Advanced JavaScript

Introduction

Sahil Khosla

Software Team Lead

Toronto, Canada

Agenda

1. Objects, Hoisting and Execution
2. Functions and IIFEs
3. Closure, Apply/Call/Bind
4. ES6 Syntax Refresher
5. Classes & Inheritance
6. Design Patterns
7. Async Programming

Section 1

Objects, Hoisting and Execution

Learning objectives of this section:

- An object in JavaScript
- What’s provided out of the box
- Execution Context
- Execution Stack
- Variable Environments
- Scope Chain
- Hoisting and Executing

What is an Object?

A collection of data in the form key/value pairs.

What is an Object?

```
name = ‘foo’
```

What is an Object?

```
{
  name: ‘foo’,
  gender: ‘male’
}
```

What is an Object?

A collection of data in the form key/value pairs.

- The key/value pairs are called properties
- The value can be some data, a function or another collection of key/value pairs
- If the value is a function then the property is called a method

What is an Object?

```
{
  name: ‘foo’,
  gender: ‘male’,
  location: {
    country: ‘Canada’,
    city: Toronto
  }
}
```

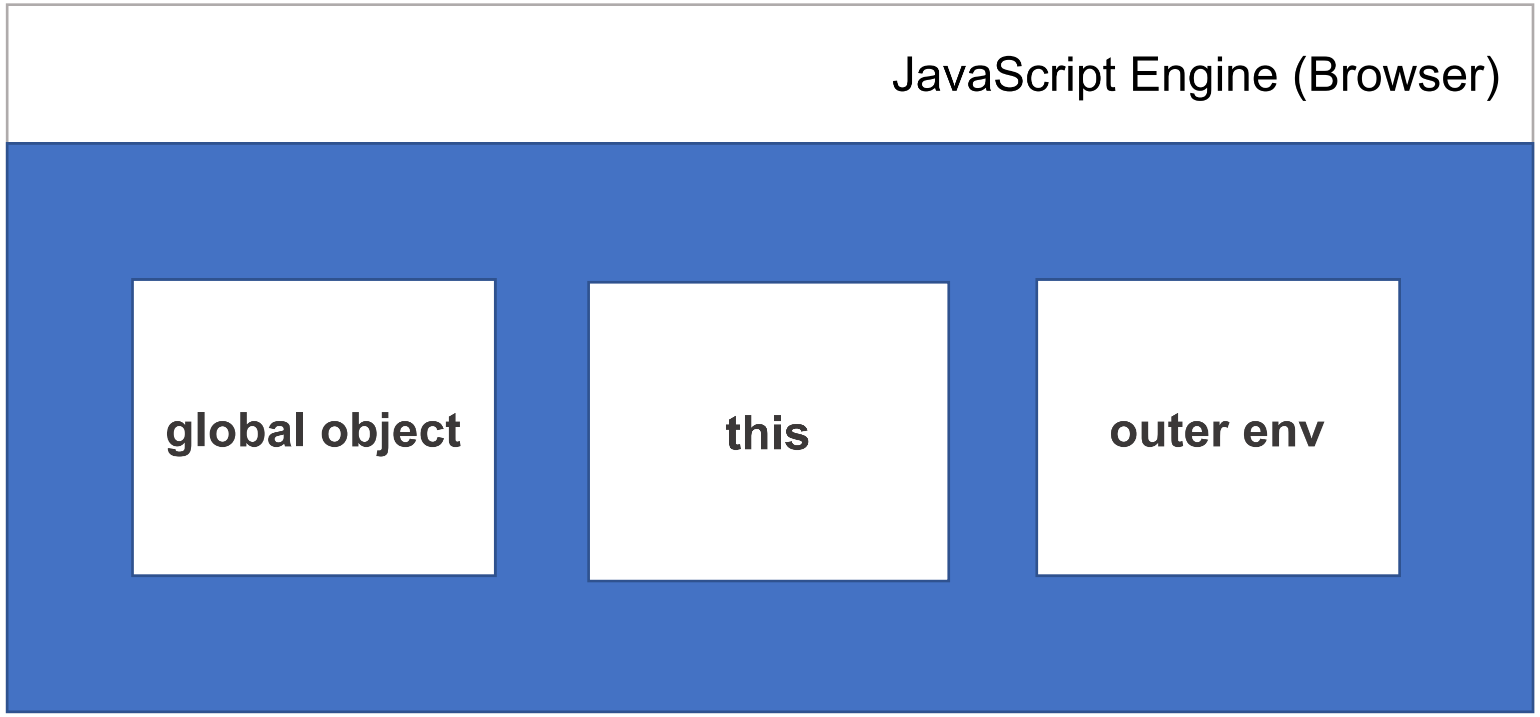
collection

What is an Object?

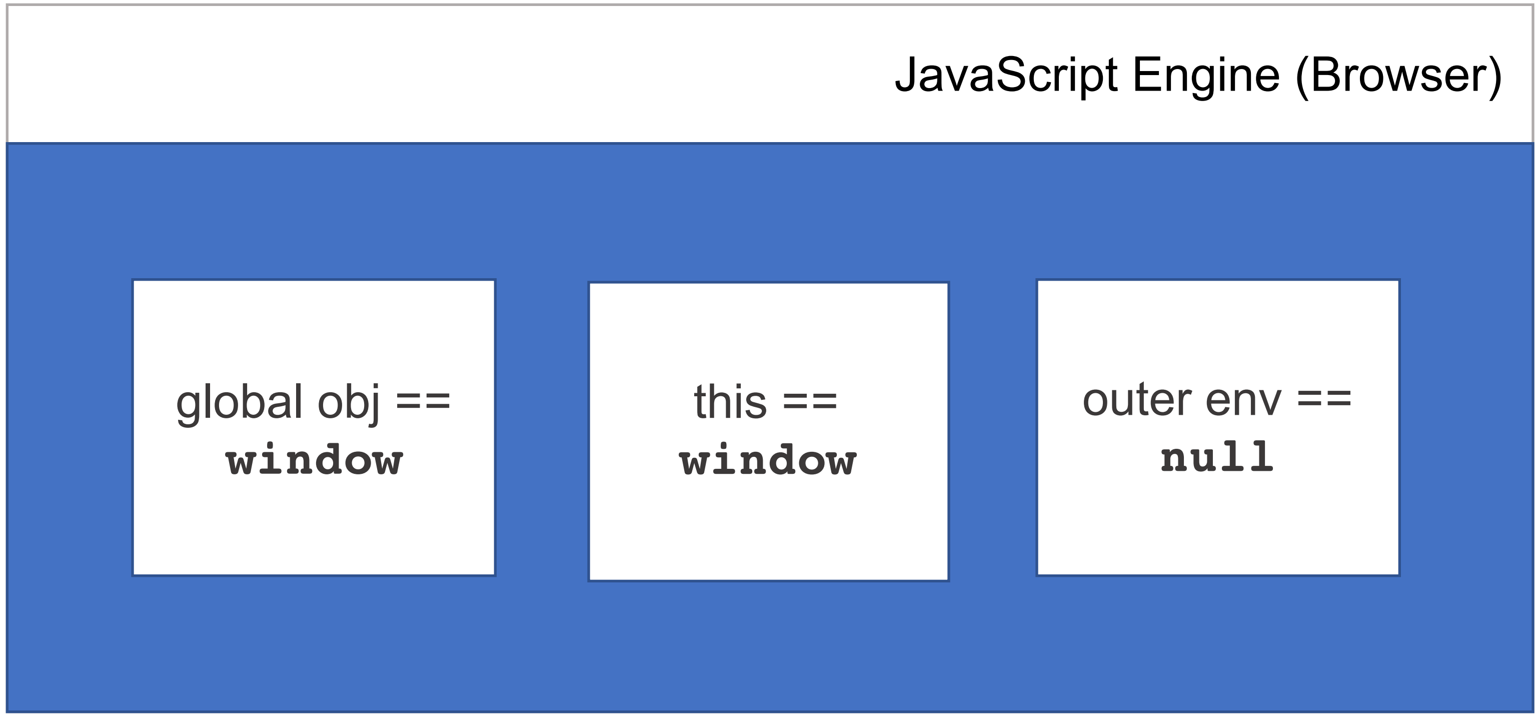
```
{
  name: ‘foo’,
  gender: ‘male’,
  logName: function() {
    console.log(this.name)
  }
}
```

