Remember to always appropriately use const, var, and let.

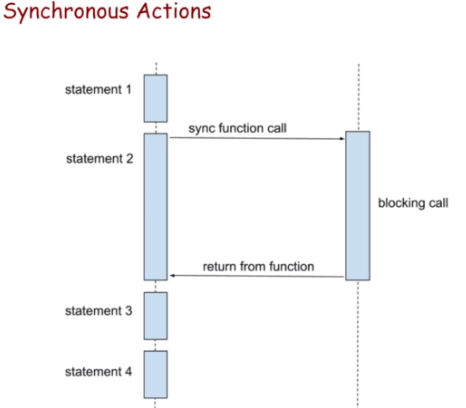
# Section 2: Differences between synchronous and asynchronous actions

* 3: Typical example of an asynchronous action

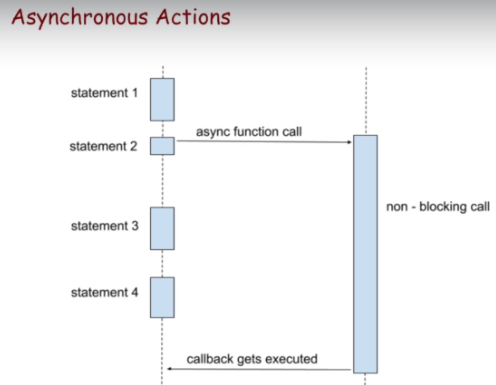
Synchronous code is executed line-by-line and is very easy to track. Asynchronous code can be executed out of order. Encoding of utf-8 means that our file contains human-readable text.

A callback is called when the function finishes executing its original code. It will take the outputs of the original function.

* 4: Synchronous vs Asynchronous actions

 Statement 3 and 4 wait until statement 2 finishes.

This means that lines of code will wait until a function is finished. This makes the function “blocking code.”

 Statement 2 doesn’t yield a blocking call. Statements 3 and 4 are waiting for the asynchronous function to start executing but not until it finishes.

Notice that we actually pass the callback to the function and when the non-blocking call finishes it gets executed. Typical examples are HTTP requests, interacting with a database, etc…

* 6: Event Loop in JavaScript. Call Stack and Message Queue

Almost every browser has its own JavaScript engine.

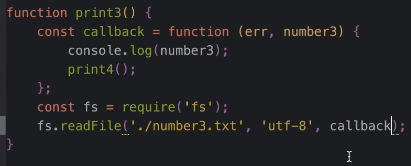
It uses two important data structures:

1. Call stack
2. When a function starts execution, a new execution context is created and placed on the call stack.
3. When the function finishes, its information is popped or removed from the call stack.
4. Message queue
5. Contains a list of tasks to be processed. Each task refers to a function that needs to be executed eventually.
6. JavaScript engine constantly checks if there are tasks in the queue. If there are some, it takes some and tries to execute them. The task that checks if there are any new processes in the message queue is called the event loop.
7. Each task in the queue is processed completely before another task can be processed.
8. Global execution context is added to call stack as first item.
9. The engine eventually comes to the line with print1(); and looks for the function and finds print1() and adds it to the call stack. The execution context is its arguments and global variables and such.
10. When it finishes print1(), it removes it from the call stack.
11. For print 2, it creates an execution context and adds it to the call stack.
12. It also adds getNumber2’s execution context in the call stack. After it returns, it gets removed.
13. When the program returns from print2, print2’s context is removed from the call stack.
14. When we enter the print3 function we add its execution context to the message queue. But readFile is an asynchronous function so we don’t wait until its done… We remove it from the call stack even though it’s still running.
15. We add print4’s execution context to the message queue and work inside of it. But, at this moment, readFile just finished and a reference to its callback is added to the message queue. The callback won’t be invoked until the call stack is empty.
16. Print4 is finished and its execution context is removed from the call stack. SO is the global execution context.
17. Finally, once the call stack is empty, the JavaScript engine checks if there is something in the message queue. There is, so it creates a new execution context for this task which is the inside of the callback. When returned from print3, the execution context is removed from the call stack.
18. The JavaScript engine checks if there is something in the call stack, and then the message queue because there was nothing there. Because there is nothing new, the JavaScript engine exits.

The reason why ‘3’ goes last isn’t because readFile takes a long time (it wouldn’t matter synchronously), it’s because readFile is asynchronous and cannot run from the message queue until every other execution context is finished from the call stack.

When executing this code, you’ll get 1 2 4 3. If you want to fix this, put print4 inside the callback.

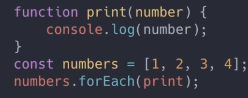
You can make a callback a function by calling it a constant inside of the function you’re about to use it in.



