



# ASYNCHRONOUS CODE COURSE

APRIL 2022

YONATAN BENEZRA

- DO NOT DISTRIBUTE -

- DO NOT COPY – ALL RIGHTS RESERVED TO THE  
AUTHOR –

## Callbacks, Promises, and Async

- Synchronous operations in JavaScript entail having each step of an operation wait for the previous step to execute completely.  
This means no matter how long a previous process takes, the subsequent process won't kick off until the previous one is completed.
- Asynchronous operations, on the other hand, are deferred operations.  
Any process that takes a lot of time to process is usually run alongside other synchronous operations and will be completed in the future.

Asynchronous Operations in JavaScript are traditionally synchronous and execute from top to bottom.

For instance, a farming operation that logs the farming process to the console:

```
// JAVASCRIPT

console.log("Plant corn");
console.log("Water plant");
console.log("Add fertilizer");
```

If we run the above code, we will have the following logged in the console:

```
// BASH

Plant corn
Water plant
Add fertilizer
```

Now, let's change that a bit so that watering the farm takes longer than planting and fertilizing:

```
// JAVASCRIPT

console.log("Plant maize");

setTimeout(function() {
  console.log("Water plant")
}, 3000);

console.log("Add fertilizer");
```

We will receive the following in the console:

```
// BASH  
  
Plant Maize  
Add fertilizer  
Water plant
```

### Why?

The `setTimeout` function makes the operation asynchronous by deferring plant watering to occur after 3 seconds.

The entire operation doesn't pause for 3 seconds in order to allow it to log "Water plant".

Rather, the system proceeds to apply fertilizers, and only then completes the "water plant" after 3 seconds.

## Functions

Functions are First-Class Objects. Before going through the rest of these lessons, it's important to keep in mind that JavaScript Functions are first-class objects. Functions have the ability to:

- Be assigned to variables (and treated as values)
- Have other functions within them
- Return other functions to be called later

## Callback Functions

When a function simply accepts another function as an argument, this contained function is known as a callback function.

Using callback functions is a core functional programming concept, and you can find them in most JavaScript code.

They can be found in simple functions, such as `setInterval`, in event listening, or when making API calls. Callback functions are written like so:

```
setInterval(function() {  
  console.log('hello!');  
}, 1000);
```

`setInterval` accepts a callback function as its first parameter, and also a time interval.

Another example using .map():

```
const list    = ['man', 'woman', 'child']

// create a new array
// loop over the array and map the data to new content
const newList = list.map(function(val) {
  return val + " kind";
});

// newList = ['man kind', 'woman kind', 'child kind']
```

In the example above, we used the .map() method to iterate through the array list. The method accepts a callback function which states how each element of the array will be manipulated. Callback functions can accept arguments as well.

## Naming Callback Functions

Callback functions can be named, or they can be anonymous functions. In our first examples, we used anonymous callback functions.

Let's look at an example of a named callback function as well:

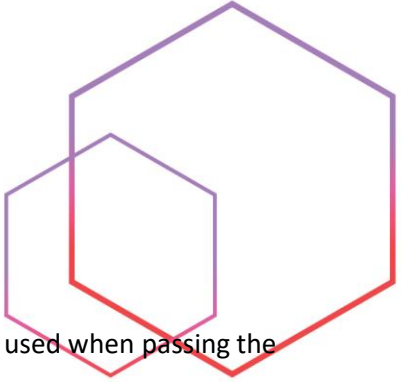
```
function greeting(name) {
  console.log(`Hello ${name}, welcome to Cyberpro-
israel!`);
}
```

The above function is assigned a name greeting and has an argument of name. Let's use this function as a callback function.

```
function introduction(firstName, lastName, callback) {
  const fullName = `${firstName} ${lastName}`;

  callback(fullName);
}

introduction('Koby', 'Naaman', greeting); // Hello Koby
Naaman, welcome to Cyberpro-israel!
```



Notice the usage of the callback? The next brackets (), after the function, are not used when passing the function as a parameter.

