Section-End Project - Lesson 2

JavaScript(ES6) Exercises

[1. Variables and Block Scopes 2](#_Toc22503)

[Ex. 1 2](#_Toc3178)

[Ex. 2 2](#_Toc30344)

[Ex. 3 3](#_Toc9964)

[2. Let & Const 4](#_Toc30835)

[Ex 4: 4](#_Toc3830)

[Ex 5: 5](#_Toc20633)

[Ex 6: 5](#_Toc18652)

[3. Arrow function and this context 5](#_Toc634)

[Ex 7 : 6](#_Toc8348)

[Ex 8 : 6](#_Toc3764)

[Ex 9 : 7](#_Toc11941)

[4. Object Literals 8](#_Toc1305)

[Ex 10 : 9](#_Toc16409)

[Ex 11 : 9](#_Toc28786)

[5. Destructuring and Rest/Spread 10](#_Toc9736)

[Object Destructuring 10](#_Toc12631)

[Nested Object Destructuring 10](#_Toc19299)

[Ex 12 : 11](#_Toc18628)

[Spread with Objects 12](#_Toc12227)

[Ex 13 : 12](#_Toc22608)

[Rest Parameters 13](#_Toc5434)

[Ex 14 : 13](#_Toc3383)

[6. Template Strings 14](#_Toc25024)

[Ex 15 : 15](#_Toc5268)

[7. Classes 15](#_Toc30976)

[OOP in JavaScript with ES6 15](#_Toc27255)

[Ex 16 : 16](#_Toc14733)

[Ex 17 : 16](#_Toc8906)

[Ex 18 : 17](#_Toc24778)

[8. Promises and Async/Await 19](#_Toc530)

[Using promises, we can write asynchronous programs in a more manageable way. Using Async/Await syntax, a promise-based asynchronous code can be written in a synchronous format which saves a lot of time and code becomes scalable. 19](#_Toc29452)

[Ex 19 : 19](#_Toc11673)

[Ex 20: 21](#_Toc29111)

[Ex 21: 23](#_Toc13695)

[Callback Hell!!! 24](#_Toc14188)

[Ex 22: 25](#_Toc4296)

[Promise to the rescue 25](#_Toc21121)

[Ex 23: 27](#_Toc8799)

[Ex 24: 28](#_Toc14423)

[Handling promise rejection in then handler 29](#_Toc22435)

[Ex 25: 29](#_Toc16465)

[Async/Await 30](#_Toc13314)

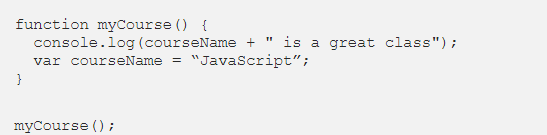
[Ex 26: 31](#_Toc2925)

Run all the code samples in its own .js files in VSCode IDE with the node command. Depending on the program(s), you can keep one .js file for each topic. You can also use the Developer Console in Chrome to test your programs.

\*\* Some exercises need only minor modifications to the previous exercise(s). Copy the previous program and make your changes as needed.

# Variables and Block Scopes

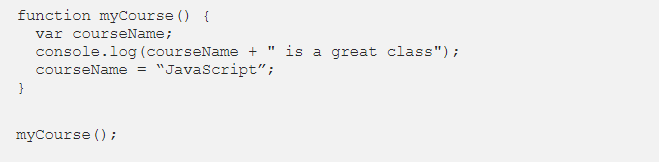
## Ex. 1



*This will output:******undefined*****

What we get is kind of strange. The reason this happened is because before it ran any of the code JavaScript did this..

## Ex. 2

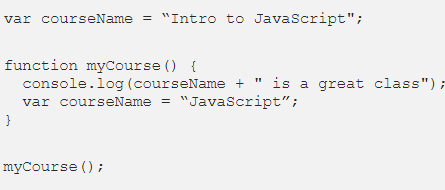


*This will output:******undefined is a great class*****

The variable is actually sitting there but it doesn’t have a value yet and this can cause problems when you create more complex applications. → *Hoisting*can turn an innocent looking declaration into a subtle bug.

Let’s see what happens if we put a variable outside of the function..

## Ex. 3



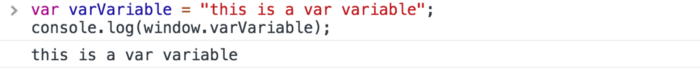
Why? because the variable of courseName is still appearing prior to the console.log since it is being hoisted, but it is not defined till after.