method

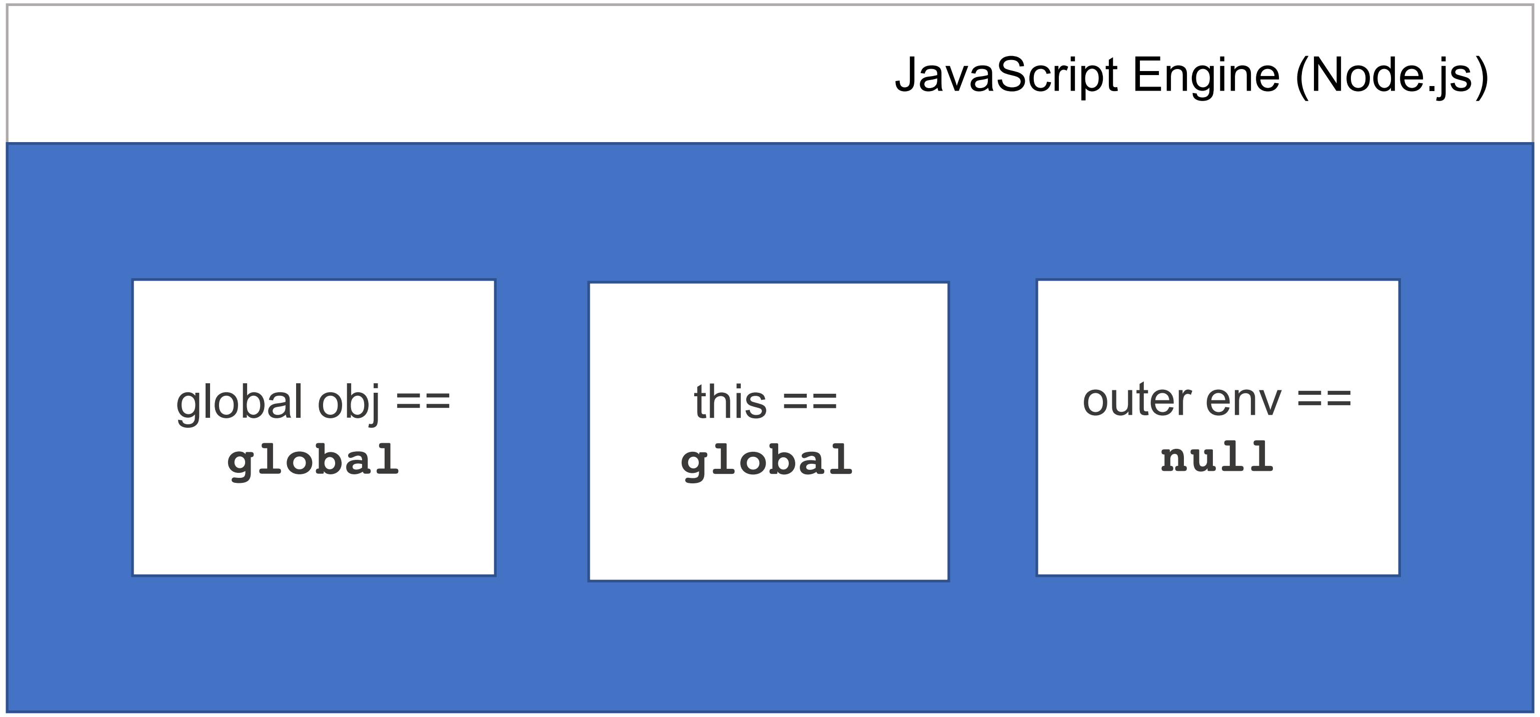
What’s provided out of the box?



What’s provided out of the box?



What’s provided out of the box?



What is the Execution Context (EC)?

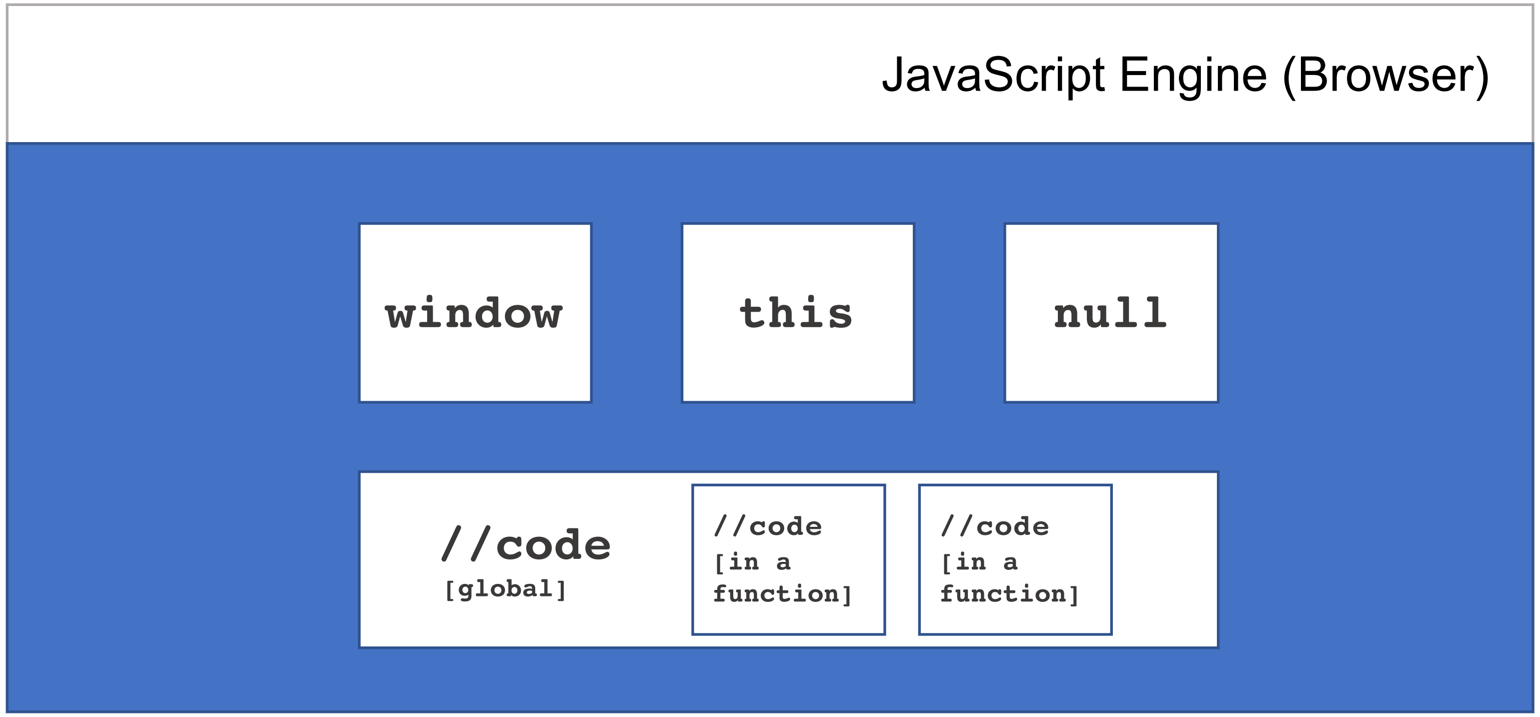
The environment in which JavaScript code is executed.

Execution Context

The environment in which JavaScript code is executed.

- This environment constitutes of variables, objects and functions available to JavaScript code being executed.
- Two important contexts:
  - Global Execution Context (default)
  - Functional Execution Context

Execution Context



What is the Execution Stack?

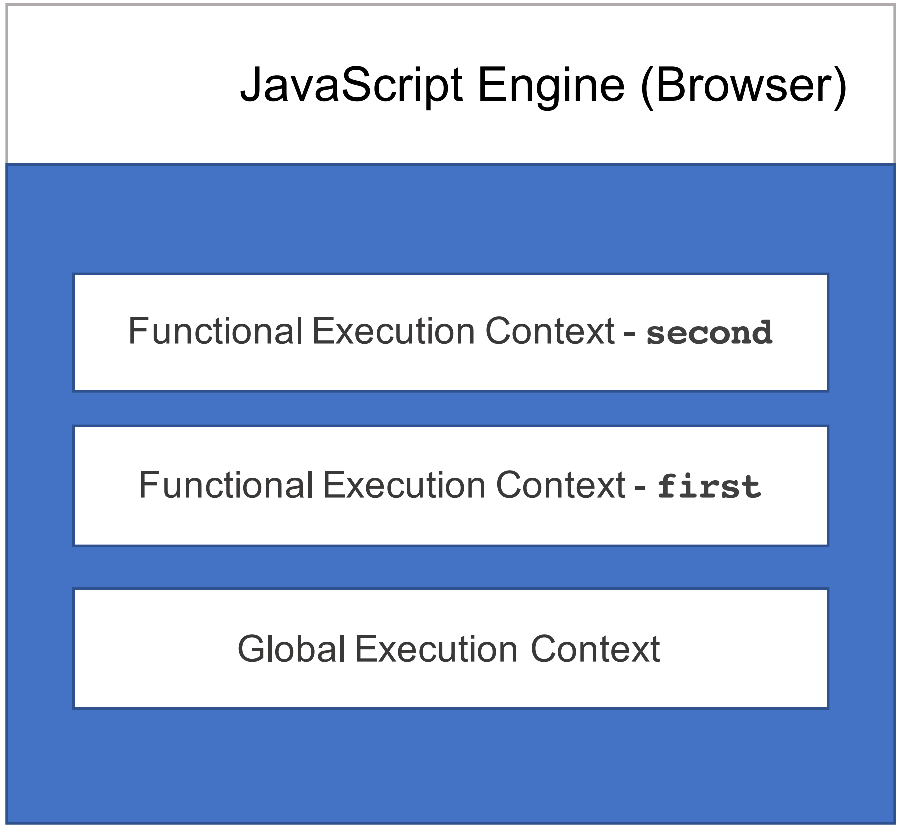
A stack with a LIFO (Last in, First out) structure, used to store all the execution context created during the code execution.

