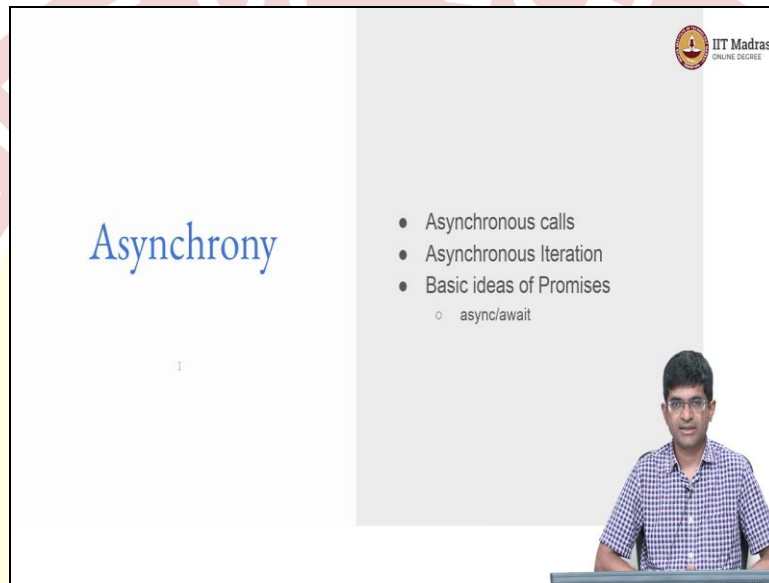


IIT Madras
ONLINE DEGREE

Modern Application Development II
Online Degree Programme
B. Sc in Programming and Data Science
Diploma Level
Prof. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology- Madras

Javascript - Asynchrony


(Refer Slide Time: 00:14)



Hello everyone welcome to modern application development part 2. One concept that is quite important to understand when you are using Javascript is this notion of what is called asynchrony. So, asynchrony basically means that multiple different events or tasks or things can happen without necessarily being tightly synchronized with each other. In other words there is no direct time relationship between them.


What exactly does that mean and why does it why is it important for us to understand that in the context of Javascript is what we look at now.


(Refer Slide Time: 00:48)


IIT Madras
ONLINE DEGREE

Function calls

- Function is like a "branch"
 - but must save present state so we can return
- Call stack:
 - Keep track of chain of functions called up to now
 - Pop back up out of stack





So, let us first look at how a function called is treated in any programming language this is not specific to Javascript. Let us say I have defined a function square of x which basically takes in a number x and is supposed to return the square of that value you can think of the function as basically some kind of a branch. So, what do I mean by a branch essentially all that I am saying over here is that there is a certain state of control flow.

I mean I have been executing the instructions in the program one by one I come to the function call and the details of what to do inside the function are not stored in the place where I am right now they are stored somewhere else. In such a way that that code can be reused I can call the function many times after defining it once. But how that is implemented in practice is that you effectively in the machine language you execute a branch.

You need to jump or branch to that new location and start executing the code over there. Now at the machine language level or assembly language level you pretty much only have branches. So, you have the ability to basically say you know take the next instruction execute it take the next instruction execute it or go somewhere else. Now go there take the next instruction over there and execute it and then again start from there right.

So, a function is pretty much like taking a branch the only difference between an arbitrary branch and a function call is that I should be able to come back to where I was before. And at the hardware level we do not really sort of provide fundamental support

for that there are certain hardware processors that may provide good support for sort of these branches with return capabilities.

But in the most part that is not really needed at the assembly language level I just need to provide the ability to branch go to a new place and start executing instructions. And the software or the compiler or whatever it is that is actually executing the code can take care of this business of remembering where the function was called from and where it needs to come back.

How that is done is that the present state of the system. What do we mean by present state? Inside a processor the most important variable that defines state is what is called the program counter. In other words it is something which says where is the processor at the moment which memory location is it reading instructions from. In addition to that it might also have a few more things which are some of the you know intermediate register values being used in the present function.

Now the program counter plus some of those things like there are certain values that need to be stored like you know what are the values of certain registers that need to be saved and so on all of those together form something called a stack frame. And this concept of stack frames is something that is common across pretty much most programming languages right.

