JavaScript :
Under The Hood (Part 1)

by Faisal Ahmed

About ME

Faisal Ahmed

Sr. Software Engineer

Tech Stack:

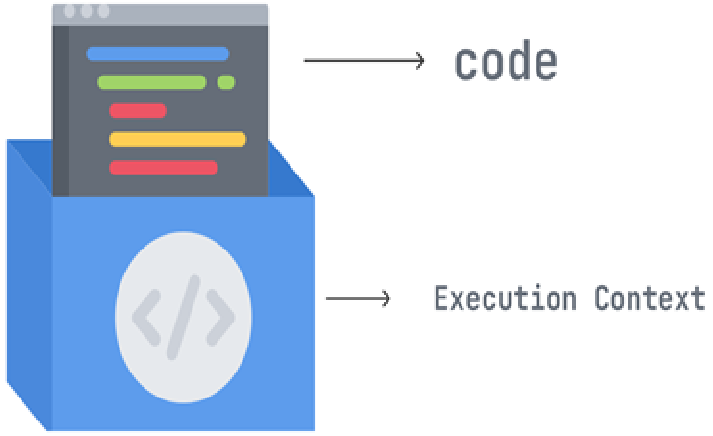
PHP, Laravel, NodeJS, JavaScript, SQL, NoSQL

Topics We Cover

- Execution Context
- JavaScript Engine

Introduction to Execution Context

The environment in which your code is running is Execution context. It is created when your code is executed



Execution Context

There are two kind of execution context in js.

Global

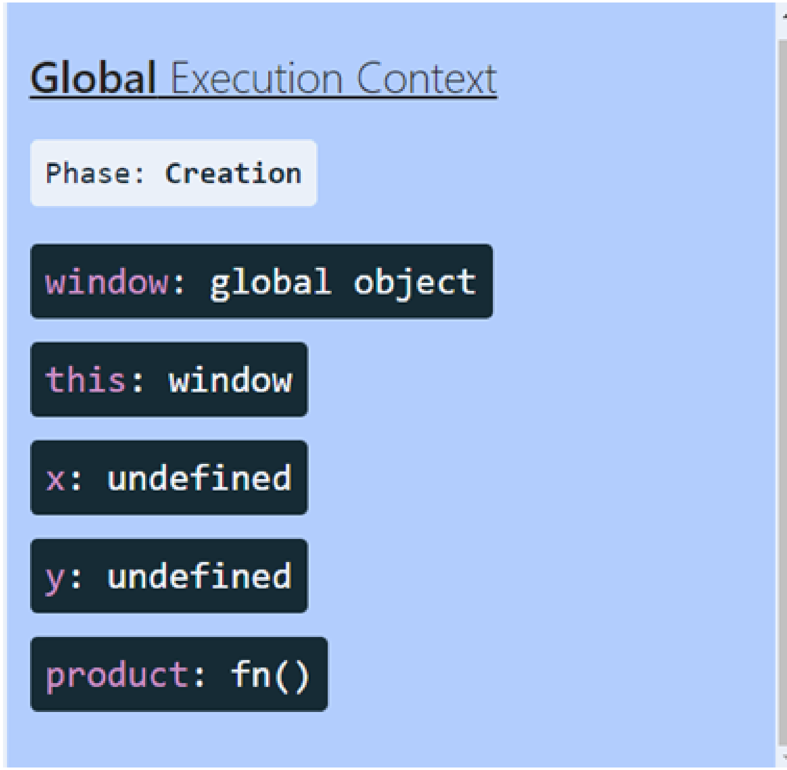
Function

Execution Context

When ever the js engine received script file, 1st create a default execution execution context known as a global execution context.

The GC is the base default execution context where all js code that is not inside of function gets executed.

While executing the script if the js engine encounters a function invocation, it creates a different type of execution context known as a function execution context within the GC to evaluate and execute the code within that function.


