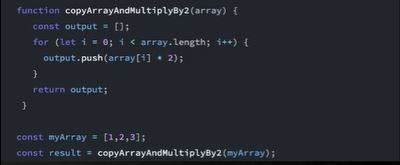
## Higher Order Function:

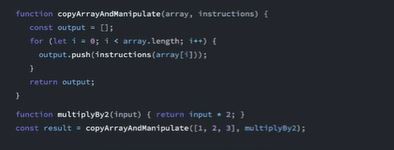
If two functions performs the same operation but slightly differ from the functionality being performed in both functions than that functionality can be passed as a parameter to the function and both functions can be combined into one and that functionality can be passed in the resulting function. This concept is known as higher order functions.

Example: Let us see two examples given below

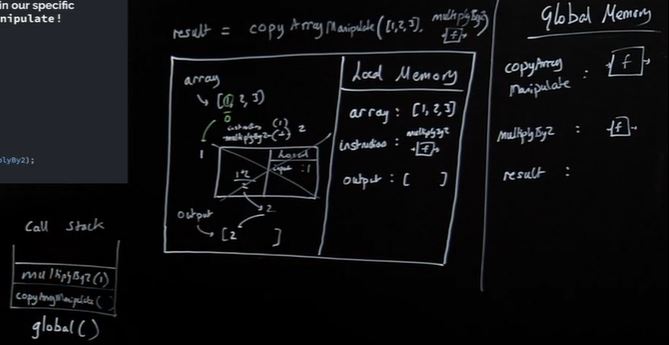




As we can see from the examples given above that most of the code is same but only the + and \* functionality is changed in the function. So we can pass this functionality in a function and pass that function as a parameter in our new generalized function.



To see how it works behind the scenes let us visualize the first iteration in the for loop and how the parameters are stored behind the scenes:



As we can see from the above image that inside the copyArrayAndManipulate function the instructions is referred to multiplyBy2 and whenever we encounter the instructions parameter we will replace it by multiplyBy2

### Note:

i) Here what we are doing with each element of the array is determined by a function defined outside the current function which is looping through the array.

ii) It is a good practice to not change any parameter pass into the function unless it is the intended functionality. If the functionality is achieved by manipulating the parameter and returning a new one than that approach should be taken.

So what we observed here is that:

Functions in javascript - first class objects

They can co-exist with and can be treated like any other javascript object

i) Assigned to variables and properties of other objects

ii) Passed as arguments into functions

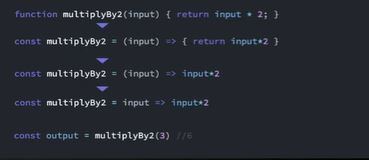
iii) Returned as values from functions

So from the above examples which is our Higher order function and which is our callback function?

Higher Order Function: The outer function that takes in a function in our higher-order function or the function that takes in a function or passes out a function i.e. copyArrayAndManipulate() function

Callback Function: The function we insert is our callback function: multiplyBy2() function

### Arrow Functions: A shorthand way to save functions



Here on the third line we see that the return along with the curly braces are removed and our function works fine without them, this is because if our function does only one thing i.e. return a value then we can skip writing the **curly braces** along with the **return** keyword. Moreover, in line four we see that the parenthesis are also removed from the input function parameter, again this is a design choice made by JS Designer i.e. if our function takes in only one parameter then we can skip **parenthesis**

So ultimately our higher order example we saw above can be rewritten as given below:



Anonymous/arrow functions:

Improve immediate legibility of your code.

But at least for our purposes here they are “syntactic sugar” - we’ll see their full effects later

Understanding how they’re working under the hood is vital to avoid confusion

## Closure:

i) Closure is the most esoteric of JavaScript concepts

ii) Enables powerful pro-level functions like ‘once’ and ‘memoize’

iii) Many JavaScript design patterns including the module pattern use closure

iv) Build iterators, handle partial application and maintain state in an asynchronous world

Functions never remember anything from their previous running. Every time they are executed they run and makes new execution context. But, what if we want a function to have a **permanent** memory attached to it?

### Functions with memories:

i) When our functions get called, we create a live store of data(local memory/variable environment/state) for that function’ execution context

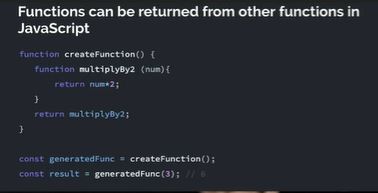
ii) When the function finishes executing, its local memory is deleted(except the returned value)

iii) But what if our functions could hold on to live data between executions?

iv) This would let our function definitions have an associated cache/persistent memory

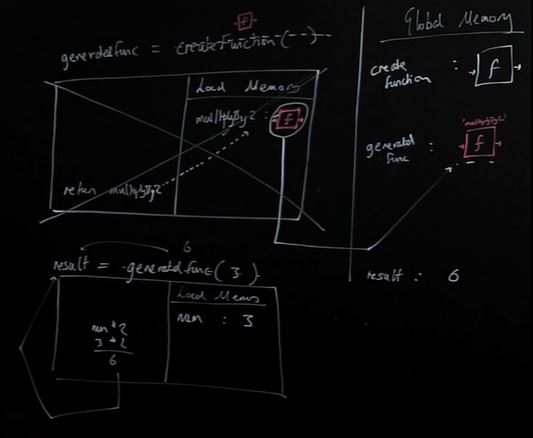
v) But it all starts with us **returning a function from another function**

We will carry on to the notion of returning a function from another function and see this from our example given below:



Here when we are executing the createFunction() it will create a new execution context and the multiplyBy2() function defined inside the createFunction() will be store in the createFunction() execution context. But, here we are returning the multiplyBy2() function and storing its value into the generatedFunc which is stored in the global memory. Thus, we have stored the multiplyBy2() function(now labelled and accessed under the name of generatedFunc()) into the global memory. So, generatedFunc is really gonna be multiplyBy2.

