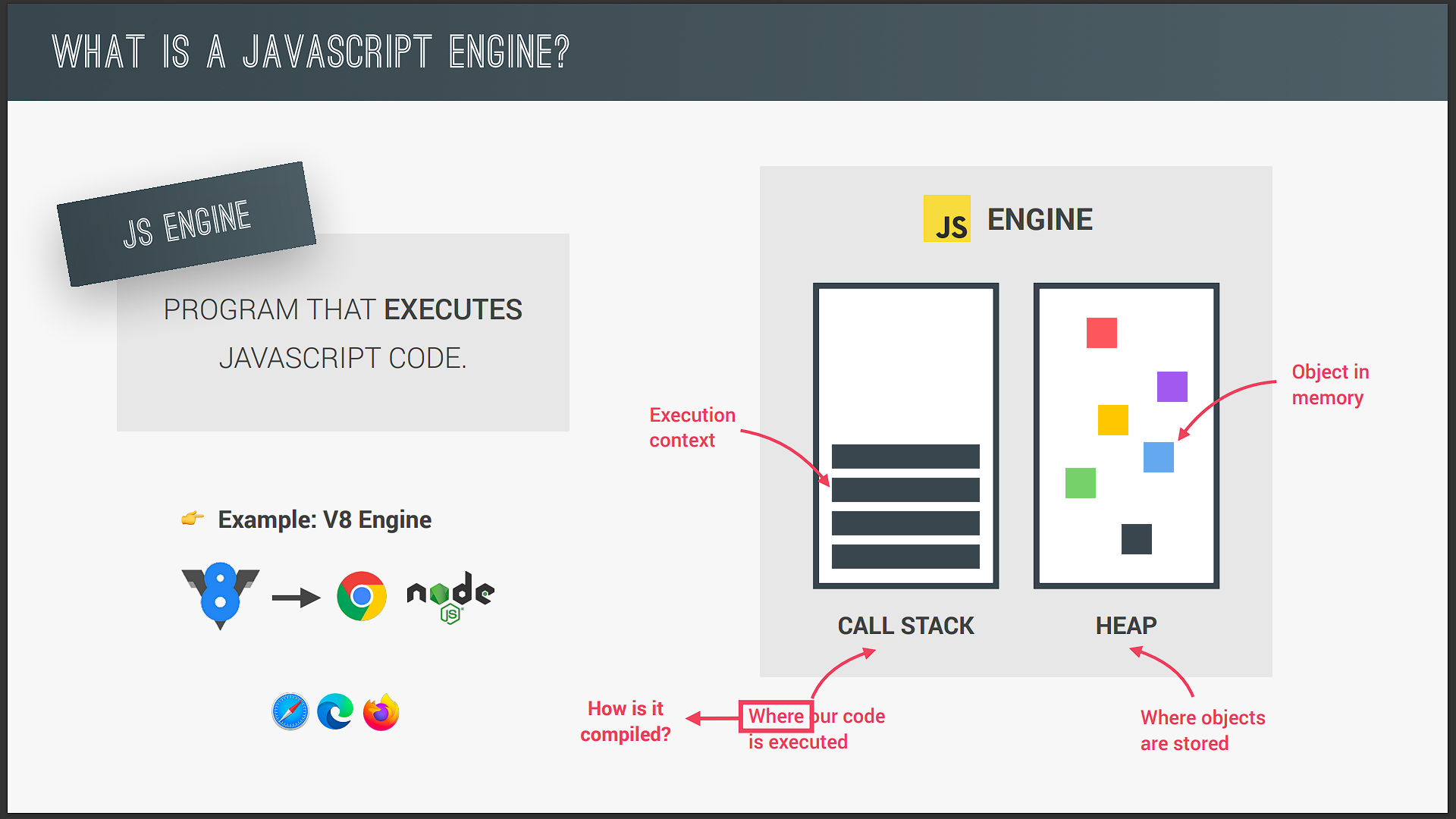
Q. And what is a JavaScript runtime? Also how is JavaScript code translated to a machine code?

So a JavaScript engine is simply a computer program that executes JavaScript code.There are a lot of steps involved in doing that but essentially executing JavaScript code is what an engine does. Now every browser has its own JavaScript engine but probably the most well known engine is Google's VEight. The V eight engine powers Google Chrome but also Node.js which is that JavaScript runtime that we talked about in the beginning of the course so the one that we can use to build server side applications with JavaScript so outside of any browser. And of course all the other browsers have their own JavaScript engines which you can look up online if you're interested.