Now that we understand ****var****, let’s walk through *what* Let & Const are, *why* they were added & *when* you should use them:

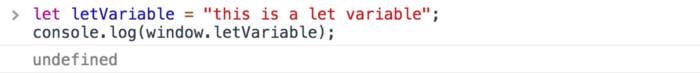
# Let & Const



* ****let**** allows you to declare variables that are limited in scope to the block, statement, or expression on which it is used. This is unlike the [var](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Statements/var" \t "https://medium.com/@LindaVivah/_blank) keyword, which defines a variable globally, or locally to an entire function regardless of block scope.  
  Ex 4:



## Ex 4:



* The****const declaration**** creates a read-only reference to a value. It does ****not**** mean the value it holds is immutable, just that the variable identifier cannot be reassigned. *Here is an example:*

## Ex 5:



* Variables declared with ****let**** can be reassigned, but can’t be redeclared in the same scope. *Here is an example:*

## Ex 6:

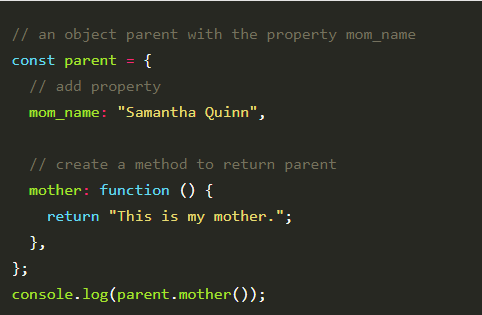


* Variables declared with ****const**** must be assigned an initial value, but can’t be redeclared in the same scope, and can’t be reassigned.

# Arrow function and this context

* [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this) represents an object that executes the current function. In short, this is defined by the function execution context. Such as how a function is called, it refers to a global object window. For example, when a function is being executed from a global object.
* You might have used this keyword in a real-life situation without realizing it. Suppose you are walking along with your mother and meet a friend along the way. This is how you would introduce your mom to your friend. This is my mother.
* Take a close look at this is that sentence. this shows a reference to your mother. this represents the mother in the current sentence. It is the same way JavaScript uses this keyword.
* Let see how JavaScript will refer to a mother using this.

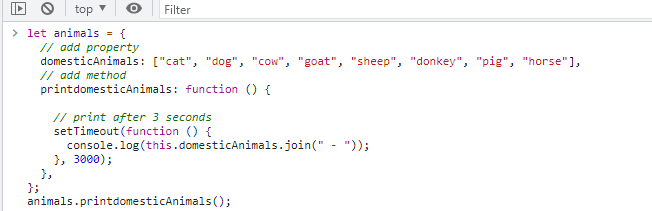
## Ex 7 :



Execute the examples below with the console. You can use the Google chrome console.

## 

## Ex 8 :



This is where the arrow function comes into play. They don’t have their own this context. When used inside the outer (enclosing) function, this keyword will point to where the function is present.

In this case, this will be attached to the outer context printdomesticAnimals() where setTimeout() is called. printdomesticAnimals() will be the enclosing context where this will be attached.

When we use the arrow function, we get the results as we expected.

## Ex 9 :



# Object Literals

Object literals are used to create an object in javascript. Enhancement in Object literals in ES2015 (ES6) release has made it a more powerful feature.

* The object in javascript can be initialized by directly using the variable name. See Example 1 below.
* Object’s method in ES5 requires function statement. This is no longer required in ES6, you can directly return statement. See Example 2 below.
* Object literals key in ES6 can be dynamic. Any Express can be used to create a key.

****RECAP****

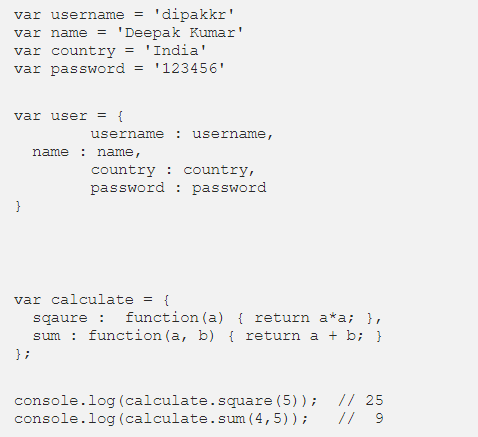
****There are three ways of creating an object in javascript.****

* Using Object literals
* Using new keyword.
* By defining object constructor and then create an object constructor type.

Let’s take a look at this example to see the working of Object literals.

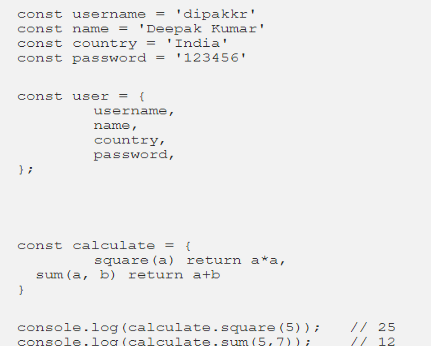
****Object Literals Without ES6 ( ES5 Supported )****

## Ex 10 :



****Object Literals with ES6****

## Ex 11 :



# Destructuring and Rest/Spread

Destructuring assignment is a syntax that allows you to assign object properties or array items as variables. This can greatly reduce the lines of code necessary to manipulate data in these structures. There are two types of destructuring: Object destructuring and Array destructuring.

## **Object Destructuring**

Object destructuring allows you to create new [variables](https://www.digitalocean.com/community/tutorials/understanding-variables-scope-hoisting-in-javascript" \l "understanding-variables) using an object property as the value.

## **Nested Object Destructuring**

An object can be nested. This means that the value of an object property can be another object, and so on.