**\*Note:**

The callback function is not run unless called by its containing function, in which case it is called back. That is why it is called a call back function.

Multiple functions can be created independently and used as callback functions, which would create multi-level functions.

When this function tree becomes too large, the code sometimes becomes incomprehensible and is not easily refactored.

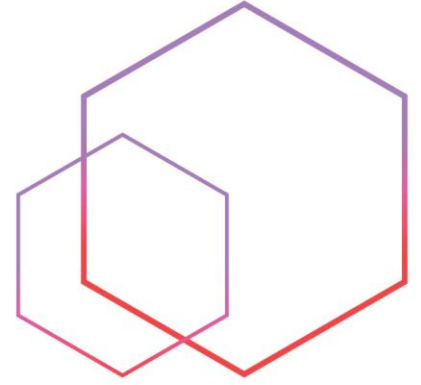
This is known as callback hell. Let's see an example:

```
// a bunch of functions are defined up here  
// let's use our functions in callback hell  
function setInfo(name) {  
  address(myAddress) {  
    officeAddress(myOfficeAddress) {  
      telephoneNumber(myTelephoneNumber) {  
        nextOfKin(myNextOfKin) {  
          console.log('done');  
//let's begin to close each function!  
        };  
      };  
    };  
  };  
};  
}
```

We are assuming these functions have been previously defined elsewhere. You can see how confusing it becomes to pass each function as a callback.

Callback functions are useful for short asynchronous operations. When working with large sets, this is not considered best practice.

Because of this challenge, Promises were introduced to simplify deferred activities.



## Promises

I promise to do this whenever that is true. If it isn't true, then I won't.

The above is a simple illustration of JavaScript Promises.  
It sounds very similar to an IF statement.

A promise is used to handle the asynchronous result of an operation.

JavaScript is designed not to wait for an asynchronous block of code to completely execute before other synchronous parts of the code can run.

For instance, when making API requests to servers, we have no idea if these servers are offline or online, or how long it will take to process the server request.

With Promises, we can defer executing a code block until an async request is completed.  
This way, other operations can keep running without interruption.

Promises have three states:

- Pending - The initial state of the Promise before an operation begins
- Fulfilled - The specified operation was completed
- Rejected - The operation was not completed; an error value is usually thrown

## Creating a Promise

The Promise object is created using the “new” keyword, and it contains the promise.

This is an executor function that has a resolve and a reject callback.

As the names imply, each of these callbacks returns a value, with the reject callback returning an error object.

```
// JAVASCRIPT

const promise = new Promise(function(resolve, reject) {
  // promise description
})
```

Let's create a promise:

```
// JAVASCRIPT

const season = true
const PlantStrawberries= new Promise(function(resolve,
reject) {
  if (season) {
    const plantDetails = {
```

```

    name:      'Home',
    location:  'Arlozorov 194',
    apartment: 2
  };

  resolve(plantDetails)
} else {
  reject(new Error('Bad season, so no planting'))
}
});

```

If season is true, resolve the promise returning the planting plantDetails.  
Else, return an error object with the data Bad season, so no planting.

## Using Promises

Using a promise that has been created is relatively straightforward.  
We chain .then() and .catch() to our Promise, as follows:

```

// JAVASCRIPT

PlantStrawberries
  .then(function(done) {
    // the content from the resolve() is here
  })
  .catch(function(error) {
    // the info from the reject() is here
  });

```

Using the promise we created above, let's take this a step further:

```

// JAVASCRIPT

const myPlant = function() {
  PlantStrawberries
    .then(function(done) {
      console.log('We are going to plant!')
      console.log(done)
    })
    .catch(function(error) {
      console.log(error.message)
    })
}

```

```
}  
  
myPlant();
```

Since the season value is true, we call myPlant(), and our console logs read:

```
We are going to plant!  
{  
  name: 'Home',  
  location: 'Arlozorov 194',  
  apartment: 2  
}
```

.then() receives a function with an argument which is the resolve value of our promise.  
.catch returns the reject value of our promise.