Let us look the execution of the code above behind the scenes:

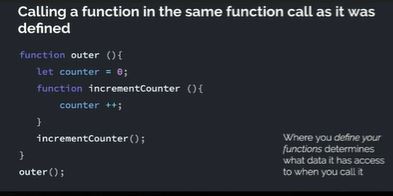


Here generatedFunc is the result of running createFunction(). And after execution the generatedFunc will be the function definition of the multiplyBy2 as we returned it in the createFunction(); So basically we can run **function definition** of multiplyBy2 **globally** but under the label **generatedFunc.** So the generatedFunc has the multiplyBy2 definition stored in the global memory and it has no relationship with the createFunction() after the line:

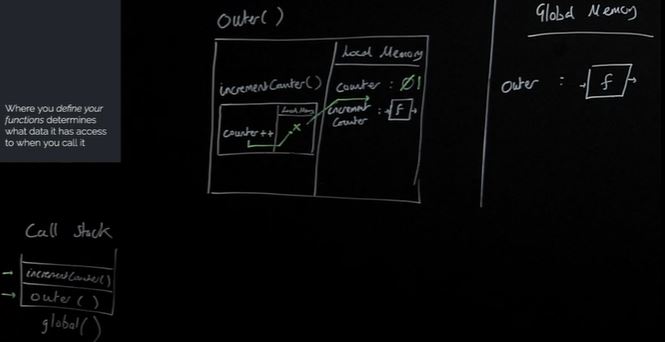
**const generatedFunc createFunction();**

There might be a question that why we defined the multiplyBy2() function inside another function only to return that function and not define the multiplyBy2() function globally. Well, we will look at the benefits of using this method to return the function and delve into the closure concept of JavaScript.

Now let us understand another concept before we learn more about closure:

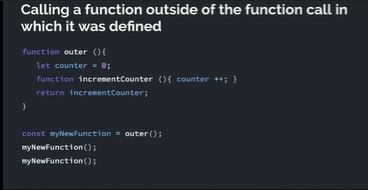


Here we will see how the above code works behind the scenes:

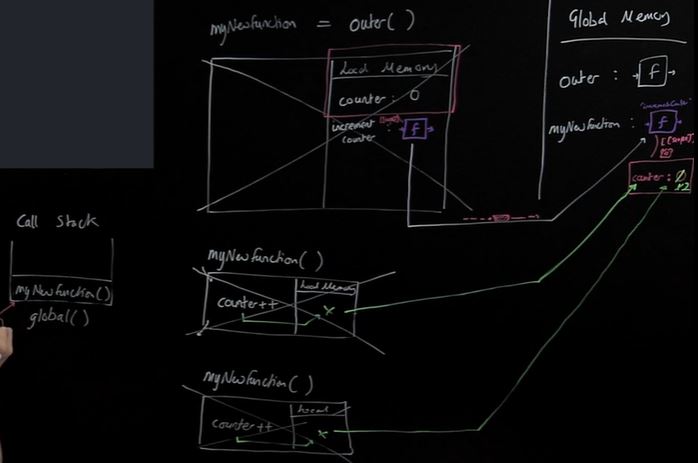


Here from the above code we **can’t** really tell if the counter++ line that is executed inside the incrementCounter() function has access to the counter variable defined in the outer() function because it is **executed** inside the outer() function or if it’s **defined** inside the outer() function. So let us figure that out

Let us rewrite the code given above to have more clear understanding and resolve the confusion we saw above about the counter variable as given below:



Here let us visualize the above code and see how it works behind the scenes:



Here we can see from the image above that when the incrementCounter got returned from the outer() function it not only returned the function definition but also returned all the **surrounding** code along with it. So when we save the incrementCounter function to the myNewFunction we not only saved the function definition of the incrementCounter but also saved the attached **backpack** to it from when that function was born(i.e. myNewFunction) with live data(i.e. Data in backpack). And in that backpack the counter is zero.. So now our myNewFunction stored the function definition and the backpack.

So, from now on if the myNewFunction could not find the counter in the local memory it will look into the backpack thereafter. So the function myNewFunction has some permanent memory(in the form of a backpack) attached to it in the global memory which we can use and manipulate.

### Let us understand some caveats around this javascript functionality:

i) The only way to access the data in the backpack is to execute the function and manipulate the data that is stored in the backpack. We can’t access the backpack directly. So now we have permanent/private data which no one can access besides by running the function we created to get access to it.

ii) Here if we were to declare a new variable in the outer() function’s local memory and if that function is not referred to in the incrementCounter function than that variable will not be stored in the backpack, because javascript knows that there is no way to access that variable as it is not referenced in the returning function(i.e. incrementCounter)

Ex:

// In the above code if the outer function is redefined as below

function outer() {

let counter = 0;

let notInBackPack = 0;

function incrementCounter() {

counter++;

}

return incrementCounter;

}

// Here in the above code the returned incrementCounter will not save the notInBackPack variable in the backpack as it is not referred anywhere in the incrementCounter function. This applies to any kind of data that is not referred to in the returning function.

