**Promises**

Q: Why Promises over Callback?

A:

1. Error handling: If there’s error in the code, Callback will still run without letting you know that there’s an error
2. You need to stack callbacks within callbacks if those callbacks are asynchronous. Will be very hard to debug

Q: Promises phrases:

A: Fulfilled (resolved): Action related to promises succeeded

Rejected: Failed

Pending: promises has not been fulfilled or rejected

Settled: promises has been fulfilled/ rejected

Promises constructors:

Q: new Promise(function(resolve, reject) {

resolve(‘hi’); // it works

resolve(‘bye’); // it doesn’t happen again

});

resolve only happens once

Promises is used to try catch wrappers around asynchronous work, not for long running operations (meaning it’s there to catch errors), not for synchronous long running operations becuz it can still block the browsers as different asynchronous events fire off

+) Used for AJAX (asynchronous work)

+) used for web workers because they are executing long, indepdent JS off the main thread

Example of Promise:

Ex:

new Promise(function(resolve, reject) {

var value = doSomething();

if (thingWorked){

resolve(value);

console.log(‘done’);

} else if (somethingWentWrong){

reject();

}

}).then(function(value){

//success!

return nextThing(value);

}).catch(rejectFunction)

// the function doesn’t stop after resolve. In fact, the “console.log(done)” will also execute

//if it’s successful (resolve) -> go to .then() function

// if not, go to “catch()”, but also if there’s any error in the code, it’ll go to catch too

// usually then and catch

// whatever value I pass into resolve() and reject() will be passed to the function called by .then and .catch

Q: What do people mean why they say “wrapping a function inside a Promise”?

A:

Example with setTimeOut:

function wait(ms){

**return** new Promise(function(resolve){

console.log(this);

window.setTimeOut(function(){

**resolve();**

},ms);

})

});

var milliseconds = 2000;

wait(milliseconds).then(finish);

function finish() {

var completion = document.querySelector('.completion');

completion.innerHTML = "Complete after " + milliseconds + "ms.";

};

means placing setTimeOut right inside Promise. A Promise comes with resolve & reject (optional but one has to be there)

The function has to return something

ALSO, resolve (in red) is called after the number of milliseconds

Q: Why is the scope of “this” the window object?

A: 1) if the code isn’t in strict mode, the value of “this” is not set by the call, so it will default to the global object

1. If it’s in strict mode (‘use strict’), then “this” is undefined

Q: What’s Node vs executing Javascript inside the browser?

A: Just different engine. So Node and Chrome use V8 engine developed by Google

Mozilla and Safari might use different engines

Vary in execution time between browsers and engines

Q: How does this code work?

Function ready() {

return new Promise(function(resolve) {

function checkState() {

if (document.readyState !== 'loading') {

resolve();

}

}

document.addEventListener('readystatechange', checkState);

checkState();

});

};

ready().then(wrapperResolved);

function wrapperResolved() {

var completion = document.querySelector('.completion');

completion.innerHTML = "Resolved!";

};

A: +) document.addEventListender takes an event and a callback.

+) ‘readystatechange’ is the name of that state’s event

+) checkState as callback checks when that event is fired

+) checkState() also checks the readystate immediately so if the state changes before creating the promise, it will still catch it

+) resolve just means that as soon as that state’s fired, mark complete

+) then(name of function) just execute that function

Q: To cancel operation Github?

A: ctrl +x

Q: More example:

A:

function get(url) {

/\*

This code needs to get wrapped in a Promise!

\*/

return new Promise(function(resolve, reject){

var req = new XMLHttpRequest();

req.open('GET', url);

req.onload = function() {

if (req.status === 200) {

// It worked!

// You'll want to resolve with the data from req.response

resolve(req.response);

} else {

// It failed :(

// Be nice and reject with req.statusText

reject(Error(req.statusText));

// reject(req.statusText);

}

};

req.onerror = function() {

// It failed :(

// Pass a 'Network Error' to reject

reject(Error('Network Error'));

};

req.send();

})

}

=>reject(Error()) to indicate it’s an error

=> Don’t pass anything in with function()

This is to test:

get('../data/earth-like-results.json')

.then((response) => {

addSearchHeader(response);

})

.catch((error) =>{

addSearchHeader('unknown');

console.log(error);

});

//so if it’s successful, then it get passed to .then()

// if not, passed to .catch

//catch always goes with (error) : catch((error =>{})

Q: .then(resolveFunc).catch(rejectFunc) is the only way to catch error?

A: No. There’s also other ways like:

Get(‘example.json’).then(resolveFunc).then(undefined, rejectFunc);

But it’s confusing

Q: Basic fetch usage?

A:

// url (required), options (optional)

fetch('https://davidwalsh.name/some/url', {

method: 'get'

}).then(function(response) {

}).catch(function(err) {

// Error :(

})

// “get” as opposed to “post”

* Fetch(url of somewhere to “fetch” from, callbackFunction). In this case that callback function contains ‘get’. The .then and .catch are optional

Q: Also

A: // Chaining for more "advanced" handling

fetch('https://davidwalsh.name/some/url').then(function(response) {

return //...

}).then(function(returnedValue) {

// ...

}).catch(function(err) {

// Error :(

});