Global Execution Context

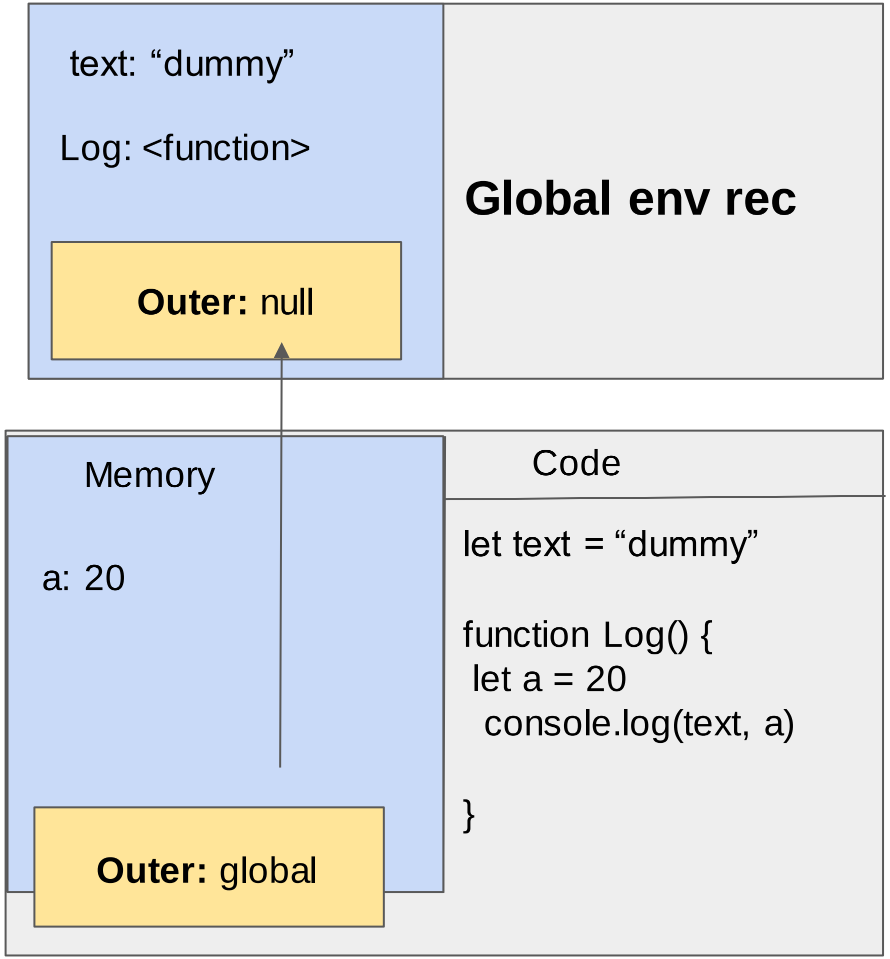
Phase: Creation

```
window: global object
this: window
x: undefined
y: undefined
product: fn()
```

Lexical Environment

Definition

A lexical environment in js is just a simple object consisting of two parts, a reference to the outer environment and environment record.



```
let text = “dummy”

function Log() {
 let a = 20
  console.log(text, a)

}
```

Lexical Environment

The outer environment reference is just what it sounds like, a reference to the parent environment in which the lexical one was created.

The second part of the lexical environment, which the environment record is basically just an identifier representing the variable environment. Which is truly fancy representation of lexical environment’s local memory.

It’s used for the initial storage of variables, arguments and function declarations.

Thread of Execution

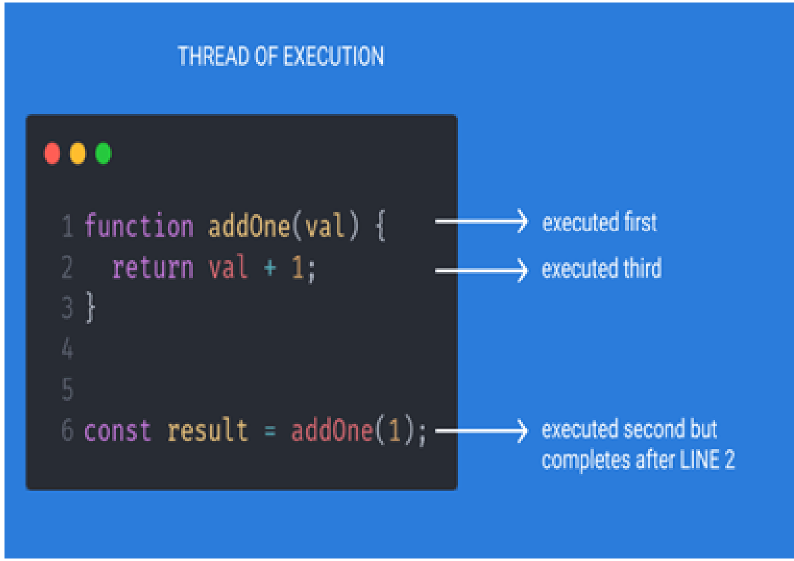
Thread of execution

Javascript goes through code line by line and executes it, known as a thread of execution.

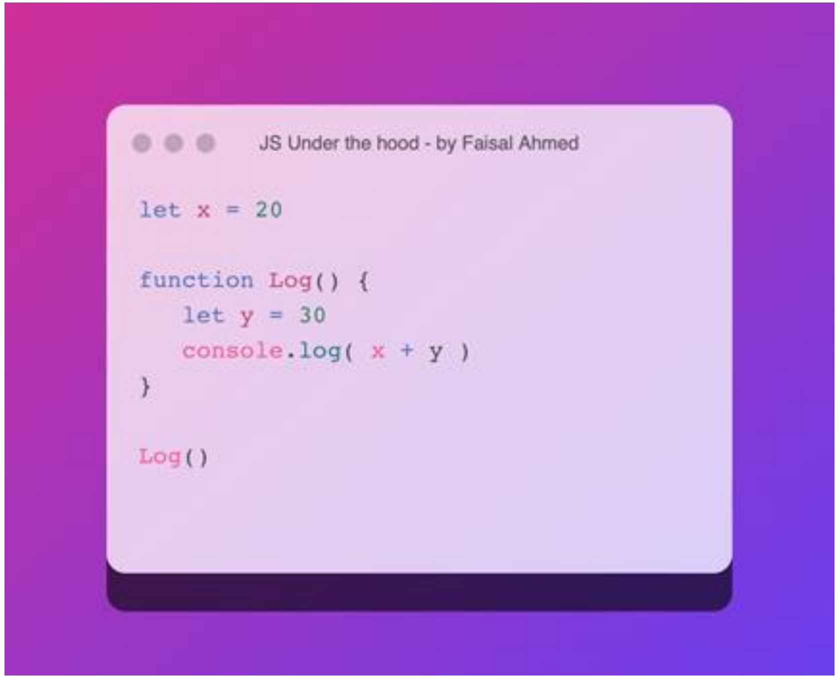
That is, it threads its way down the code, top to bottom, and executes each line.