And what happens is that every programming language the runtime of the language maintains something called a call stack which means that whenever I have a function call the present program counter and any other register values that you want to save get pushed onto that call stack. Meaning that they are just stored into certain locations in memory and we use a specific data structure called a stack which you should be familiar with from a data structures course.

Now what does the stack do the basic idea behind a stack is that we have some locations and what we do is that let's say that my main function is what is currently at the bottom of the stack the moment I call a new function right I sort of create some new space after all its just some memory. So, I need to sort of move some pointers around move the values of pointers around and say that now I am executing the function f.

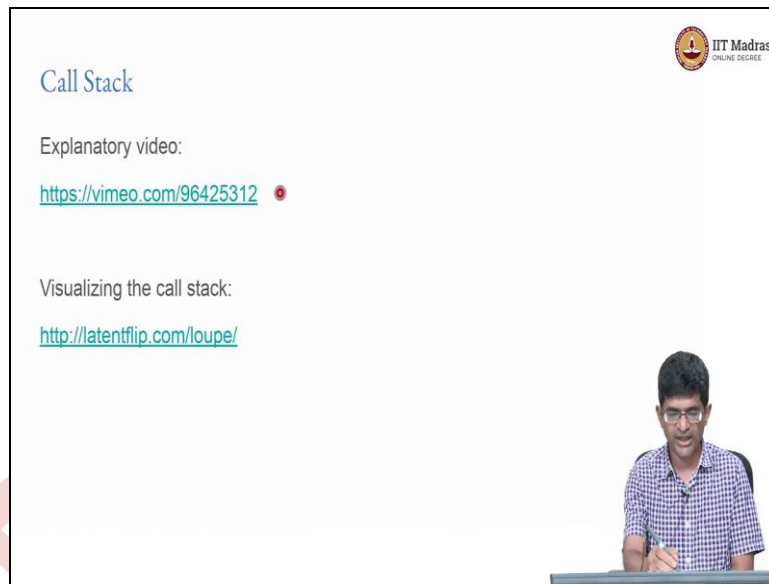
Let us say that the function f in turn executes another function g I go further up over there and that function g in turn calls a value h or calls a function h each of these is what's called a stack frame right and in this case what we are saying is we are essentially nested four levels deep I am inside the function main I started from here I then go on to call the function f that in turn calls g that calls h.

So, I have gone through like four levels of nesting. So, that I am four levels deep inside a function. How do I get back to main I cannot just dump all of these other things because I first need to return to wherever I was in g before I called h and then execute from the next instruction after h. Once I am done with g I exit and go back to f and continue with f. Once I am done with f I exit and go back to main and continue with main.

Normally once I am done with main as well I should exit the program altogether and then go back to you know the operating system or whatever it was that triggered this in the first place right. So, this call stack in other words is used in order to keep track of the chain of functions that have been called up to any point in the execution of the system. And as we can see over here what happens is that you know I have these four functions main then f then g then h I return from h pop out into g return from g pop out into f return from f pop out into main.

And at some point of course I would also exit from main but that is will go back to the operating system and you can ask what happens when you exit the operating system you basically shut down the machine that is ultimately what the operating system is doing is just running a loop where it never exits. The important point here is that function calls act like branches and we need to use this call stack in order to save the state of the branch. So, that we know where to come back.

(Refer Slide Time: 07:13)



