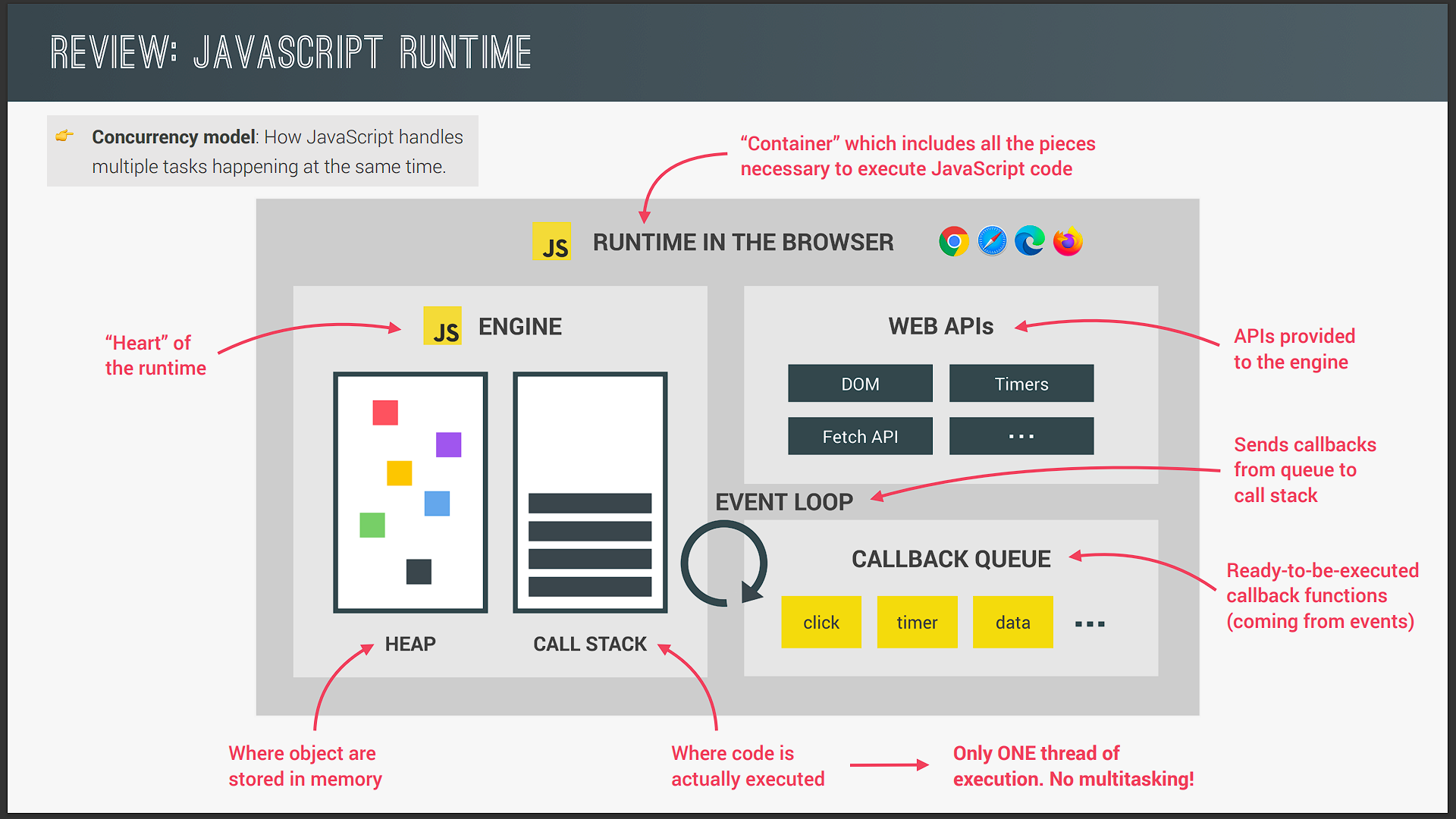
To start let's quickly review the JavaScript runtime that we talked about way back in the course just to make sure that the rest of this lecture will make sense to you.