Javascript synchronous which means it moves to the next line only when the execution of the current line is completed.

And it’s therefore called a single-threaded language, which means it can execute one command at a time in a specific order, serially.



```
1 function addOne(val) {
2   return val + 1;
3 }
4
5
6 const result = addOne(1);
```



```
let x = 20

function Log() {
   let y = 30
   console.log( x + y )
}

Log()
```

Thread of execution

In the corresponding piece of code, the order of execution will be as follows.

At each point, the code is executed within one of the execution contexts.

The global one, or a function one.

As js passes code it does one of two things:

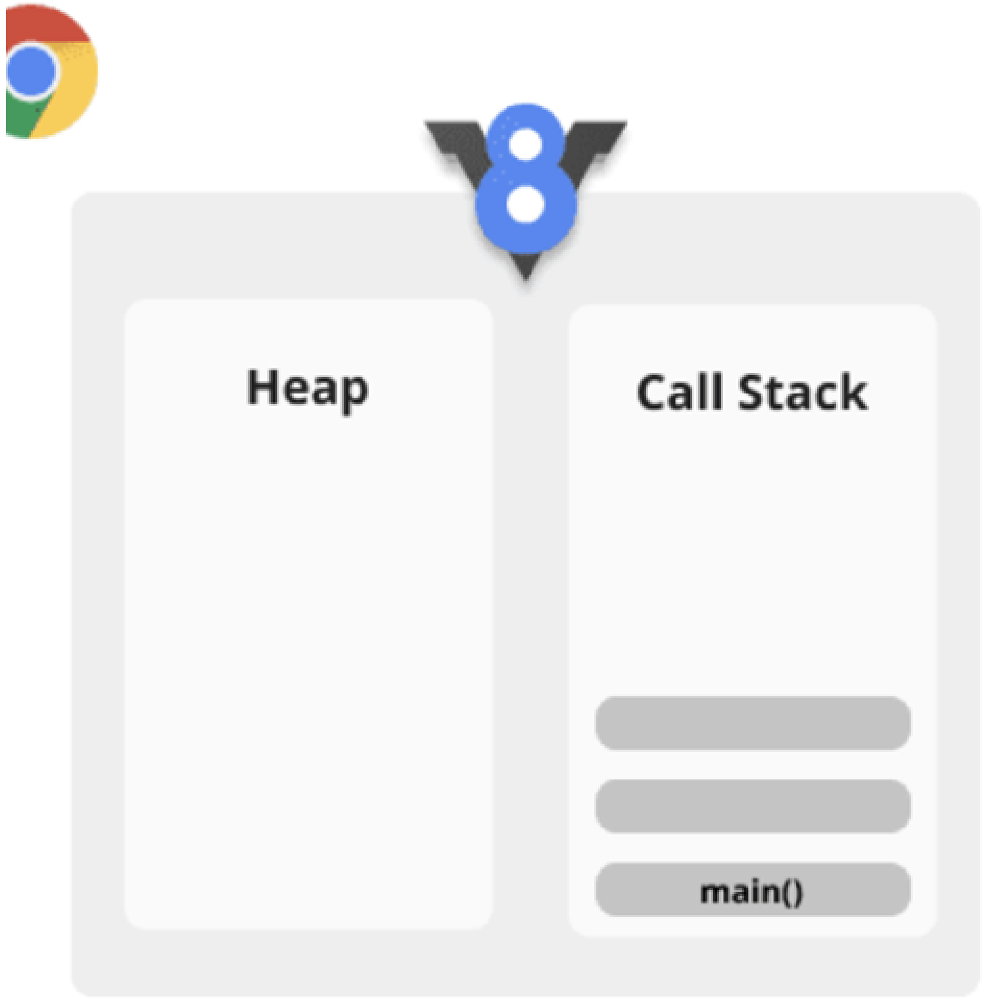
1. Stores code in memory
2. Executes code

Call Stack

V8

The v8 engine has two important components, a call stack, and a heap.

JavaScript, being a single-threaded programming language, it can only perform one thing at a time and has only one call stack.