Call Stack

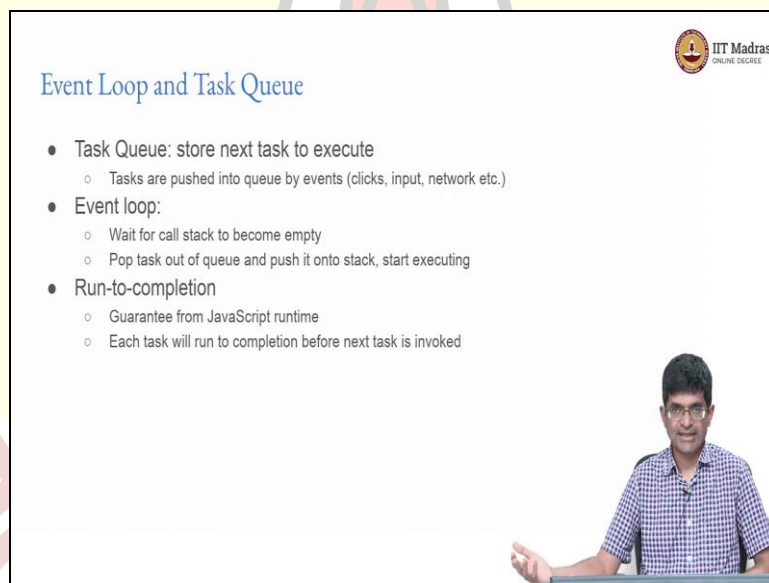
Explanatory video:
<https://vimeo.com/96425312>

Visualizing the call stack:
<http://latentflip.com/loupe/>

The slide features the IIT Madras logo in the top right corner. A presenter is visible in the bottom right corner of the slide frame.

Now there is a very good explanatory video on this which uses this concept called visualizing the call stack which we can briefly take a look at.

(Refer Slide Time: 07:27)



Event Loop and Task Queue

- Task Queue: store next task to execute
 - Tasks are pushed into queue by events (clicks, input, network etc.)
- Event loop:
 - Wait for call stack to become empty
 - Pop task out of queue and push it onto stack, start executing
- Run-to-completion
 - Guarantee from JavaScript runtime
 - Each task will run to completion before next task is invoked

The slide features the IIT Madras logo in the top right corner. A presenter is visible in the bottom right corner of the slide frame.

So, now we know what a call stack is right and we can sort of visualize what is the pattern that is going to through a call stack primarily the call stack only builds up and gets multiple entries when I have nested function calls one inside the other. But in addition to that Javascript also has something else called an event loop and a task queue. This task queue is it's not a stack as the name suggests it is a queue.

So, what does that actually mean it basically means that I can think of a set of tasks that are sort of being pushed into a queue there is d1 d2 d3 and so on. What are these tasks they could be functions. So, they could be something which basically says do something

in the code and each one of these when t1 is ready it then goes on to the call stack, t1 in turn could invoke something else which continues on the call stack that in turn could call something else.

Note that while t1 is on the call stack t2 and t3 are waiting there is nothing else happening over there. Only after all of this has sort of emptied out and the stack has become empty. Do I pick up the next value t2 and push that onto the stack. So, in other words the call stack is maintaining the functionality within a given function call but once all the functions have been finished and have exited.

Obviously in this case I am not talking about main as being the function I am considering that there is some notion of tasks and the task goes on to the call stack invokes other functions exits all of them and then finally leaves the call stack. At that point the Javascript runtime goes back and looks for the next task to execute t2 comes onto the call stack in turn calls other functions they also exit t2a goes away t3 comes on to the call stack and so on.

So, what is happening over here where are these t1 t2 t3 coming from they are coming either from some internal notion of what is generating tasks or possibly from events right. It could be events that are being triggered somewhere maybe a mouse click something being typed into a text box or a network request let us say that I tried to fetch a page from somewhere the network request finally returns and that causes a new task to be pushed onto the task queue.

And that is an important thing to think about what happens when I make a network request in Javascript is that there are two ways of handling it either immediately the runtime has to you know just freeze until the network request works or it can just push it onto the task queue and rather send it off to the runtime. And the runtime then waits until the network result has come back and then pushes.

The new task to process the network result onto the task queue that is a very sort of important idea to keep in mind. Effectively what is happening over here is we have this event loop which is first of all waiting for the call stack to become empty and then it

pops a task out of the queue and pushes it onto the stack and starts executing. Where are these tasks coming into the queue from events.

Those events could be user interaction network or other kinds of things which are basically external to the Javascript runtime itself or related to the runtime but not part of the execution of the code itself. And effectively what this means is that this is the key to what we will later use in terms of asynchronous processing. Now there is one reason why this whole thing becomes very important to keep in mind which is that Javascript guarantees something called run to completion semantics for any task.

