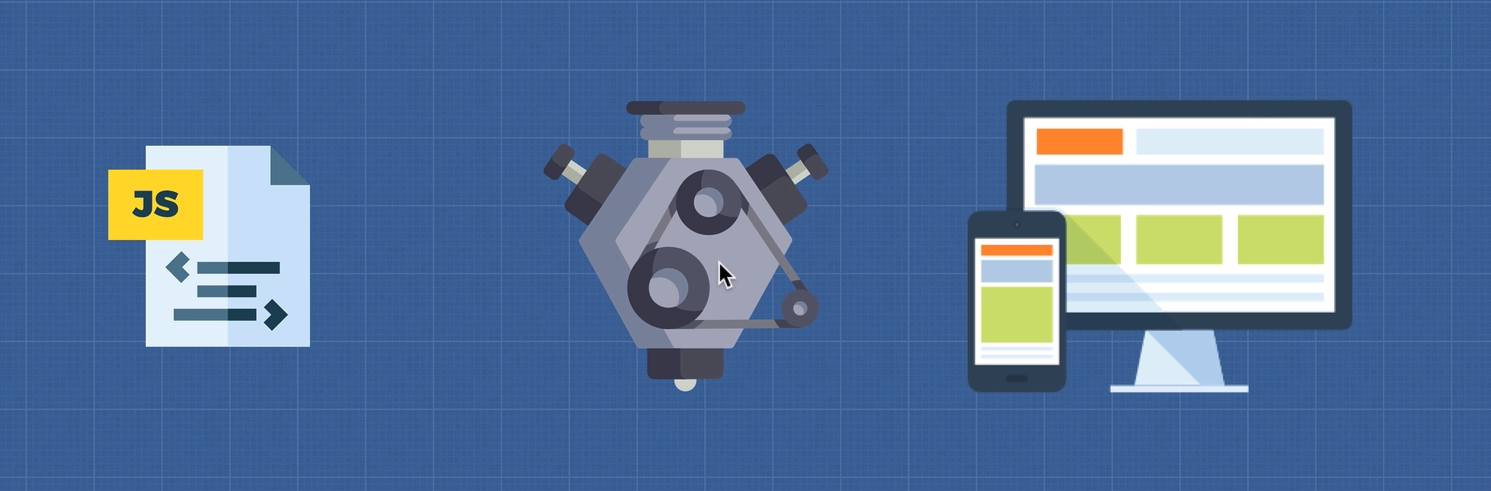
JavaScript: Advanced Concepts

# JavaScript Foundation

## JavaScript Engine

1. JS is a single threaded language.
2. It is an interpreted language.

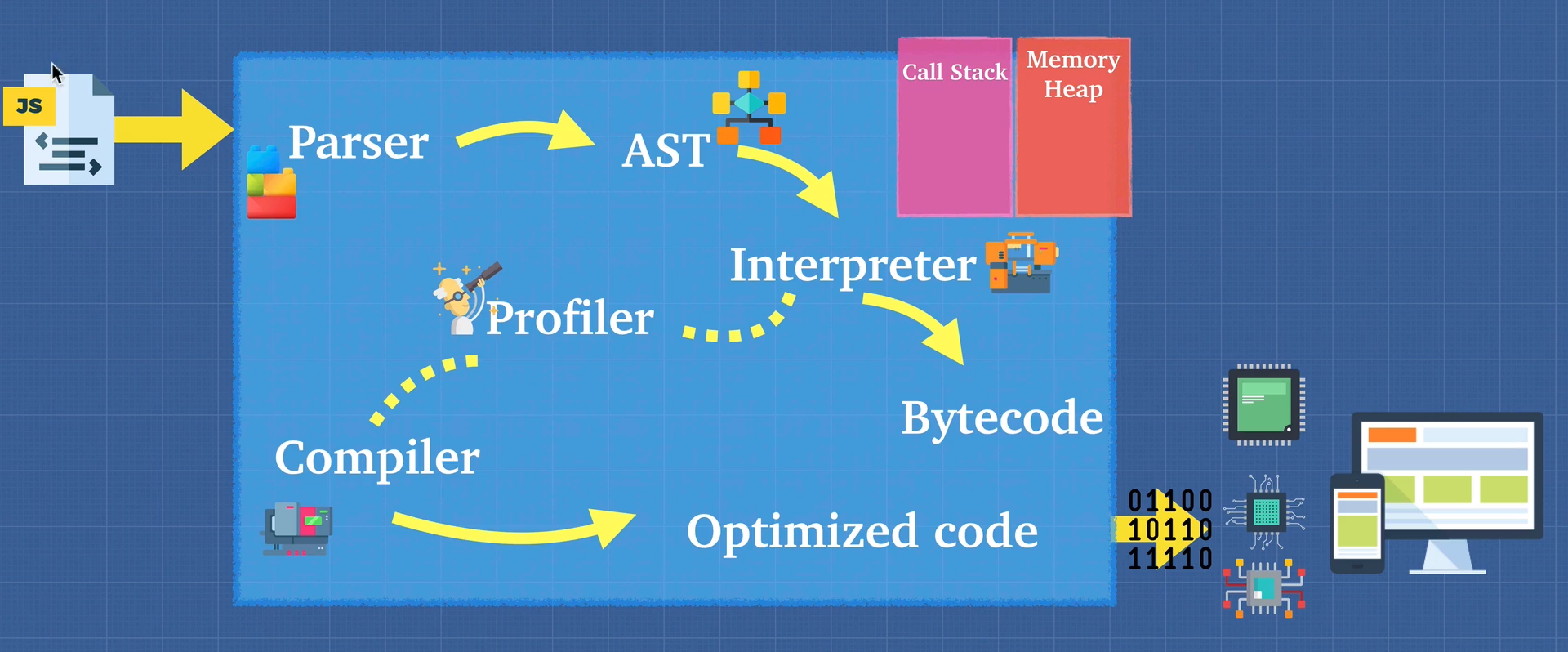
A JS Engine helps the computer understand a JS file by translating it into machine code.



There are various JS engines available, example, V8, Chakra, Chakran, etc.

Google’s V8 enables JS to run fast. It was released in 2008.