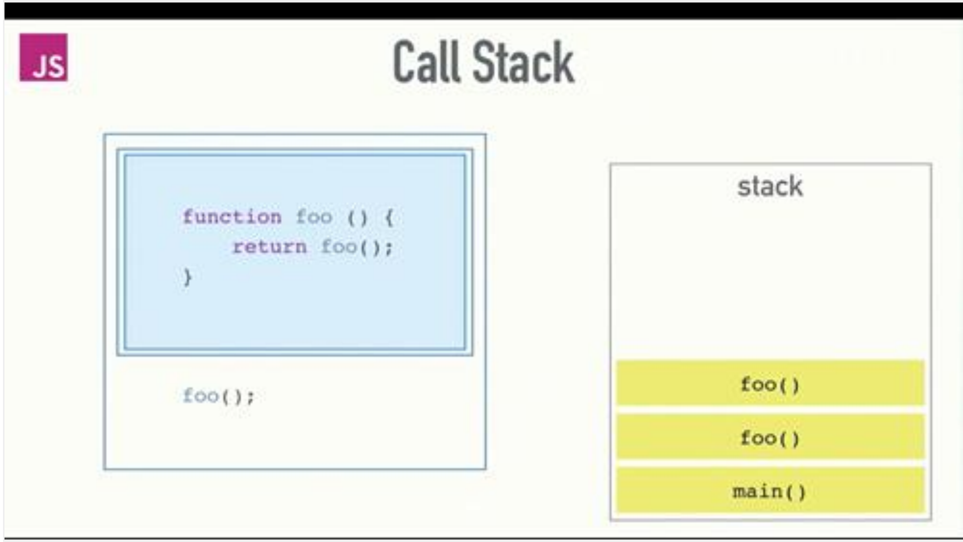


Story for illustration purposes only

Call stack

Call stack is basically a last in, first out data structure that is used for function calls that record where we are in the program.

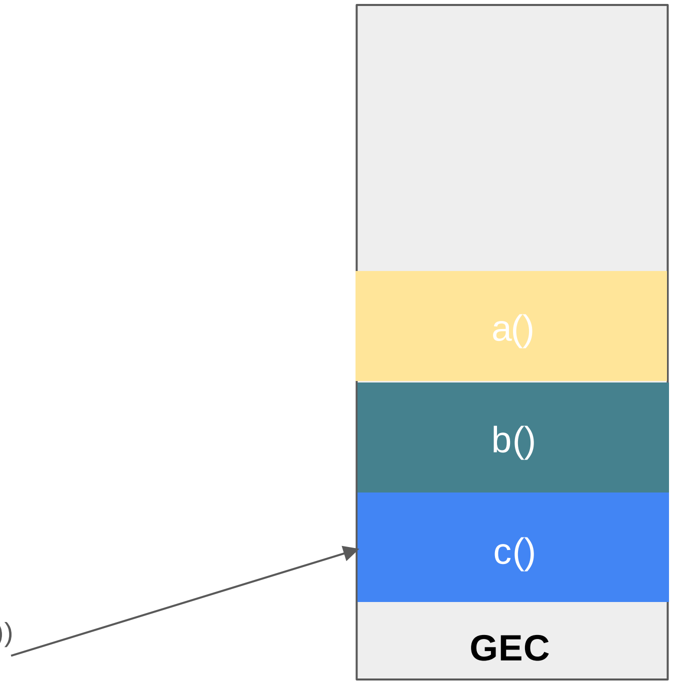
Let’s take a look an example and see how the call stack works.



```
function foo () {
    return foo();
}

foo();
```

Call Stack

```
function a() {

 return 1

}

function b() {

 return a() + 1

}

function c() {

 return b() + 1

}

console.log(c())
```



Consider the simple js code. It start from line number one.
The js engine manages the execution context.
The execution context is basically created in two phase,

1. Creation phase
2. Execution phase

Let's take a look at the execution phase for this specific code example. It start execution on line number one, and reads every single function one by one until it encounters function invocation, which is at the bottom of the code. Once function invocation encountered a separate execution context is created inside or within the global execution context, and this happens for every single function that it is going through. Once all the functions are read, it starts beginning to execute them one by one from the top, since the call stack is a last in, first out data structure

Hoisting

Hosting

Js run everything inside an execution context. It the environment variable in which our code is run.

There are two phases:

1. Creational phase
2. Executional phase

During the creation phase the js engine does a lot behind the scenes.

- Scope management
- Setting “this” value
- Encounter variable object

Variable Object

The variable object is an object-like container created within an execution context. It stores the variables and function declarations defined.

Hosting

Memory

Name	Value
sum	<function>
a	undefined
b	uninitialized

```
console.log(a)
console.log(b)
console.log(sum())

function sum(a, b){
  return a + b
}

var a = 20
let b  = 10
```

This process of storing variables and function declarations in memory prior to the execution of code is known as hoisting.

Recap:

Functions and variables are stored in memory in an execution context before we execute our code. This is called hoisting.

JS Engine

Stages of Compilations

There are seven prominent stages inside a JS engine.