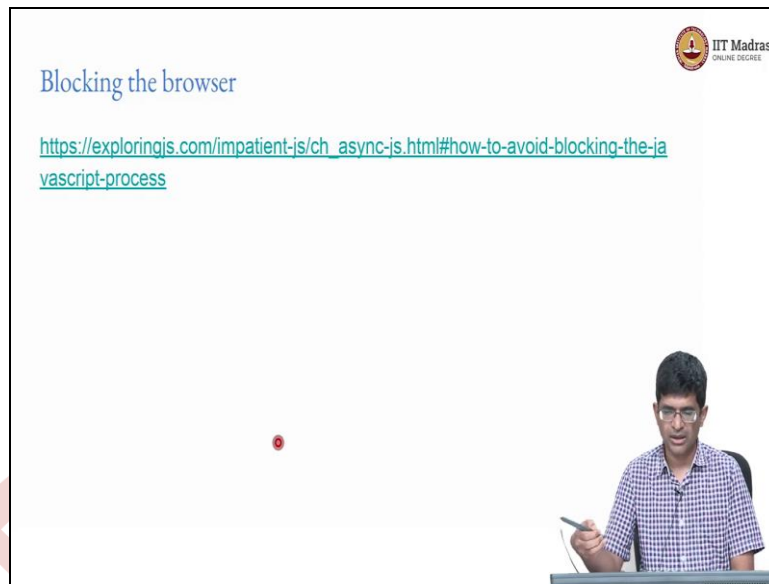
What that means is once a task has made it onto the call stack the task has to complete before the next task can be fetched think about that for a moment what it means is that if I somehow manage to get one task into the task queue. And the task you that task then gets on to the call stack and inside that if I write a function which basically says you know just sleep right.

Or just keep chugging away write a for loop for one to whatever is the biggest number that the system can handle pretty much you are going to make the system hang. Because until that task completes Javascript will never go and fetch the next task and that is in fact true it is quite possible to crash a system just by writing poor code which does not sort of you know give up its position for others to execute.

On the other hand because of the way by which these kind of tasks are handled Javascript also has the ability to say that look if I have a long running task somewhere then I will just hand it off to somebody else who has to finally push a task back into this queue and therefore I will be able to clear out this stack allowing the next task to be processed.

This requires a little bit of thought to understand clearly there will be like multiple examples that we do come across at some point. So, it is worth sort of thinking about as we move forward.

(Refer Slide Time: 13:14)



A sort of simple example of this you know is given on this page exploringjs dot com right and what we can see over here is that in fact this exploringjs.com has quite a I mean a very good review of Javascript and a good explanation of most of the behaviour. Along with that they also have you know an example out here which sort of demonstrates this whole business of what it means to block a particular thread in Javascript and we can run this code.

(Video Start: 13:29)

So, what we can see is there is this code which you know it is not particularly interesting by itself but what we can see is that if I click on some on the button over here you can see that each time I click on the button there is some kind of a response. So, there is this change in colour it clicks and so on. But now what happens if I click on this block for five seconds. Now during that time the whole thing says blocking and in fact even the cursor has not changed in shape until the done comes.

At which point I can go back and once again click the button. So, how do we understand what exactly happened over here the thing is when I click this block for five seconds it is calling a Javascript loop which is just basically going to sit over there for five seconds checking the current date and seeing whether five seconds have elapsed or not until that time it does not let go of the task.