Sometimes this backpack is also referred to as C.O.V.E (Closed over variable Environment). We can also call the backpack as P.L.S.R.D (Persistent Lexically Scope Referenced Data). Industrially this backpack is known as closure (i.e. function’s closure is function’s backpack which is myNewFunction’s backpack in above example)

So the ‘backpack’ (or ‘closure’) of live data is attached incrementCounter(then to myNewFunction) through a hidden property known as [[scope]] which persists when the inner function is returned out.

### Multiple Closure Instances:

Here running the outer function again will create a new backpack in the example given below:

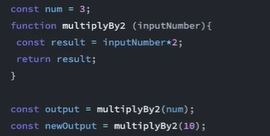


Individual backpacks: If we run ‘outer’ again and store the ‘incrementCounter’ function definition in ‘anotherFunction’, this new incrementCounter function was created in a new execution context and therefore has a brand new independent backpack

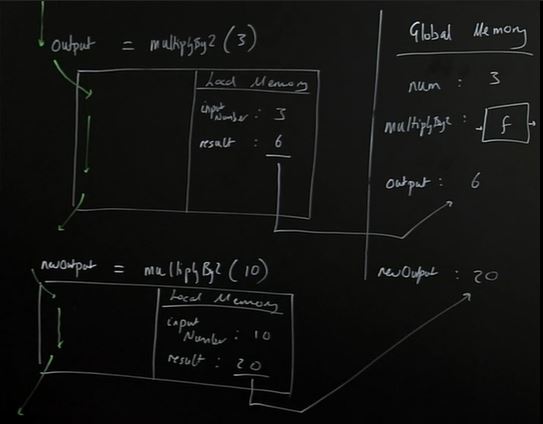
## Asynchronous JavaScript:

JavaScript is synchronous language but the asynchronicity can be achieved by making certain tweaks on the single thread of execution in a JavaScript program. Before we move let us remind ourselves how JavaScript executes code:

Example:



Now let us see how the JavaScript executes this code behind the scenes:



Asynchronicity is the backbone of modern web development in JavaScript yet:

JavaScript is:

1. Single threaded (one command runs at a time)
2. Synchronously executed (each line is run in order the code appears)

So what if we have a task:

1. Accessing Twitter’s server to get new Tweets that takes a long time
2. Code we want to run using those tweets

Challenge: We want to wait for the tweets to be stored in tweets so that they’re there to run displayTweets on - but no code can run in the meantime

Now let us look at another example and try to determine its output:



Here we would think that the printHello function will execute after waiting for 1s(1000ms) and log “Hello” to the console and thereafter it logs “Me first!” to the console, but in reality it prints “Me first!” on the console and then it prints “Hello”

Now let us look at a different example:



Here we would think that it would log “Hello” first to the console as there is the delay of only 0 ms in the setTimeout() function but in reality it prints “Me first!” on the console and then it prints “Hello”

Now we will look at the other aspect of the JavaScript web browser:

JavaScript is not enough - We need new pieces(some of which aren’t JavaScript at all)

Our core JavaScript engine has 3 parts:

1. Thread of execution
2. Memory/variable environment
3. Call stack

We need to add some new components:

1. Web browsers APIs/Node background APIs
2. Promises
3. Event loop, Callback/Task queue and micro task queue

Web Browser contains different tools apart from JavaScript like: Devtools console, Sockets, Network requests, HTML DOM, Timer

There are some functions in JavaScript which can be termed as **facade** cause they look like they are the JavaScript functions but they are the features provided by the web browser.

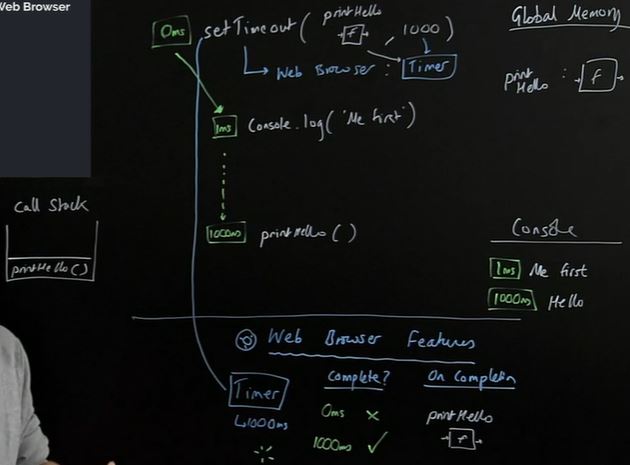
So JavaScript has different functions to access different features provided by the web browser as given below:

Devtools console(console), Network requests(xhr/fetch), HTML DOM(document), Timer(setTimeOut)

Let us look at an example:



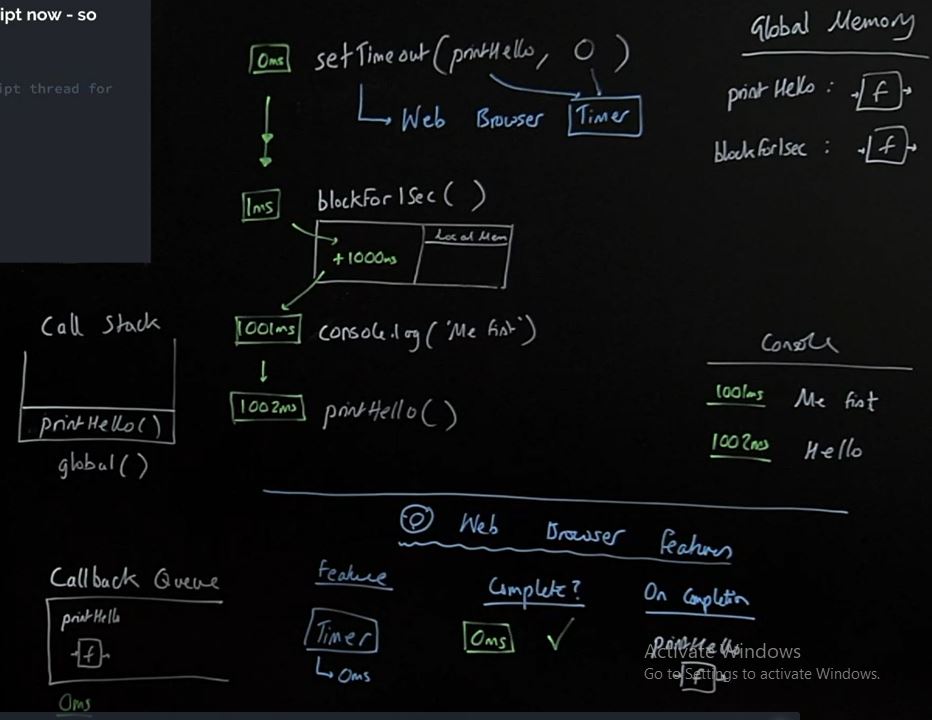
Here we will see what happens behind the scene when the code above executes:



Here we would argue if the setTimeOut() function has 0ms and then if we execute the code just by changing the time from 1000ms to 0ms, then we would print “Hello” first followed by “Me first” **but** that is not the case. So in order to determine when the code from the setTimeOut() function will run and when the code from the JavaScript current thread of execution in a program will run we will go through some of the rules.



Let us see how the above code works behind the scenes:



When we are dealing with the world outside of the JS the callback queue comes into play. In the code above, after the setTimeout() function is executed and ready to print the printHello() function it will wait in the callback queue after the current thread of execution is finished i.e. after the program has finished execution. After the code is executed i.e. after printing the “Me first!” on the console the awaiting function in the callback queue i.e. printHello() will be executed

### **Note**:

JavaScript checks if the call stack is empty i.e. is there any global code to run if there is then it will execute that first and thereafter it will execute the function’s in the callback queue.

So the feature that checks at every line of code to see if the global call stack is empty and if the callback queue should be executed is known as “Event Loop”. So this is how the Global Call Stack, Callback queue, and Event Loop worked till ES6.

ES5 Web Browser APIs with callback functions:

Problems:

1. Our response data is only available in the callback function - Callback hell
2. Maybe it feels a little odd to think of passing a function into another function only for it to run much later

Benefits:

1. Super explicit once you understand how it works under-the-hood

## ES6+ Solution (Promises)

Using two-pronged ‘facade’ functions that both:

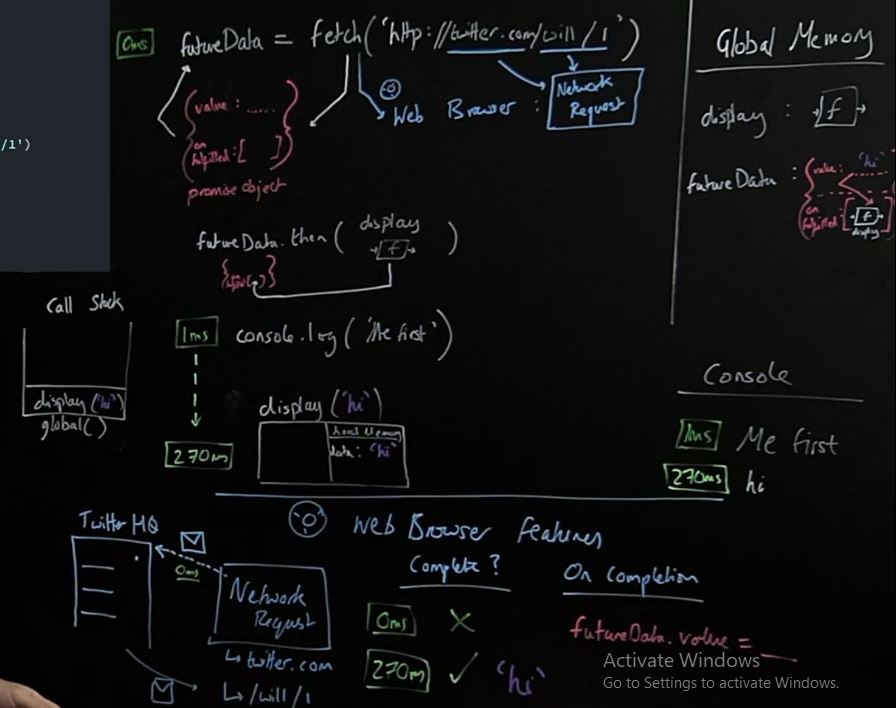
i) Initiate background web browser work and

ii) Return a placeholder object(promise) immediately in JavaScript

Let us look at an example:



Let us visualize this how it works behind the scenes:



The fetch() function will return out the promise object. The promise object contains two properties i.e. **value**, **onFullFilment**. So the fetch() function has two consequences one in the JavaScript i.e. the object returned and the other consequence in the web browser where it fetches the data. So before we used to use the xhr function to request data but after ES6+ we use the fetch function to fetch the data from the internet. So the fetch function uses the Network Request feature of the Web Browser. So these two consequences are intimately linked to each other such that when the request is made and the **response** is available we can access the response in the **value** property of the **promise** object which was initially **empty**.

So we do not know when the data will be available after the request through the fetch() function is made over the internet. We do know that the data will be stored in the value property of the promise object, but do not know when the data will be available.

