# 2022.5.5 JavaScript如何在幕后工作

## 90.JavaScript引擎和运行时

### 什么是JavaScript引擎？

1. JavaScript引擎 只是一个执行JavaScript代码的计算程序

2. 代码在执行之前涉及很多步骤，但本质上执行JavaScript代码是引擎的作用

3. 每个浏览器都有自己的JavaScript引擎，最知名的引擎是谷歌的V-8引擎

4. V-8引擎为Google Chrome和Node.js提供动力，这就是我们在课程开始时谈到的JavaScript运行时，因此我们可以使用它来使用JavaScript构建服务器端应用程序，因此可以在任何浏览器之外

### JavaScript编译、解释、即时编译

#### 区别

1. 编译，在编译时，整个源码一下子转换成了机器码，然后将此机器代码写入可在任何计算机上执行的可移植文件，最后执行

a) 启动需要更多的时间，因为它必须在一开始就经过那个编译步骤，但是循环中的代码运行得更快，因为它不需要为每次通过该循环重复编译

b) 另一个区别是编译器有更多时间查看代码并对其进行编辑，以便它运行得更快，这些编辑称为优化

c) 解释器在运行时完成其它工作，因此在翻译阶段不会花费太多时间来找出这些优化

2. 解释，有一个解释器贯穿源代码并逐步执行，代码是同时读取和执行的

a) 可以快速启动和运行，开始运行代码之前，您不必完成整个编译步骤，您只需开始翻译第一行并运行它

b) 注意当您运行相同的代码时，就会出现使用解释器的弊端，例如您处于循环中，你必须一遍又一遍地做同样的翻译

3. 即时编译，是编译和解释之间的混合，我们有常规提前编译的两个步骤，但是没有可移植文件

a) 作为摆脱解释器效率低下的一种方式，--解释器每次通过循环时都必须不断地重新翻译代码--，浏览器开始混合编译器

b) 不同的浏览器执行此操作的方式略有不同，但基本思路是相同的，他们为JavaScript引擎添加了一个新部分，称为监视器（也称为分析器），该监视器在代码运行时监视它，并记下它运行了多少次以及使用了哪些类型

c) 采用warm和hot标签，进行优化，当遇到相同代码时，直接打开编译版本

https://zhuanlan.zhihu.com/p/99395691

https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/ 详细介绍三者之间区别

#### 区别总结

1. 编译，编译、机器代码和可移植文件、执行，注意可以在编译后执行

2. 解释，同时编译和执行，注意也需要转换为机器代码

3. 即时编译，编译、机器代码、执行，注意是立即执行 (在代码编译的过程中就已经开始执行了)

a) 下面JavaScript现代引擎"即时编译"详细解释了执行为什么发生在编译中

b) 上方链接详细说明了即时编译的warm和hot标签优化机制！！！

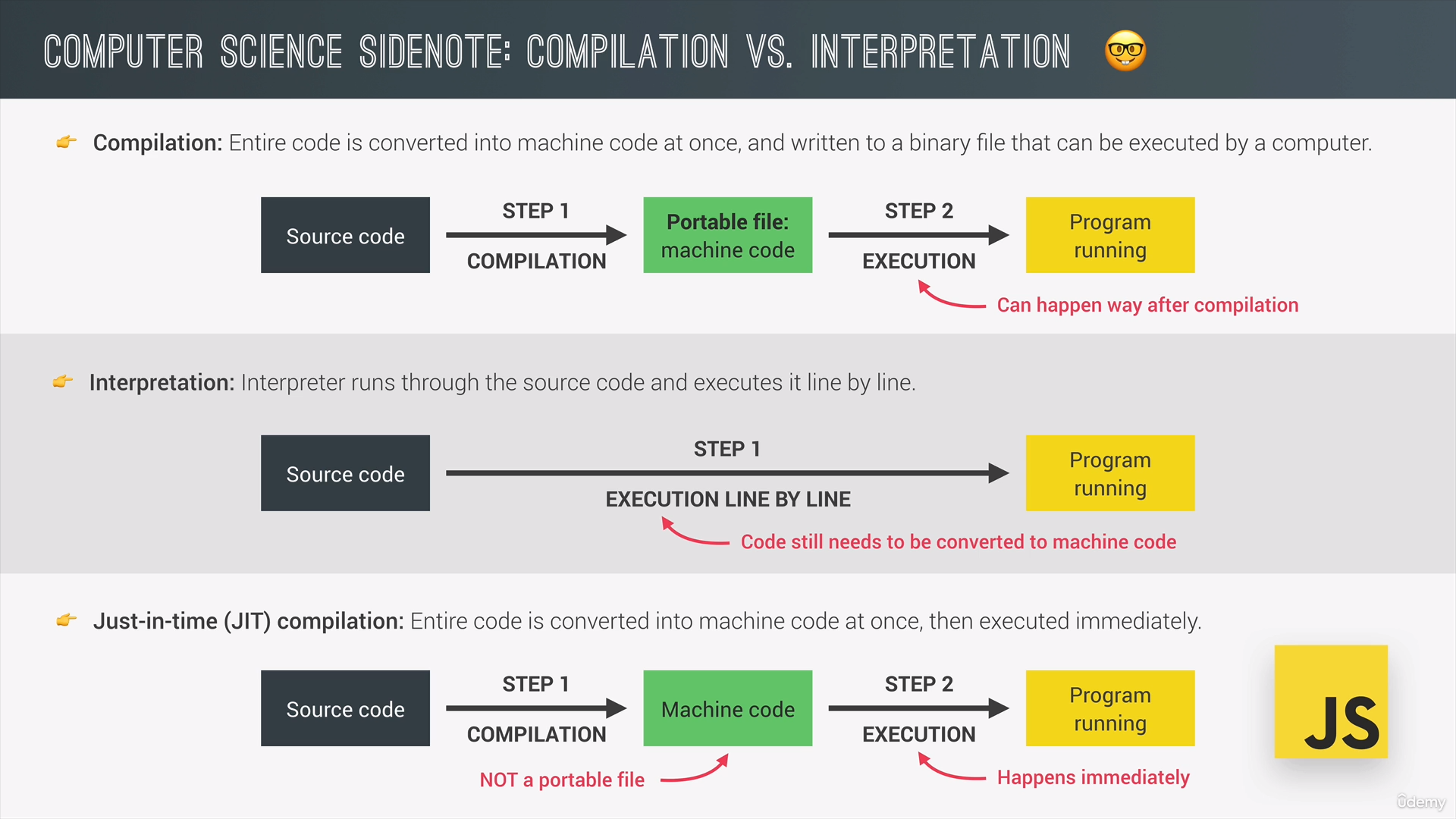
c) 即时编译，在编译过程中会更快，因为会先执行一版未非常优化的机器代码，在程序执行期间不断编译和优化，优化又分为warm和hot两种，因此性能会成倍的增加，非常的哇塞，详情还是要看上方链接，目前看的头疼 ------- 等待笔记