They are as follows ->

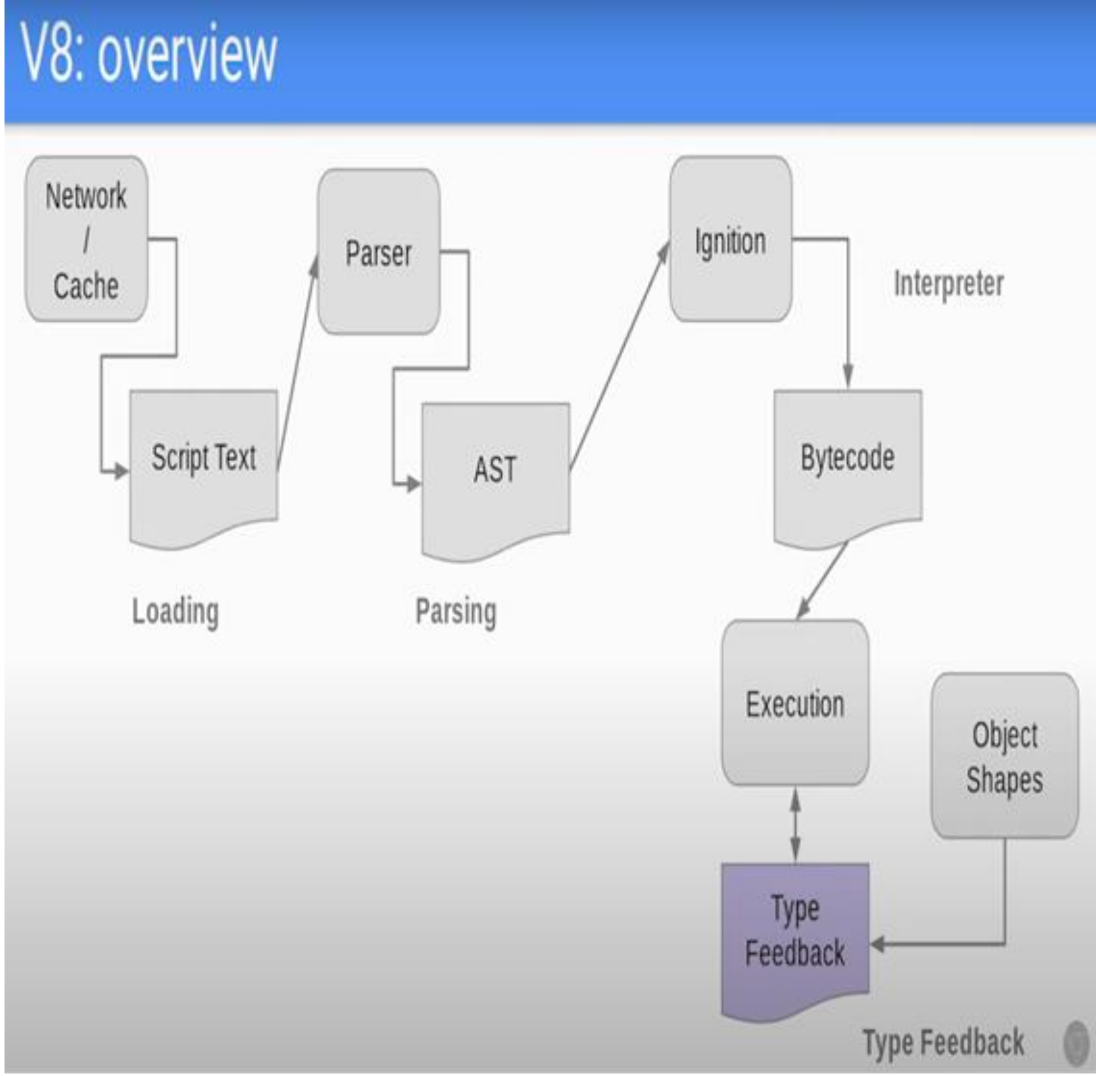
Code from this source gets loaded from either the network, cache, or an installed service worker. The response is the requested script as a stream of bytes, which the byte stream decoder takes care of! The byte stream decoder decodes the stream of bytes as it's being downloaded.

The byte stream decoder creates tokens from the decoded stream of bytes.