Execution Stack

```
function second() {
  // code
}

function first() {
  second();
}

first();
```

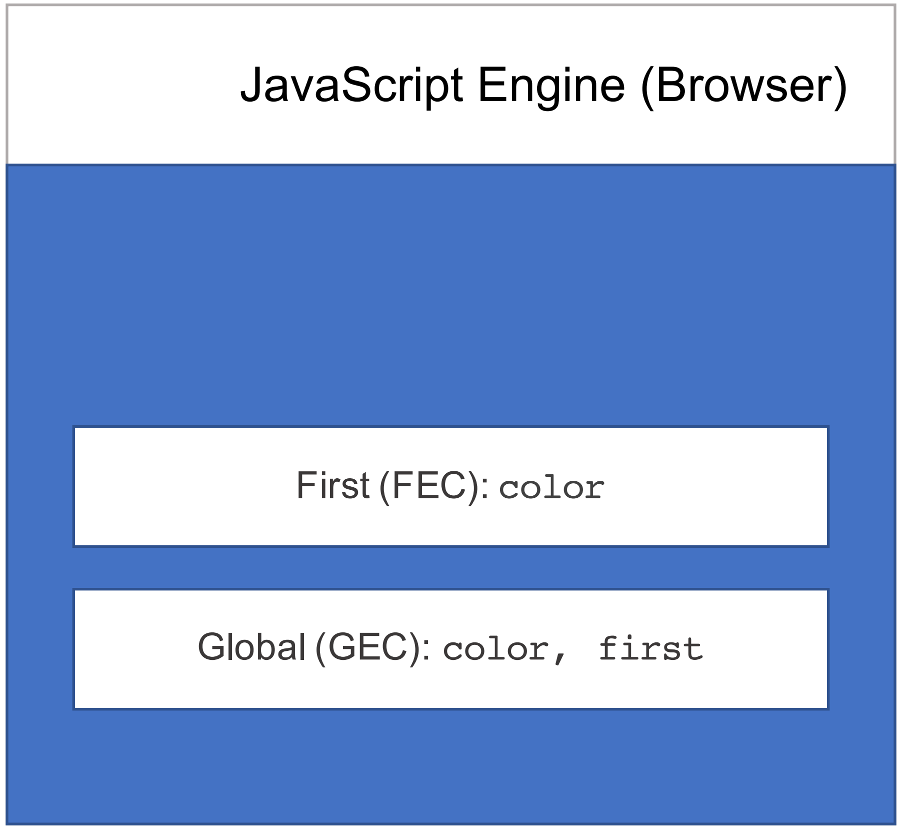

JavaScript Engine (Browser)
Functional Execution Context - second
Functional Execution Context - first
Global Execution Context

Execution Stack / Variable Environments

```
var color = 'red';

function first() {
  var color = 'green';
  console.log(color);
}

first();
console.log(color);
```

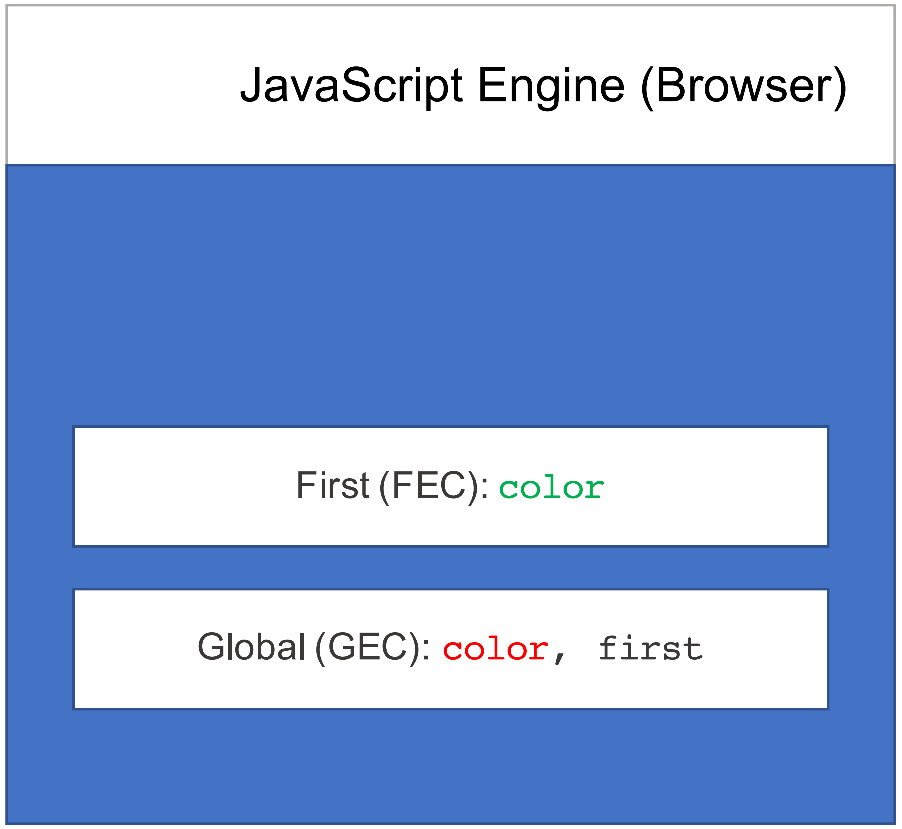


Execution Stack / Variable Environments

```
var color = ‘red’;

function first() {
  var color = ‘green’;
  console.log(color);
}

first();
console.log(color);
```

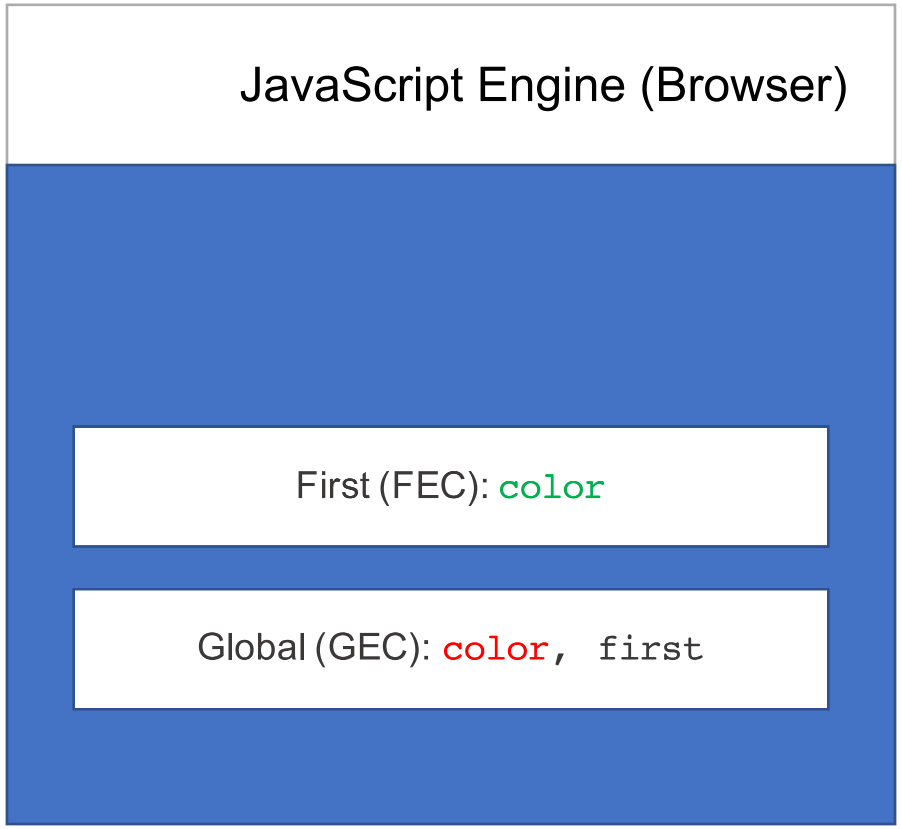


Execution Stack / Variable Environments

```
var color = ‘red’;

function first() {
  var color = ‘green’;
  console.log(color);  //green
}

first();
console.log(color);  //red
```



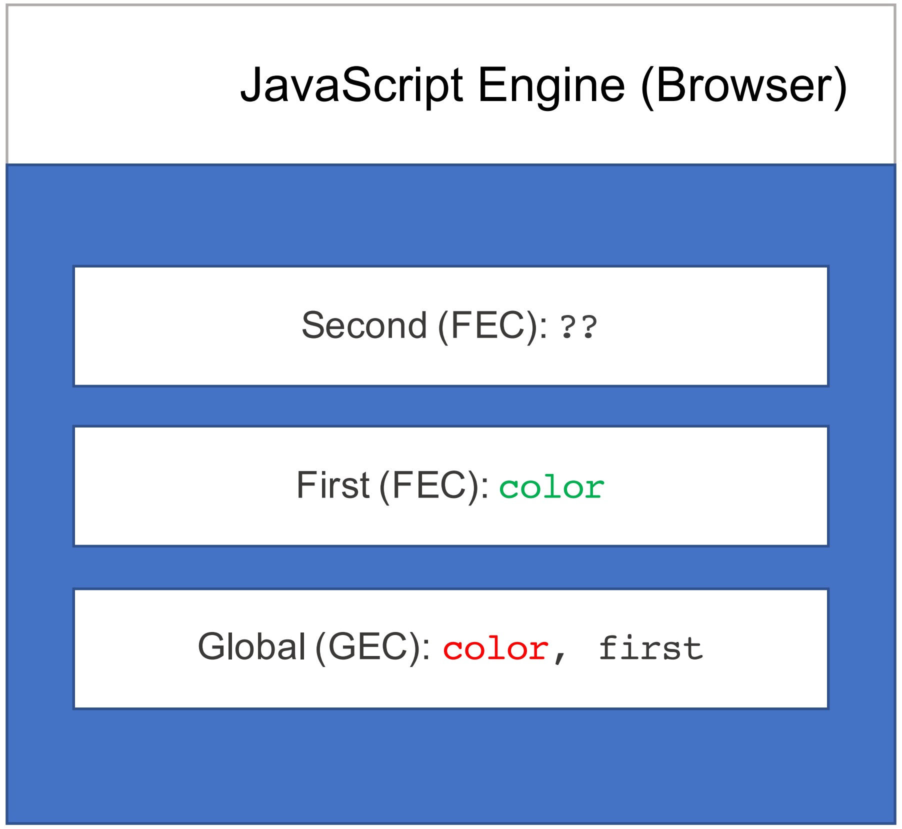
Execution Stack / Variable Environments

```
var color = 'red';

function first() {
  var color = 'green';
  console.log(color);
  second();
}

function second() {
  console.log(color);
}

first();
console.log(color);
```