#### js不仅仅是解释型语言

1. JavaScript曾经是一种纯粹的解释型语言，但解释型语言的问题在于它们比编译语言慢得多，这对于JavaScript来说过去是可以的，但现在有了现代JavaScript和我们今天构建和使用的成熟的Web应用程序，低性能是不再可以接受

2. 我们仍然有常规提前编译的两个步骤，但是没有可移植的文件可以执行，编译后立即执行，这对于JavaScript来说是完美的，因为它确实比逐行执行代码要快的多

#### 区别对比图



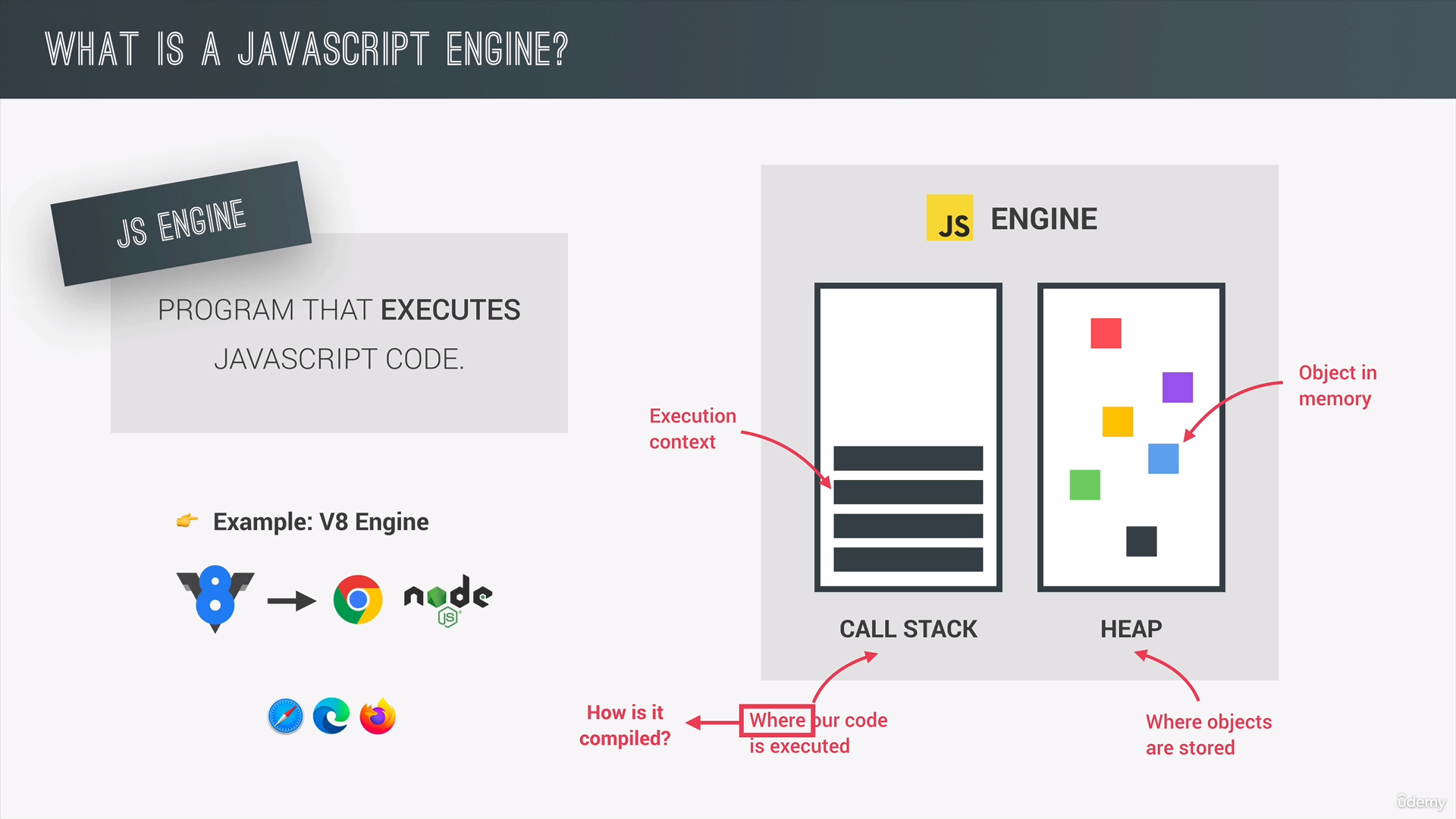
### JavaScript引擎组件包含什么？

1. 任何JavaScript引擎总是包含一个调用栈和一个堆，调用堆栈是我们的代码使用称为执行上下文的东西实际执行的地方

2. 堆是一个非结构化的内存池，它存储我们应用程序需要的所有对象 ----等待笔记

3. 通过这个引擎，我们已经回答了我们代码在哪里执行

4. 有问题是，如何将代码编译成机器代码，以便之后可以实际执行



### JavaScript现代引擎“即时编译”

#### 即时编译步骤

1. JavaScript代码进入引擎时，第一步解析代码，这实际上意味着读取代码，在解析过程中，代码会被解析为一种称为抽

象语法树或AST的数据结构

a) 其工作原理是将每一行代码拆分成对语言有意义的片段，例如const、function关键字、变量名、参数等，然后以结构化的方式将这些片段保存到树中

b) 此步骤还检查是否存在任何语法错误，并且生成树梢后将用于生成机器代码

2. 第二步进行编译，将生成的AST编译成机器代码

3. 第三步立即执行，执行发生在JavaScript引擎执行堆栈中(调用堆栈)

4. 第二步与第三步之间，JavaScript有一个优化策略，会在开始执行前，创建一个未非常优化的机器代码版本，为了能够尽快执行

a) 在后台中代码已经运行，在程序执行期间，未优化的机器代码会被重新编译和优化，并且每次优化后，未优化的代码都会被简单地扫描以获取新的更优化的代码，当然不会停止执行 (在这其中还有warn和hot标签优化机制--深层次优化)

