**Introduction**

Often in web development, we need to handle asynchronous actions— actions we can wait on while moving on to other tasks. We make requests to networks, databases, or any number of similar operations. JavaScript is non-blocking: instead of stopping the execution of code while it waits, JavaScript uses an [event-loop](https://youtu.be/8aGhZQkoFbQ) which allows it to efficiently execute other tasks while it awaits the completion of these asynchronous actions.

Originally, JavaScript used callback functions to handle asynchronous actions. The problem with callbacks is that they encourage complexly nested code which quickly becomes difficult to read, debug, and scale. With ES6, JavaScript integrated native [promises](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) which allow us to write significantly more readable code. JavaScript is continually improving, and [ES8](https://en.wikipedia.org/wiki/ECMAScript#8th_Edition_-_ECMAScript_2017) provides a new syntax for handling our asynchronous action, *async...await*. The async...await syntax allows us to write asynchronous code that reads similarly to traditional synchronous, imperative programs.

The async...await syntax is [syntactic sugar](https://en.wikipedia.org/wiki/Syntactic_sugar)— it doesn’t introduce new functionality into the language, but rather introduces a new syntax for using promises and [generators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Generator). Both of these were already built in to the language. Despite this, async...await powerfully improves the readability and scalability of our code. Let’s learn how to use it!

**Instructions**

**1.**

In this lesson, you’ll be given a code-editor to write your code and a bash terminal to execute your code. To execute your program, you’ll type node app.js in the terminal and hit enter (or return). You’ll press the “Check Work” button to submit your code for evaluation.

Check out the code we provided. It shows three ways of accomplishing the same task: reading and printing from two files in a specified order:

* The first version uses callback functions.
* The second version uses native promise syntax
* The third version uses async...await.

Don’t worry about understanding the async...await syntax— that’s what you’ll learn in this lesson! If you haven’t already, type node app.js into the terminal and press enter. Then, press “Check Work”.

const fs = require('fs');

const promisifiedReadfile = require('./promisifiedReadfile');

// Here we use fs.readfile() and callback functions:

fs.readFile('./file.txt', 'utf-8', (err, data) => {

  if (err) throw err;

  let firstSentence = data;

  fs.readFile('./file2.txt',  'utf-8', (err, data) => {

    if (err) throw err;

    let secondSentence = data;

    console.log(firstSentence, secondSentence)

  });

});

// Here we use native promises with our "promisified" version of readfile:

let firstSentence

promisifiedReadfile('./file.txt', 'utf-8')

  .then((data) => {

    firstSentence = data;

    return promisifiedReadfile('./file2.txt', 'utf-8')

  })

  .then((data) => {

    let secondSentence = data;

    console.log(firstSentence, secondSentence)

  })

  .catch((err) => {console.log(err)});

// Here we use promisifiedReadfile() again but instead of using the native promise .then() syntax, we declare and invoke an async/await function:

async function readFiles() {

  let firstSentence = await promisifiedReadfile('./file.txt', 'utf-8')

  let secondSentence = await promisifiedReadfile('./file2.txt', 'utf-8')

  console.log(firstSentence, secondSentence)

}

readFiles()

**The async Keyword**

The async keyword is used to write functions that handle asynchronous actions. We wrap our asynchronous logic inside a function prepended with the async keyword. Then, we invoke that function.

async function myFunc() {

// Function body here

};

myFunc();

We’ll be using async function declarations throughout this lesson, but we can also create async function expressions:

const myFunc = async () => {

// Function body here

};

myFunc();

async functions always return a promise. This means we can use traditional promise syntax, like .then() and .catch with our async functions. An async function will return in one of three ways:

* If there’s nothing returned from the function, it will return a promise with a resolved value of undefined.
* If there’s a non-promise value returned from the function, it will return a promise resolved to that value.
* If a promise is returned from the function, it will simply return that promise

async function fivePromise() {

return 5;

}

fivePromise()

.then(resolvedValue => {

console.log(resolvedValue);

}) // Prints 5

In the example above, even though we return 5 inside the function body, what’s actually returned when we invoke fivePromise() is a promise with a resolved value of 5.

Let’s write an async function!

**Instructions**

**1.**

We provided a function withConstructor() which takes in a number. If the number is 0, it returns a promise that resolves to the string 'zero'. If the number is not 0, it returns a promise that resolves to the string 'not zero'. Take a moment to understand this function and the code that follows. When you’re ready to run it, type node app.js in to the terminal and press enter.