So, here is when the onFullFilment property comes in cause when the data will be available in the value property the JS will run the **onFullFilment()** property of the promise object. So any function we put on the **onFullFilment** array will **run** as soon as the **value(response)** is **available**. Moreover, the data in the value property will be **automatically available** as a **parameter** to **any function** present in the **onFullFilment()** method of that object.

So in the above example the display function has a data parameter automatically available to it which contains the data stored in the value property of that object. But, we **cannot** push a function directly to the onFullFilment() method because it is the **hidden property**. So, the then() function will put the given function into the onFullFilment(hidden property) of the promise object which will execute when the response is available on the value property of the promise object.

### *then* method and functionality to call on completion:

Any code we want to run on the returned data must also be saved on the promise object

Added using the **.then** method to the hidden property **‘onFullfilment’**

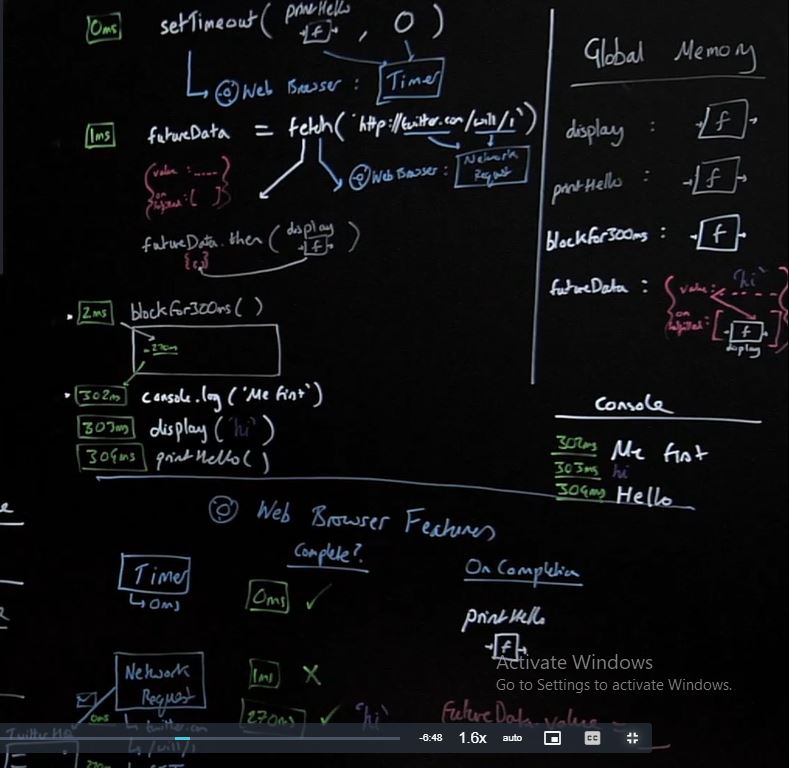
Promise objects will automatically trigger the attached function to run (with its input being the returned data)

Let us look at the other in order to understand how the callstach works involving the promise functions:



Here we have two functions setTimeout() our old function where we do not have control to track the data and our promise function(fetch) where we can track what we can do when the response is available.

Let us visualize this in order to figure out what is happening behind the scenes:



Here we can see that when our **setTimeout()** function is ready it will push the printHello() function to the **callback/task queue**. But, what’s important to note here is that when the response is available in the value property of the promise object and the function display is ready to run while the function **blockFor300ms() is executing** in the JavaScript the ready-to-execute **display function** will wait in a new queue called the **microtask queue**.

### Note:

So, the important thing to note here is that the **Event Loop** executes the current global queue i.e. **the global execution context stack first** and then they will run the **Microtask queue** and at last the **Event loop** will run the **Callback queue**.

In sum, we have two queues. Any function that is executed via the **old-school facade** function i.e. setTimeout() will **wait** in the **callback** queue when they are ready to be executed(when the background work is completed). While any function that is attached to a **promise object** via the **then** method and then **auto triggered** to run within **JavaScript** when the **response** is available via the **two pronged** function like **fetch** will wait in the **microtask** queue.

**Execution priority:**

**Global code(stack) → Microtask queue → Callback queue**

The part where the JavaScript uses the web browser to perform background tasks are known as the APIs and the link to MDN where we can find different APIs available to JavaScript is [here](https://developer.mozilla.org/en-US/docs/Web/API)

So in order to determine if a function is going to the **microtask queue** or the **callback queue** we can determine that by looking if a function takes in another function directly like **setTimeout()** function we can say that the input function will end up in the **callback queue**, but if our function is to two-pronged function which return a **promise object** it will end up in the **microtask queue** i.e. the function given as an input to the **then() function**

### Promises (Benefits):

Cleaner readable style with pseudo-asynchronous style code

Nice error handling process

**Note**: If we have any kind of error from the request made through the **fetch()** function it will not auto-trigger any of the function present in the **onFulfilment** method, but there is another property associated with promise object i.e. **onRejection** and as it is the **hidden property** JavaScript gives us the **catch** function to put function in that property, which will eventually get auto-trigger if an error occurs while making the request. On the other hand, there is another way to auto-trigger a function if an error occurs i.e. passing a second argument to the **then** function

### Summary:

We have rules for the execution of our asynchronously delayed code:

1. Hold promise-deferred function in a microtask queue and callback function in a task queue (Callback queue) when the Web Browser Feature(API) finishes
2. Add the function to the call stack (i.e. run the function) when:

* Call stack is empty & all global code run(Have the Event loop check this condition)