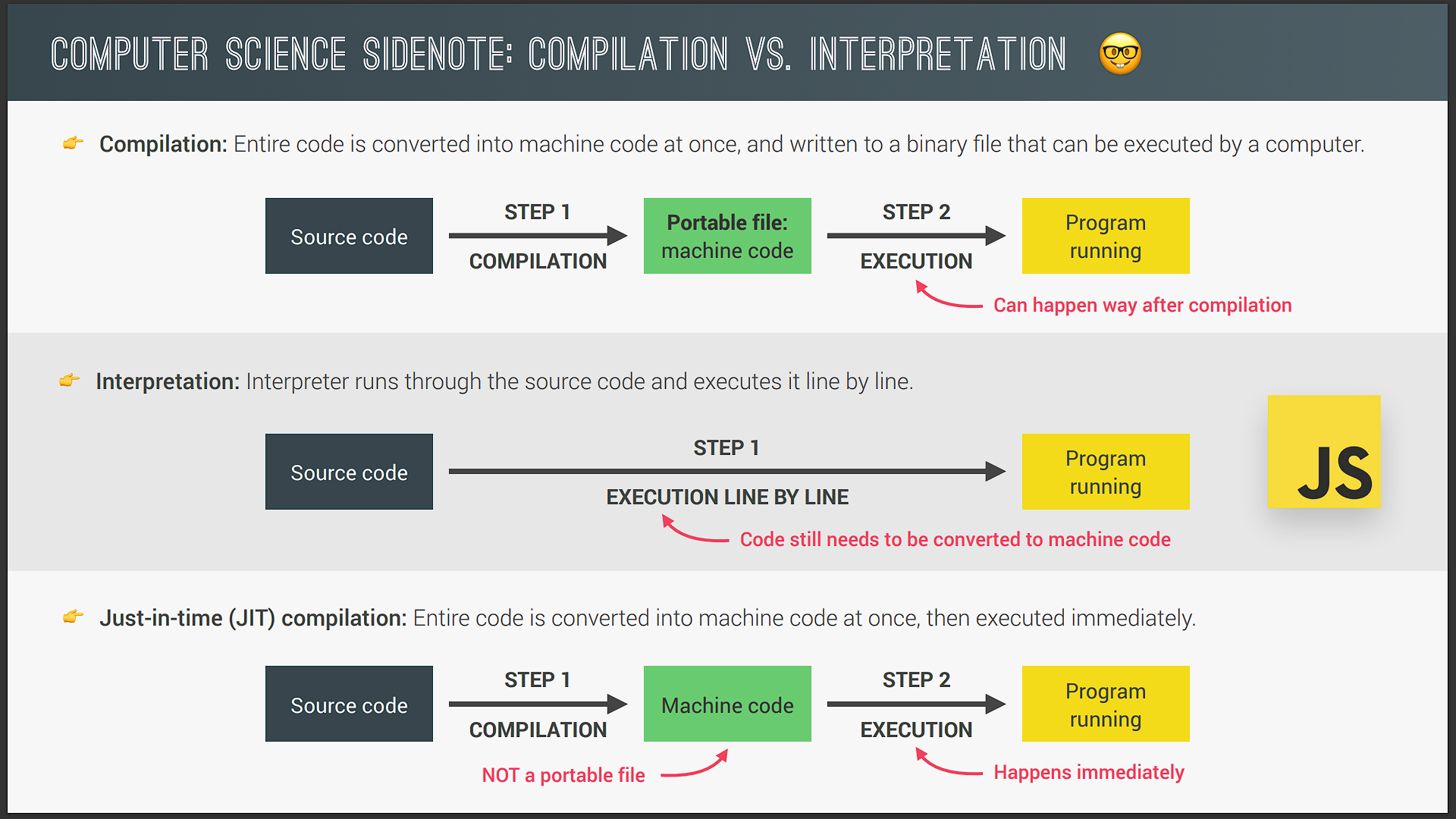


Anyway it's quite easy to understand what an engine is but what's most important is to actually understand its components and how it works.

So any JavaScript engine always contains a call stack and a heap. The call stack is where our code is actually executed using something called execution contexts. Then the heap is an unstructured memory pool which stores all the objects that our application needs. Alright so with this look at the engine we have answered where our code is executed. But now the question is how the code is compiled to machine code so that it actually can be executed afterwards.