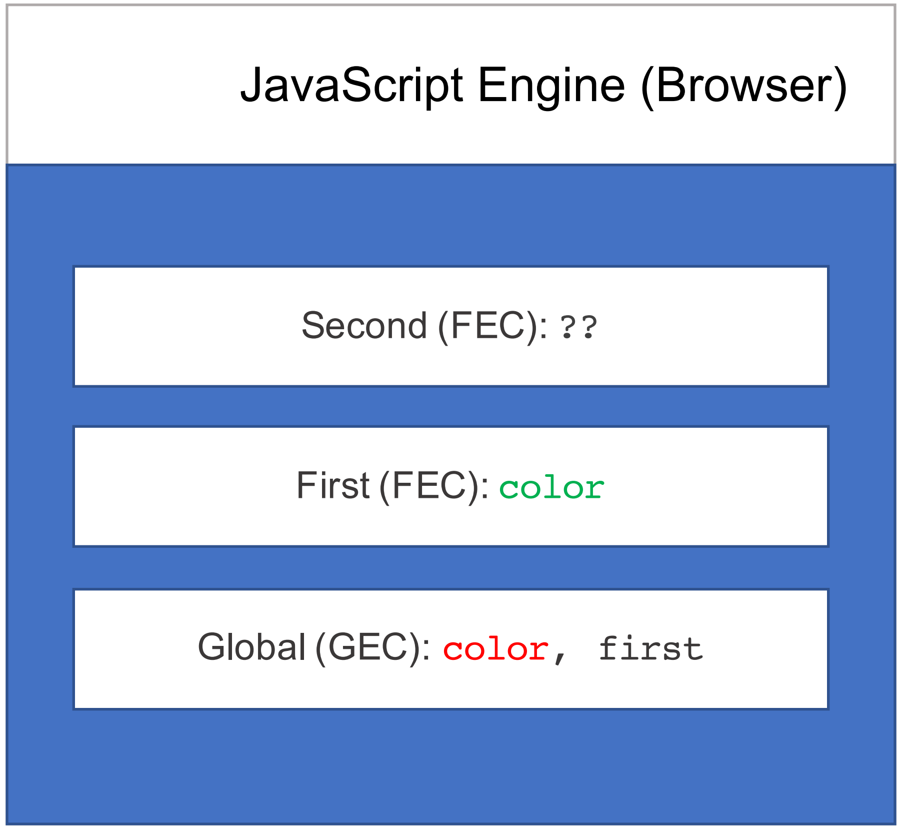
Execution Stack / Variable Environments

```
var color = ‘red’;

function first() {
  var color = ‘green’;
  console.log(color); //green
  second();
}

function second() {
  console.log(color);
}

first();
console.log(color); //red
```



Outer Environment

Environment where the code is sitting, not where it's invoked from.

Scope Chain

The scope chain is a way to link to all variables and functions that the current execution context has access to.

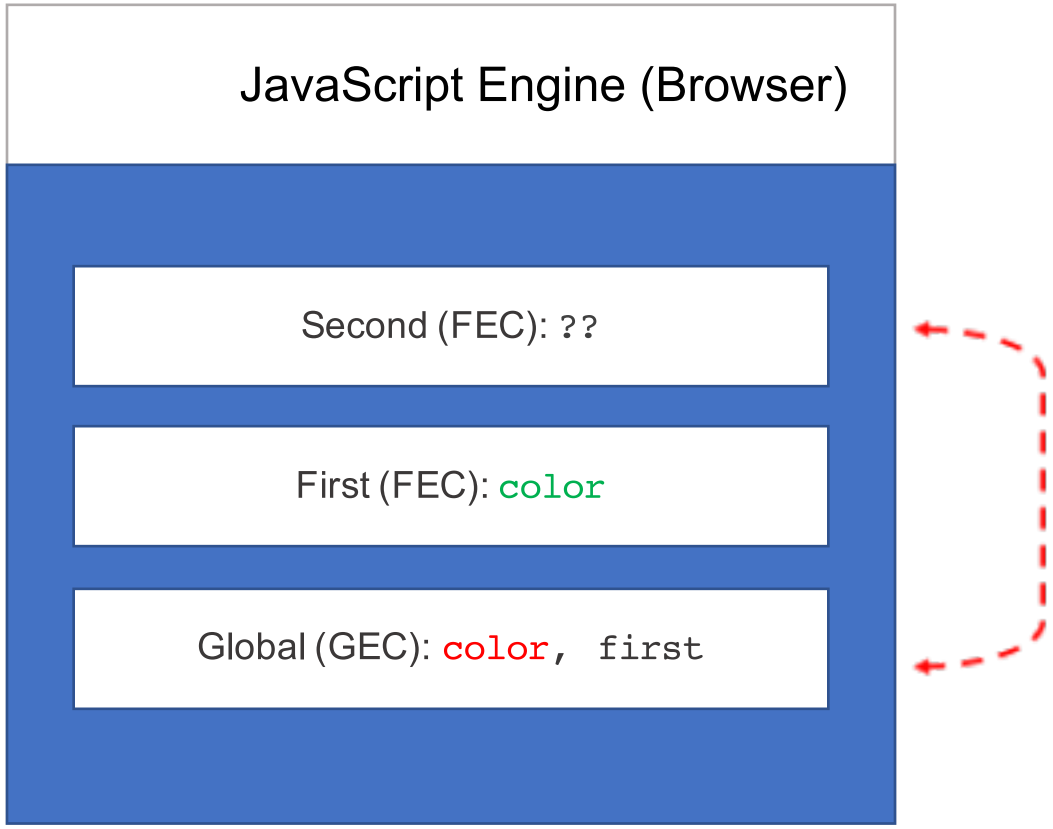
Variable Environments

```
var color = ‘red’;

function first() {
  var color = ‘green’;
  console.log(color); //green
  second();
}

function second() {
  console.log(color); //red
}

first();
console.log(color); //red
```



Hoisting vs. Execution

Hoisting is nothing but the “setup” phase, required before the code is executed.

Hoisting vs. Execution

```
var name = 'foo';

function getName() {
  console.log(name);
}

getName();
```

Hoisting Phase	Execution Phase
Allocate space for variable called **name** in memory	Assign value of **foo** to variable called **name**
Load **getName** function in memory	
// no-op	Invoke **getName** function.

Section 2

Functions and IIFEs

Learning objectives of this section:

- What are functions?
- Difference between objects and functions
- Different ways to create a function
- IIFEs and their execution context