# Section 3: Callback Function in JavaScript

A callback is any function that is passed as an argument to another function, and then invoked from within that original function.

Functions are objects.

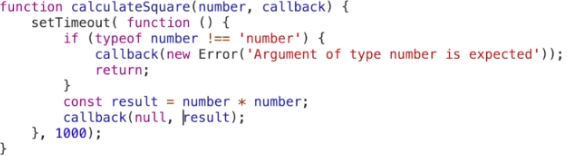
 We don’t invoke print right away, because there aren’t parentheses. It will be invoked by the forEach method. It will be invoked for times: once for each number in the array.

Keep in mind that a callback is not necessarily asynchronous.

Try/catch cannot be used to throw an error for an asynchronous function. The reason why is because the try/catch surrounding the asynchronous call will have already been executed. Errors can instead be handled through arguments of the callback.

Typically a first argument in the callback will be “error”. If something goes wrong in an asynchronous function, then the callback gets called with the first argument specifying what has happened. If there is no error, it will return “error” as null. This is known as an Error First callback.

To do this, you need to callback an error as a return to the function.



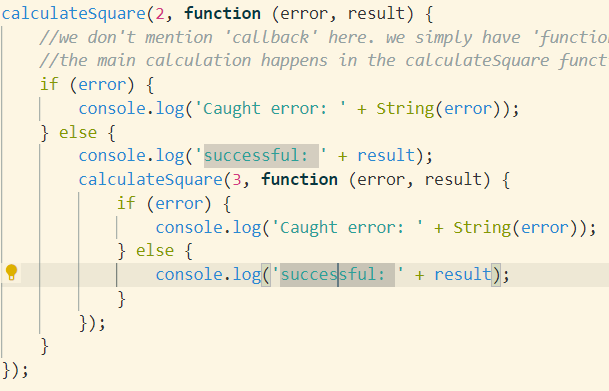
Some pros of callbacks:

1. It’s simple.
2. Popular.

Some cons of callbacks:

1. It’s sometimes hard to understand the code.
2. Callback hell: several callbacks that depend on each other.

Callbacks in callbacks is hard to read. Use promises instead.



If you want to treat asynchronous calls synchronously, put the second function inside of the callback of the first function. People created promises because of callback hell.

# Section 4: Testing Callback Functions

You can setup a test environment with Node, Mocha and Chai (for comparing results with expected outcomes)

1. Install mocha

It’s a JavaScript test framework on Node.js and in the browser. ‘npm I mocha’

1. Install Chai

A BDD/TDD assertion library for node and the browser. It can be used with any JavaScript testing framework.

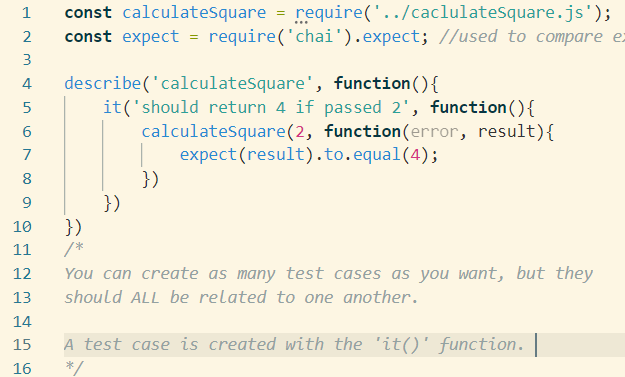
1. Finish setup

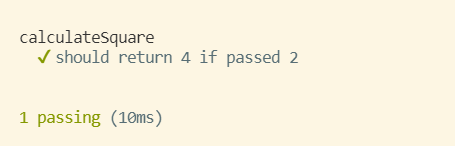
Replace the ‘test’ script value with “mocha”

1. Create a “test” folder

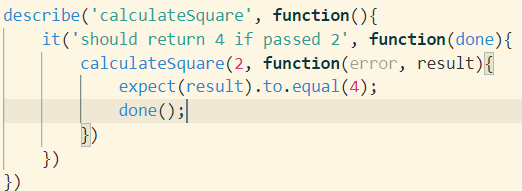
This is where all tests will be run when “mocha” script is ran.

…





The problem is that it’s testing it as though it were synchronous. We can test an asynchronous function by passing an argument to the function that contains our test. Call the ‘done’ function when our task is finished. Execute the done function after all expectations are checked. Mocha should understand that an asynchronous function is being called and will wait.



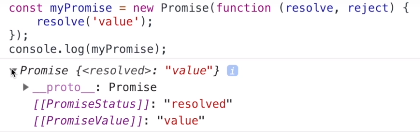
# Section 5: Promise in JavaScript

A promise is a special JavaScript object that represents an eventual result of an asynchronous action. A promise is a proxy for a value that we don’t have yet.