Let's consider the user object below. It has a property called department with the value as another object. But let's not stop here! The department has a property with the key address whose value is another object. Quite a real-life scenario, isn't it?

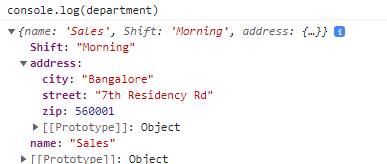
## Ex 12 :



How do we extract the value of the department property? Ok, it should be straight-forward by now.

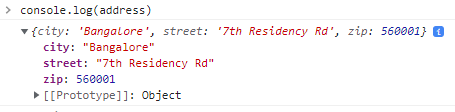


And here's the output when you log department:



But, let's go one more nested level down. How do we extract the value of the address property of the department? Now, this may sound a bit tricky. However, if you apply the same object destructuring principles, you'll see that it's similar.





In this case, department is the key we focus on and we destructure the address value from it. Notice the {} around the keys you want to destructure.

Now it's time to take it to the next level. How do we extract the value of city from the department's address? Same principle again!



The output when you log city is "Bangalore".

It can go any level nested down.

The rule of thumb is to start with the top-level and go down in the hierarchy until you reach the value you want to extract.

## Spread with Objects

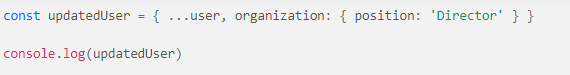
When working with objects, spread can be used to copy and update objects.

One important thing to note with updating objects via spread is that any nested object will have to be spread as well. For example, let’s say that in the user object there is a nested organization object:

## Ex 13 :



If you tried to add a new item to organization, it would overwrite the existing fields:



This would result in the following:

Output

id: 3

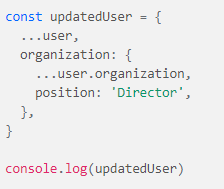
name: "Ron"

organization: {position: "Director"}

If mutability is not an issue, the field could be updated directly:

user.organization.position = 'Director'

But since we are seeking an immutable solution, we can spread the inner object to retain the existing properties:



This will give the following:

Output

id: 3

name: "Ron"

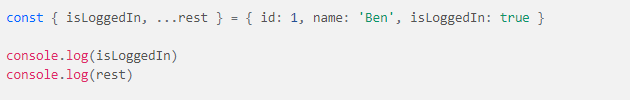
organization: {name: "Parks & Recreation", city: "Pawnee", position: "Director"}

## Rest Parameters

The last feature is the rest parameter syntax. The syntax appears the same as spread (...) but has the opposite effect. Instead of unpacking an array or object into individual values, the rest syntax will create an array of an indefinite number of arguments.

Rest can be used when destructuring objects:

## Ex 14 :



Giving the following output:

Output

true

{id: 1, name: "Ben"}

In this way, rest syntax provides efficient methods for gathering an indeterminate amount of items.

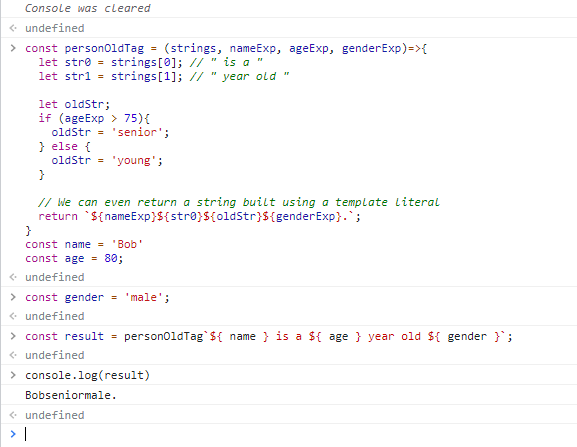
# Template Strings

Tagged templates are great for converting strings to anything you want since it’s just a regular function. However, it is a special function because the first parameter is an array containing the constant parts of the string. The rest of the parameters contain the returned values that each expression returns. This is great for manipulating the string and transforming the returned values to what we want.

The return value of tags can be anything you want. So we can return strings, arrays, objects, or anything else.

As we see in the function below, in the string that we put beside the personOldTag, we first interpolated the name, then the age , then the gender. So in the parameters, they also appear in this order — nameExp, ageEx, and genderExp. They are the evaluated versions of the name, age, and gender in the template string.

## Ex 15 :



# Classes

## OOP in JavaScript with ES6

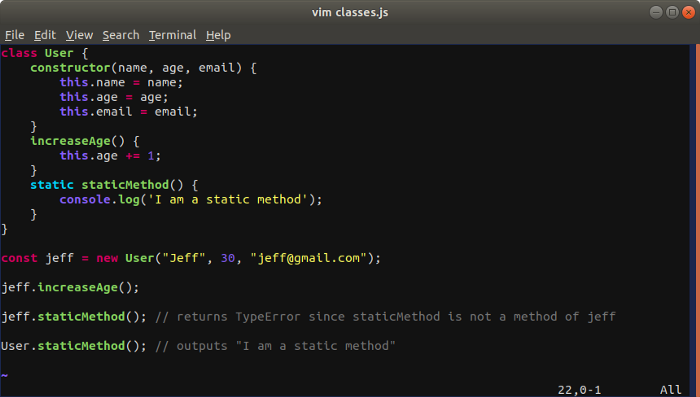
In ES6 we can create ****classes****. If you’ve come from a language like PHP or Python, this will be familiar to you.