**2.**

Write an async function, withAsync() which reproduces the functionality of withConstructor(). Though your function will return a promise, it should not construct the promise using the new keyword. Instead, it should rely on the fact that an async function automatically returns a promise.

When you’re ready, check your work to move on to the next step.

Hint

Remember that an async function **returns** a promise with a resolved value equal to the return value of that function. The function:

function nativePromise(){

return new Promise((resolve, reject) => {

resolve('yay!');

})

}

Could be written:

async function asyncPromise(){

return 'yay!';

}

**3.**

Now test your code! Uncomment the test code we wrote at the bottom of **app.js**. In the terminal, type node app.js and press enter to execute the code.

function withConstructor(num){

  return new Promise((resolve, reject) => {

    if (num === 0){

      resolve('zero');

    } else {

      resolve('not zero');

    }

  })

}

withConstructor(0)

  .then((resolveValue) => {

  console.log(` withConstructor(0) returned a promise which resolved to: ${resolveValue}.`);

})

// Write your code below:

async function withAsync(num){

  if (num === 0){

      return 'zero';

    } else {

      return 'not zero';

    }

}

withAsync(100)

  .then((resolveValue) => {

  console.log(` withAsync(100) returned a promise which resolved to: ${resolveValue}.`);

})

# The await Operator

In the last exercise, we covered the async keyword. By itself, it doesn’t do much; async functions are almost always used with the additional keyword await inside the function body.

The await keyword can only be used inside an async function. await is an operator: it returns the resolved value of a promise. Since promises resolve in an indeterminate amount of time, await halts, or pauses, the execution of our async function until a given promise is resolved.

In most situations, we’re dealing with promises that were returned from functions. Generally, these functions are through a library, and, in this lesson, we’ll be providing them. We can await the resolution of the promise it returns inside an async function. In the example below, myPromise() is a function that returns a promise which will resolve to the string "I am resolved now!".

async function asyncFuncExample(){

let resolvedValue = await myPromise();

console.log(resolvedValue);

}

asyncFuncExample(); // Prints: I am resolved now!

Within our async function, asyncFuncExample(), we use await to halt our execution until myPromise() is resolved and assign its resolved value to the variable resolvedValue. Then we log resolvedValue to the console. We’re able to handle the logic for a promise in a way that reads like synchronous code.

**Instructions**

**1.**

In the provided code, we’ve required in the function brainstormDinner(). This function expects no arguments and returns a new promise with a resolved value of a string representing a meal. (You can look at the code for this function by navigating to the **library.js** file.)

Look at the nativePromiseDinner() function in **app.js**. Take a moment to understand this function and predict its expected outcome. In the next step, you’ll be recreating its functionality using async...await rather than native promises.

It’s not necessary to execute the nativePromiseDinner() function, but, if you’d like to, check out the hint for some guidance. Press “Check Work” when you’re ready to move on to the next step.

Hint

To see the nativePromiseDinner(), invoke the nativePromiseDinner() function in your **app.js**. Then, type node app.js in the terminal and press enter, which will execute the program.

**2.**

Fill in the body of the announceDinner() function so that it has the same functionality as nativePromiseDinner(). It should wait for the promise returned from brainstormDinner() to resolve, and then log a string to the console in the same format as did nativePromiseDinner(). You’ll need to use the await operator inside your function.

When you’re ready, check your work to move on to the next step.

Hint

We’ll want to create a variable to hold the resolved value of the promise returned from brainstormDinner().

let meal = await brainstormDinner();

Then we can write our console.log()

console.log(`I'm going to make ${meal} for dinner.`);

**3.**

At the bottom of **app.js**, beneath the function declaration, invoke the announceDinner() function.

Hint

We invoke an async function the same way we invoke a normal function:

someAsyncFunction();

**4.**

In the terminal type node app.js and press enter to execute your code.

const brainstormDinner = require('./library.js')

// Native promise version:

function nativePromiseDinner() {

  brainstormDinner().then((meal) => {

    console.log(`I'm going to make ${meal} for dinner.`);

  })

}

// async/await version:

async function announceDinner() {

  // Write your code below:

  let meal = await brainstormDinner();

  console.log(`I'm going to make ${meal} for dinner.`);

}

announceDinner()

**Writing async Functions**

We’ve seen that the await keyword halts the execution of an async function until a promise is no longer pending. Don’t forget the await keyword! It may seem obvious, but this can be a tricky mistake to catch because our function will still run— it just won’t have the desired results.