### How JS Engine works?

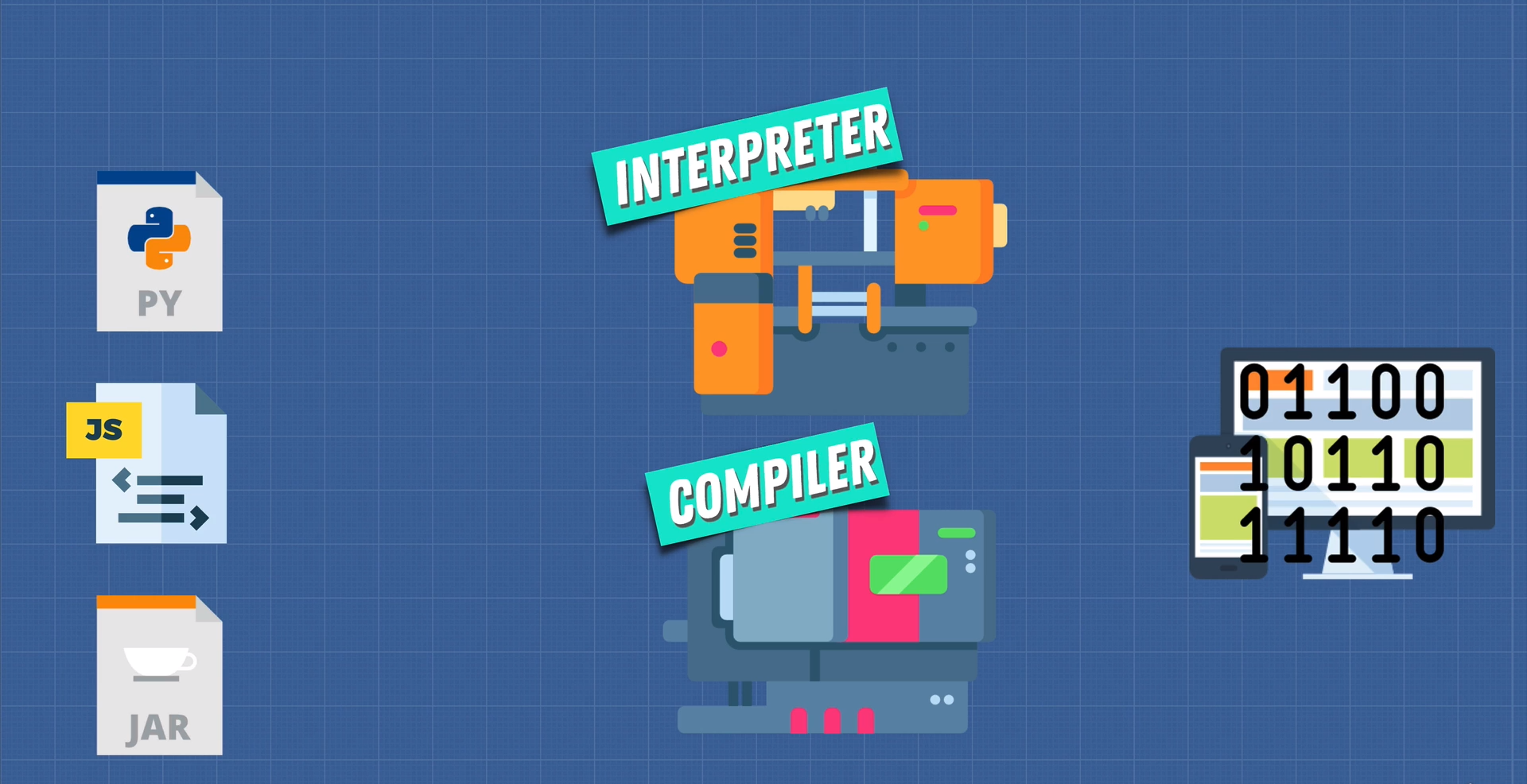


1. The first step is to do a **lexical analysis** of the JS code. This is done by a **parser** which tokenize the code.
2. Then an **AST (Abstract Syntax Tree)** is formed from the tokens.
3. Then an interpreter spits out byte code. This process is called **ignition**.
4. At the same time, a **Profiler** checks for places where the code can be optimized. Wherever, it finds such a code it passes it to the compiler to optimize that code. Example, a for loop.
5. Now the compiler replaces the optimized code in the byte code. The output, byte code + optimized code = machine code that is dumped out.

***JavaScript can thus be an interpreted language or compiled language or both depending on the engine.***

**NOTE: A JS Engine should conform to ECMAScript standard.** [List of ECMAScript engines - Wikipedia](https://en.wikipedia.org/wiki/List_of_ECMAScript_engines)

## Interpreter & Compiler



### Interpreter

* Reads file line by line

### Compiler

* Works ahead of time.
* Compiles down at beginning into machine code.
* However, at the end, the machine code is interpreted by the CPU

### Pros & Cons

* Interpreter starts up quickly and is suited for JS as the browsers need to work fast.
* Startup is slow in a compiler but it runs faster at runtime because the entire code is in machine code, i.e., compiler does optimization.

Taking the pros of both interpreter and compiler, a new compiler was developed called the **JIT Compiler**. V8 uses JIT Compiler.

## Babel & TypeScript

Have you heard of Babel or TypeScript? They are heavily used in the Javascript ecosystem and you should now have a good idea of what they are:  
  
[Babel](https://babeljs.io/) is a JavaScript compiler that takes your modern JS code and returns  browser compatible JS (older JS code).  
[Typescript](https://www.typescriptlang.org/) is a superset of JavaScript that compiles down to Javascript.  
  
Both of these do exactly what compilers do: Take one language and convert into a different one!