In the context of our of what we saw with regard to the call stack what it basically means is that one task has come on to the call stack and it is just spinning over there checking

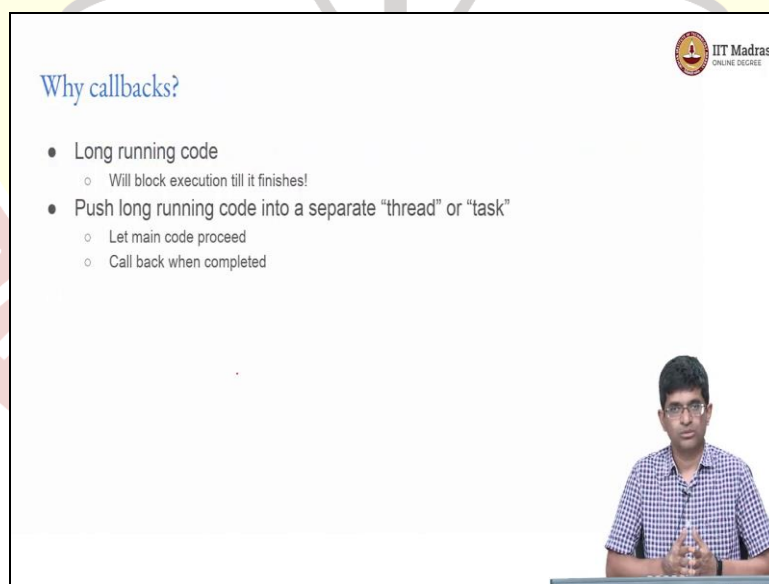
the current date and time and seeing whether five seconds have elapsed until that time nobody else can move. No function can proceed the tasks cannot proceed even if you try clicking somewhere on the browser the system will simply not respond.

Now you might have seen this in multiple cases eventually the browser sort of pops up a window saying the page is unresponsive what do you want to do wait or force quit. This is precisely what is happening something has happened over there which is basically running for in such a way that it is just freezing the browser. Now of course browsers have some ways of working around it they put some kind of a separate timeout which is completely outside of the Javascript runtime.

And that timeout sort of triggers something it is sort of like an operating system interrupt in some ways if you look at it. All that that is doing is basically going in there and telling the system look you have been waiting for too long is this a problem do we need to kill this page and that is pretty much the only way to get out of it if you do end up with a page that actually tries to hang like this.

(Video End: 16:16)

(Refer Slide Time: 16:17)



The video frame displays a presentation slide with the title "Why callbacks?" in blue text. The slide contains two main bullet points, each with sub-points:

- Long running code
 - Will block execution till it finishes!
- Push long running code into a separate "thread" or "task"
 - Let main code proceed
 - Call back when completed

In the bottom right corner of the video frame, a man with glasses and a checkered shirt is visible, sitting at a desk and speaking. The IIT Madras logo is in the top right corner of the slide.

So, now all of this brings us to this notion of something called a callback. And a callback is used precisely to handle this kind of scenario. Let us say that we have a long running code which is not just some kind of a for loop which is not doing anything useful but is actually something that is important for us. Examples nowadays could even be something like you know you have a full machine learning an inference application

maybe a tensorflow dot js you have implemented an entire machine learning deep neural network.

And the inference of that is taking long, long meaning it might even be a few seconds but from the point of view of the Javascript it means that the page is going to be unresponsive until it finishes. Another sort of very common example and in fact one of the basic reasons why Javascript became popular in the first place is. I want to update part of a page and I do that by fetching information from somewhere else on the internet.

Now fetching depends on a lot of things it depends on whether the remote server is up is it under load is your network congested is your machine working sort of fast enough or is your own local machine overloaded. Lots of things like that can affect how fast you get a response on a network request. During that time you do not want your browser to completely hang.

So, you make an asynchronous request and the way that that is done is using something called a callback. You provide a function right to the network request function which says once the network request has completed successfully invoke this other function call me back. So, what you can do is you can effectively use this approach in order to push long running code into a separate task.