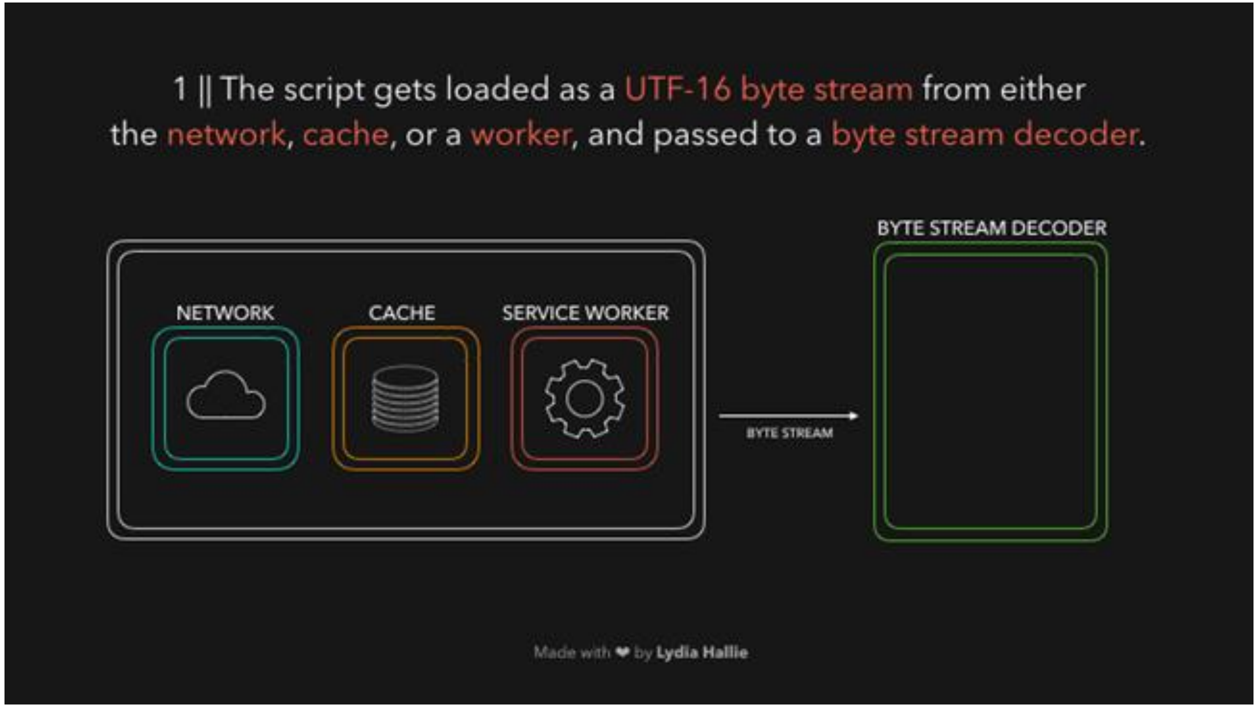
The parser divides the code into multiple tokens based on certain keywords that it recognizes.

It's converted into an abstract syntax tree, a tree-like structure that represents functions, conditionals, scopes, etc. The AST is passed to the interpreter which converts the code into byte code. At the same time engine is actually running the js code as well. Byte code is used by the optimizing compiler along with profiling data. The optimizing compiler makes certain assumptions based on the profiling data and produces a highly optimized machine code.

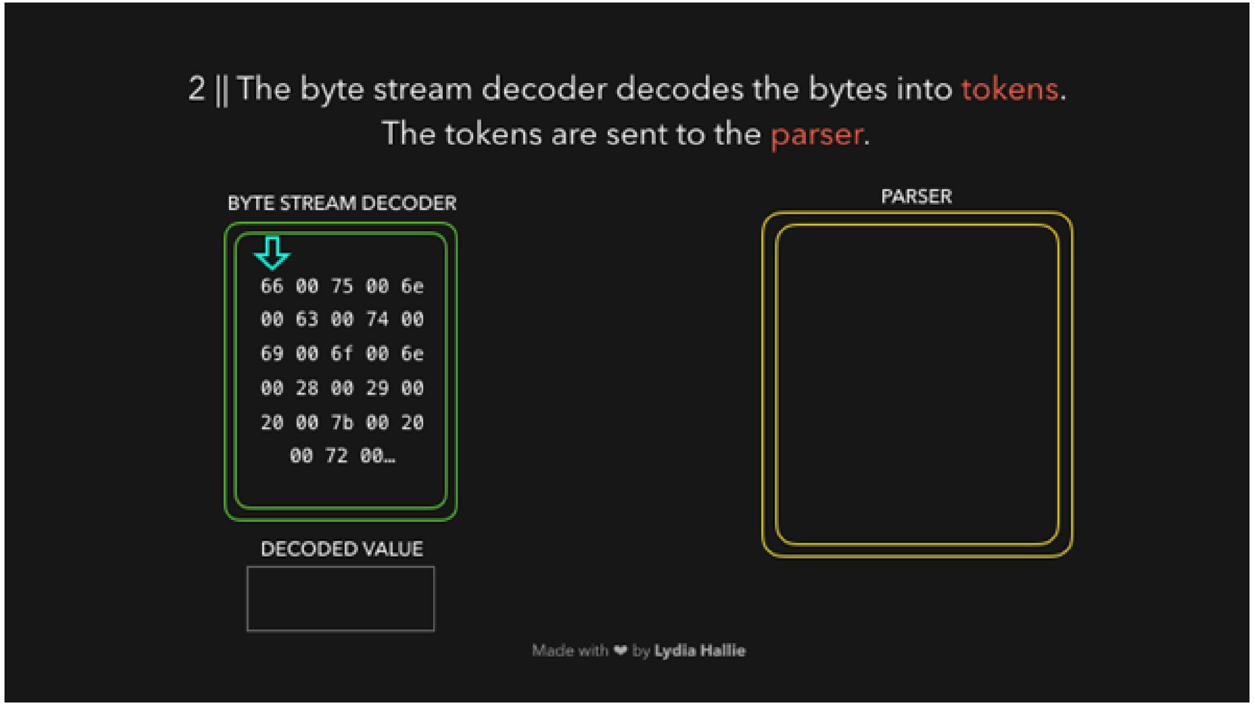
This is how parser work.