**\*Note:**

Promises are asynchronous.

Promises in functions are placed in a micro-task queue and run when other synchronous operations complete.

## Chaining Promises

Sometimes, we may need to execute two or more asynchronous operations based on the result of preceding promises.

In such cases, promises can be chained.

Using our created promise once again, let's order a taxi if we are going to plant.

To do so, we will create an additional promise:

```
// JAVASCRIPT  
  
const orderTaxi = function(plantDetails) {  
  return new Promise(function(resolve, reject) {  
    const message = `Get me a taxi ASAP to  
    ${plantDetails.location}, we are going to plant!`;  
  
    resolve(message)  
  });  
}
```

This promise can be shortened to:



```
// JAVASCRIPT
```

```
const orderTaxi = function(plantDetails) {  
  const message = `Get me a taxi ASAP to  
  ${plantDetails.location}, we are going to plant!`;   
  return Promise.resolve(message)  
}
```

We then chain this promise to our earlier Plant operation, as follows:

```
// JAVASCRIPT
```

```
const myPlant= function() {  
  plantStrawberries  
    .then(oderTaxi)  
    .then(function(done) {  
      console.log(done);  
    })  
    .catch(function(error) {  
      console.log(error.message)  
    })  
}  
  
myPlant();
```

Once the orderTaxi promise is chained with .then, the subsequent .then utilizes data from the previous one.

## Async and Await

An async function is a modification of the syntax used to write promises.

It makes writing promises easier.

An async function returns a promise -- if the function returns a value, the promise will be resolved with the value, but if the async function triggers an error, the promise is rejected with that value.

Let's see an example of an async function:

```
// JAVASCRIPT
```

```
async function myRide() {  
  return '2017 Dodge Charger';  
}
```

Another example of a function that achieves the same goal, only in promise format:

```
// JAVASCRIPT
```

```
function yourRide() {  
  return Promise.resolve('2017 Dodge Charger');  
}
```

From the above statements, myRide() and yourRide() are equal and will both resolve to "2017 Dodge Charger". Also, when a promise is rejected, an async function is represented as follows:

```
// JAVASCRIPT
```

```
function foo() {  
  return Promise.reject(25)  
}
```

```
// is equal to  
async function() {  
  throw 25;  
}
```

## Await

Await is only used with an async function.

The await keyword is used in an async function to ensure that all promises returned in the async function are synchronized, meaning, they wait for each other.

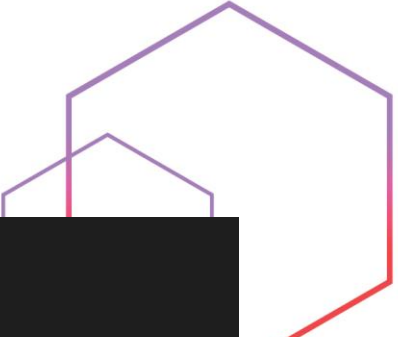
Await eliminates the use of callbacks in .then() and .catch().

In using async and await, async is prepended when returning a promise, and await is prepended when calling a promise.

“try” and “catch” are also used to get the rejection value of an async function. Let's see how this works with our plant example:

```
// JAVASCRIPT
```

```
async function myPlant() {  
  try {  
  
    let plantDetails = await plantStrawberries;  
    let message = await orderTaxi(plantDetails);  
  }  
}
```



```
    console.log(message);  
  } catch(error) {  
    console.log(error.message);  
  }  
}
```

Finally, we will call our async function:

```
// JAVASCRIPT  
  
(async () => {  
  await myPlant();  
})();
```

This is an arrow function syntax.

## Conclusion

Understanding the concepts of Callbacks, Promises, and Async/Await can be rather confusing. So far, we have seen how they work when carrying out asynchronous operations in JavaScript.

They will often come in handy when making API requests and during event handling.

You can learn more about using [promises here](#) and [async functions here](#).