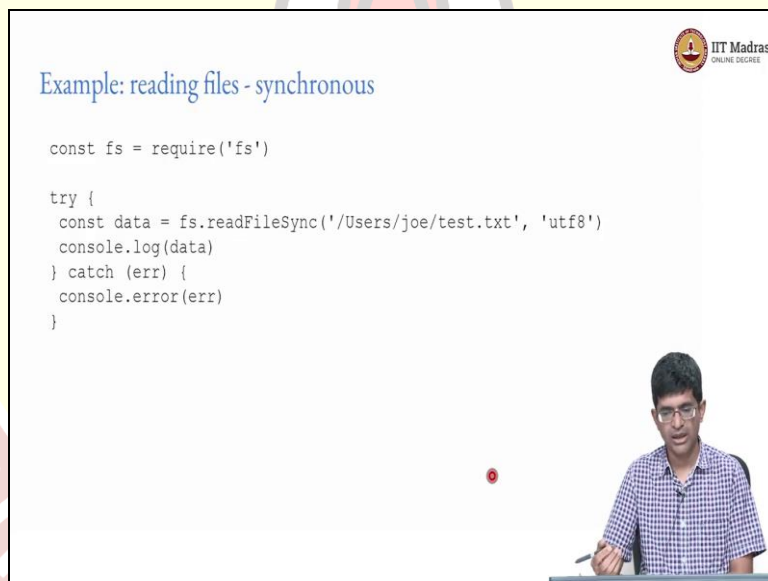
I also use this word thread over here conceptually it is very similar to the idea of threads which you might have encountered in other programming languages whether it is java or C or even python has a notion of you know threading. And the idea is somewhat similar you push the long running task into a separate something else which is not going to block my main flow and I have some way by which I can wait for that task to complete.

When I have done everything else I just sit over there and check has the task completed if not I go do something else and come back I keep on waiting for it over there. And this callback effectively says that you know once the long running task has completed I will trigger something which will let the main code know that yes you know work is done now go and update the page or you know do something else right.

Javascript does this without necessarily being multi-threaded. It just uses the event loop in order to realize the same functionality but it is very powerful what it means is that input output for example in the Javascript engine is almost completely asynchronous. Meaning that reading files reading from the network various other kinds of input or output that are required by the system can all just be handed off to asynchronous function calls.

Which will eventually trigger callbacks and as a result of this a single threaded Javascript engine can actually handle a very large number of requests it can effectively give the impression of being something running in parallel. Because during the time that these multiple tasks are sort of queuing up over there each one of them is taking its own time to complete during that time the engine continues to proceed and gives the feeling that the overall system is very fast and responsive.

(Refer Slide Time: 19:50)



```
Example: reading files - synchronous

const fs = require('fs')

try {
  const data = fs.readFileSync('/Users/joe/test.txt', 'utf8')
  console.log(data)
} catch (err) {
  console.error(err)
}
```

So, asynchronous code by itself is a little difficult to demonstrate directly and but on the other hand we will be coming across a lot of different uses of asynchronous code later. For the time being all I am going to do is leave you with a couple of examples of what the code might look like. And this is something which comes from it is used more in sort of the node.js the interpreted form rather the command line interpreter form of Javascript where we are saying that we want to be able to read something from the file system.

So, there is a package called fs you would notice that I am using this require over here you know this is basically a piece of code that has been sort of directly given from one of

the node js examples ideally you should not be doing this you would probably want to use an import instead. So, require should not really be needed in this thing you should be able to import instead.

But there are certain cases where you need to use require simply because the libraries are still in the older module format. But the important point is this is a module and the module has a method called read file sync or synchronously read a file. So, what we are doing is that we have this try and catch which is similar to the python try except or the Javascript try catch or the C++ try catch which essentially what it does is it tries to execute this meaning that it tries to synchronously read a file from this file name.

And once that is done it will take the data and log it out to the console and if that did not work it should give an error on the console. What will happen when we run this is that this fs dot read file sync will block meaning that until it is able to actually read the file nothing else in the Javascript engine is going to proceed. Now that is fine but what happens if let us say the file is on a network file system or it is a very large file something else happens that basically makes that reading process a little slow.

During that time there is nothing that can happen the system just has to wait. Now when you are using Javascript as a sort of backend system this might even be acceptable because you are not talking about live interactivity most of the time you are basically looking at something which maybe it needs to read that file and should not proceed without being able to read it that might contain let us say a password secret or something like sort.

And you know unless you have that ready inside the code you cannot really move forward. But the alternative is what is called the asynchronous approach from the same module there is also this thing called read file from the name you can sort of infer that you know the read file the asynchronous form is what is expected to be the default it does not have read file async in this it just says read file.