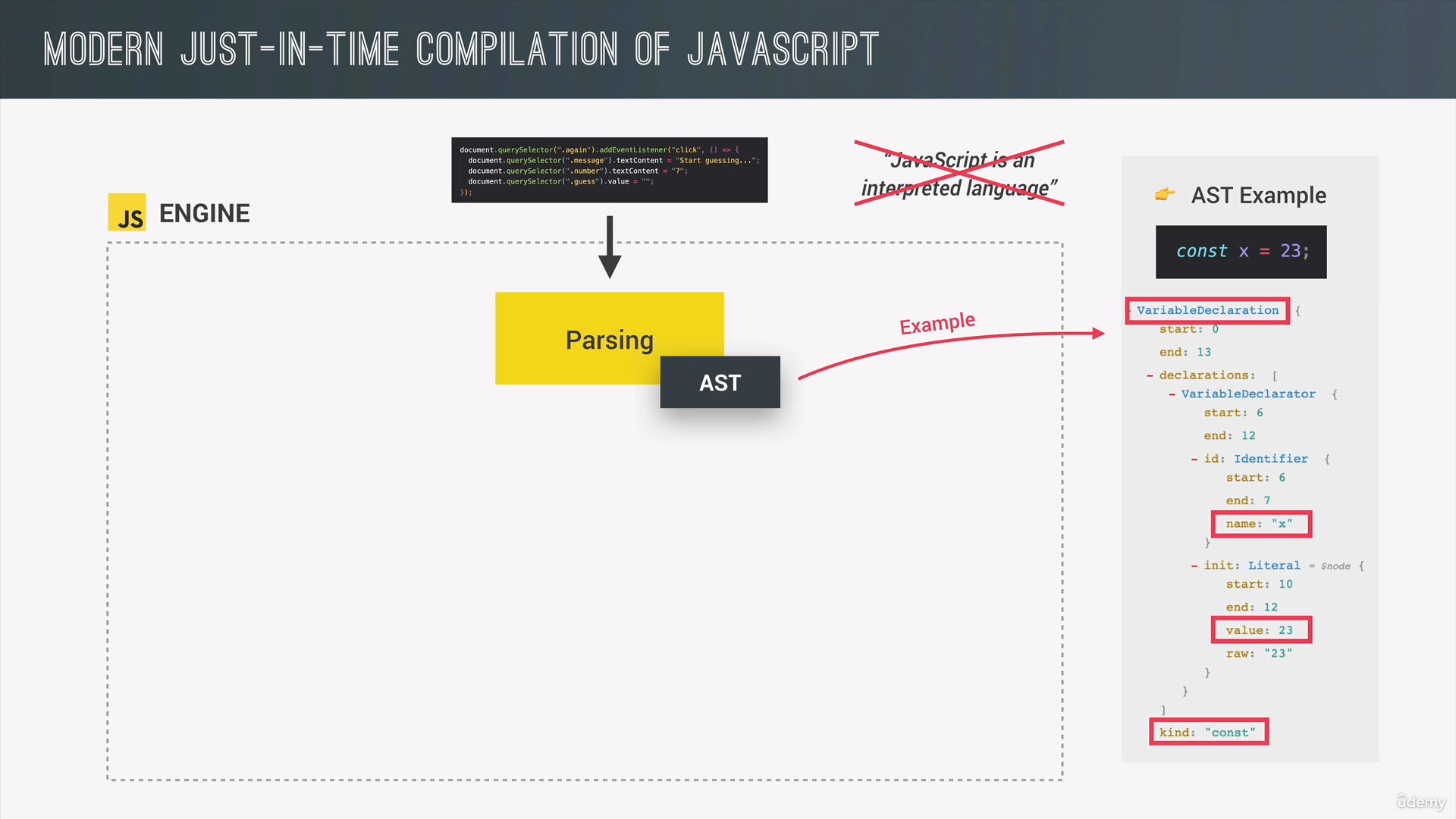
步骤图解释：

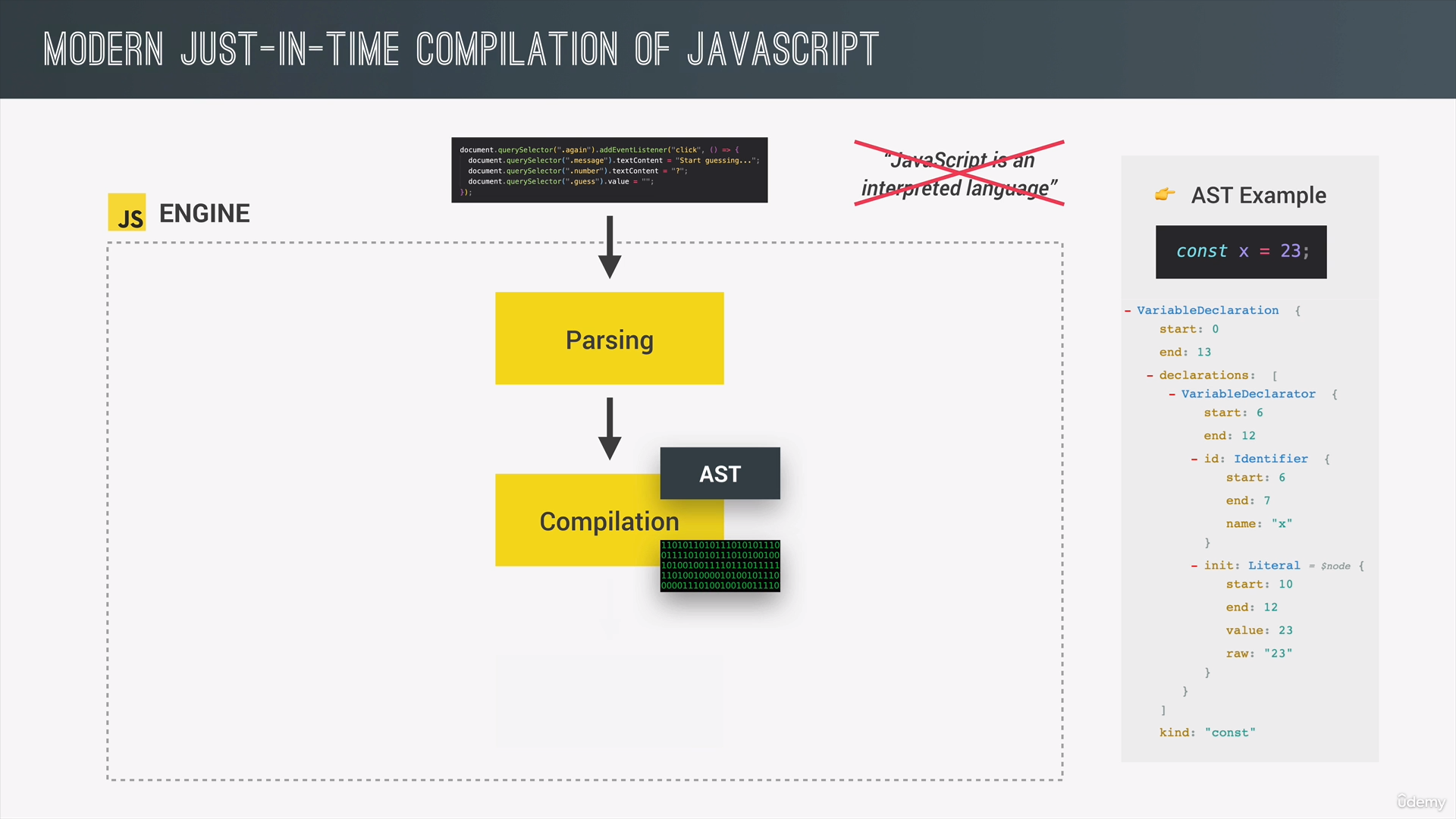
1. 即时编译过程使用现代引擎(如V-8)非常快速，所有这些解析、编译和优化都发生在引擎内部的一些特殊线程中，我们无