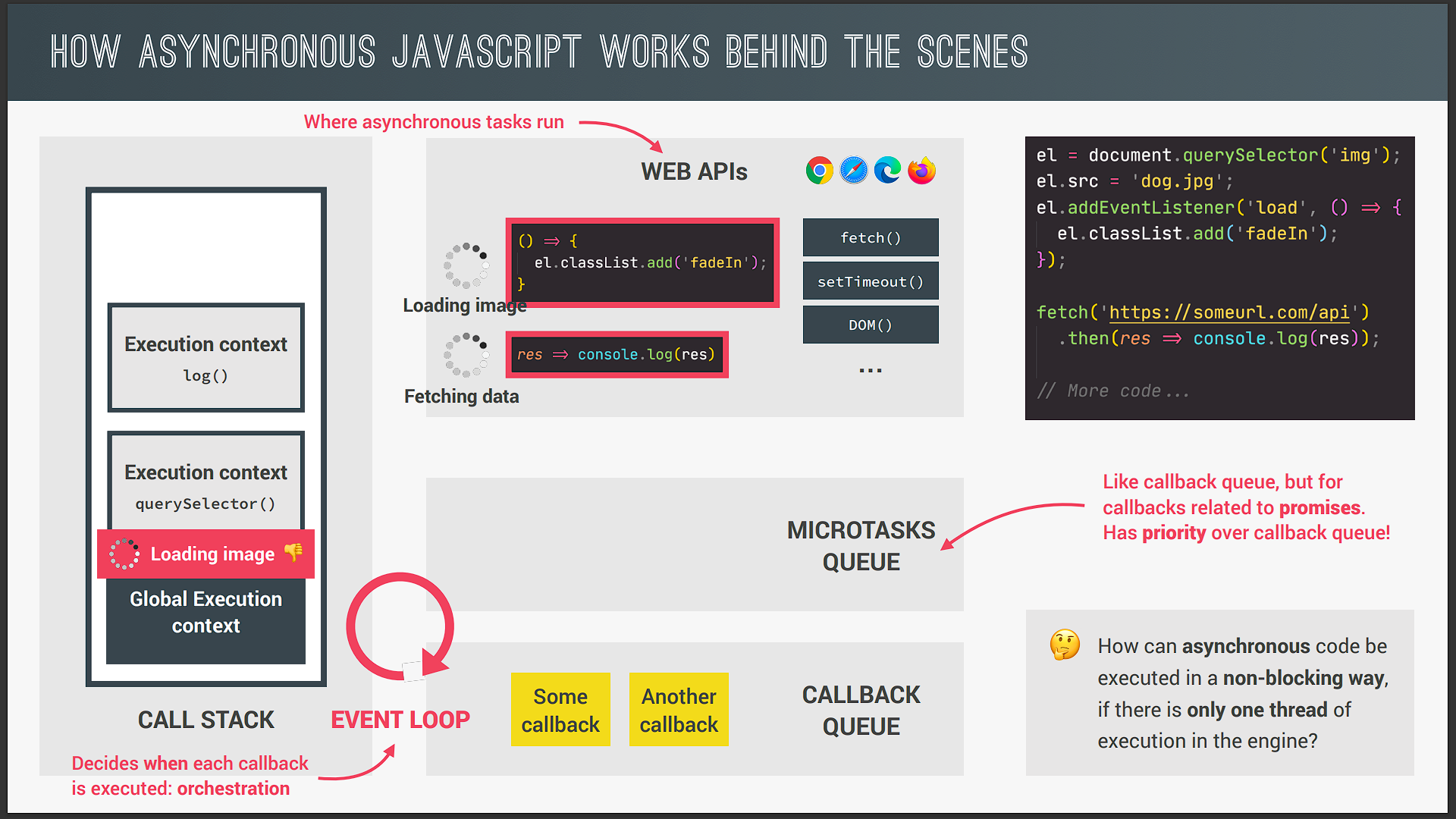


So a JavaScript runtime is basically a container which includes all the different pieces that are necessary to execute JavaScript code. Now the heart of every JavaScript runtime is the engine. So this is where code is actually executed and where objects are stored in memory. So these two things happen in the call stack and in the heap.

Now what's important to note here is that JavaScript has only one threat of execution. And so it can only do one thing at a time. There is absolutely no multitasking happening in JavaScript itself. Now other languages like Java can execute multiple pieces of code at the same time but not JavaScript. But anyway next we have the web APIs environment. These are some APIs provided to the engine but which are actually not part of the JavaScript language itself. So that's things like the DOM timers the fetch API the geolocation API and so on and so forth.

