CALLBACK, PROMISE & ASYNC/AWAIT IN JAVASCRIPT

INTRODUCTION

- JavaScript is **single threaded**, meaning that two bits of script cannot run at the same time; they have to run one after another.
- In JavaScript we cannot run multiple parallel threads like a .NET/Java application would afford you.
- With JavaScript, instead of relying on threads, we rely on the ability to perform actions and continuing moving along with the execution of our code.
- On browsers, JavaScript shares a thread with a load of other stuff that differs from browser to browser.

> *"As a human being, you're multithreaded. You can type with multiple fingers, you can drive and hold a conversation at the same time. The only blocking function we have to deal with is sneezing, where all current activity must be suspended for the duration of the sneeze."*

WHAT IS A CALLBACK FUNCTION?

- We learned that JavaScript, functions are objects. Can we pass objects to functions as parameters? Yes.
- So, we can also pass functions as parameters to other functions and call them inside the outer functions.

```
function print(callback) {
    callback();
}
```

- In above example the print( ) function takes another function as a parameter and calls it inside.
- This is valid in JavaScript and we call it a “callback”.
- So a function that is passed to another function as a parameter is a callback function.

HOW TO CREATE A CALLBACK

- To understand what I've explained above, let me start with a simple example. We want to log a message to the console but it should be there after 3 seconds.

1

```
const message = function() {
    console.log("This message is shown after 3 seconds");
}

setTimeout(message, 3000);
```

OR

2

```
setTimeout(function() {
    console.log("This message is shown after 3 seconds");
}, 3000);
```

OR

3

```
setTimeout(() => {
    console.log("This message is shown after 3 seconds");
}, 3000);
```

4

```
function functionOne(x) { alert(x); }

function functionTwo(var1, callback) {
    callback(var1);
}

functionTwo(2, functionOne);
```

WHAT IS A PROMISE?

- The **Promise** object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.
- A **Promise**
  - is a proxy for a value not necessarily known when the promise is created.
  - it allows you to associate handlers with an asynchronous action's eventual success value or failure reason.
    - asynchronous methods return values like synchronous methods: instead of immediately returning the final value
    - asynchronous method returns a *promise* to supply the value at some point in the future.

- In JavaScript promises have been around for a while in the form of libraries, such as:
  - Q
  - when
  - WinJS
  - RSVP.js

QUICK EXAMPLE

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then…

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

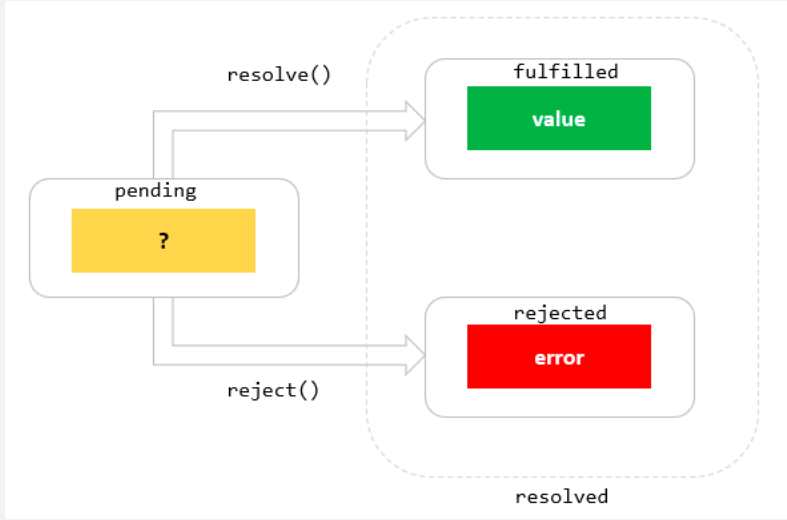
Declaration

Call

```
promise.then(function(result) {
  console.log(result); // "Stuff worked!"
}, function(err) {
  console.log(err); // Error: "It broke"
});
```