We’re going to explore this using the following function, which returns a promise that resolves to 'Yay, I resolved!' after a 1 second delay:

let myPromise = () => {

return new Promise((resolve, reject) => {

setTimeout(() => {

resolve('Yay, I resolved!')

}, 1000);

});

}

Now we’ll write two async functions which invoke myPromise():

async function noAwait() {

let value = myPromise();

console.log(value);

}

async function yesAwait() {

let value = await myPromise();

console.log(value);

}

noAwait(); // Prints: Promise { <pending> }

yesAwait(); // Prints: Yay, I resolved!

In the first async function, noAwait(), we left off the await keyword before myPromise(). In the second, yesAwait(), we included it. The noAwait() function logs Promise { <pending> } to the console. Without the await keyword, the function execution wasn’t paused. The console.log() on the following line was executed before the promise had resolved.

Remember that the await operator returns the resolved value of a promise. When used properly in yesAwait(), the variable value was assigned the resolved value of the myPromise() promise, whereas in noAwait(), value was assigned the promise object itself.

**Instructions**

**1.**

Take a look at the provided code. We required in the shopForBeans() function from **library.js** which returns a promise with a resolved value of a string representing a type of bean, eg. ‘kidney’.

Next, we declared the getBeans() function. Let’s summarize its intended functionality:

* Log '1. Heading to the store to buy beans...' to the console.
* Capture the resolved value of the promise returned when we invoke shopForBeans().
* The promise returned from shopForBeans() prints a string in the format '2. I bought [the resolved type of beans] beans because they were on sale.'
* Finally, the function prints a string in the format '3. Great! I'm making [the bean type] beans for dinner tonight!' to the console.

Execute **app.js** in the terminal to see if getBeans() is working as intended.

Hint

To execute a file, we can type node someFileName.js in the terminal and press enter.

**2.**

Uh oh, getBeans() is logging '3. Great! I'm making [object Promise] beans for dinner tonight!' to the console. Take a moment to understand why this is happening, and then refactor getBeans() so it works as intended. Check your work when you’re ready to move on to the next step.

Hint

getBeans() is missing the keywords async and await. The keyword async should precede the function keyword, and the await operator precedes a promise. Remember that in our case, this is the promise returned from shopForBeans().

**3.**

To test that the code is now functioning properly, execute **app.js** in the terminal again.

Hint

To execute a file, we can type node someFileName.js in the terminal and press enter.

const shopForBeans = require('./library.js');

async function getBeans() {

  console.log(`1. Heading to the store to buy beans...`);

  let value = await shopForBeans();

  console.log(`3. Great! I'm making ${value} beans for dinner tonight!`);

}

getBeans();

**Handling Dependent Promises**

The true beauty of async...await is when we have a series of asynchronous actions which depend on one another. For example, we may make a network request based on a query to a database. In that case, we would need to wait to make the network request until we had the results from the database. With native promise syntax, we use a chain of .then() functions making sure to return correctly each one:

function nativePromiseVersion() {

returnsFirstPromise()

.then((firstValue) => {

console.log(firstValue);

return returnsSecondPromise(firstValue);

})

.then((secondValue) => {

console.log(secondValue);

});

}

Let’s break down what’s happening in the nativePromiseVersion() function:

* Within our function we use two functions which return promises: returnsFirstPromise() and returnsSecondPromise().
* We invoke returnsFirstPromise() and ensure that the first promise resolved by using .then().
* In the callback of our first .then(), we log the resolved value of the first promise, firstValue, and then return returnsSecondPromise(firstValue).
* We use another .then() to print the second promise’s resolved value to the console.

Here’s how we’d write an async function to accomplish the same thing:

async function asyncAwaitVersion() {

let firstValue = await returnsFirstPromise();

console.log(firstValue);

let secondValue = await returnsSecondPromise(firstValue);

console.log(secondValue);

}

Let’s break down what’s happening in our asyncAwaitVersion() function:

* We mark our function as async.
* Inside our function, we create a variable firstValue assigned await returnsFirstPromise(). This means firstValue is assigned the resolved value of the awaited promise.
* Next, we log firstValue to the console.
* Then, we create a variable secondValue assigned to await returnsSecondPromise(firstValue). Therefore, secondValue is assigned this promise’s resolved value.
* Finally, we log secondValue to the console.

Though using the async...await syntax can save us some typing, the length reduction isn’t the main point. Given the two versions of the function, the async...await version more closely resembles synchronous code, which helps developers maintain and debug their code. The async...await syntax also makes it easy to store and refer to resolved values from promises further back in our chain which is a much more difficult task with native promise syntax. Let’s create some async functions with multiple await statements!