And what it says is same thing read from this file name but it also provides this as a callback look at this piece of code effectively what it is saying is it has two parameters error data a right arrow function and this curly bracket. Block of execution which means

that all of this starting from this error data until this curly bracket is an anonymous function. In this case that anonymous function is being used as a callback.

And this is a perfect use case for anonymous functions because you know I am not going to call this function by itself it is there its only purpose is to be used as a callback. So, why even bother to give it a name directly give it give its functionality as part of the code itself. What does it do? The callback basically says is going to get two parameters error and data.

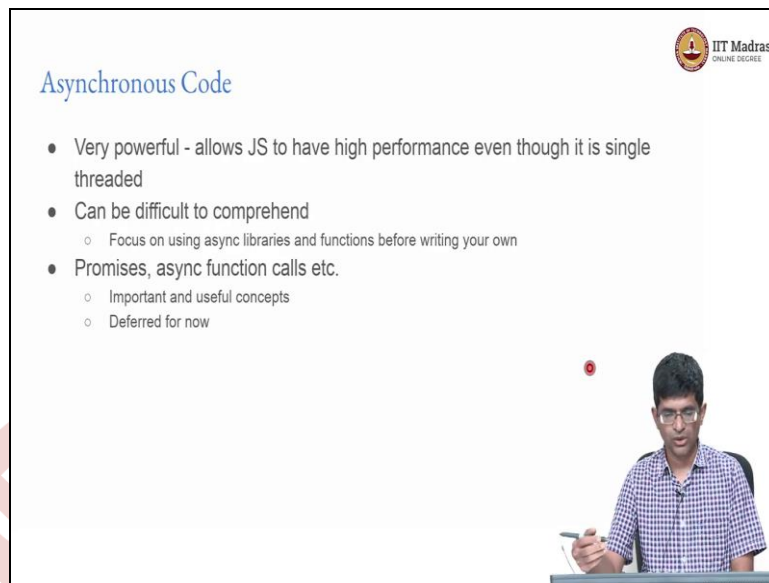
This will be called only after this read file has asynchronously completed meaning that when I actually call this fs dot read file it will exit from here and you know proceed. But once all of this is done it will come back and basically say ok you know execute this function. If there was an error it will put an error and return but if it did there was no error it will log the data.

So, the asynchronous form of writing the code essentially uses this notation in order to implement a callback you could have defined this as a named function and just given that function name over here that of course will work. It is just that given that you are never going to use this function in any other context it is a perfect place to use an anonymous function and this is a common sort of programming pattern that you will see in Javascript.

So, asynchronous programming in other words is something that you will extensively come across as we go forward in Javascript. It takes a little getting used to but you need to be sort of prepared for that and to sort of think you know what why is it fundamentally there it is there because Javascript's primary purpose is user interaction which means that it does not want to slow down.

And if it is not slowing down then ideally anything that is long running which might affect the user interface should be pushed out into a separate task. Javascript excels at this because of the fact that there is an event loop that can pull out those tasks and basically put them back into the call stack as in when required. So, as long as there is some way to push new tasks into that task queue the rest of the event loop basically takes care of executing things in as clean a manner as possible.

(Refer Slide Time: 25:40)



Asynchronous Code

- Very powerful - allows JS to have high performance even though it is single threaded
- Can be difficult to comprehend
 - Focus on using async libraries and functions before writing your own
- Promises, async function calls etc.
 - Important and useful concepts
 - Deferred for now

So, there are a few other concepts in asynchronous code in particular there is this concept of promises which have been brought up in the newer ECMAScript versions. And in fact once we use the `async` and `await` keywords. The programming the way that you write asynchronous programs actually comes almost very close to how you would write a regular program.

It is just that you need to think about the fact that yes this is an asynchronous operation or a synchronous operation. Having said that we are not really going to get into the details of how promises are implemented or even how they work at the moment. At some point depending on necessity we will see what is required of these concepts in order to understand certain pieces of code.

But a full explanation of how they are implemented why they are implemented in a certain way is beyond the scope of what we are doing here.