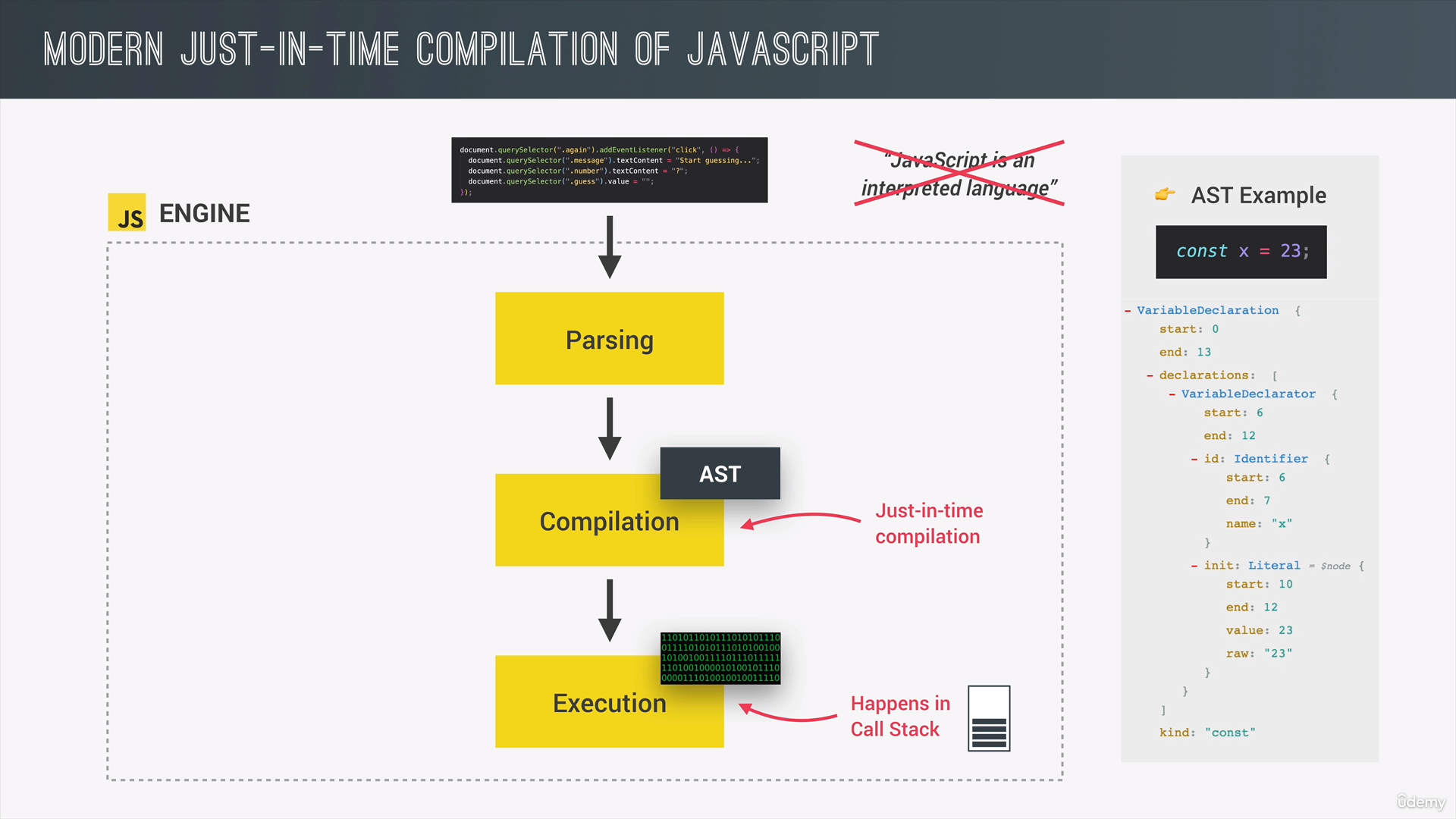
法从代码中访问这些线程，注意执行发生在JavaScript主线程中

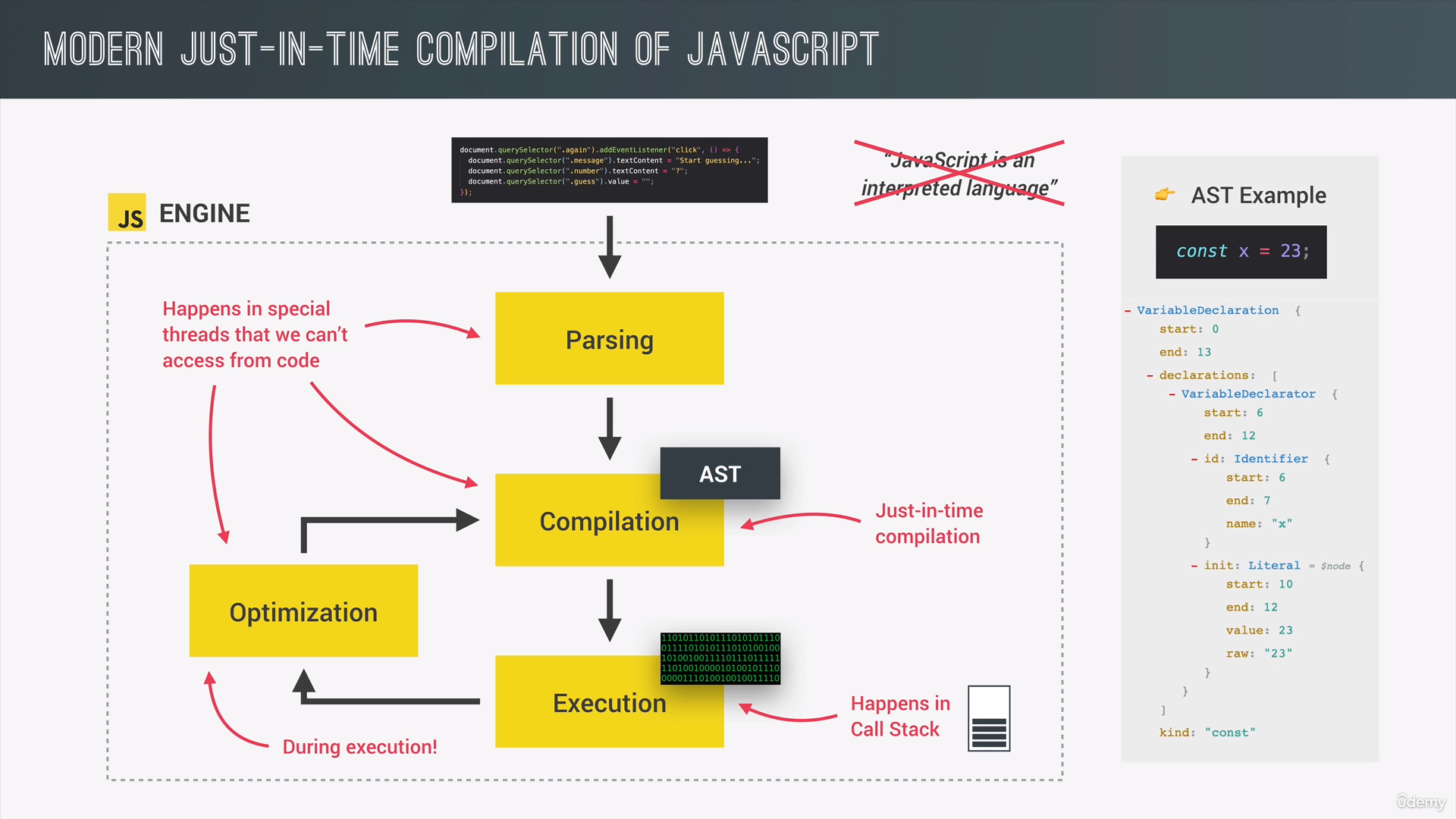
2. 因此特殊线程与主线程分离，主线程是代码运行到调用堆栈中执行我们自己的代码，现在不同的引擎以稍微不同的方式实现，简而言之，这就是现代即时编译在JavaScript中的样子

#### 即时编译步骤图









### JavaScript运行时

1. 将JavaScript运行时想象成一个大盒子或一个大容器，其中包含我们在这种情况下使用JavaScript所需的所有东西，即在浏览器中

2. 对于任何JavaScript的核心，运行始终是一个JavaScript引擎，没有引擎就没有运行时，也根本没有JavaScript

#### js运行时包含webApi、回调队列、事件循环

1. 程序运行仅靠JavaScript引擎是不够的，我们还需要访问Web API

a) 使用与DOM或计时器甚至console.log相关的所有内容

b) 本质上，Web API是提供给引擎的功能，但实际上并不是JavaScript语言本身的一部分

c) JavaScript只是通过全局窗口对象来访问这些API(全局窗口就是window对象)，但是，Web API也是运行时的一部分仍然有意义的，因为运行时就像一个包含我们需要的所有JavaScript相关内容的盒子