Stages of Compilations

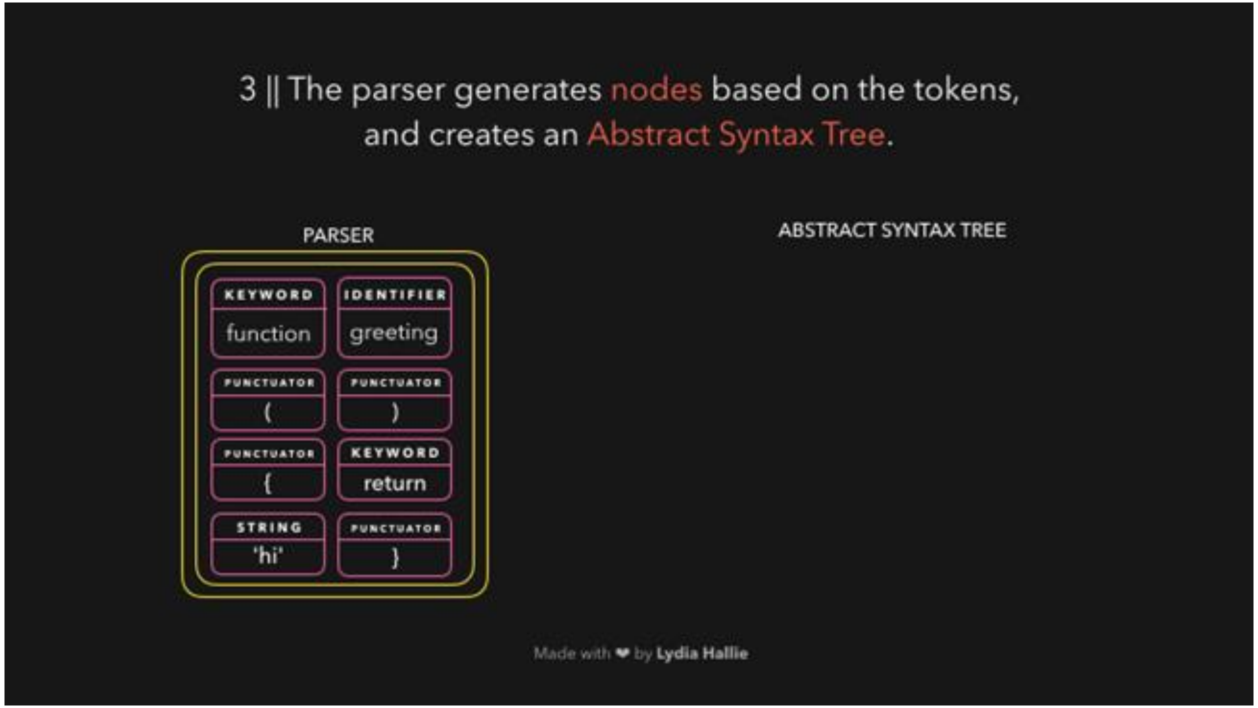

1 || The script gets loaded as a UTF-16 byte stream from either the network, cache, or a worker, and passed to a byte stream decoder.

