Asynchronous Programming

# Real life example of Asynchronous Vs Synchronous

**SYNCHRONOUS**

You are in a queue to get a movie ticket. You cannot get one until everybody in front of you gets one, and the same applies to the people queued behind you.

**ASYNCHRONOUS**

You are in a restaurant with many other people. You order your food. Other people can also order their food, they don't have to wait for your food to be cooked and served to you before they can order. In the kitchen restaurant workers are continuously cooking, serving, and taking orders. People will get their food served as soon as it is cooked.

# Showing Asynchronous programming in JavaScript

console.log("first");

console.log("second");

Easy right? The code runs in the order we write it, no problems.

Asynchronous programming looks like code running in a different order to which we write it.

setTimeout(() => console.log("first"), 1000);

console.log("second");

This is an easy way to force asynchronous programming for an example, in this example “second” would be printed out before “first”.

Now realistically we probably want our code to run the first way, when the code is executed in the order that we write it. A solution could be:

setTimeout(() => {

console.log("first");

console.log("second");

}, 1000)

A lot of times this might be ok, however we may need to make lots of requests.

We might want to make an asynchronous call to a database to extract some information – and then we work with the data when we get it back by making an API call with it, and then based on that response we need to make another API call...

setTimeout(() => {

console.log("first");

setTimeout(() => {

console.log("second");

setTimeout(() => {

console.log("third");

// ...

}, 1000);

}, 1000);

}, 1000);

Hopefully you can appreciate this is an unsustainable way of writing code

* Ugly
* Hard to reason
* Hard to maintain

# Callbacks

* Callbacks are the basic way of handling Async execution
* A callback is a function that is to be executed after another function has finished executing — hence the name ‘call back’.
* Callbacks are founded on the functional programming paradigm, particularly Higher Order Functions
  + *A function that has a function as an argument*
* Passing what you want the async operation to do after it’s done.
* Example – Making a web request to the website, however we have to wait for the response, if we paused execution until that happened it’s just wasted execution time, we could do other things while we wait by using it in an async way!

function example(callback) {

webRequest(); //pretend method

callback();

}

function log() {

console.log("Done");

}

example(log);//Execution

//Opposed to

webRequest();

console.log("Done");

**Another Example of Asynchronous Programming**

var fs = require("fs");

fs.readFile('input.txt', (err, data) => {

if (err) {

return console.error(err);

}

console.log(data.toString());

});

console.log("Program Ended");

In the example above the function we are passing in as our callback is an anonymous function, we have also used the ES6 syntax to describe our function.

# Promises

Promises are a better way of handling asynchronous programming.

A promise is a value that represents something that will happen – it is a proxy value

We can then call methods on this promise and say:

* When this thing happens
* Do something else

We will have a function that returns a Promise object, for this example don’t worry about how a Promise is created, its working with the Promise that is important.

waitASecond = () => {

return new Promise( resolve => setTimeout(() => resolve(), 1000));

}

waitASecond()

.then(() => console.log("first"));

So in the example above the waitASecond function does something that takes 1000ms, this could represent retrieincg data from a databse or an API Call.

As this function returns a promise, then when the promise is ‘resolved’ the function after .then() is executed.

A great feature of these .then() ‘s is that we can chain them together!

This is because .then() returns a new promise that resolves with whatever was returned by the function that we ran in the body of our .then()

waitASecond()

.then(() => console.log("first"))

.then(() => console.log("second"));

We don’t have to write all our code in one function body, we can also chain together promises – all of this allows us to escape callback hell

waitASecond()

.then(() => console.log("first"))

.then(waitASecond)

.then(() => console.log("second"));

We can create these async chains getting and resolving data and allows us to write it in this style of

* just do this
* then this
* then this

Without resorting to callback hell and nesting functions within one another.

Ok so whats next

How do we do something with what our promise resolves with?

If a promise gives us something back, can I do something with it?

function resolvesWithOne() {

return new Promise((resolve) => resolve(1));

}

waitASecond()

.then(() => console.log("first"))

.then(waitASecond)

.then(resolvesWithOne)

.then((result) => console.log(result + 1));

resolvesWithOne immediately resolves with the value 1 – again don’t worry how this works – it just does.

Result is what the previous promise resolved with – we are then using result in the following .then()

waitASecond()

.then(() => console.log("first"))

.then(waitASecond)

.then(resolvesWithOne)

.then((result) => result + 1)

.then((result) => console.log(result));

Still chaining

Often very useful when we have these long promise chains to have some logging to see how the process is unfolding.

# More examples of Promises

const aCondition = true;

const createPromise = new Promise(

function (resolve, reject) {

if (aCondition) {

const user = {

name: 'Bert',

email: 'bert@gmail.com'

};

resolve(user);

} else {

const reason = new Error('Rejected');

reject(reason);

}

}

);

const consumePromise = function () {

createPromise

.then((user) => {

console.log(user);

})

.catch(function (error) {

console.log(error.message);

});

};

consumePromise();