2. 一个典型JavaScript运行时还包括一个所谓的回调队列

a) 这是一个包含所有准备执行的回调函数的数据结构，例如我们将事件处理函数附加到DOM元素上

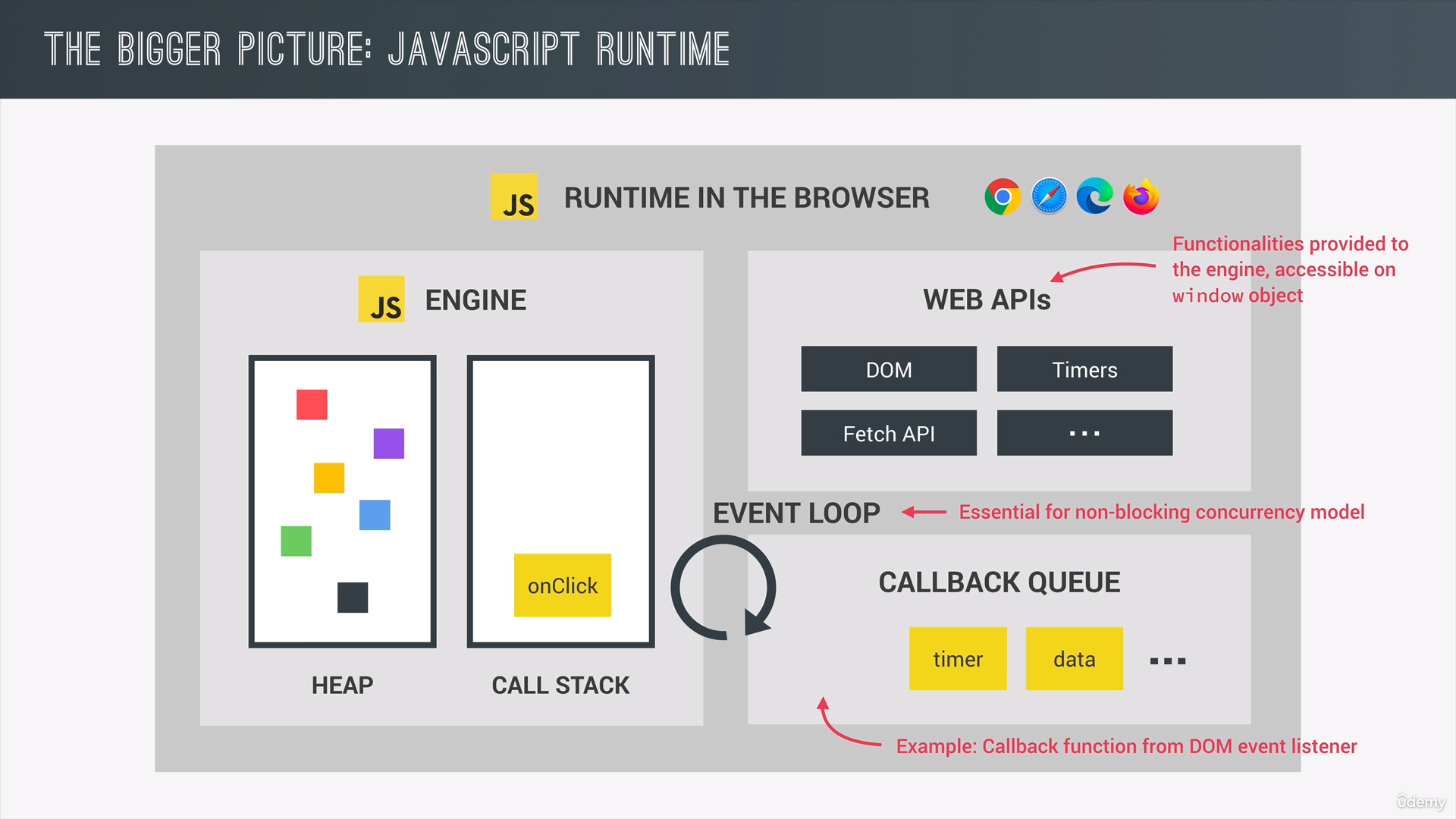
b) 事件处理函数也叫回调函数，事件发生后的第一件事就是回到函数被放入回调队列，然后当堆栈为空时，回调函数被传递到堆栈以便它可以执行