## WebAssembly

It is a standard for compiling the JS code for browsers. This can remove the need for a JS Engine as there will be a standard just like .exe files.

## Call Stack and Memory Heap

### Call Stack

It is a register where the instructions of the program are stored. It’s a LIFO data structure. Since JS has a single call stack, i.e., V8, it is a single threaded language. It also makes JS synchronous.

### Memory Heap

It’s a place where variables and other memory allocations are stored and released. There is no order for storage.

## Garbage Collection

JS is a garbage collected language, i.e., it automatically frees up the memory heap from variables which are no longer in use. This ensures there is no memory leak. In low level programming languages like C, the memory management is done by the programmer which gives more control to them but can also be dangerous if not implemented properly.

JS uses **mark and sweep** algorithm to remove unreferenced variables.

## Memory Leaks



## JavaScript Runtime



Each browser comes with a web API which can do DOM manipulation, API calls, and set timeouts. These web APIs are asynchronous. When the call stack encounters a web API call, it moves to the next statement and assigns the async call to Web API. Once the Web API finishes its work, the results are sent to the **callback queue**. An event loop constantly checks if the call stack is empty or not and if the entire file has been run. If it is empty and the entire file has been run, the results of call back queue are placed on the call stack. This is how asynchronous calls are handled in JS.

## Node.js

It is a JavaScript runtime built upon Chrome’s V8 JS engine. It’s a C++ program which provides a JS runtime outside a browser. Node.js can access users file system, etc which the conventional browser cannot do.

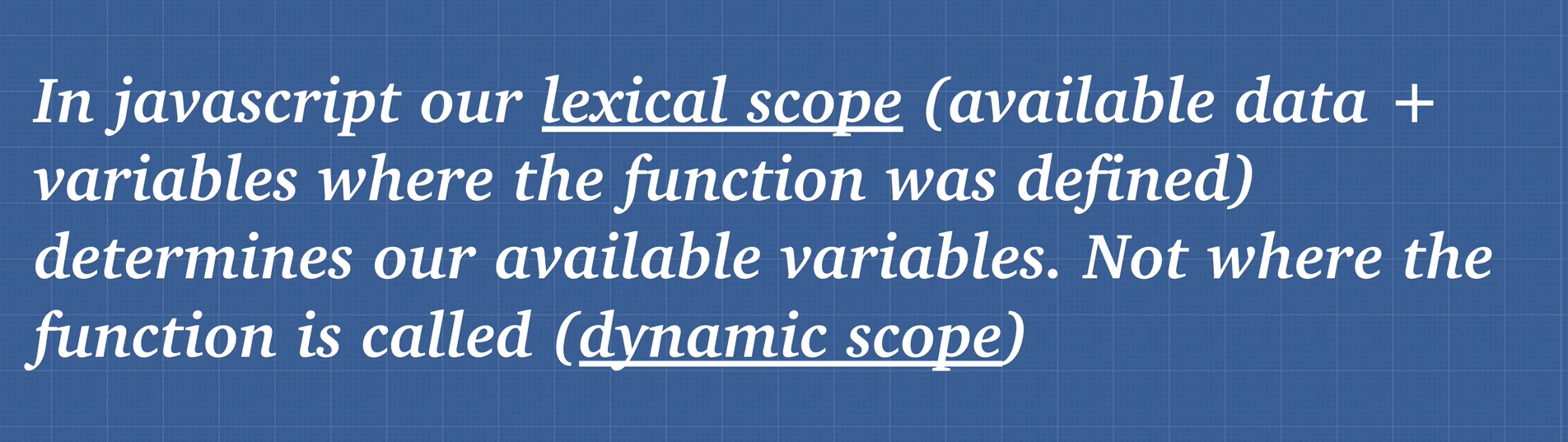
# JavaScript Foundation 2

## Execution Context

Execution context is the place where the code is run by a JS engine. By default, an engine gives **global execution context**, where you get a **global object** and a ***this*** keyword. Both point to the same thing. In a browser, the global object is *window* and both *this* & *window* are same thing.