1. Prioritize function in the microtask queue over the callback queue

## Classes, Prototypes - Object Oriented JavaScript:

* An enormously popular paradigm for structuring our complex code
* Prototype chain - the feature behind the scenes that enables emulation of OOP but is a compelling tool in itself
* Understanding the difference between \_\_proto\_\_ and prototype
* The new and class keywords as tools to automate our object and method creation

Functions on objects are known as **“methods”**

There are number of ways to create an objects like given below:



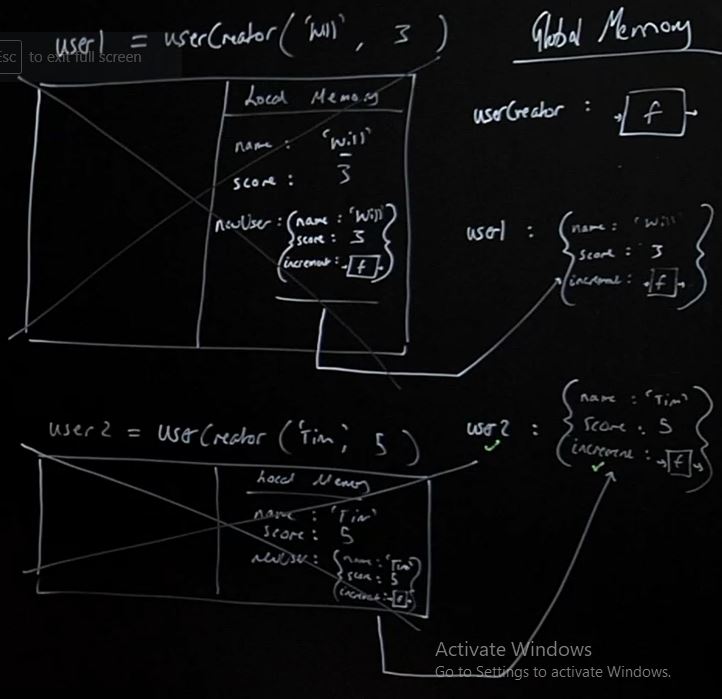




### Solution 1: Generate objects using a function



Let us visualize how the code above works behind the scenes:

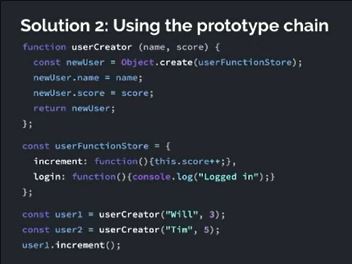


The code above is not efficient as it creates a new function i.e. increment() function every time an object is created. We need to look at an approach where only the parameters and parts of the function are changed and the same function is referenced among all the objects created. Moreover, if we want to add a new functionality we need to add to each user object.

So, we need a way in which all of our objects refer to the same function defined only once.

### Solution 2: Using the prototype chain

* Store the increment function in just one object and have the interpreter, if it doesn’t find the function on user1, look up to that object to check if it’s there
* Link user1 and functionStore so the interpreter, on not finding increment, makes sure to check up in functionStore where it would find it
* Make the link with Object.create() technique



Let us visualize what is going on behind the scenes:

In the above example **this** is an implicit parameter that is always available on the method. And **this** parameter always refers to the object on the left which access the method using the **dot** notation

In the above example the Object.create() does in fact create an empty object, but note that we have passed in the userFunctionStore as the parameter which will be linked to the hidden property of the object through the \_\_proto\_\_ and it has stored in it the link to userFunctionStore, so while accessing the increment method, if it does not find the link in the current object it will look up the proto property which has a link to it.

So, it is the feature provided by the JavaScript, when it does not find a given property on an object it does not panic, in fact it looks up to its **\_\_proto\_\_** property where we can find the userFunctionStore

So the argument that the Object.create() receives is always stored in the \_\_proto\_\_ property

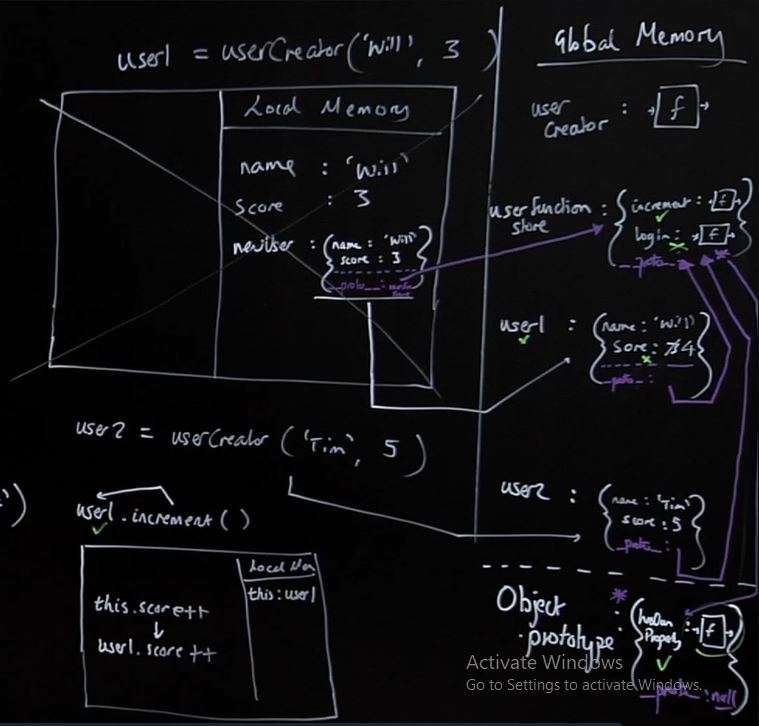
Note that this hidden property we can in fact see in our browser if on our chrome devtools we look up the object:



Moving on , what if we want to confirm our user1 has the property score in the above code. So let’s see:



Let us visualize this below:



So there is a big **Object.prototype** that is available to **all** of our objects. Moreover, this Object.prototype is available to all the objects through the **\_\_proto\_\_ property**. So if we create any object it has the **default** hidden(accessible in chrome dev tools) **\_\_proto\_\_ property** available to it, which points to the **Object.prototype**. So the hasOwnProperty is a property of Object.prototype. So what we are doing with our **Object.create()** is **taking control** of the \_\_proto\_\_ property. Moreover, the **Object.prototype** also has a **\_\_proto\_\_ property** but it’s **null**

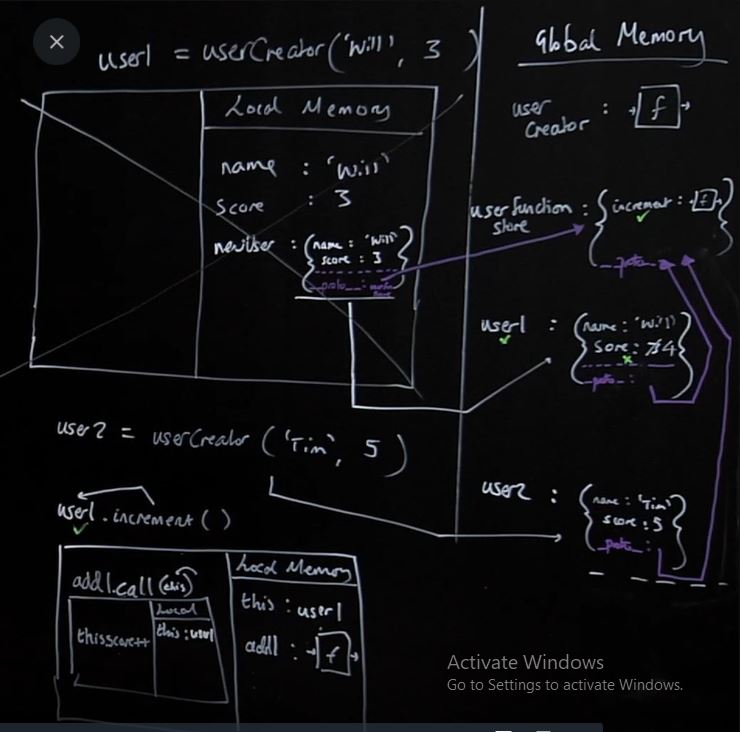
Now let us understand about **this** keyword by modifying the above examples as given below:



Here what if we want to add another method inside our increment function. So let use consider another example in fact which has this situation and see how **this** keyword is to be used:

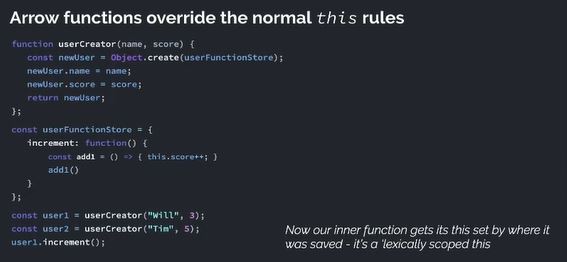


Let us visualize how this works behind the scenes:

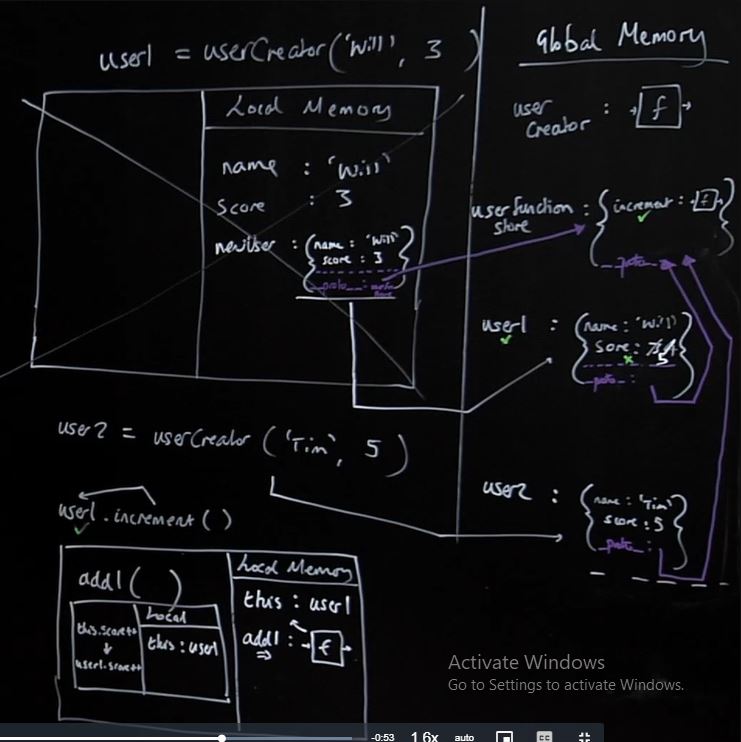


But the problem here is that the **this** implicit parameter that is by default available to the userFunctionStore is not by default available to the add1 function. So **this** in the add1 function refers to the global windows object which is **undefined**

Now we will see the arrow function style as it is lexically scoped it’s function stores all the data when it was defined. So here when it was defined this referred to the user1 which will preserve the state of this when the function is called. Let us see that example below:



Let us visualize how this works behind the scenes:



Here in the above example we can see that now the **this** keyword inside the add1 function works as intended. Note that we should **not** declare the increment function as the arrow function. As, the increment function is defined globally the **this** inside the increment function will refer to the **this** in global and not **this** of the object.