The ****class****function basically creates a template that we can use to create objects later. The ****constructor()**** method is a special method called when an instance of the *User* class is created. This is essentially the same as the *User* function we defined in the pre-ES6 example.

****Methods:****

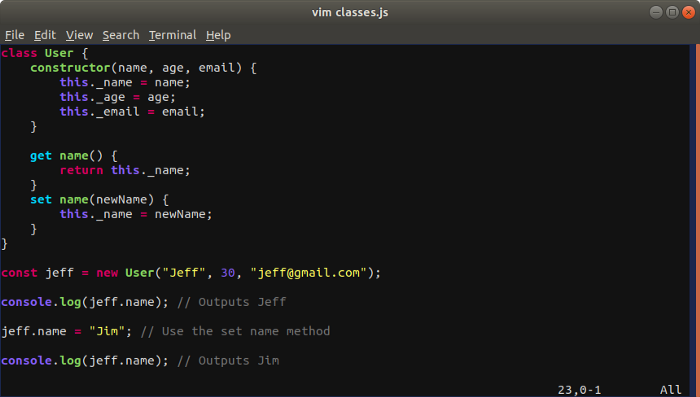
Classes can contain instance and ****static**** methods. A static method is a function that is bound to the class, not an object. A static method cannot be called from an instance of the class.Class methods can be defined as follows:

## Ex 16 :



Let’s take a quick look at ****getters****and ****setters****. One of the core concepts of OOP is ****encapsulation.****An important part of ****encapsulation**** is that data (object properties) should not be directly accessed or modified from outside the object. To access or modify a property we would use a ****getter**** (access) or a ****setter**** (modify), which are specific methods we define in our class. Let’s take a look at a quick example:

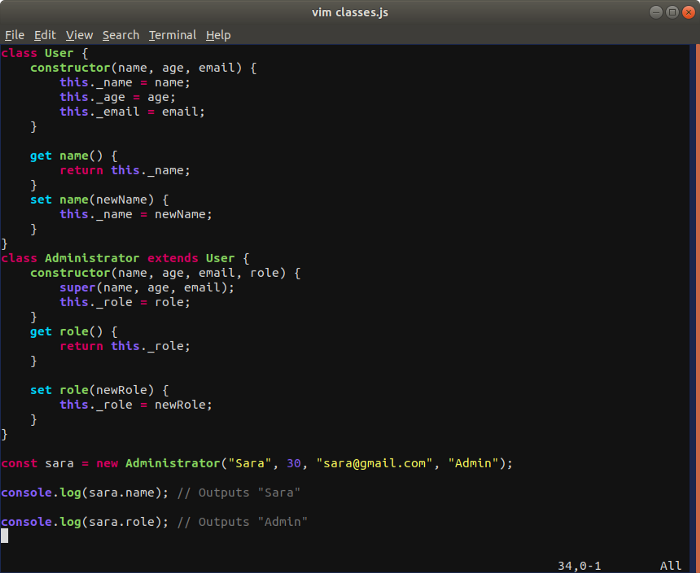
## Ex 17 :



Currently there is no native support for private properties in JavaScript (it is possible to mimic private properties but we’re not going to go into that). So all the properties we’ve declared can be directly accessed from outside the object. But following ****encapsulation**** is best practice for OOP.

****Inheritance:****Classes can also inherit from other classes. The class being inherited from is called the ****parent,**** and the class inheriting from the parentis called the ****child****. In our example, another class, let’s say *Administrator*, can inherit the properties and methods of the *User* class:

## Ex 18 :



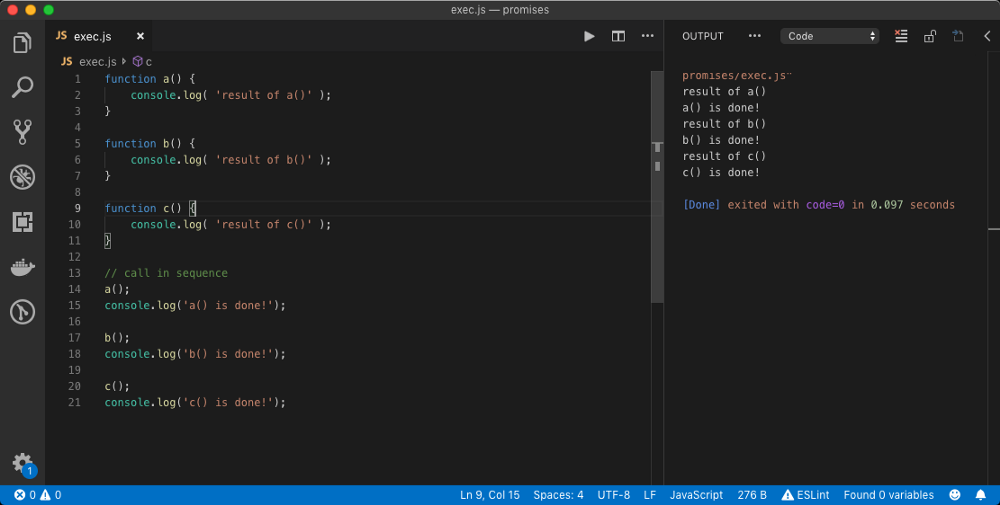
In the above example, *User* is the parent and *Administrator*is the child. There’s a few important things to note. Firstly, when we create the child class we need to state that it ****extends**** the parent class. Then we need to pass whatever properties we want to inherit from the parent to the child’s constructor, as well as any new properties that we will define in the child class. Next, we call the ****super**** method. Notice that we pass it the values that we pass the child class when creating the *sara* object. These values are defined in the parent’s constructor so we need to run it in order for the values to be instantiated. Now we can define our child class’s properties and methods.