3. 这通过事件循环的东西发生，所以基本上事件循环从回调队列中获取回调函数并将它们放入调用堆栈中，以便它们可以执行

a) 事件循环是 JavaScript 的非阻塞并发模型

b) 后续详细介绍

#### 运行图



示例：来自DOM事件侦听器的回调函数

非阻塞并发模型的关键

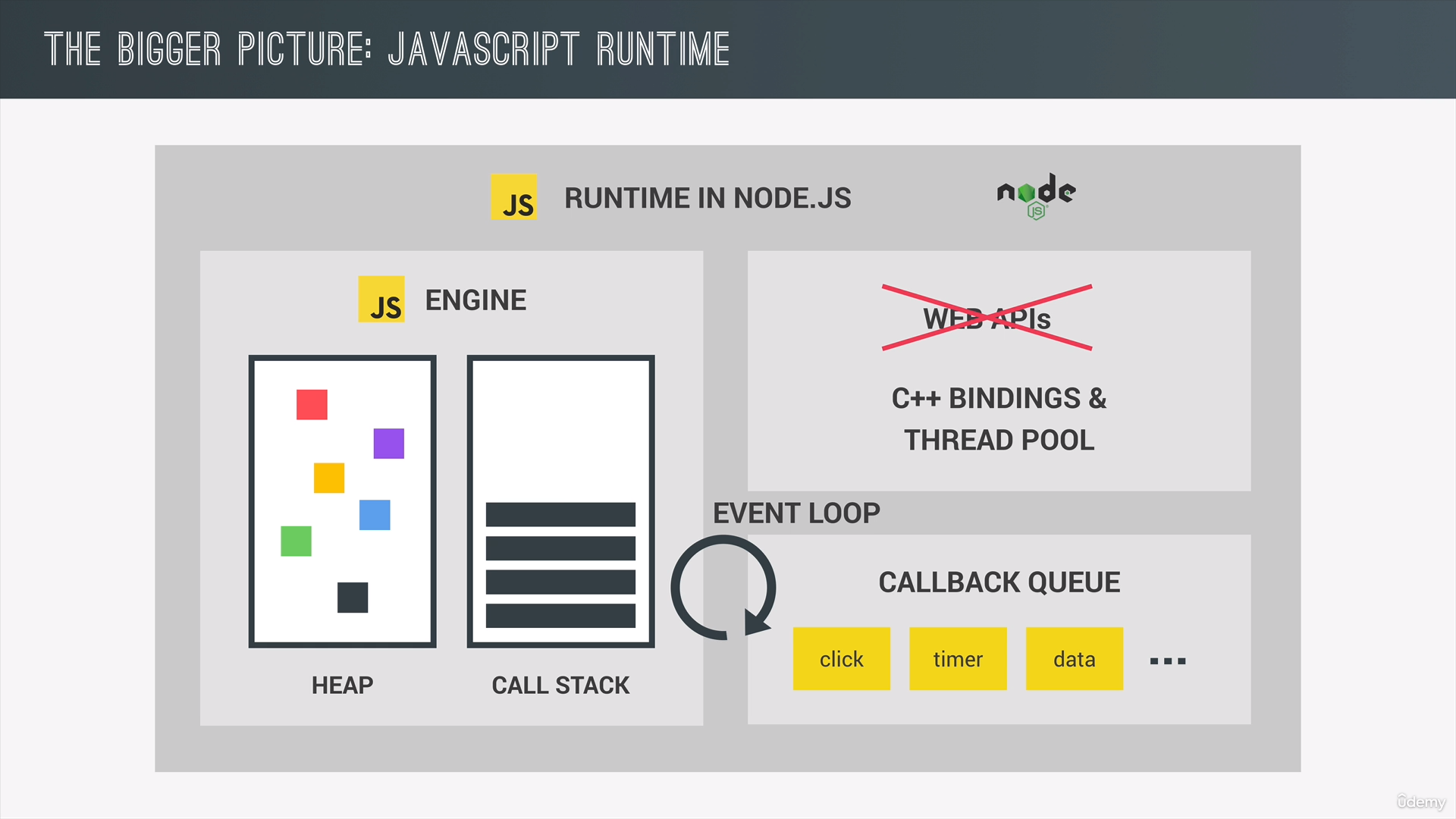
提供给引擎的功能，可在窗口对象上访问

### node.js可以让JavaScript运行在浏览器之外

1. Node.js是一个单线程、开源、跨平台的运行时环境，用于构建快速且可扩展的服务器端和网络应用程序。它运行在V8 JavaScript运行时引擎上，它使用事件驱动的非阻塞I/O架构，这使其高效且适用于实时应用程序。

2. 由于我们没有浏览器，我们不能拥有web API，因为提供这些的是浏览器，相反，我们有多个C++绑定和一个所谓的线程池





## 91.执行上下文和调用栈

1. JavaScript代码是如何执行的？我们已经知道代码执行在调用堆栈中，在引擎中

2. 假设我们的代码刚刚完成编译，会创建一个所谓的全局执行上下文，代码在创建的上下文中执行

### 执行上下文

#### 抽象概念

1. 执行上下文是一个抽象的概念，但我基本上将其定义为执行一段JavaScript的环境，它就像一个盒子，存储了要执行的某些代码的所有必要信息

a) 必要信息：局部变量或参数

b) JavaScript代码总是在执行上下文中运行

2. 以购买披萨实际行为举例

a) 为了让这更直观一点，让我们假设您在外卖店点了一份披萨，通常那个披萨是装在盒子里的，它还可能附带一些其他你吃披萨所必需的东西，比如餐具或收据，这样你可以在吃披萨之前付钱

b) 披萨是要执行的JavaScript代码，而盒子当然是我们披萨的执行上下文，那是因为吃披萨发生在盒子里，这就是吃披萨的环境，盒子里还有餐具和收据，这是吃披萨或执行代码所必需的

3. 任何JavaScript项目中，无论它有多大，都只有一个全局执行上下文，它始终作为默认上下文存在，并且是执行顶级代码的地方

#### 工作流程

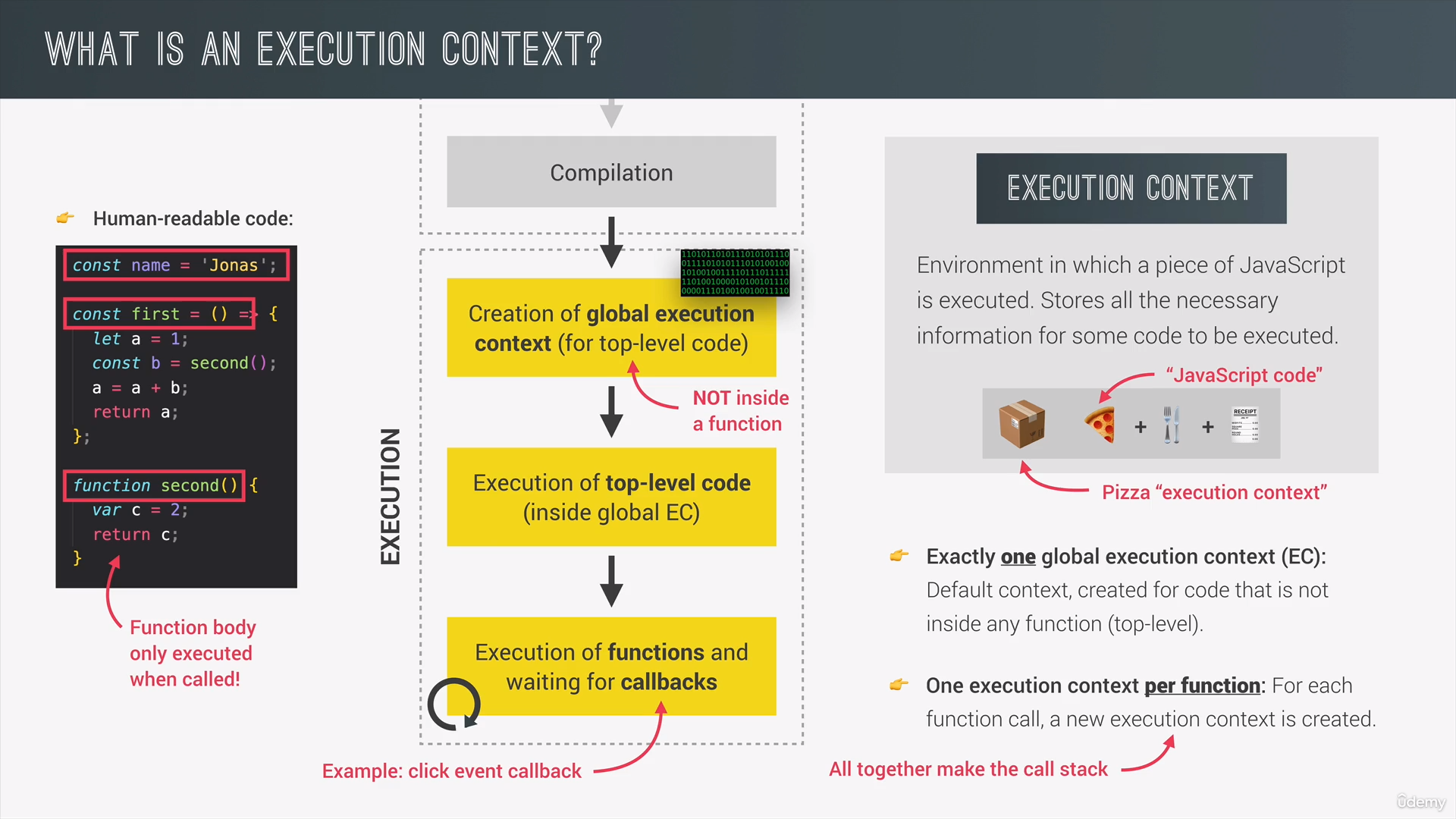
1. 创建全局执行上下文，因此有了可以执行顶层代码的环境

2. 执行顶层代码，执行就是cpu处理它收到的机器代码，当顶层代码完成，函数最终开始执行，这就是它的工作原理

3. 执行函数并等待回调，函数调用会创建执行上下文，其中包含运行该函数所需的信息，当然方法也是如此，因为他们只是附加到对象的函数，函数执行完毕后，引擎基本上会一直等待回调函数到达，以便执行这些函数.