But first we need to make a quick computer science side note here and talk about the difference between compilation and interpretation. So in the last lecture we learned that the computer's processor only understands zeros and ones and that's therefore every single computer program ultimately needs to be converted into this machine code and this can happen using compilation or interpretation. So in compilation the entire source code is converted into machine code at once. And this machine code is then written nto a portable file that can be executed on any computer.

So we have two different steps here



First the machine code is built and then it is executed in the CPU so in the processor. And the execution can happen way after the compilation of course. For example any application that you're using on your computer right now has been compiled before and you're now executing it way after it's compilation.

Now on the other hand in interpretation there is an interpreter which runs through the source code and executes it line by line. So here we do not have the same two steps as before. Instead the code is read and executed all at the same time. Of course the source code still needs to be converted into machine code but it simply happens right before it's executed and not ahead of time. Now JavaScript used to be a purely interpreted language but the problem with interpreted languages is that they are much much slower than compiled languages.

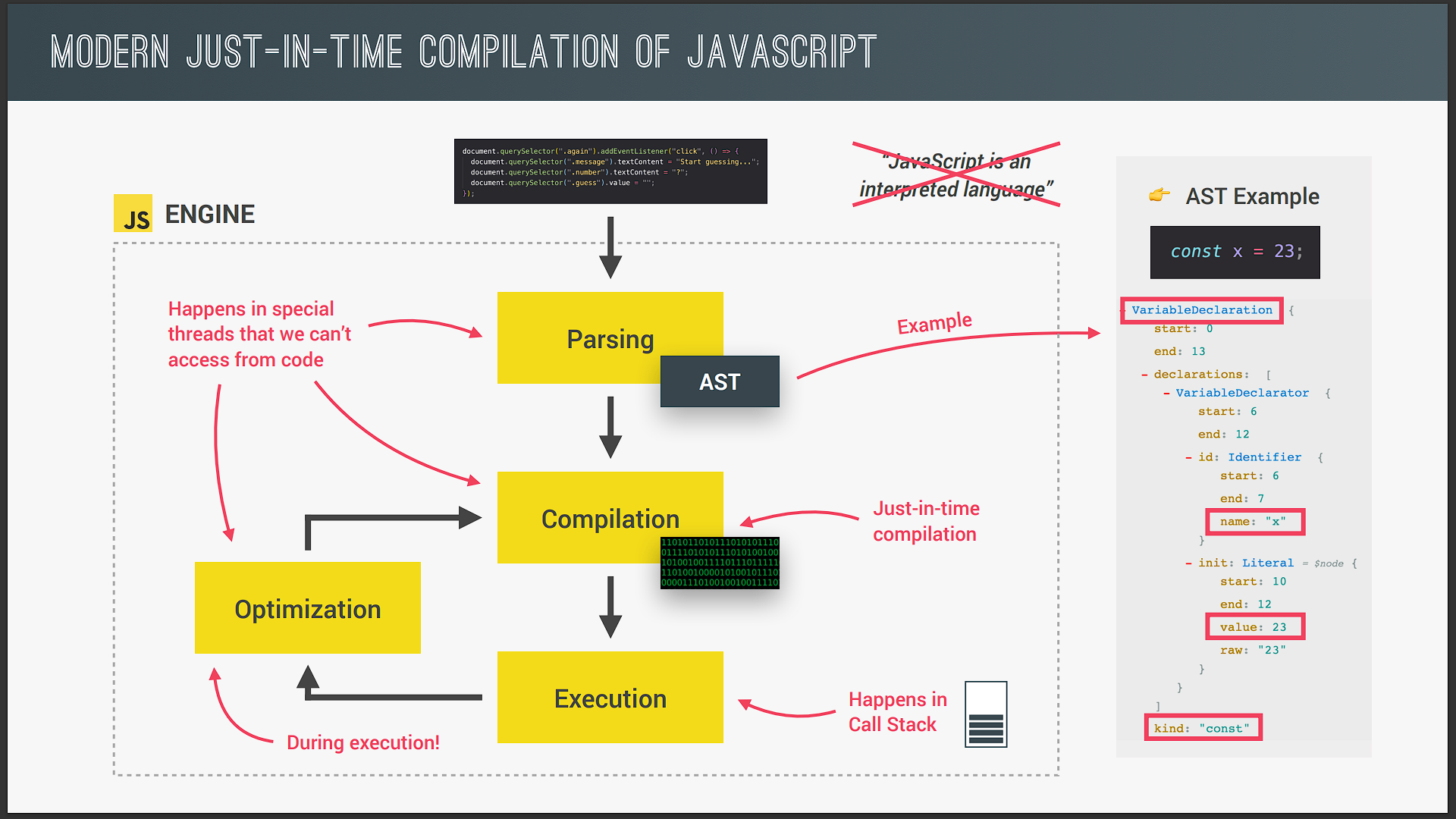
This used to be okay for JavaScript but now with modern JavaScript and fully fledged web applications that we built and use today low performance is no longer acceptable. Just imagine you were using Google maps in your browser and you were dragging the map and each time you dragged it would take one second for it to move. That would be completely unacceptable right?