**Instructions**

**1.**

Take a look at the provided code. We require in three functions: shopForBeans(), soakTheBeans(), and cookTheBeans(). These functions each return a promise.

* shopForBeans() expects no arguments and returns a promise which will resolve to a string of a bean type.
* soakTheBeans() expects a bean type string as an argument and returns a promise which resolves to a boolean indicating whether or not the beans are softened.
* cookTheBeans() expects a boolean as an argument and, if that value is true, returns a promise which will resolve to a string announcing that dinner is ready.

If you’d like, look at the **library.js** file to see how these functions work. Press “Check Work” to move on to the next step.

**2.**

In the following checkpoints, you’ll create an async function that handles the three dependent promises we just walked through. It will simulate shopping for, soaking, and then cooking the beans. To get started, declare an async function, makeBeans(). You can leave the function body blank for now.

Check your work to move on to the next step.

Hint

To declare an async function:

async function funcName() {

}

**3.**

Inside your function, declare a variable, type, assigned to the resolved value of shopForBeans(). This must be accomplished in one statement.

Hint

This is the syntax for using the await operator:

const variableName = await returnsPromise();

**4.**

Next inside your function, declare a variable, isSoft, assigned to the resolved value of soakTheBeans(). Don’t forget: the soakTheBeans() function expects an argument. Make sure to pass the expected value into soakTheBeans().

Hint

The soakTheBeans() function expects a bean type string which is exactly what the shopForBeans() function’s promise resolves to! You should pass your type variable into soakTheBeans().

let isSoft = soakTheBeans(type);

**5.**

Next inside your function, declare a variable, dinner, assigned to the resolved of cookTheBeans(). Don’t forget: the cookTheBeans() function also expects an argument. Make sure to pass the expected value into cookTheBeans().

Stuck? Get a hint

**6.**

Next inside your function, log your dinner variable to the console.

**7.**

Beneath your function declaration, invoke the makeBeans() function.

**8.**

Let’s see your async function in action! In the terminal, type node app.js and press enter to execute the code.

const {shopForBeans, soakTheBeans, cookTheBeans} = require('./library.js');

// Write your code below:

async function makeBeans() {

  let type = await shopForBeans();

  let isSoft = await soakTheBeans(type);

  let dinner = await cookTheBeans(isSoft);

  console.log(dinner);

}

makeBeans();

# Handling Errors

When .catch() is used with a long promise chain, there is no indication of where in the chain the error was thrown. This can make debugging challenging.

With async...await, we use try...catch statements for error handling. By using this syntax, not only are we able to handle errors in the same way we do with synchronous code, but we can also catch both synchronous and asynchronous errors. This makes for easier debugging!

async function usingTryCatch() {

try {

let resolveValue = await asyncFunction('thing that will fail');

let secondValue = await secondAsyncFunction(resolveValue);

} catch (err) {

// Catches any errors in the try block

console.log(err);

}

}

usingTryCatch();

Remember, since async functions return promises we can still use native promise’s .catch() with an async function

async function usingPromiseCatch() {

let resolveValue = await asyncFunction('thing that will fail');

}

let rejectedPromise = usingPromiseCatch();

rejectedPromise.catch((rejectValue) => {

console.log(rejectValue);

})

This is sometimes used in the global scope to catch final errors in complex code.

**Instructions**

**1.**

For convenience, we’ve been working with a lot of promises that never reject, but this isn’t very realistic!

This time we’ve required in a function, cookBeanSouffle() which returns a promise that resolves or rejects randomly. When it resolves, the promise resolves with a value of 'Bean Souffle' and, when it rejects, it rejects with a value of 'Dinner is ruined!'. If you’re interested, you can see how the function works by looking in the **library.js** file.

Press “Check Work” when you’re ready to move on to the next step.

**2.**

Declare an async function, hostDinnerParty(). Inside your function, create a try...catch statement. The catch statement should specify an identifier, error. You can leave both the try and catch blocks empty.

Hint

Your function should look like this:

async function functionName() {

try {

}

catch (error) {

}

}

**3.**

Inside your try block, log a string in the following format: '[resolved value of cookBeanSouffle() promise] is served!' ie. 'Bean Souffle is served!'. Make sure to await the cookBeanSouffle() promise. For more guidance, check out the hint.