Next up there is the callback queue and this is a data structure that holds all the ready to be executed callback functions that are attached to some event that has occurred. Finally whenever the call stack is empty the event loop takes callbacks from the callback queue and puts them into call stack so that they can be executed. So the event loop is the essential piece that makes asynchronous behavior possible in JavaScript. It's the reason why we can have a non blocking concurrency model in JavaScript. And a concurrency model is simply how a language handles multiple things happening at the same time. But now how does this non blocking concurrency actually work? And why is the event loop so important? Well let's find out. And let's focus on the essential parts of the runtime here. So that's the call stack the event loop the web APIs and to callback queue.

So as you know by now a JavaScript engine is built around the idea of a single threat. But if there was only one thread of execution in the engine then how can asynchronous code be executed in a non blocking way? Well that's actually the big question that we're gonna answer in this video. So essentially you will learn how the JavaScript concurrency model really works behind the scenes using all the parts of the JavaScript runtime that you already know. And as always we will do this by looking at a real code example. So let's walk through the code line by line and I will keep updating the call stack as we go however you already know how to call stack works. And so it's best that you focus more on the code and on the web APIs and callback queue.



Okay but now let's start by selecting this image element. And then in the next line we set the source attribute of that image to dog.jpg. And as we learned before this will now start to load this image asynchronously in the background. But this time we can actually understand what that mysterious background actually is. So as you already know everything related to the DOM is not really part of JavaScript but of the web APIs. And so it's in a web APIs environment where the asynchronous tasks related to the DOM will run. And in fact the same is true for timers AJAX calls and really all other asynchronous tasks.

So again these asynchronous tasks will all run in the web API environment of the browser. Now if the image would be loading in a synchronous way it would be doing so right in the call stack blocking the execution of the rest of the code. But as we already learned that would be terrible. And that's why loading images in JavaScript is asynchronous. So it does not happen in the call stack. So not in the main thread of execution but really in the web APIs environment as I mentioned before. Now if we want to do something after the image has finished loading then we need to listen to the load event. And so that's exactly what we do in the next line of code. So here we attach an event listener to the load event of that image and pass an a callback function as always. In practice this means to register this callback in the web APIs environment exactly where the image is loading. And to callback will stay there until the load event is emitted. So we're handling asynchronous behavior here with a callback just as we learned before but anyway let's go back to the code. And so in the next line we make an AJAX call using the fetch API. And as always the asynchronous fetch operation will happen in the web APIs environment. And again that's because otherwise we would be blocking the call stack and create a huge lag in our application.

Finally we use the then method on the promise returned by the fetch function. And this will also register a callback in the web API environment so that we can react to the future resolved value of the promise. So this callback is associated with a promise that is fetching the data from the API. And that's gonna be important later on.

So with this we have now executed all the top level of code. So all the code that is not inside any callback function in asynchronous way. We also have the image loading in the background and some data being fetched from an API. And so now it's time for this to get really interesting. Let's say the image has finished loading and therefore the load event is emitted on that image. What happens next is that the callback for this event is put into callback queue. And the callback queue is basically an ordered list of all the callback functions that are in line to be executed. And you can think of this callback queue as a to do list that you would write for yourself with all the tasks that you have to complete. So the callback queue is also a to do list of a kind but with tasks that the call stack will eventually have to complete.

Now in this example there are no other callbacks in the queue yet but there could be of course. So if there were already other callbacks waiting in line then this new callback would of course go straight to the end of the queue. And there it would sit patiently for its turn to finally run. And this actually has big implications. So imagine that you set a timer for five seconds. And so after five seconds that timer's callback will be put on the callback queue right. And let's say there were already other callbacks awaiting. And let's also say that it took one second to run all of those callbacks. Then in that case your timers callback would only run after six seconds and not after five. So these six seconds are the five seconds that passed for the timer plus the one second that it took to run all the other callbacks that were already waiting in line in front of your timer. So what this means is that the timers duration that you define is not a guarantee. The only guarantee is that the timers callback will not run before five seconds but it might very well run after five seconds have passed. So it all depends on the state of the callback queue. And also another queue that we're gonna learn about in a second.