**Note: Do not use the arrow functions on the methods of an objects, but we can use it inside the methods to define other methods as this is lexically scoped**

This is not the standard way to create the object, but after having this understanding we can now look at the next solution and be able to understand it better

Now let us see the **“new”** keyword. The new keyword automates lot of stuff we saw above and the things that are automated by using the **“new”** keywords is given below

### Solution 3: Introducing the keyword that automates the hard work - new

When we call the function that returns an object with new in front we automate 2 things:

1. Create a new user object
2. Return the new user object

Ex: const user1 = new userCreator(“Eva”, 9)

const user2 = new userCreator(“Tim”, 5)

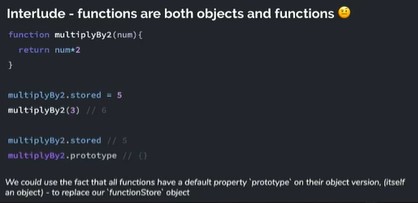
But now we need to adjust how we write the body of userCreator - how can we

* Refer to the auto-created object?
* Know where to put our single copies of our functions?

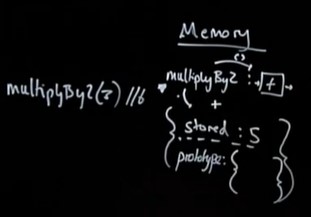
Let us look at the code we looked at previously in order to determine how the new keywords automates a lot of work for us. Let us look at the screenshot below:



But before we delve deep into it let us look at an interlude of javascript i.e. **functions are both objects and functions**



So let us visualize this below:



We should note here that we can access the function part of a function through the **parenthesis** and the object bit through the **dot** notation

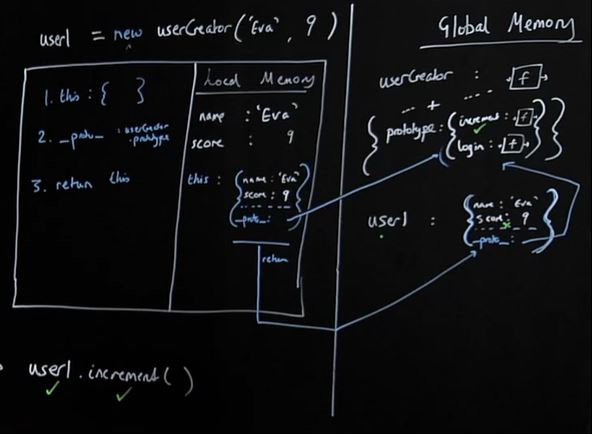
We should also note that all the functions have the default **prototype** property that is available on the object part of all the functions and it is not a hidden property. This property is an **empty** object.

So the question is after declaring the new keyword where we will keep the shared functions that are accessible by all the objects. The answer is that we will be declaring the **shared functions** or **property** that is **accessible** by **all** of the **objects** in the **prototype property** of the **function** before which we use the **new** keyword

Let us again understand this concept which the help of an example given below:



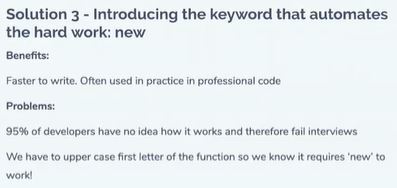
Let us visualize this below:



Here we answer the question as to where we will keep all the shared functions that are available to all of the userCreator objects. So, the answer to the question is that we will be defining all the functions on the **object** part of the userCreator function’s **prototype** property i.e. **increment** and **login** function in the above example.

So finally the consequence of adding the new keyword before the function results in the following three changes:

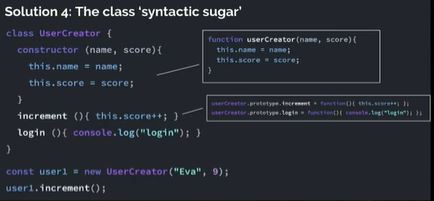
1. The **new** keyword defines an **empty** object
2. The **new** keyword uses **this** keyword which refers to the **empty** object
3. As all the object in the javascript has the **hidden \_\_proto\_\_** property, the **hidden \_\_proto\_\_ property** of the **empty** object refers to the **prototype property** of the **object part** of the **userCreator function**
4. Finally the **new** keyword **returns** out the object i.e. **returns this** or **the object assigned to this**



So as we can see from the image above that most of the time we do not have any idea if a function requires a new keyword before calling. So the developers made the decision to add the class ‘syntactic’ sugar in order to remove that ambiguity and let us know as developers where we can use the new keyword. So we are progressing from the **function** to **class** but under the hood it’s working the **same** but now we see that defining a class gives us **explicit information** that when calling this you are **required** to use the **new** keyword before.

### Solution 4: The class ‘syntactic sugar’

We’re writing our shared methods separately from our object ‘constructor’ itself (off in the userCreator.prototype object). Other languages let us do this all in one place. ES2015 lets us do so too



So before we created the function and object part of a function separately and now we have used the class keyword to have a function + object combo where we can initialize and object and access it’s function part as well using the new keyword. So the constructor is the function bit of the function + object combo, then all the function i.e. increment and login listed below the constructor are in the object part of the combo and it will store it in the prototype property of the object bit of the function + object combo