Hint

You can declare a variable, we called ours dinner, assigned to the resolved value of the promise returned by cookBeanSouffle(). Then use string interpolation and the dinner variable to log the string 'Bean Souffle is served!' to the console:

try {

let dinner = await cookBeanSouffle();

console.log(`${dinner} is served!`);

}

**4.**

Now let’s fill in the catch block! First log the error to the console, and then log the string: 'Ordering a pizza!' to the console.

Hint

Your catch block should contain the following code:

console.log(error);

console.log('Ordering a pizza!');

**5.**

Awesome! Beneath your function declaration, invoke hostDinnerParty().

**6.**

Let’s see your function in action. In the terminal type node app.js and press enter to run the code. If you keep executing **app.js** you’ll see the results of the promise resolving or rejecting.

const cookBeanSouffle = require('./library.js');

// Write your code below:

async function hostDinnerParty() {

 try {

   let dinner = await cookBeanSouffle();

   console.log(`${dinner} is served!`);

 }

 catch(error){

   console.log(error);

   console.log('Ordering a pizza!');

 }

}

hostDinnerParty();

**Handling Independent Promises**

Remember that await halts the execution of our async function. This allows us to conveniently write synchronous-style code to handle dependent promises. But what if our async function contains multiple promises which are not dependent on the results of one another to execute?

async function waiting() {

const firstValue = await firstAsyncThing();

const secondValue = await secondAsyncThing();

console.log(firstValue, secondValue);

}

async function concurrent() {

const firstPromise = firstAsyncThing();

const secondPromise = secondAsyncThing();

console.log(await firstPromise, await secondPromise);

}

In the waiting() function, we pause our function until the first promise resolves, then we construct the second promise. Once that resolves, we print both resolved values to the console.

In our concurrent() function, both promises are constructed without using await. We then await each of their resolutions to print them to the console.

With our concurrent() function both promises’ asynchronous operations can be run simultaneously. If possible, we want to get started on each asynchronous operation as soon as possible! Within our async functions we should still take advantage of *concurrency*, the ability to perform asynchronous actions at the same time.

Note: if we have multiple truly independent promises that we would like to execute fully in parallel, we must use individual .then() functions and avoid halting our execution with await.

**Instructions**

**1.**

Take a look at the provided code. We require in four functions: cookBeans(), steamBroccoli(), cookRice(), and bakeChicken(). These functions each return a promise which will resolve to a string representing a part of a meal.

If you’d like, look at the **library.js** file to see how these functions work. Press “Check Work” when you’re ready to move on.

**2.**

Declare an async function, serveDinner(). Create four variables:

* vegetablePromise which should be assigned the return value of steamBroccoli()
* starchPromise which should be assigned the return value of cookRice()
* proteinPromise which should be assigned the return value of bakeChicken()
* and sidePromise which should be assigned the return value of cookBeans()

These variables should be assigned the promise objects themselves not their resolved values.

Hint

In order to make use of concurrency, we’re going to omit the await operator! Your code should include:

const vegetablePromise = steamBroccoli();

const starchPromise = cookRice();

const proteinPromise = bakeChicken();

const sidePromise = cookBeans();

**3.**

Next console.log() a string in the following format: Dinner is served. We’re having [resolved value of the vegetablePromise], [resolved value of the starchPromise], [resolved value of the proteinPromise], and [resolved value of the sidePromise]. ie. ‘Dinner is served. We’re having broccoli, rice, chicken, and beans.’

You’ll need to await each promise, but there are a few different ways you can accomplish the desired functionality. Check out the hint if you want some guidance.

Hint

One option is to create a second set of variables with assigned values of await-ing the promise variables you made in the last checkpoint and then using string concatenation or string interpolation with those values. Another is to use the await operator directly inside your string construction:

`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${await proteinPromise}, and ${await sidePromise}.`

**4.**

Awesome! Let’s see your function in action. Beneath your function declaration, invoke serveDinner().

**5.**

In the terminal type node app.js and press enter to run the code.

let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js')

// Write your code below:

async function serveDinner() {

 let vegetablePromise = steamBroccoli();

 let starchPromise = cookRice();

 let proteinPromise = bakeChicken();

 let sidePromise = cookBeans();

 console.log(`Dinner is served. We're having ${await vegetablePromise}, ${await starchPromise}, ${await proteinPromise}, and ${await sidePromise}.`)

}

serveDinner()

# Await Promise.all()

Another way to take advantage of concurrency when we have multiple promises which can be executed simultaneously is to await a Promise.all().