JavaScript Functions

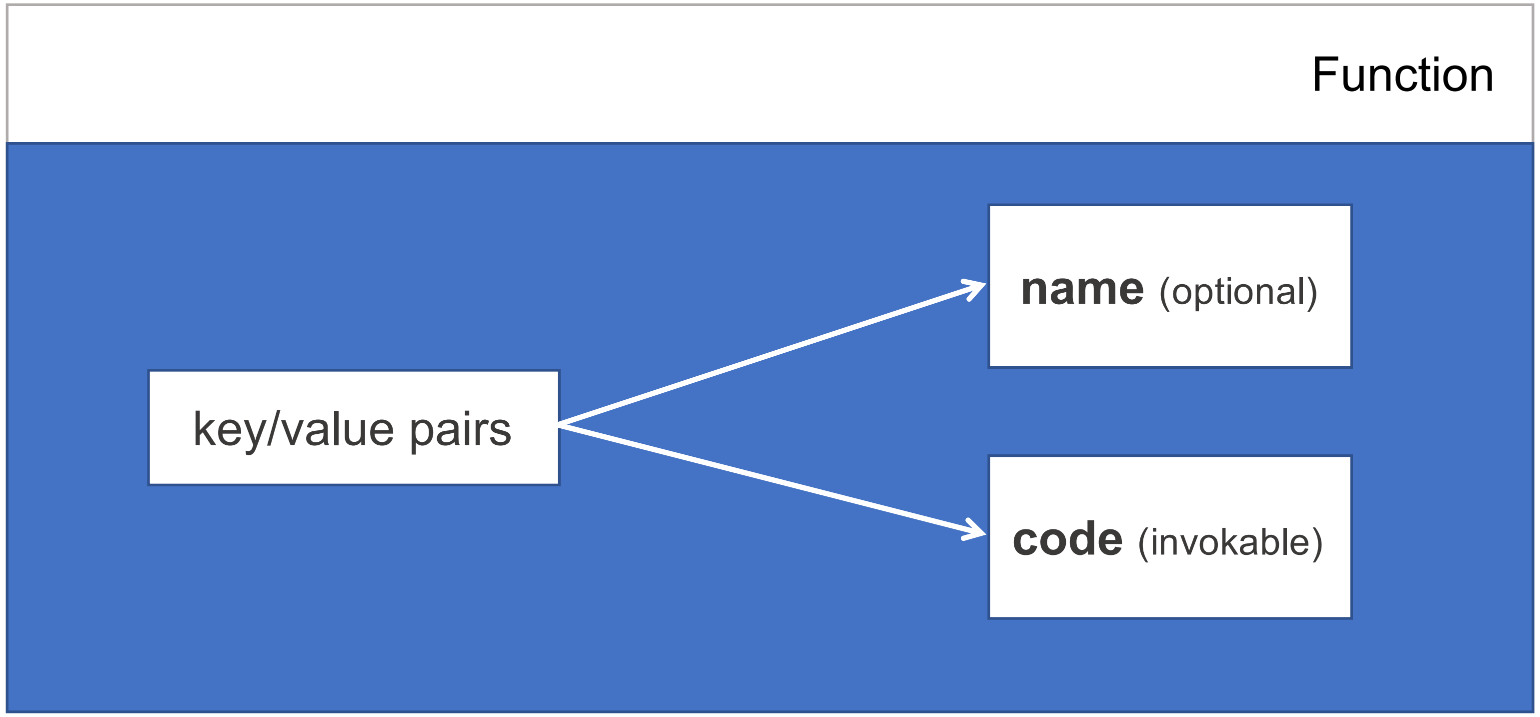
In JavaScript, functions are first-class objects.

JavaScript Functions

In JavaScript, functions are first-class objects.

- They can have properties and methods just like other objects.
- They have additional properties such as their name and code (which can be invoked)
- Can be assigned or passed around just like variables and objects

JavaScript Functions



Functions – Declaration vs. Expression

Declaration

```
function sayHi() {
  // code
}
```

Expression

```
var sayHello = function() {
  // code
}
```

IIFEs

Immediately **I**nvoked **F**unction **E**xpressions

Regular Function

```
function startGame() {
  // init()
  console.log(‘started…’);
}

startGame();
```

IIFE Example

```
(function startGame() {
  // init()
  console.log(‘started…’);
})();
```

IIFE Advantages

Immediately Invoked Function Expressions

- Prevent the global namespace from polluting in case a function is not required again
- Keep the variables required by the function to stay enclosed within the ()

Section 3

Closure + Apply, Call & Bind

Learning objectives of this section:

- How is ‘this’ decided
- Understand closure in JavaScript
- How IIFEs come to the rescue
- Using Apply, Call & Bind

JavaScript ‘this’

this

- Each function has access to a **`this`** variable.
  - It points the environment where the function is sitting
- If function is in the global execution context, then **`this`** points to the **`window`** object
- If function is within an object, then **`this`** points to that **`object`**

Closure

Anytime you create a function within an another function, you have created a closure.

Closure

Anytime you create a function within an another function, you have created a closure.

- Closure has access to the variables it needs from the enclosing function even after the enclosing function has finished executing.
- The inner function along with the variables stored in memory for a later used form the ”closure”

Apply, Call & Bind

```
var obj = {target: ‘div’}

function logTarget() {
  console.log(this.target)
}

logTarget()
```

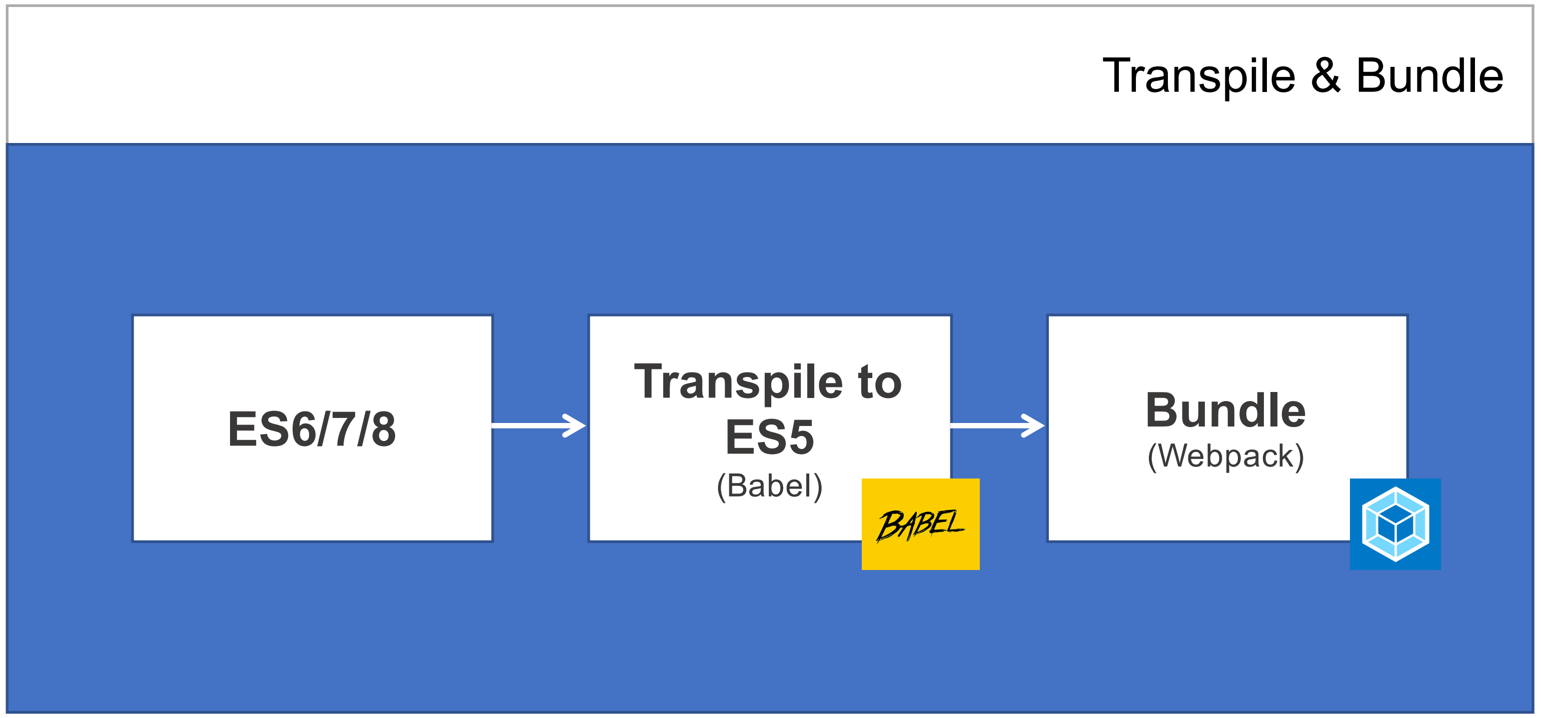
Section 4

ES6 Syntax Refresher

Learning objectives of this section:

- Const/Let
- Template Strings
- Object Literals
- Arrow Functions
- Spread Syntax
- Destructuring
- ES6 Array Methods

Setup


Transpile & Bundle
ES6/7/8
Transpile to ES5
(Babel)
BABEL
Bundle
(Webpack)

Let’s code…

4

Section 5

Classes & Inheritance

Learning objectives of this section:

- Prototype and Prototype Chain
- Prototypical Inheritance
  - ES5 Classes (old)
  - ES6 Classes (new)

What is a prototype?

5.1a

What is a prototype?