STATES OF PROMISE

- A Promise is in one of these states:
  - ***pending***: initial state, neither fulfilled nor rejected.
  - ***fulfilled***: meaning that the operation completed successfully.
  - ***rejected***: meaning that the operation failed.
  - ***settled*** - meaning that operation has fulfilled or rejected

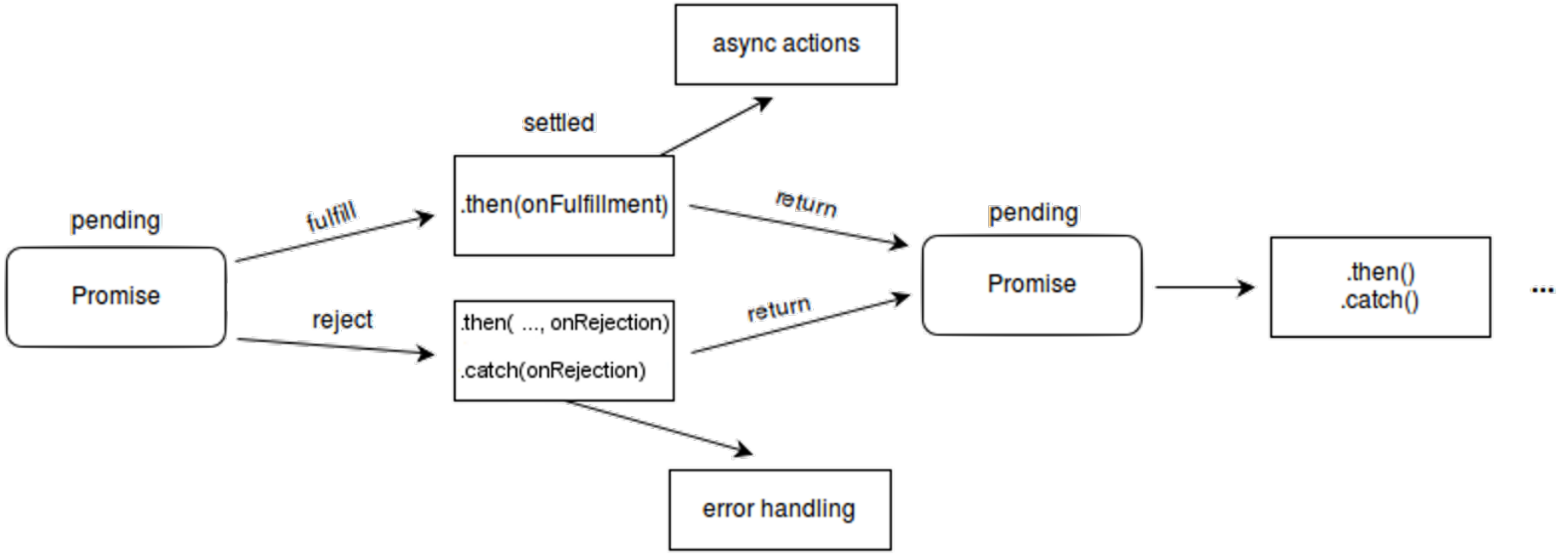


- A pending promise can either be *fulfilled* with a value or *rejected* with a reason (error).

- When either of these options happens, the associated handlers queued up by a promise's then method are called.

- If the promise has already been fulfilled or rejected when a corresponding handler is attached, the handler will be called, so there is no race condition between an asynchronous operation completing and its handlers being attached.

