a promise that contains the value of the JSON returned
- Thus, we can call a then method on it, following which we will invoke another then call on the promise returned. This promise includes the value of JSON.parse .
- Eventually, we will take that value and display it in an alert.

Promise and Callback

- Additionally, the use of anonymous inline functions in a callback can make reading the call stack very tedious.
- Also, when it comes to debugging, exceptions that are thrown back from within a deeply nested set of callbacks might not propagate properly up to the function that initiated the call within the chain, which makes it difficult to determine exactly where the error is located.
- Moreover, it is hard to structure a code that is based around callbacks as they roll out a messy code like a snowball. We will end up having something like the following code sample but on a much larger scale:

A promise comes with some guarantees

- Callbacks will never be called before the completion of the current run of the JavaScript event loop.
- Callbacks added with then() even after the success or failure of the asynchronous operation, will be called, as above.
- Multiple callbacks may be added by calling then() several times. Each callback is executed one after another, in the order in which they were inserted.
- One of the great things about using promises is chaining.

Promise and Callback


```
function readJSON(filename, callback) {
    fs.readFile(filename, function (err, result) {
        if (err) return callback(err);
        try {
            result = JSON.parse(result, function (err, result) {
                fun.readAsync(result, function (err, result) {
                    alert("I'm inside this loop now");
                    });
                alert("I'm here now");
                });
            } catch (ex) {
        return callback(ex);
        }
    callback(null, result);
    });
}
```

The sample code in the previous example is an excerpt of a deeply nested code that is sometimes referred to as the *pyramid of doom*. Such a code, when it grows, will make it a daunting task to read through, structure, maintain, and debug.

Rami Sarieddine**, JavaScript Promises Essentials**

Promise and Callback

- Simply put, the promises pattern will allow the asynchronous programming to move from the continuation-passing style that is widespread to one where the functions we call return a value, called a promise, which will represent the eventual results of that particular operation.

Promise and Callback


```
call1(function (value1) {
    call2(value1, function(value2) {
        call3(value2, function(value3) {
            call4(value3, function(value4) {
                // execute some code
            });
        });
    });
});
```

To:

```
Promise.asynCall(promisedStep1)
.then(promisedStep2)
.then(promisedStep3)
.then(promisedStep4)
.then(function (value4) {
    // execute some code
});
```

Rami Sarieddine**, JavaScript Promises Essentials**

Promises’s benefits

- It is easier to read as with the usage of cleaner method signatures
- It allows us to attach more than one callback to a single promise
- It allows for values and errors to be passed along and bubble up to the caller function
- It allows for chaining of promises

The Promise API and Its Compatibility

- Both handlers onFulfilled and onRejected that must be called asynchronously.

Another Promise Example

- function successCallback(result) {
- console.log("Audio file ready at URL: " + result);
- }

- function failureCallback(error) {
- console.log("Error generating audio file: " + error);
- }

- createAudioFileAsync(audioSettings, successCallback, failureCallback);
- …modern functions return a promise you can attach your callbacks to instead

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

```
new Promise((resolve, reject) => {
    console.log('Initial');

    resolve();
})
.then(() => {
    throw new Error('Something failed');

    console.log('Do this');
})
.catch(() => {
    console.log('Do that');
})
.then(() => {
    console.log('Do this, no matter what happened before');
});
```

Sequential composition with promise

- Sequential composition is possible using some clever JavaScript:
- [func1, func2, func3].reduce((p, f) => p.then(f), Promise.resolve())
- .then(result3 => { /* use result3 */ });
- Basically, we reduce an array of asynchronous functions down to a promise chain equivalent to:

```
Promise.resolve().then(func1)
.then(func2)
.then(func3);
```

Promise Example


```
var promiseObj = function (time) {
        return new Promise(function (resolve) {
                setTimeout(resolve, time);
                });
};
promiseObj(3000).then(function () {
        alert("promise 1");
        }).then(function () {
           alert("another promise");
        }
);
```

Promise Example

- Definition and Usage
- The setTimeout() method calls a function or evaluates an expression after a specified number of milliseconds.

Promise Example Test


```
var promiseTest = new Promise(function (resolve) {
        // JSON.parse will throw an error because of invalid JSON
        // so this indirectly rejects
        resolve(JSON.parse("json"));
});
promiseTest.then(function (data) {
                alert("It worked!" + data);
        },function (error) {
                //error handler
                        alert(" I have failed you: " + error);
});
```

Different Usage of promise

- promise.then(handler1, handler2);
  promise.then(handler1).catch(handler2);
- The above lines of code include promise , then , and catch methods with two handlers: handler1 and handler2 .
- These two calls are not equivalent
  - the first line will not call handler2 if an error occurs in handler1. That is because, if the promise is fulfilled, handler1 will be invoked, and if the promise is rejected, handler2 will be invoked. But if handler1 throws error , handler2 will not be called in return.
  - Meanwhile, in the second line, handler2 will be invoked if either promise is rejected or handler1 throws an exception. Since catch() is merely a sugarcoat for then(null, handler) , the second line is identical to the following, which can make this conundrum clearer:
    promise.then(handler1).then(null, handler2);

Questions ?