Now many people still think that JavaScript is an interpreted language but that's not really true anymore. So instead of simple interpretation modern JavaScript engine now use a mix between compilation and interpretation which is called justintime compilation. This approach basically compiles the entire code into machine code at once and then executes it right away.

So we still have the two steps of regular ahead of time compilation but there is no portable file to execute. And the execution happens immediately after a compilation. And this is perfect for JavaScript as it's really a lot faster than just executing code line by line.

Now I skimmed over some details here but this is really all you need to know. Anyway let's now understand how this works in the particular case of JavaScript.

So as a piece of JavaScript code enters the engine the first step is to parse the code which essentially means to read the code. During the parsing process the code is parsed into a data structure called the abstract syntax tree or AST. This works by first splitting up each line of code into pieces that are meaningful to the language like the const or function keywords and then saving all these pieces into the tree in a structured way. This step also checks if there are any syntax errors and the resulting tree will later be used to generate the machine code.



Now sometimes I get asked if this tree has anything to do with the DOM tree and the answer is a very clear no. So this tree has absolutely nothing to do with the DOM. It is not related in any way. It's just a representation of our entire code inside the engine.

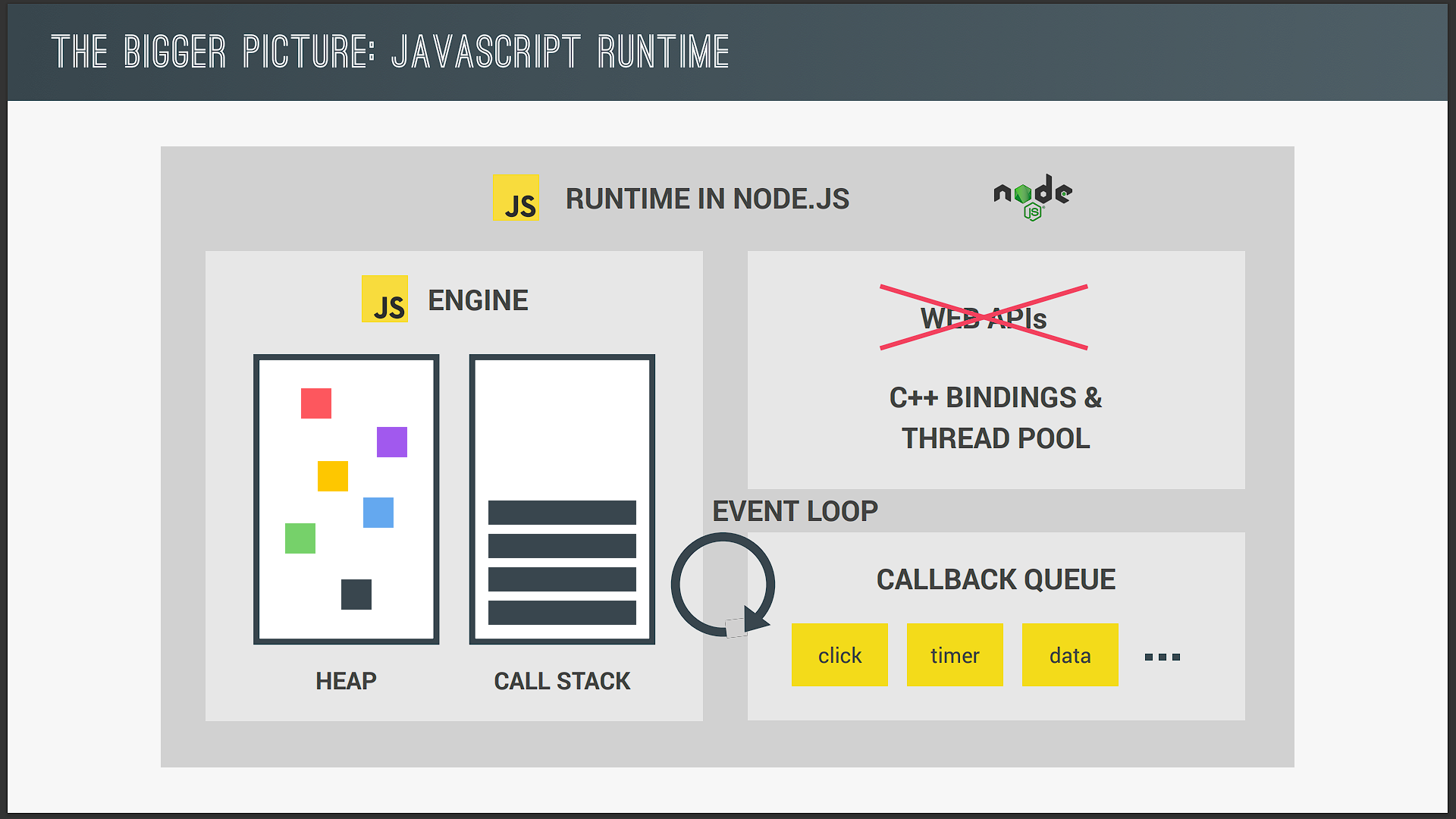
Anyway the next step is compilation which takes the generated AST and compiles it into machine code just as we learned in the previous slide. This machine code then gets executed right away because remember modern JavaScript engine use justintime compilation. And remember execution happens in the JavaScript engines call stack but we will dig deeper into this in the next lecture. All right so far so good. We have our code running so we can finish here Right? Well not so fast because modern JavaScript engines actually have some pretty clever optimization strategies. What they do is to create a very unoptimized version of machine code in the beginning just so that it can start executing as fast as possible. Then in the background this code is being optimized and recompiled during the already running program execution. And this can be done most of the times and after each optimization the unoptimized code is simply swept for the new more optimized code without ever stopping execution of course. And this process is what makes modern engines such as the VEight so fast and all this parsing compilation and optimization happens in some special threads inside the engine that we cannot access from our code. So completely separate from the main thread that is basically running into call stack executing our own code.