- As the Promise.prototype.then() and Promise.prototype.catch() methods return promises, they can be chained.


async actions
settled
pending
fulfill
.then(onFulfillment)
return
pending
Promise
Promise
.then()
.catch()
...
reject
.then(..., onRejection)
.catch(onRejection)
return
error handling

EXAMPLE - PROMISE

```
promiseObject.then(onFulfilled, onRejected);
```

```
function makePromise(completed) {
    return new Promise(function (resolve, reject) {
        setTimeout(() => {
            if (completed) {
                resolve("I have completed learning JS.");
            } else {
                reject("I haven't completed learning JS yet.");
            }
        }, 3 * 1000);
    });
}
```

```
let learnJS = makePromise(true);

learnJS.then(
    success => console.log(success),
    reason => console.log(reason)
);
```

CHAINED PROMISES

- The methods promise.then(), promise.catch(), and promise.finally() are used to associate further action with a promise that becomes settled. These methods also return a newly generated promise object, which can optionally be used for chaining; for example, like this:

```
const myPromise =
        (new Promise(myExecutorFunc))
         .then(handleFulfilledA, handleRejectedA)
         .then(handleFulfilledB, handleRejectedB)
         .then(handleFulfilledC, handleRejectedC);

// or, perhaps better ...
const myPromise =
         (new Promise(myExecutorFunc))
        .then(handleFulfilledA)
        .then(handleFulfilledB)
        .then(handleFulfilledC)
        .catch(handleRejectedAny);
```

SUMMERY

- A promise is an object that returns a value in the future.
- A promise starts in the pending state and ends in either **fulfilled** state or **rejected** state.
- Use **then()** method to schedule a callback to be executed when the promise is fullfiled, and **catch()** method to schedule a callback to be invoked when the promise is rejected.
- Place the code that you want to execute in the **finally()** method whether the promise is fulfilled or rejected.

CALLBACK-BASED PATTERN:

Promises	Callbacks
Promises allow us to do things in the natural order. First, we run `loadScript(script)`, and `.then` we write what to do with the result.	We must have a `callback` function at our disposal when calling `loadScript(script, callback)`. In other words, we must know what to do with the result *before* `loadScript` is called.
We can call `.then` on a Promise as many times as we want. Each time, we're adding a new "fan", a new subscribing function, to the "subscription list". More about this in the next chapter: Promises chaining.	There can be only one callback.

ASYNC AND AWAIT

- Async/Await is just syntactical sugar for promises.
- They're a bit more declarative and easier to read but at the end of the day, they return a Promise object just the same.
- Understanding promises, means you already understand Async/Await.

QUICK EXAMPLE

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: 'resolved'
}

asyncCall();
```

ASYNC KEYWORD

- You use the async keyword with a function to represent that the function is an asynchronous function. The async function returns a promise.

- The syntax of async function is:

```
async function name(parameter1, parameter2, ...paramaterN)
{
    // statements
}
```

Here,

- **name** - name of the function
- **parameters** - parameters that are passed to the function

-

EXAMPLE - ASYNC

```
// async function example

async function f() {
    console.log('Async function.');
    return Promise.resolve(1);
}

f();
```

Output

```
Async function.
```

```
async function f() {
    console.log('Async function.');
    return Promise.resolve(1);
}

f().then(function(result) {
    console.log(result)
});
```

Output

```
Async function
1
```

AWAIT KEYWORD

- The await keyword is used inside the async function to wait for the asynchronous operation.
- The syntax to use await is:

```
let result = await promise;
```

- The use of await pauses the async function until the promise returns a result(resolve or reject) value.

EXAMPLE - AWAIT

```
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {

    // wait until the promise resolves
    let result = await promise;

    console.log(result);
    console.log('hello');
}

// calling the async function
asyncFunc();
```

Output

```
Promise resolved
hello
```

ERROR HANDLING

```
// a promise
let promise = new Promise(function (resolve, reject) {
    setTimeout(function () {
    resolve('Promise resolved')}, 4000);
});

// async function
async function asyncFunc() {
    try {
        // wait until the promise resolves
        let result = await promise;

        console.log(result);
    }
    catch(error) {
        console.log(error);
    }
}

// calling the async function
asyncFunc(); // Promise resolved
```

BENEFITS OF USING ASYNC FUNCTION

- The code is more readable than using a callback or a promise.
- Error handling is simpler.
- Debugging is easier.

THANK YOU