# Promises and Async/Await

## Using promises, we can write asynchronous programs in a more manageable way. Using Async/Await syntax, a promise-based asynchronous code can be written in a synchronous format which saves a lot of time and code becomes scalable.

JavaScript executes code in a single thread, which makes it blocking. Let’s take a simple example of calling three functions in series.

## Ex 19 :



As we can see from the above result, each function call and console.log statement is executing in series AKA in ****a synchronous manner****. This means until the function a has returned, the next line of code won’t be called. ****By default, a function with no return statement returns undefined value****.

Using Web APIs, some JavaScript jobs can be transferred to other threads. For example, handling of AJAX request should be done on a different thread, else our main thread would be blocked until the network response is received. This would be a horrible UX for the user, as his/her screen would just freeze for several seconds or minutes.

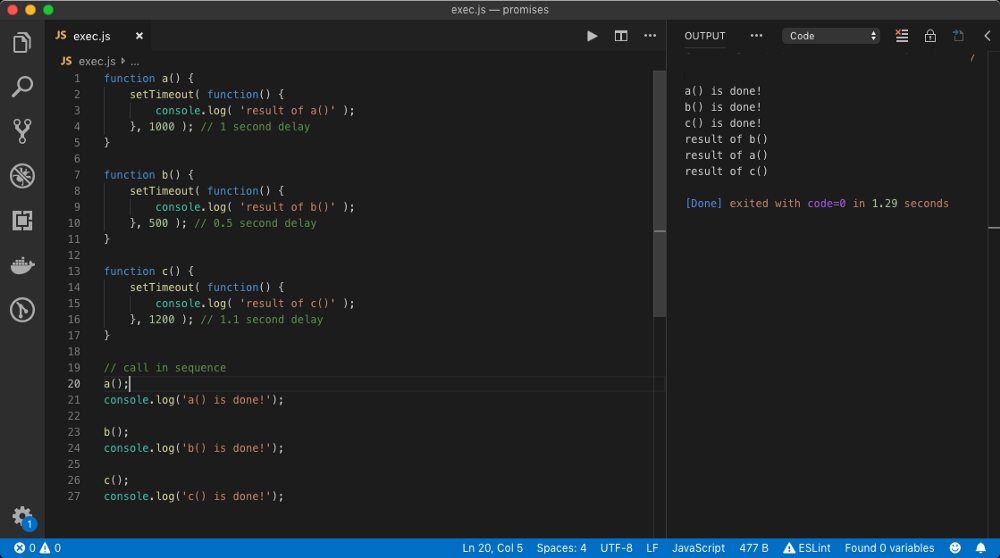
Web APIs are APIs that extend JavaScript functionality to perform asynchronous tasks. For example, ****setTimeout**** is a Web API that performs some action after a given delay. To understand how Web APIs work, or least how setTimeout works, you should check out my article on the **event loop**.

*Web APIs is not a part of the JavaScript standard. They are not included in the JavaScript engine. Instead, they are provided by the browser or server-side JavaScript frameworks like******Node.js*****

Basically, setTimeout(callback, delay) function takes a callback and stores it temporarily. It waits for delay given in milliseconds and then pushes the callback function in the stack ****once the stack is empty****. Then the callback function gets executed. This is basically how all Web APIs work.

Natively, most Web APIs are callback based. They need a callback function to notify when an asynchronous operation is done. Let’s add asynchronous nature to our earlier example and see the problem with it.

## Ex 20:



In the above example, we have added some delay in the log statements of each function. Hence even though we are calling function a, b and c in series, their logs will show up in the order of the least delay, because callback function with least delay provided to setTimeout will execute first.

So in theory, this is what’s happening. The function a is called on line no. 20, which calls the setTimeout function which registers the callback function containing console.log('...done!') statement and then it returns.

Then console.log statement on line no. 21 is called. Then the function b is called in the same manner the function a was called and so on. Once the delay time is passed, setTimeout gives the callback function to the ****event loop**** to be called as soon as possible and the event loop ****queues**** the task in a so-called ****task queue**** or ****macrotask queue****.