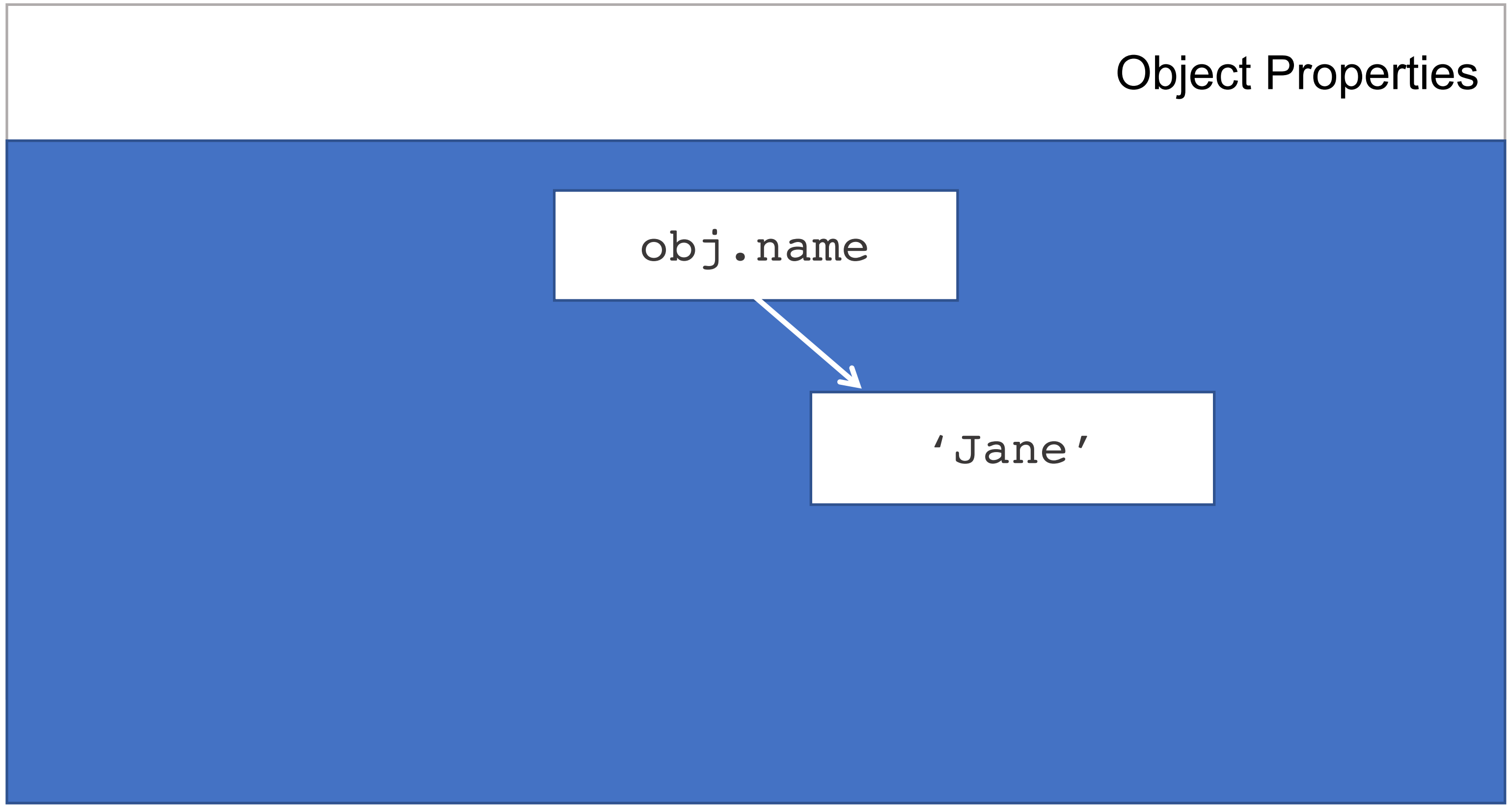
A first, typical or preliminary model of something, especially a machine, from which other forms are developed or copied.

What is a prototype in JavaScript?

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

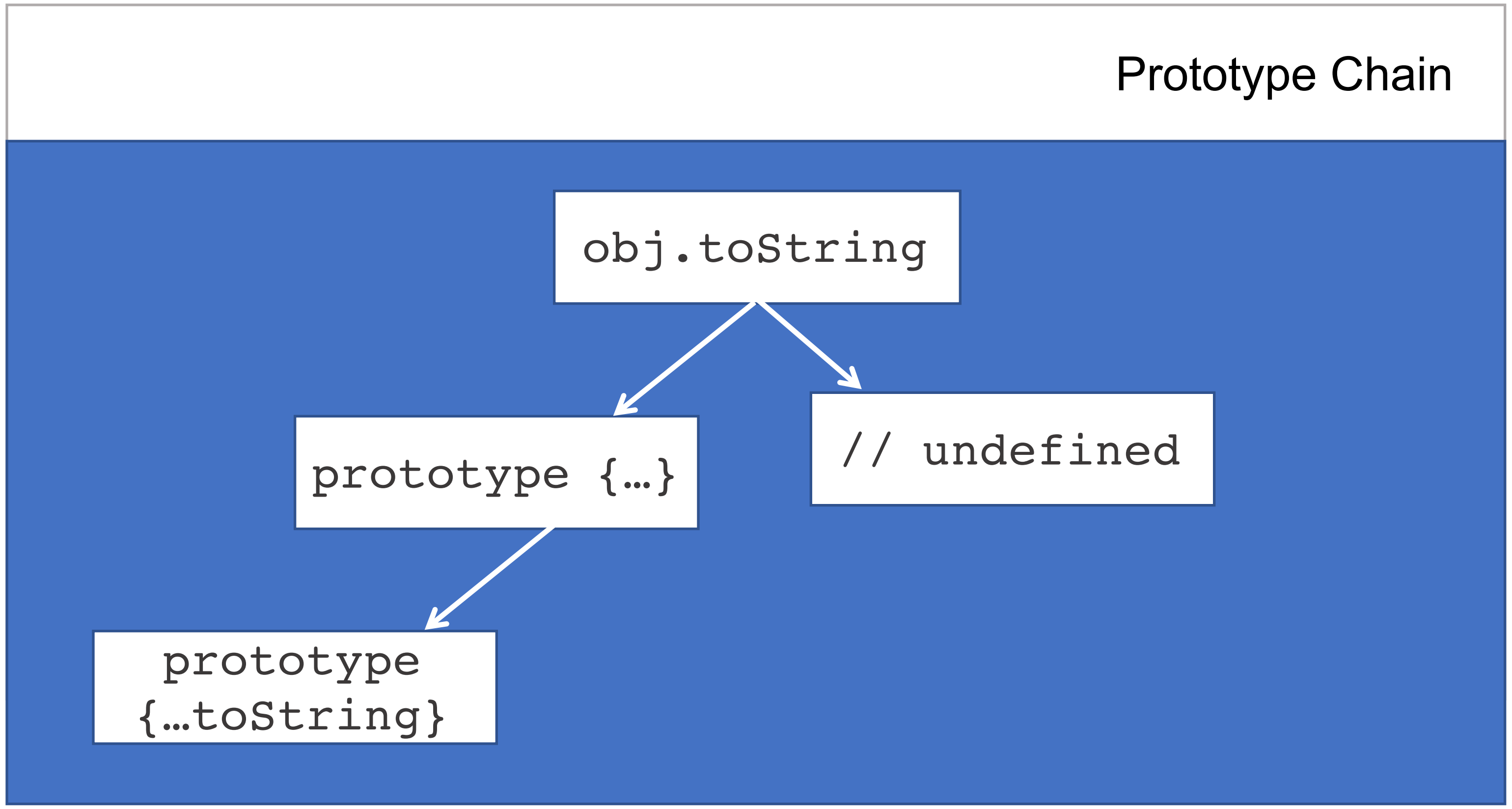
What is a prototype in JavaScript?

```
var obj = {
  name: ‘Jane’
}
```



What is a prototype in JavaScript?

```
var obj = {
  name: ‘Jane’
}
```



What is a prototype in JavaScript?

`__proto__`

What is a prototype in JavaScript?

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

Everything is an Object!

How to tell if a property is on the object or proto?

```
var obj = {
   name: ‘Jane’
}
```

```
obj.hasOwnProperty(‘name’);
// true

obj.hasOwnProperty(‘toString’);
// false
```

Why bother about prototypes?

5.1d

Classes

5.2

Section 6

1. Objects, Hoisting and Execution
2. Functions and IIFEs
3. Closure, Apply/Call/Bind
4. ES6 Syntax Refresher
5. Classes & Inheritance
6. Design Patterns
7. Async Programming

Design Patterns

Learning objectives of this section:

- Module Pattern
- Revealing Module Pattern
- Singleton

Design Patterns

A design pattern is a general repeatable solution to a commonly occurring problem in software design.

Why learn design patterns?

A design pattern is a general repeatable solution to a commonly occurring problem in software design.

- No need to re-invent the wheel
- Read faster by recognizing patterns
- Shared vocabulary
- Consistency in large code bases

Why learn design patterns?

A design pattern is a general repeatable solution to a commonly occurring problem in software design.

Let’s code…

6

Section 7

1. Objects, Hoisting and Execution
2. Functions and IIFEs
3. Closure, Apply/Call/Bind
4. ES6 Syntax Refresher
5. Classes & Inheritance
6. Design Patterns
7. Async Programming

Async Programming

Learning objectives of this section:

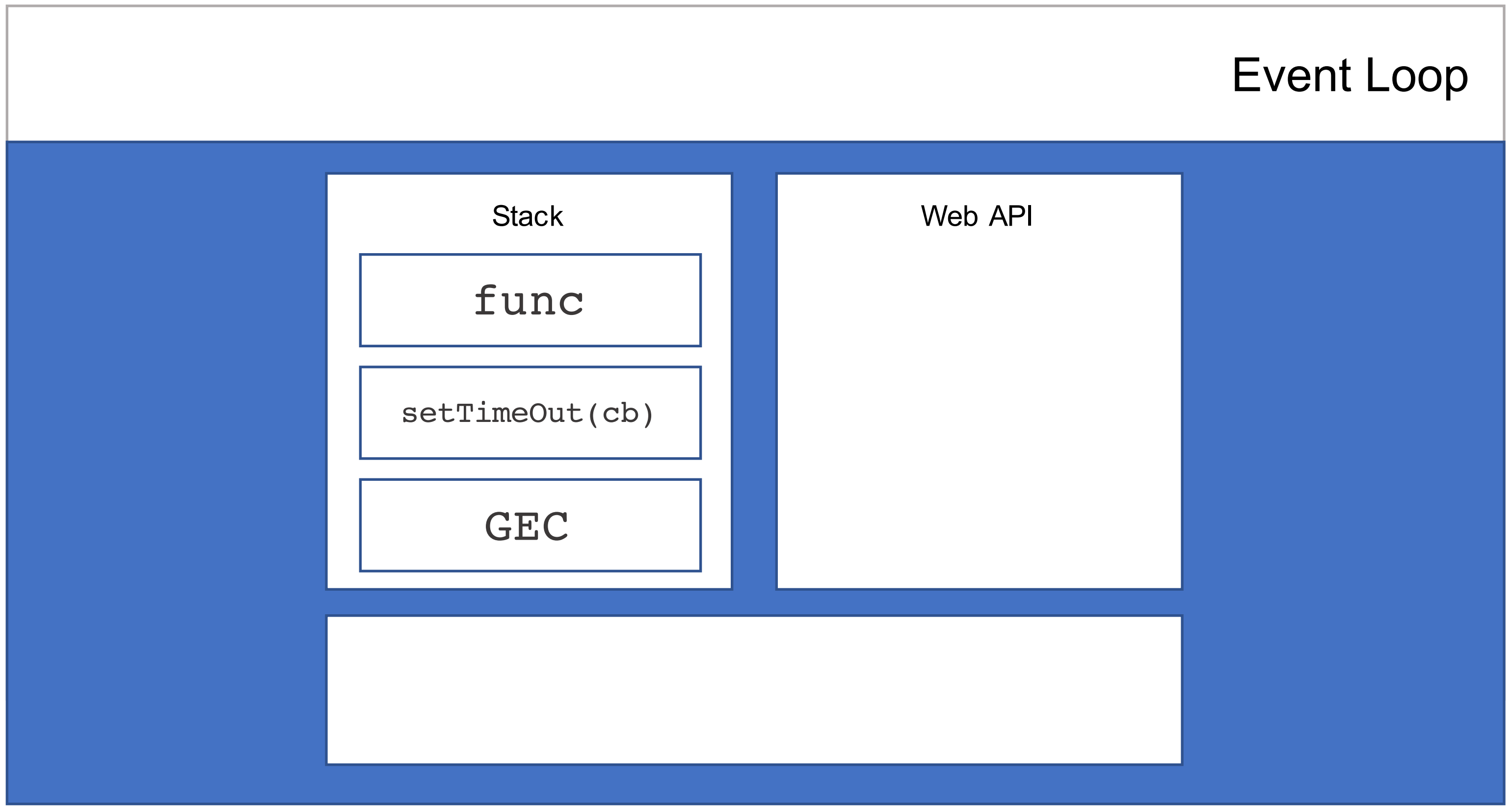
- JavaScript Event Loop
- Callbacks
- Callback Hell
- Promises
- Using Promises
- Async/Await

What is a Web API?

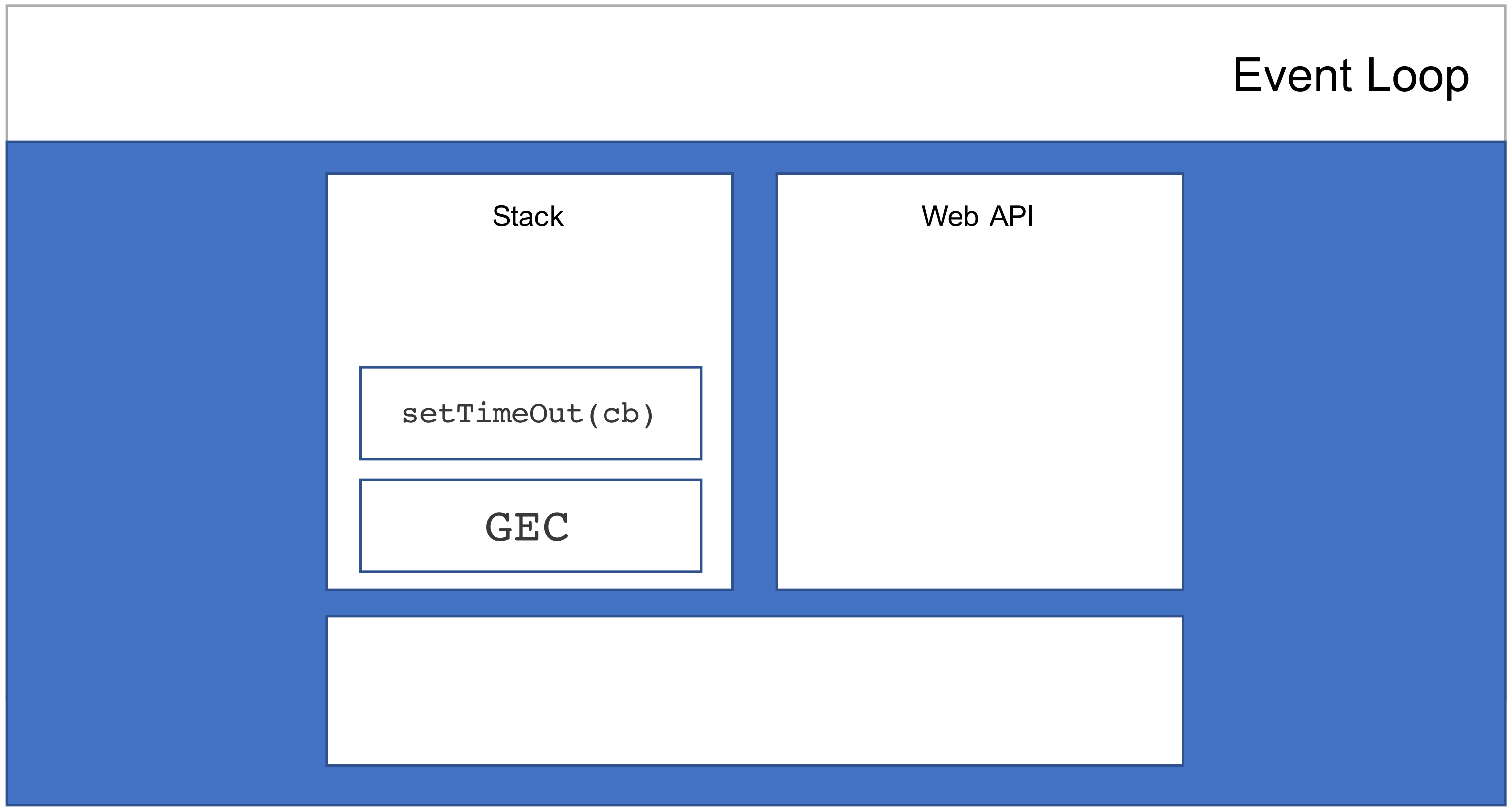
A Web API, in the context of the browser, simply is an API, provided by the browser and that we can communicate with using JavaScript in order to solve our front-end problems.

Even though these APIs are accessible with JavaScript, their implementation is in the language that the browser uses, for example, for Google Chrome it is C++.