4. 所有执行上下文一起构成了我之前提到的调用堆栈

#### 概念、工作流程图



#### 变量环境、作用域链、this关键字组成

1. 变量环境，在这个环境中，我们所有变量和函数声明都被存储了，还有一个特殊的参数对象，该对象包含传递给当前执行上下文所属函数的所有参数

a) 每个函数在调用函数时都会得到自己的执行上下文，所以基本上所有在函数内部以某种方式声明的变量，最终都会在它的变量环境中

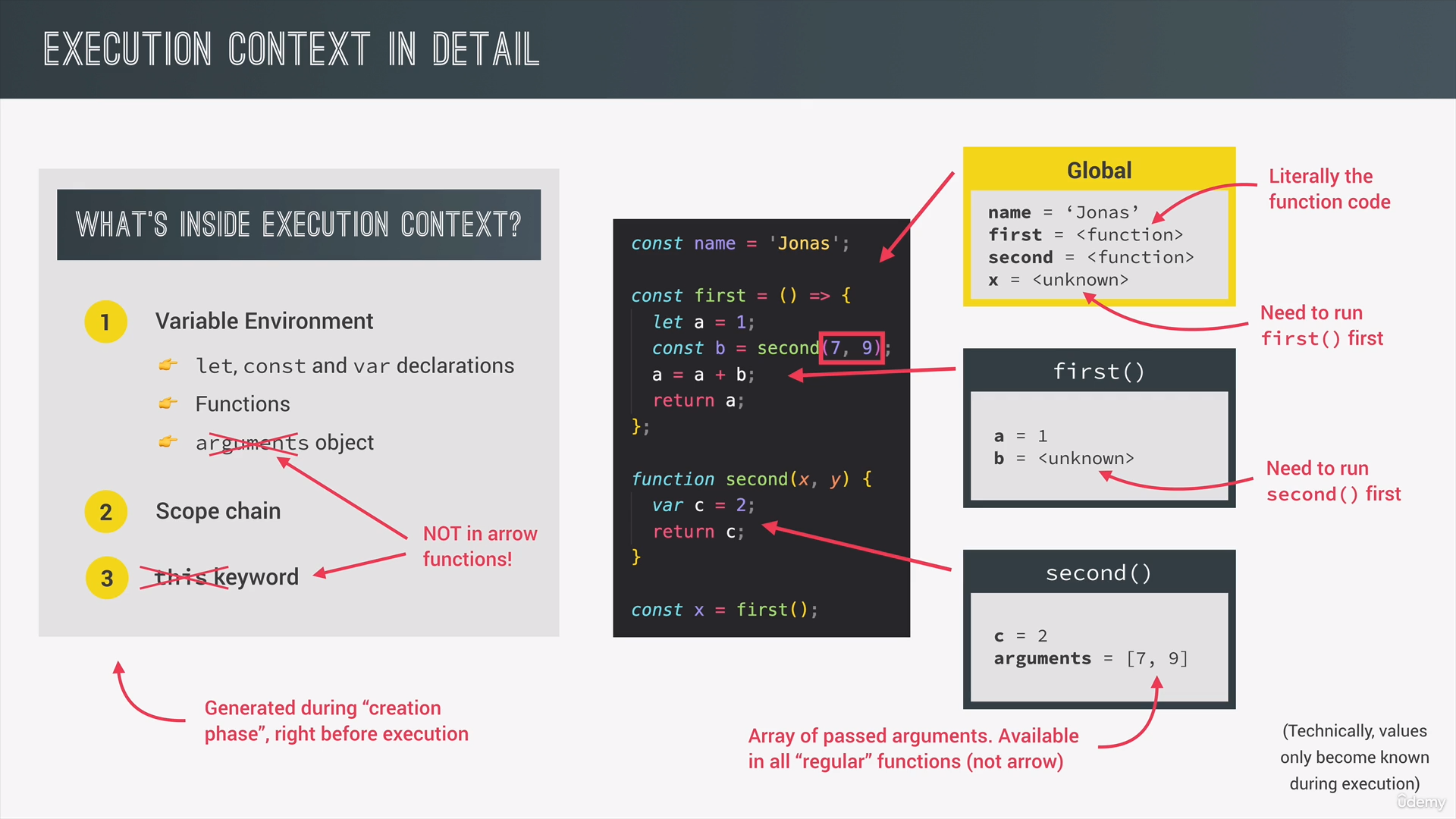
2. 作用域链，作用域链基本上由对位与当前函数之外的变量引用组成，可以访问函数外部的变量