The ****event loop**** is endlessly running a single-threaded loop that runs on the ****main JavaScript thread**** and listens for the different events. Its job is to accept callback functions and execute them on the main thread. Since the event loop runs on the main thread, if the main thread is busy, the event loop is basically ****dead**** for that time.

The ****macrotask**** queue is a queue of the callback function waiting to be executed. The ****event loop**** pushes the oldest queued callback functions (FIFO) from the macrotask queue on to the main call ****stack**** one at the time where they are executed synchronously. The event loop only pushes a callback function to the stack ****when the stack is empty**** or when the main thread is not busy.

The stack will become empty when all synchronous function calls are executed. This is why console.log statements outside our functions are executed first because they were pushed on the stack before console.log statements inside the callback functions.

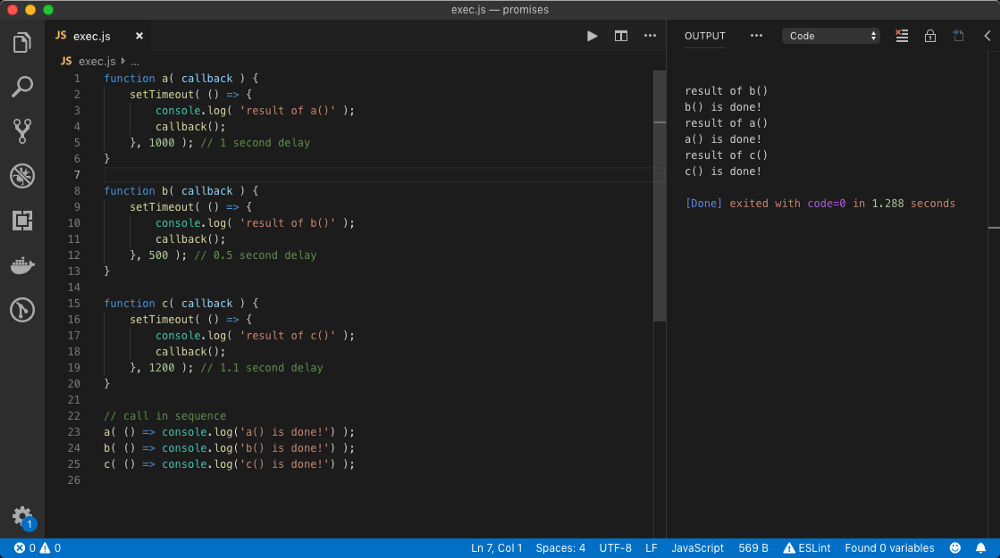
This is why all the console.log statements outside the functions are getting printed in a synchronous manner while console.log statements inside functions are getting called in order of their time delay.

So, here is a tricky question? How can we print console.log statement for each function call only after console.log statement inside that function is printed? The final outcome should be as below.

result of a()  
a() is done!  
result of b()  
b() is done!  
result of c()  
c() is done!

The answer lies in the callback function itself. A callback function according to its definition is a function that will be called when a job is finished. Let’s pass a callback function to each of our functions a, b, and c which contains console.log statement in it.

## Ex 21:



In the above example, we have replaced function declaration syntax with ES6 fat arrow function syntax just for simplicity. Then we passed a callback function which has a console.log('...done!') statement to each function that will perform an async task (*setTimeout*).

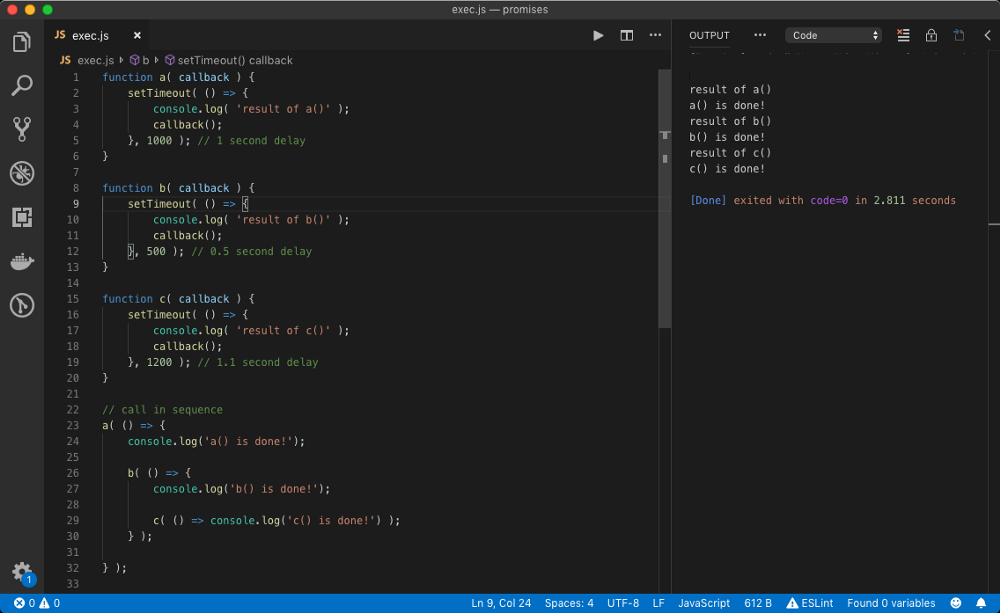
In the callback of setTimeout, we called the callback function received as an argument which indeed prints the done console.log('...done!') statement. This is how we can make sure that a job should execute after an async job is finished, that is, by ****passing a callback function as an argument****.