Now different engines implements in slightly different ways but in a nutshell this is what modern justintime compilation looks like for JavaScript. And the next time someone tells you JavaScript is an interpreted language you just show them this slide so that they can learn how it really works.

Alright so we looked at the JavaScript engine and how it works behind the scenes in quite some detail. Now to round off this lecture let's also take a look at what a JavaScript runtime is. And in particular the most common one which is the browser and by doing this we can get the bigger picture of how all the pieces fit together when we use JavaScript. And so this is a really important slide.

So we can imagine a JavaScript runtime as a big box or a big container which includes all the things that we need in order to use JavaScript in this case in the browser. And to heart of any JavaScript runtime is always a JavaScript engine. So exactly the one we've been talking about. That's why it makes sense to talk about engines and runtimes together. Without an engine there is no runtime and there is no JavaScript at all. However the engine alone is not enough. In order to work properly we also need access to the web APIs and we talked about web APIs before remember?

So that's everything related to the DOM or timers or even the console.log that we use all the time. So essentially web APIs are functionalities provided to the engine but which are actually not part of the JavaScript language itself. JavaScript simply gets access to these APIs through the global window object. But it still makes sense that the web APIs are also part of the runtime because again a runtime is just like a box that contains all the JavaScript related stuff that we need.



Next a typical JavaScript runtime also includes a so called callback queue. This is a data structure that contains all the callback functions that are ready to be executed. For example we attach event handler functions to DOM elements like a button to react to certain events right? And these event handler functions are also called callback functions okay. So as the event happens for example a click the callback function will be called. And here is how that actually works behind the scenes.

So the first thing that actually happens after the event is that the callback function is put into the callback queue.

Then when the stack is empty the callback function is passed to the stack so that it can be executed. And this happens by something called the event loop. So basically the event loop takes callback functions from the callback queue and puts them in the call stack so that they can be executed. And remember how I said in the last lecture that the event loop is how JavaScript's nonblocking concurrency model is implemented? Well here is an overview of how that works.

Alright so as they already said the focus in this course is on JavaScript in the browser and that's why we analyzed the browser JavaScript runtime. However it's also important to remember that JavaScript can exist outside of browsers for example in Node.js. And so here is what the node JS JavaScript runtime looks like. It's pretty similar but since we don't have a browser of course we can't have the web APIs because it's the browser who provides these. Instead we have multiple C ++ bindings and a so called thread pool. Now details don't matter here at all. I just want you to know that different JavaScript runtimes do exist.