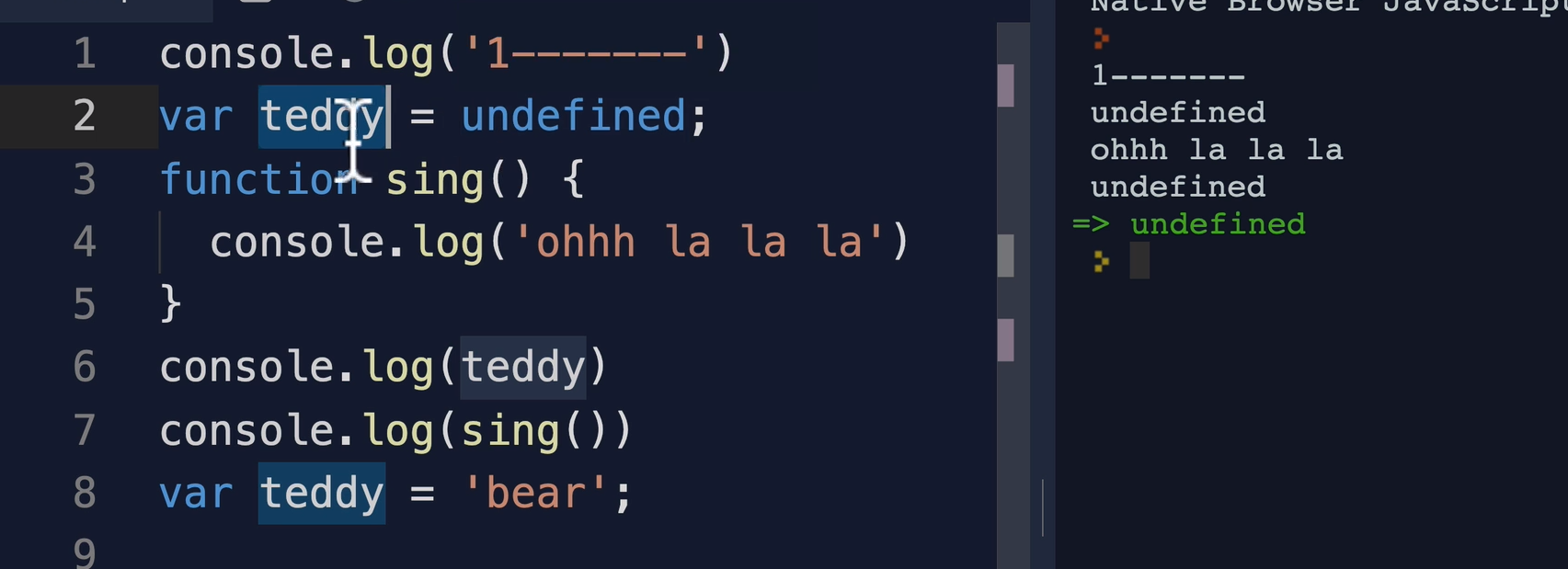
On top of the global execution context, functional contexts are added. When a program is closed, the global execution context is the last thing to popped out from call stack.

## Lexical Environment



## Hoisting

The process of assigning space in the memory heap for *var* and *function* keywords before the program is executed, is called hoisting.



*Var* is partially hoisted. They are initialized with *undefined* status. Functions are fully hoisted, i.e., you can call the function before it is defined in a file.

Hoisting happens on each execution context. So, if there is a same variable name in a function like a global variable name, then the function variable name would be assigned undefined instead of global value.

**Sometimes hoisting can allow bugs to creep in. To avoid hoisting, use *let*** **and *const* keywords.**

## Function Invocation

### Arguments keyword

Just like we get *this* in the global context, we get *this* in a function context along with *arguments*.



However, using *arguments* confuses the JS engine. To avoid this use *(…args)* which can be used in function scope as an array of arguments, just like in Python *\*args*.