## Callback Hell!!!

But still, we haven’t resolved the issue. What we want is, ****the jobs should execute in series****. What we can do to make sure a() completes first and then b() and then c(). ****Welcome to callback hell!****

The simple idea is to call b() in the callback of a because that’s where we know that a() has done its job and similarly call c() in the callback of a.

## Ex 22:



In the above example, we have nested the callback which guarantees the series operation of the function. This nesting of callbacks is called as ****Callback Hell****. This is literally a hell because if we had more than 3 function calls, the code could get messier pretty quickly.

## 

## Promise to the rescue

Promises make our job a little easier when it comes to writing complicated asynchronous programs. A promise is an object which has then and catch methods on it. One of this method gets called when the promise returns a value or an error. Let’s understand how to create a promise.

A promise object is created from Promise constructor/class which needs a callback function AKA ****executor function**** ( *either in ES5 syntax or a fat arrow function*). This callback function receives the resolve and reject function arguments, either of which we must envoke with an ****optional payload****.

var myPromise = new Promise( ( resolve, reject ) => {  
 resolve( 'successPayload' );  
 // reject( 'errorPayload' );  
} );myPromise  
.then( successCallback )  
.catch( errorCallback )  
.finally( finallyCallback );

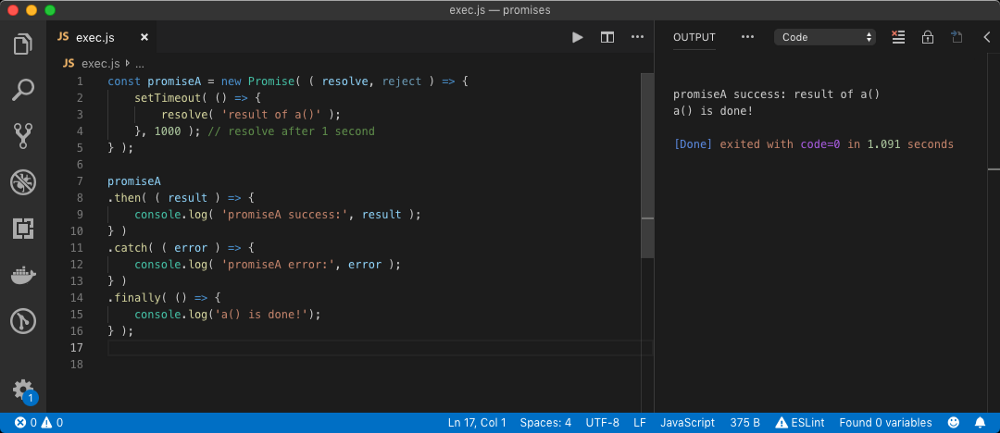
From the above syntax, we can see that we have to call either the resolve function with an optional success payload or reject function with optional error payload. These function can be called from within an async callback as well, like from within a callback of setTimeout function. If no payload passed to these function, the payload is undefined.

When resolve is called, then method (****AKA handler****) is called which executes successCallback with successPayload while if reject is called, catch handler is called which executes errorCallback with errorPayload. finally handler is always called (*without any payload*) when the ****promise is settled****, which means either resolve or reject is called.

*finally is called before then or catch handler if it appears first i.e, before then or catch handler.*

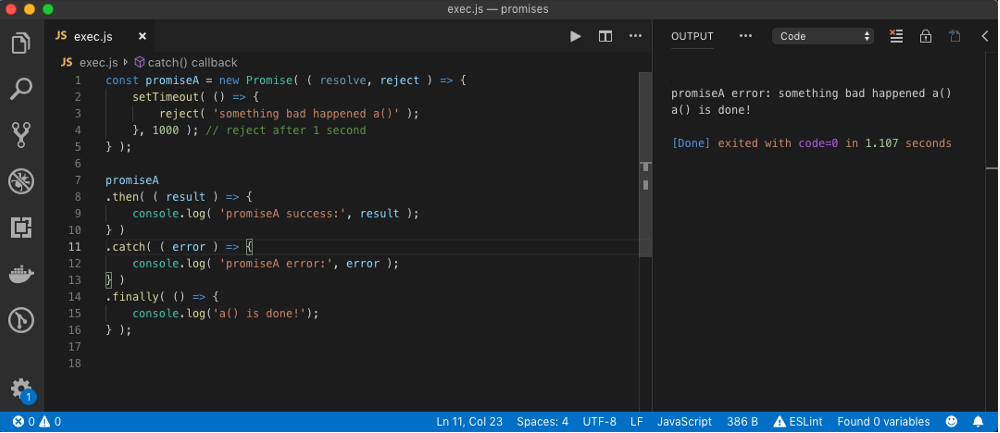
As we can see, then, catch and finally methods are **chainable**, we will see why that is in later topics. Let’s introduce a promise in the previous example.