a) 后续详细学习有关范围界定和范围链

3. this关键字，为了跟踪范围链，它存储在每个执行上下文中，最后每个上下文还获得一个称为this关键字的特殊变量

a) 后续会详细介绍this关键字

小结： 执行上下文的内容、变量环境、作用域链和this关键字都是在所谓的创建阶段生成的，这发生在执行之前，注意箭头函数没有参数对象和this关键字



传递的参数数组。可用于所有“常规”功能（非箭头）

### 调用堆栈

#### 意义

1. 让引擎可以在数百个执行上下文中，跟踪我们调用函数的顺序

2. 让引擎知道它目前在执行中的位置

3. 调用堆栈与内存堆一起构成了JavaScript引擎本身

#### 作用

1. 将执行上下文堆叠在一起，以便跟踪程序执行中的位置，因此堆栈顶部执行的上下文是我们正在运行的上下文

#### 工作原理--代码示例

1. 一旦代码被编译，创建一个全局执行上下文，开始执行顶部代码

2. 遇到函数调用将创建一个新的执行上下文进入堆栈，知道遇到return语句

3. return代表执行完毕，该函数的执行上下文将从堆栈中弹出，并从计算机内存中消失

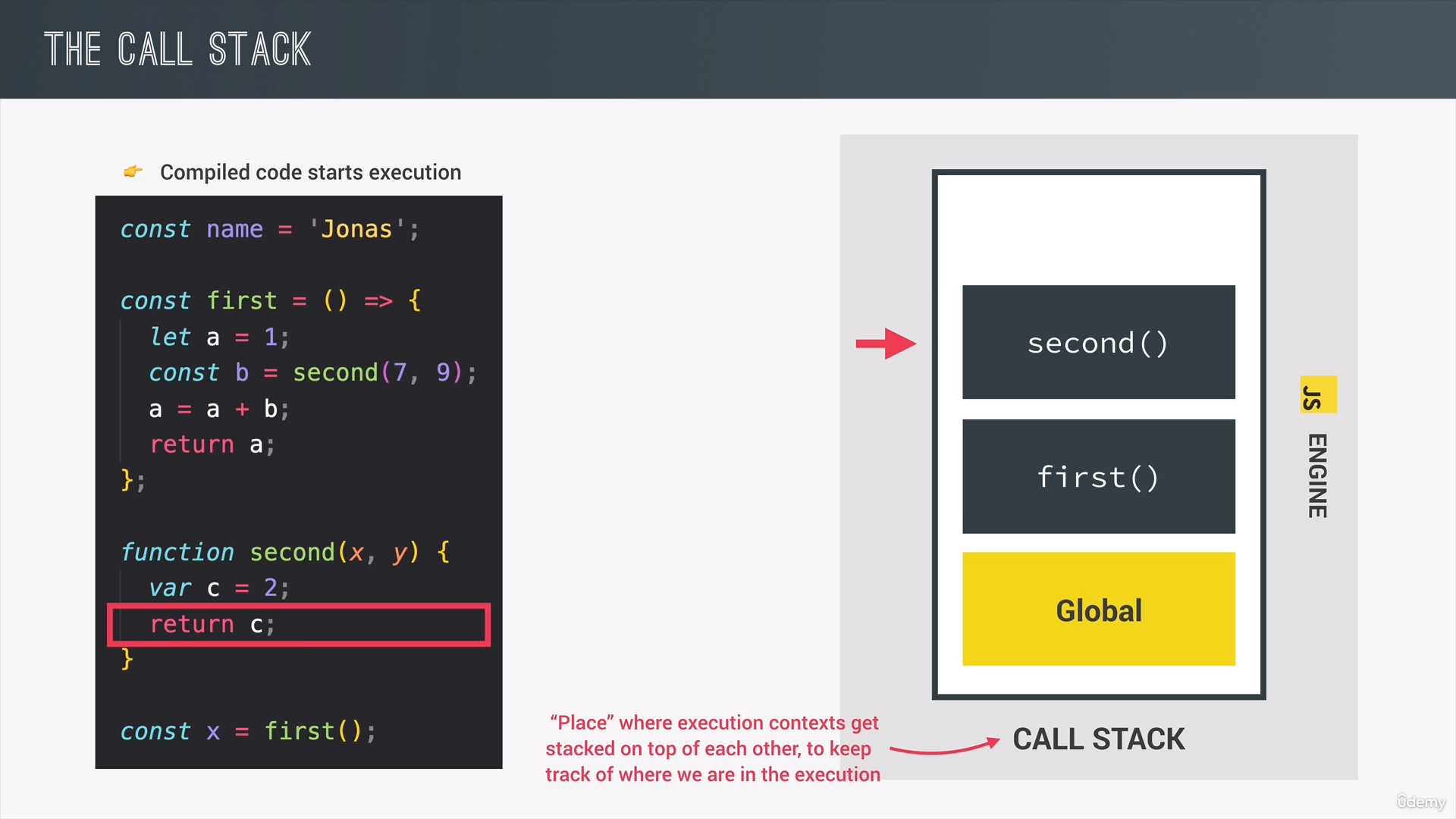
a) 注意弹出的执行上下文可能会一直存在于内存中

b) 后续会详细说明

4. 无论如何，接下来会发生的是，先前的执行上下文现在将再次成为活动的执行上下文

a) 从这里知道调用堆栈是如何追踪执行上下文顺序的

b) 调用堆栈比喻成JavaScript引擎的映射，因为调用堆栈保证了执行顺序永远不会丢失，就像地图一样



将执行上下文堆叠在一起的位置，以跟踪我们在执行中的位置

#### 小结

1. 执行上下文全部结束后，只有全局执行上下文会一直在调用堆栈中

2. 只有关闭浏览器选项卡或浏览器窗口时才会将全局执行上下文弹出

3. JavaScript代码在调用堆栈中运行，更准确的是在调用堆栈的执行上下文中运行

## 92.作用域和作用域链

1. 作用域控制了JavaScript引擎如何组织和访问我们程序的变量

### 范围的概念

1. 范围界定，变量在哪里？或者我们在哪里可以访问某个变量而在哪里不能访问？

2. 词法作用域，词法范围意味着变量的组织和访问方式完全由程序代码中函数和块的位置控制

a) 写在另一个函数内部的函数可以访问父函数的变量

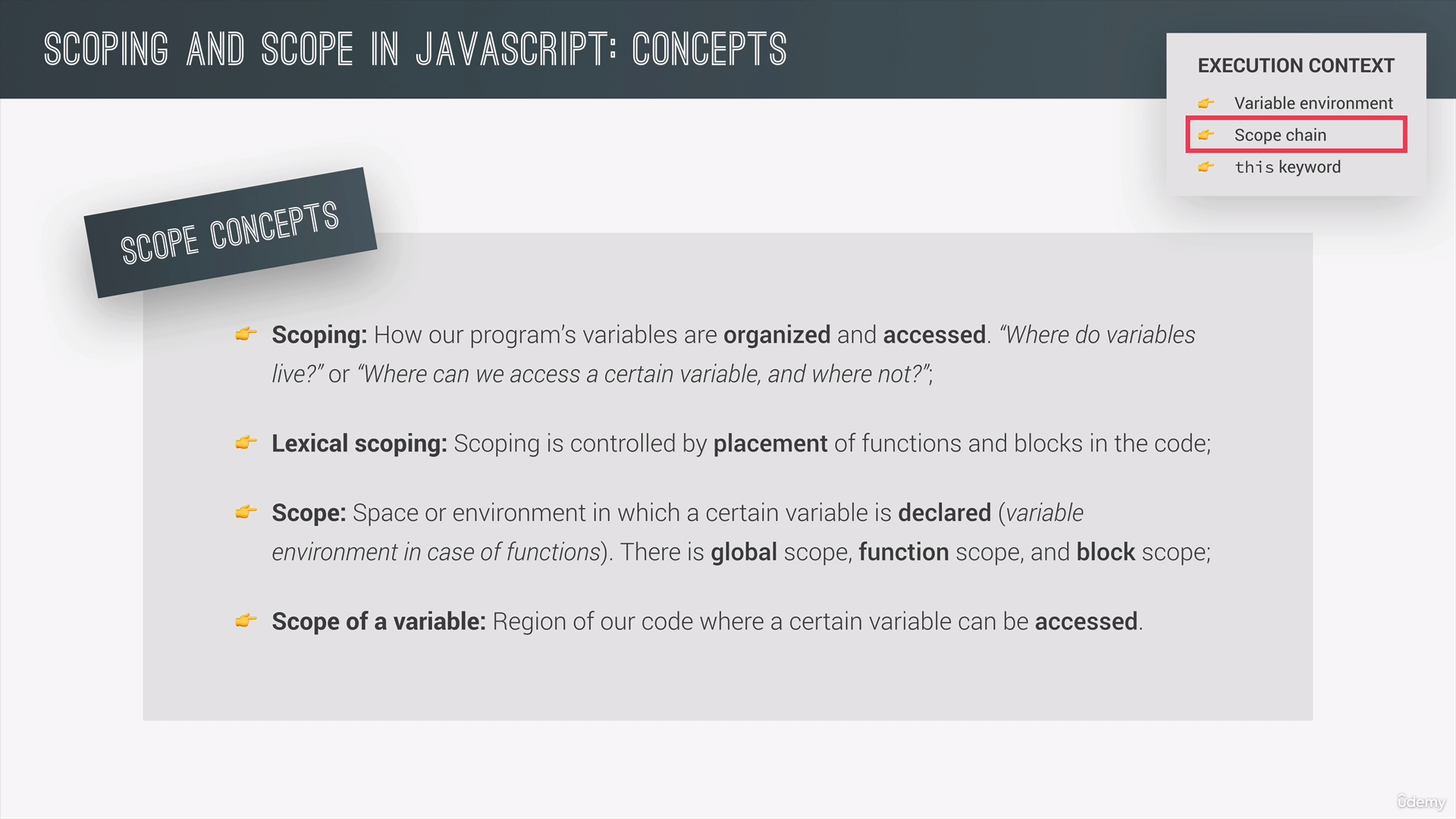
3. 范围，声明某个变量的空间或环境

a) 在函数中，本质上是存储在函数执行上下文中的变量环境

b) 范围和变量环境基本是一样的

4. 变量的范围，变量的作用域基本上是我们代码的整个区域，可以访问某个变量

总结：范围与变量的范围有细微的差异



变量的范围：代码中可以访问某个变量的区域

范围：声明某个变量的空间或环境（函数的情况下为变量环境）。有全局作用域、函数作用域和块作用域

范围界定：我们的程序变量是如何组织和访问的。变量住在哪里？“或者我们在哪里可以访问某些变量，在哪里不能访问？

词法作用域：作用域由函数和块在代码中的位置控制