For loading a file, we have an asynchronous problem. We can pass a callback, which will return a value when the loading is finished. Or, we can return a promise, which represents the result of the function… It’s a substitute of the value, and can be used right away even though the value isn’t ready yet.

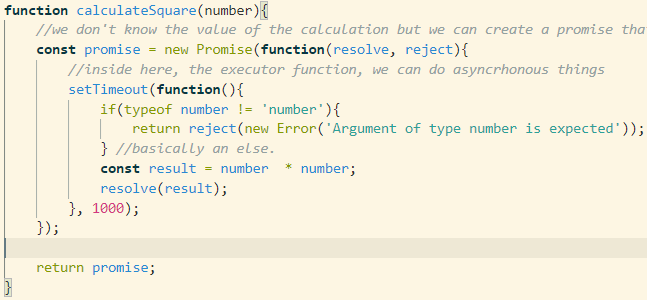
A Promise consists of a PromiseStatus and a PromiseValue (should eventually contain the real value of the promise). A promise can have the state of ‘pending’ (neither fulfilled nor rejected.. PromiseValue will be undefined), which can then either become ‘fulfilled’ (success, value contains a real value) or ‘rejected’ (failure, value will contain error message).

A promise has one necessary argument called an executor, which is a function that has two arguments: resolve and reject (these are functions too).

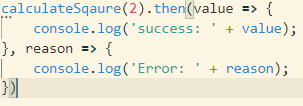


If you can resolve/reject more than once in the same promise… Fulfilled state cannot be changed. Reject cannot be resolved or be changed.

To actually use the PromiseStatus/Value (they are private), use the ‘Then’ method. It has two arguments that are used to determine what should be done if fulfilled or rejected.



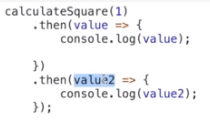
Make sure to return a reject.

 The two functions onFulfilled and onRejected.

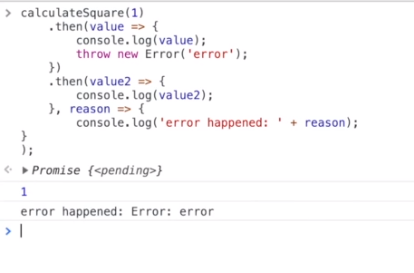
Promisifying a function is just refactoring it so that it returns a promise. For capitalization, resolve if setting the first letter to uppercase is successful but reject if the input isn’t a string. Make sure to set an if statement with reject first so the execution basically acts as an if/else for rejection/resolve.

1. Create a promise and return it.
2. Create the resolve/reject executor function
3. In the executor, create a case for the reject and make sure to return the reject.
4. Finally, return a resolve.
5. When calling the function, use a ‘.then’ and handle the case for the resolve, reject.

You can also chain promises. ‘value’ can be piped into ‘value2’ if you return it.

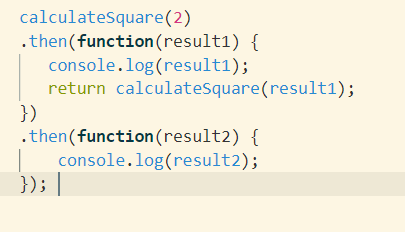


Though, to pipe it to the next promise, you have to return it in the promiseValue.

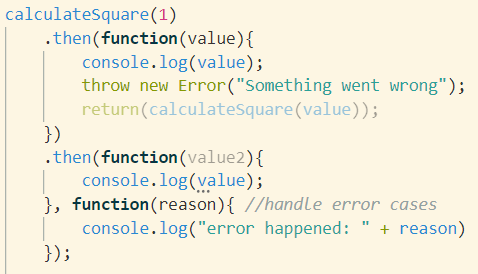


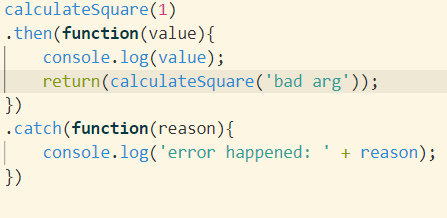
You can also throw errors and catch them with the ‘reason’ clause.

This is how we can avoid callback hell with promises: we can very clearly explain and read what is happening:

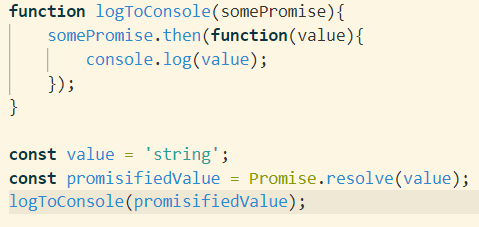


In cases where we want to handle errors, we can simply use .then(function(value)). We have to add ‘, reason’ arrow function after the brackets of initial ‘value’ arrow function. We can also do catch.