Stages of Compilations

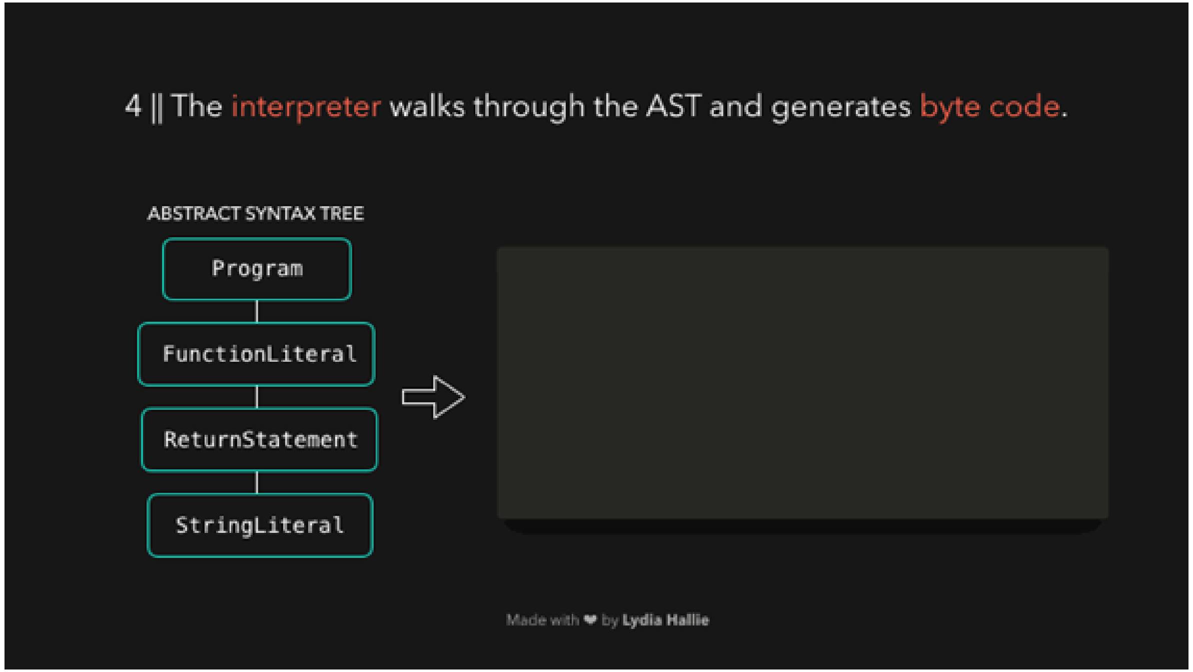

2 || The byte stream decoder decodes the bytes into tokens.
The tokens are sent to the parser.
BYTE STREAM DECODER
PARSER
DECODED VALUE
Made with ❤ by Lydia Hallie

```
66 00 75 00 6e
00 63 00 74 00
69 00 6f 00 6e
00 28 00 29 00
20 00 7b 00 20
   00 72 00…
```

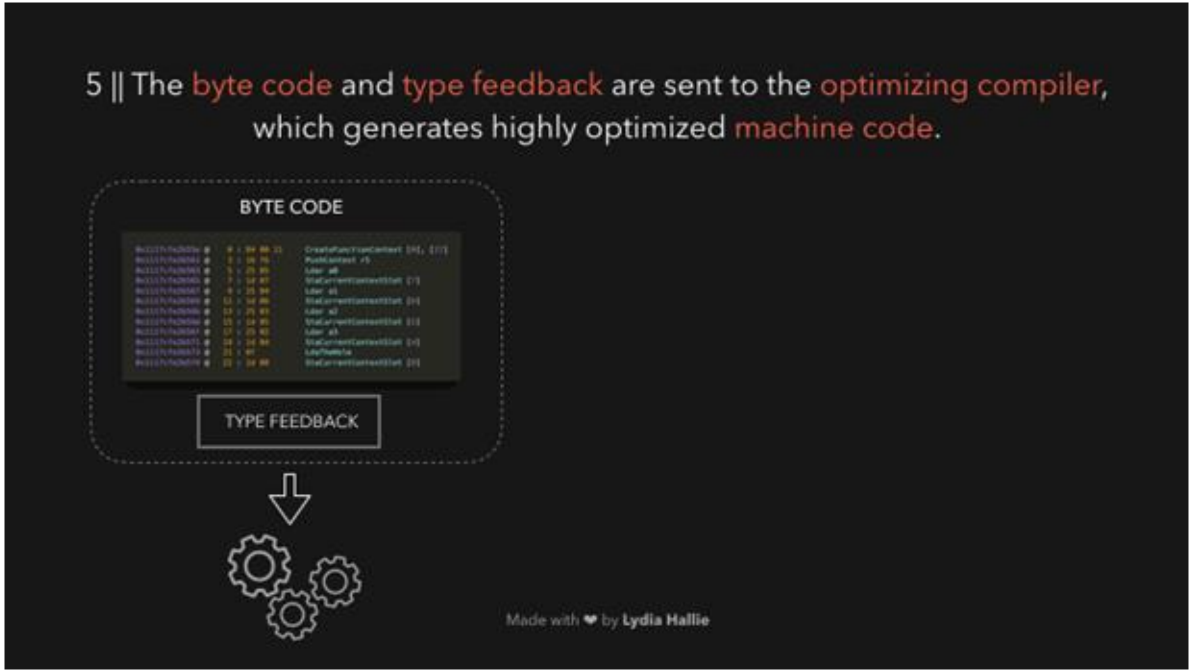
Stages of Compilations



Stages of Compilations


4 || The interpreter walks through the AST and generates byte code.
ABSTRACT SYNTAX TREE
Program
FunctionLiteral
ReturnStatement
StringLiteral
Made with ❤ by Lydia Hallie

Stages of Compilations



5 || The byte code and type feedback are sent to the optimizing compiler, which generates highly optimized machine code.

```
[illegible] @    0 : [illegible]    CreateFunctionContext [illegible]
[illegible] @    3 : [illegible]    PushContext r5
[illegible] @    5 : [illegible]    Ldar a0
[illegible] @    7 : [illegible]    StaCurrentContextSlot [illegible]
[illegible] @    9 : [illegible]    Ldar a1
[illegible] @   11 : [illegible]    StaCurrentContextSlot [illegible]
[illegible] @   13 : [illegible]    Ldar a2
[illegible] @   15 : [illegible]    StaCurrentContextSlot [illegible]
[illegible] @   17 : [illegible]    Ldar a3
[illegible] @   19 : [illegible]    StaCurrentContextSlot [illegible]
[illegible] @   21 : [illegible]    LdaTheHole
[illegible] @   22 : [illegible]    StaCurrentContextSlot [illegible]
```

Thank You