## Ex 23:



In the above example, we have created promiseA which ****fulfils**** after 1000ms. Since our promise fulfils, then handler executes the callback first and then finally handler executes its callback.

## Ex 24:

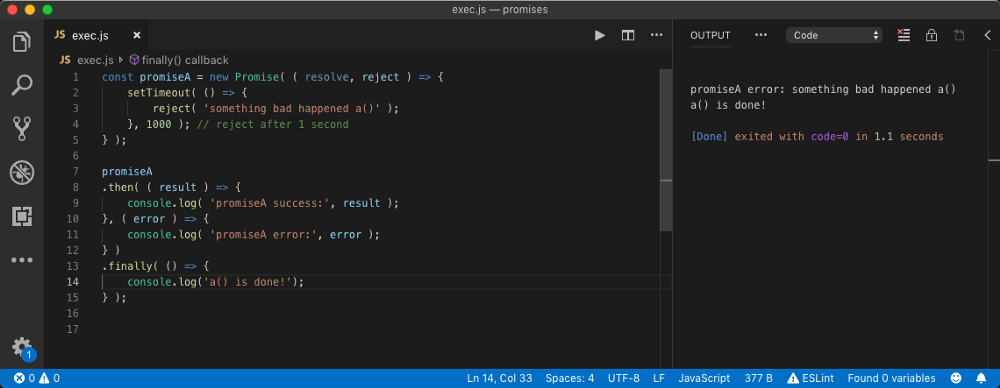


In the above example, we purposefully ****rejected**** the promise after 1000ms after which catch handler executed its callback and then finally handler.

## Handling promise rejection in then handler

You do not need to have a catch handler to handle promise rejection. The then handler takes the second argument which should be a function to handle the promise rejection.

## Ex 25:



But it is standard practice to use the catch method handler to handle promise rejection instead of using then method. If you use both catch and then method to handle promise rejection, catch handler will be ignored.

# Async/Await

Async/Await is a fancier syntax to handle multiple promises in synchronous code fashion. When we put async keyword before a function declaration, it will return a promise and we can use await keyword inside it which blocks the code until promise it awaits resolves or rejects.

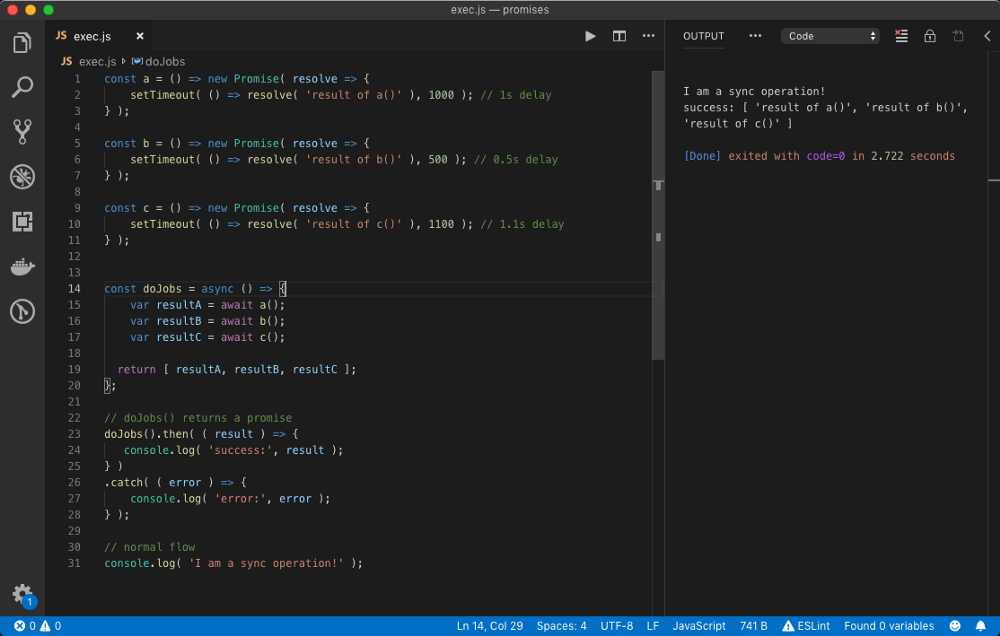
async function myFunction() {  
 var result = await new MyPromise();  
 console.log( result );  
}myFunction(); // returns a promise

In the above example, we have created a function myFunction which has async keyword on it. This keyword makes it asynchronous, which means when this function is called, a promise is returned and normal code execution will commence as usual.

We can say, await keyword inside a async function blocks the execution of JavaScript in that function context until the promise it is awaiting is settled. This gives us cleaner syntax to work with promises in a synchronous fashion.

Let’s take our previous example and use async/await style in it.

## Ex 26:



In the above example, we have returned an array of results of different promise resolutions inside the async function. This will be the payload of the promise async function returns if it fulfills.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*