We can pass an array of promises as the argument to Promise.all(), and it will return a single promise. This promise will resolve when all of the promises in the argument array have resolved. This promise’s resolve value will be an array containing the resolved values of each promise from the argument array.

async function asyncPromAll() {

const resultArray = await Promise.all([asyncTask1(), asyncTask2(), asyncTask3(), asyncTask4()]);

for (let i = 0; i<resultArray.length; i++){

console.log(resultArray[i]);

}

}

In our above example, we await the resolution of a Promise.all(). This Promise.all() was invoked with an argument array containing four promises (returned from required-in functions). Next, we loop through our resultArray, and log each item to the console. The first element in resultArray is the resolved value of the asyncTask1() promise, the second is the value of the asyncTask2() promise, and so on.

Promise.all() allows us to take advantage of asynchronicity— each of the four asynchronous tasks can process concurrently. Promise.all() also has the benefit of failing fast, meaning it won’t wait for the rest of the asynchronous actions to complete once any one has rejected. As soon as the first promise in the array rejects, the promise returned from Promise.all() will reject with that reason. As it was when working with native promises, Promise.all() is a good choice if multiple asynchronous tasks are all required, but none must wait for any other before executing.

**Instructions**

**1.**

In this exercise, we require in the same four functions as in the last exercise: cookBeans(), steamBroccoli(), cookRice(), and bakeChicken().

These functions each return a promise which will resolve to a string representing a part of a meal. You can check them out in the **library.js** file. Press “Check Work” to move on to the next step.

**2.**

You’re going to create a very similar function to the serveDinner() function you created in the last exercise. This time, you’ll use Promise.all()!

Create an async function serveDinnerAgain(). Inside your function, declare a variable foodArray and assign it the resolved value of the promise returned from Promise.all().

Remember that Promise.all() takes in an array of promises. Pass in an array containing the steamBroccoli(), cookRice(), bakeChicken(), and cookBeans() functions in that order.

Hint

Don’t forget to await Promise.all()! Your code may follow a similar pattern to this:

async function myFunction() {

let myArray = await Promise.all([returnsPromise1(), returnsPromise2(), returnsPromise3() ]);

}

**3.**

Next console.log() a string in the following format: Dinner is served. We’re having [first item in foodArray ], [second item in foodArray ], [third item in foodArray], and [fourth item in foodArray ]. eg. ‘Dinner is served. We’re having broccoli, rice, chicken, and beans.’

There are a few different ways you can accomplish the desired functionality. Check out the hint if you want some guidance.

Hint

Now that you have an array holding each of the promises’ resolved values, you can use string interpolation or concatenation to log the required string.

One option is to save variable for each value:

let vegetable = foodArray[0];

let starch = foodArray[1];

let protein = foodArray[2];

let side = foodArray[3];

console.log(`Dinner is served. We're having ${vegetable}, ${starch}, ${protein}, and ${side}.`);

If you’re confident in your logic but this step isn’t passing, make sure to double check your spelling and punctuation.

**4.**

Ok great! Now let’s see your function in action. Beneath your function declaration, invoke serveDinnerAgain().

**5.**

In the terminal type node app.js and press enter to run the code.

let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js')

// Write your code below:

async function serveDinnerAgain(){

  let foodArray = await Promise.all([steamBroccoli(), cookRice(), bakeChicken(), cookBeans()]);

  console.log(`Dinner is served. We're having ${foodArray[0]}, ${foodArray[1]}, ${foodArray[2]}, and ${foodArray[3]}.`)

}

serveDinnerAgain()

# Review

Awesome work getting the hang of the async...await syntax! Let’s review what you’ve learned:

* async...await is syntactic sugar built on native JavaScript promises and generators.
* We declare an async function with the keyword async.
* Inside an async function we use the await operator to pause execution of our function until an asynchronous action completes and the awaited promise is no longer pending .
* await returns the resolved value of the awaited promise.
* We can write multiple await statements to produce code that reads like synchronous code.
* We use try...catch statements within our async functions for error handling.
* We should still take advantage of concurrency by writing async functions that allow asynchronous actions to happen in concurrently whenever possible.

let {cookBeans, steamBroccoli, cookRice, bakeChicken} = require('./library.js')

// Write your code below:

async function serveDinnerAgain(){

  let foodArray = await Promise.all([steamBroccoli(), cookRice(), bakeChicken(), cookBeans()]);

  console.log(`Dinner is served. We're having ${foodArray[0]}, ${foodArray[1]}, ${foodArray[2]}, and ${foodArray[3]}.`)

}

serveDinnerAgain()