Now another thing that's important to mention here is that the callback queue also contains callbacks coming from DOM events like clicks or key presses or whatever. Now we learned before that DOM events are not really asynchronous behavior but they still use the callback queue to run their attached callbacks. So if a click happens on a button with addEventListener then what will happen is just like what I illustrated here with the asynchronous load event. But anyway now it's time to finally learn about the event loop. So here is what the event loop does. It looks into the call stack and determines whether it's empty or not. Except of course for the global context then if the stack is indeed empty which means that there's currently no code being executed then it will take the first callback from the callback queue and put it on the call stack two will be executed. And this is called an event loop tick. So each time the event loop takes a callback from the callback queue. We say that there was an event loop tick. So as we can see here the event loop has the extremely important task of doing coordination between the call stack and to callbacks in the callback queue.

So the event loop is basically who decides exactly when each callback is executed. We can also say that the event loop does the orchestration of this entire JavaScript runtime. Another thing that becomes clear from this whole explanation is that the JavaScript language itself has actually no sense of time. That's because everything that is asynchronous does not happen in the engine. It's the runtime who manages all the asynchronous behavior. And it's the event loop who decides which code will be executed next. But the engine itself simply executes whatever code it has given.

Okay so this is of course a lot to take in. So let's try to recap what's happened here. So the image started loading asynchronously in the web APIs environment and not in the call stack right. We then used addEventListener to attach a callback function to the image load event. And this callback is basically or asynchronous code it's code that we deferred into the future because we only want to execute it once the image has loaded. And in the meantime the rest of the code kept running. Now addEventListener did not put the callback directly in the callback queue. It simply registered the callback which then kept waiting in the web APIs environment until the load event was fired off. Only then the environment put the call back into queue. Then while in the queue the callback kept waiting for the event loop to pick it up and put it on the call stack. And this happened as soon as the callback was first in line and the call stack was empty. And that's it actually.

So all this happened so that the image did not have to load in the call stack but in the background in a non blocking way. So in a nutshell the web APIs environment the callback queue and the event loop all together make it possible that asynchronous code can be executed in a non blocking way even with only one thread of execution in the engine. Wow that was already a lot to understand but we're not done yet. Because we still have to fetch function getting data from the AJAX call in the background. And this is basically happening with a promise.

Remember now with promises things work in a slightly different way which is why I included this promise example as well. So let's say that the data has now finally arrived. And so the fetch is done. Now callbacks related to promises like this one that we registered with the promises then method. Do actually not go into the callback queue. So again this callback did we still have here which is coming from a promise will not be moved into the callback queue. Instead callbacks of promises have a special queue for themselves which is the so called microtasks queue. Now what is special about the microtasks queue is that it basically has priority over the callback queue. So at the end of an event loop tick so after a callback has been taken from the callback queue the event loop will check if there are any callbacks in the microtasks queue. And if there are it will run all of them before it will run any more callbacks from the regular callback queue.

And by the way we call these callbacks from promises microtasks. And therefore the name microtasks queue. And there are actually other microtasks but that's not relevant here. So going back to our example currently we actually do have a microtask sitting in a microtasks queue the call stack is also empty. And therefore the event loop will now take this callback and put it in the call stack just like it does with callbacks from the callback queue. And it doesn't matter if the callback queue is empty or not. So this would have worked the exact same way even if there were some callbacks in the callback queue. And again that's because microtasks always have priority. In practice this means that microtasks can basically cut in line before all other regular callbacks. Now if one microtask adds a new microtask then that new microtask is also executed before any callbacks from the callback queue. And this means that the microtasks queue can essentially starve the callback queue. Because if we keep adding more and more microtasks then callbacks in the callback queue can never execute. Now this is usually never a problem but I just wanted to mention this possibility here anyways who knows maybe this will be an interview question for you someday. And if so you'd now know the answer. But anyway as you can hopefully see the idea of running asynchronous code with regular callbacks and with microtasks coming from promises is very similar. The only difference is that they go into different queues and that the event loop gives microtasks priority over regular callbacks. All right and thats finally it. that's all you need to know about how asynchronous JavaScript really works behind the scenes.

And I'm sure that this knowledge is gonna be extremely helpful and valuable to you. Because you're gonna be way more confident writing asynchronous code now. And also you will ace any job interview question about asynchronous JavaScript. And actually so many JavaScript developers don't know anything about this. So I'm sure that this knowledge will put you into the top % or even top % of JavaScript developers. And that's just amazing on itself right. But anyway let's no finish here and try out some of this stuff in practice so that you see for yourself that I didn't just make this stuff up.