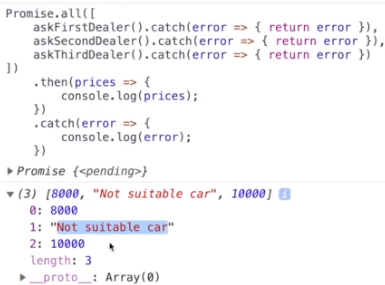
You can turn any JavaScript value into a promise. This is called ‘Promise.resolve(anyValue);’



You can also create a rejected promise as a value…

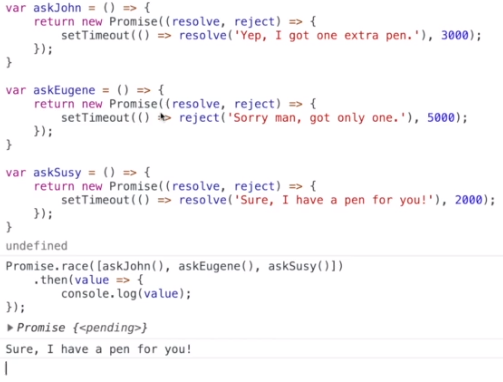
A very practical use of promises comes with calling multiple promises at the same time. Multiple different HTTP requests at the same time. ‘Promise.all([array of functions that use promises]).then(values =>’ is the format. Values is an array of the returned promiseValues. The array is returned when all values are loaded. If the elements in the array are non-promises, it simply returns them in the values.

If you add a .catch block to the Promise.all function, then if there is a single error it will ignore all other values. Having catch blocks in the list of promises is possible because the catch block itself returns a promise.



This lets you get the value of each individual promise even if there is an error. However, if one of the promises are itself already a rejection, the Promise.all will ignore everything and go straight to the catch block.

It is possible to execute promises in parallel but only get results from the fastest one. Use Promise.race()

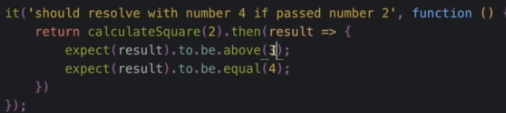


# Section 6: Testing promises in JavaScript

When we tested callbacks, we only needed Node.js, Mocha, Chai. With promises, we need to include Chai As Promised.

Mocha will timeout after 2 seconds have passed, but we can set the time by using ‘this.timeout(timeInMs)’. You have to specify a timeout for each it() case .

You can’t have to expectations at once normally. However, you can make an arrow function for multiple expectations.



# Section 7: Async Await in JavaScript

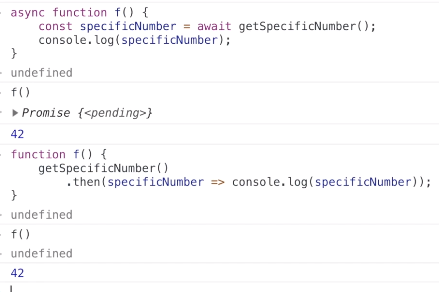
Async//await allows an asynchronous function to be structured in a way similar to an ordinary synchronous function. Async is just put infront of the function tag.

Async functions always returns a promise. If not returning a promise, a promise is wrapped around the return value. The purpose of async is to define that a function will return a promise.

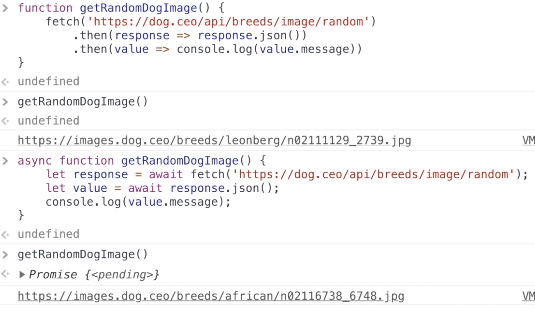


**The await keyword will wait until a promise is resolved or rejected. Using await in an async function means that the specific function that contains the await keyword will stop until a promise either resolves or rejects.** If you’re trying to generate a graph that needs data, in the graphing async function, you must await for the data to be loaded before continuing. Although the graphing function itself may not be asynchronous, it needs to use async because it will contain await.

Of course, you should be able to do the same thing no matter what asynchronous methodology you use. This example shows how you can get the same output whether you use plain promises or async/await…



People like async/await because it’s more readable. Up to you.



Async/await won’t work with node.js… You need to make your JS file of a “.mjs” extension…

You can use a try/catch block to catch errors as if they were normal functions. We can also still use .catch.

Instead of synchronously trying to complete async functions with this image’s format, invoke them in parallel with the next image’s format.