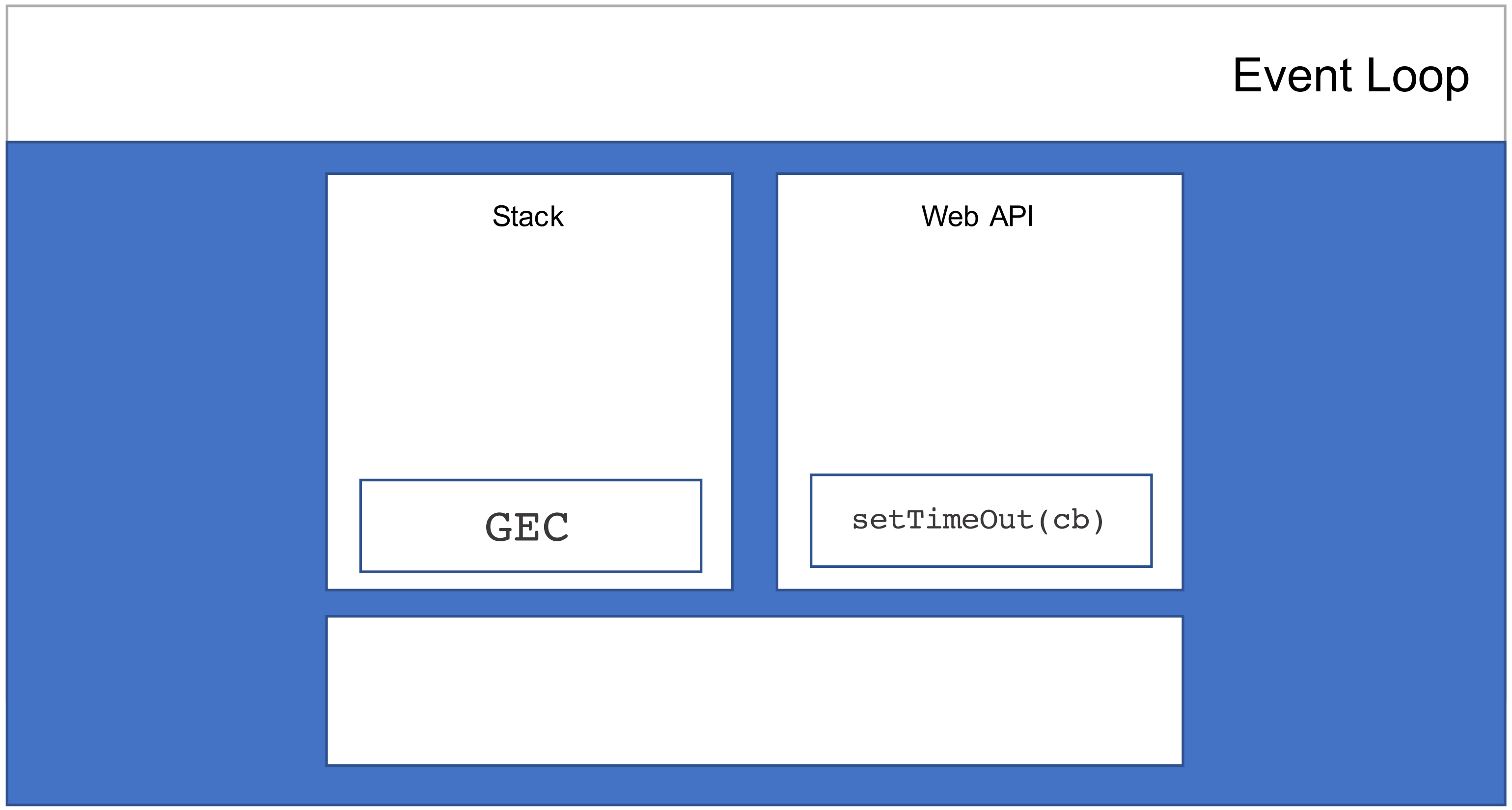
JavaScript Event Loop



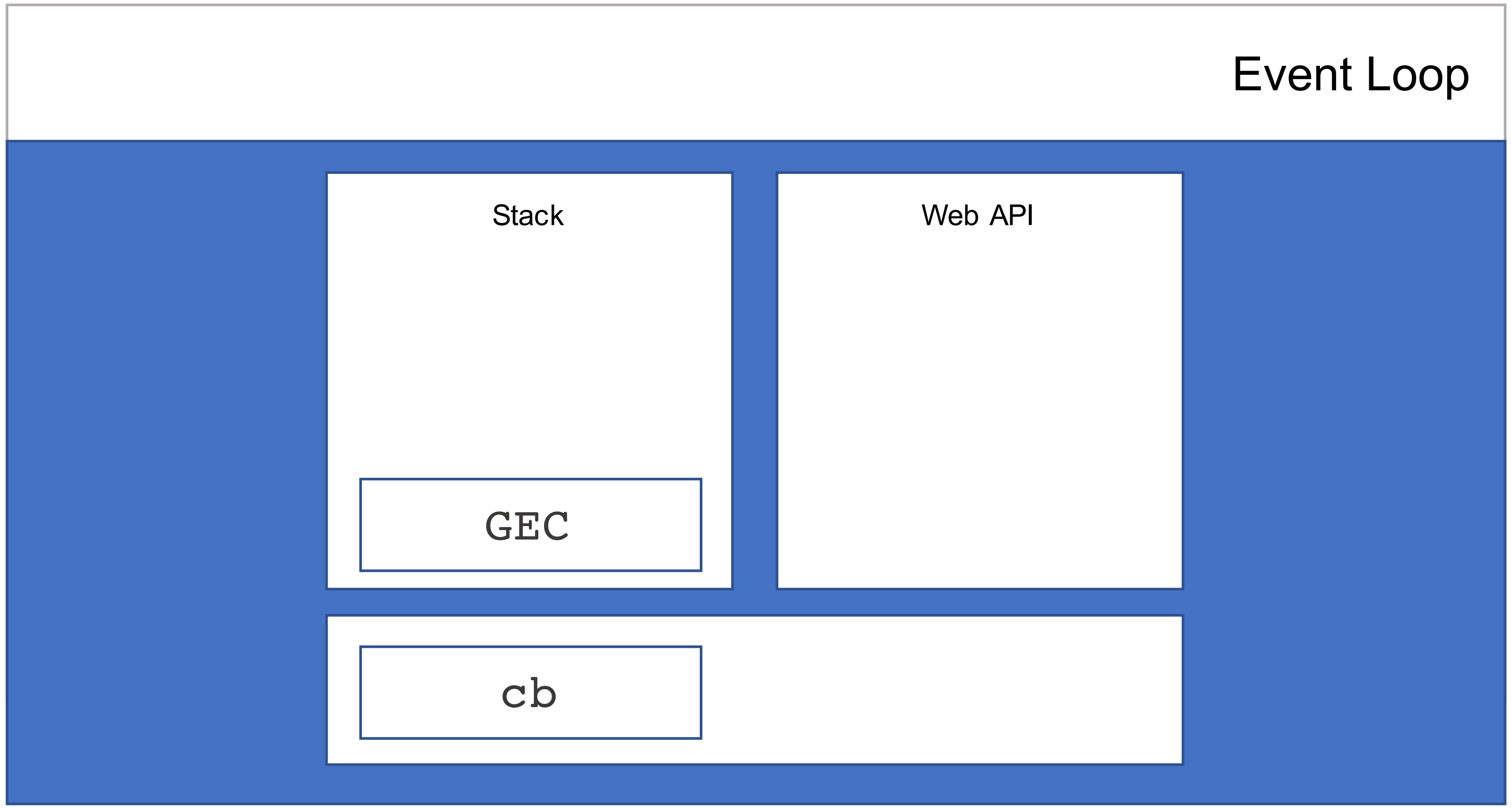
JavaScript Event Loop



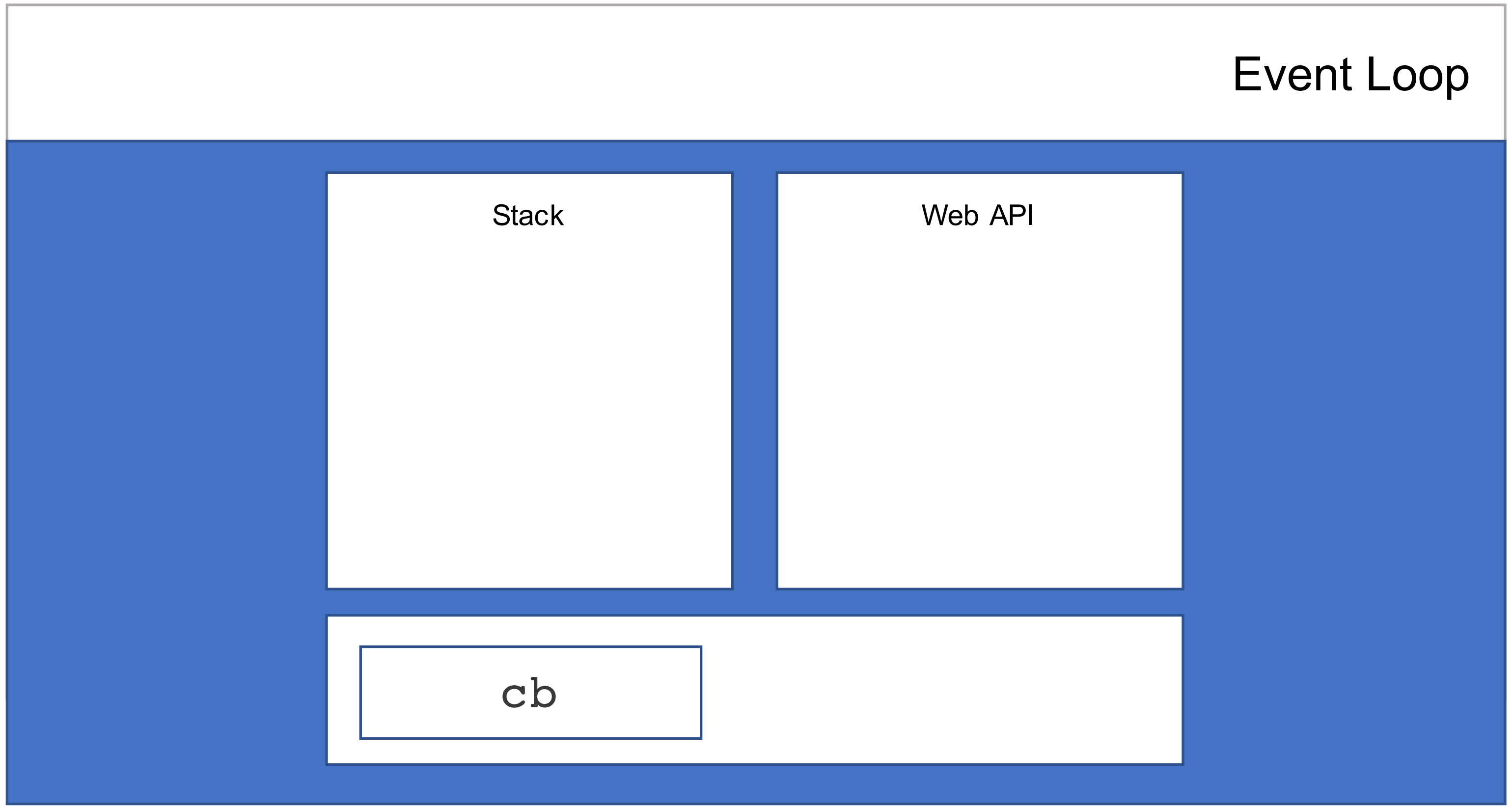
JavaScript Event Loop



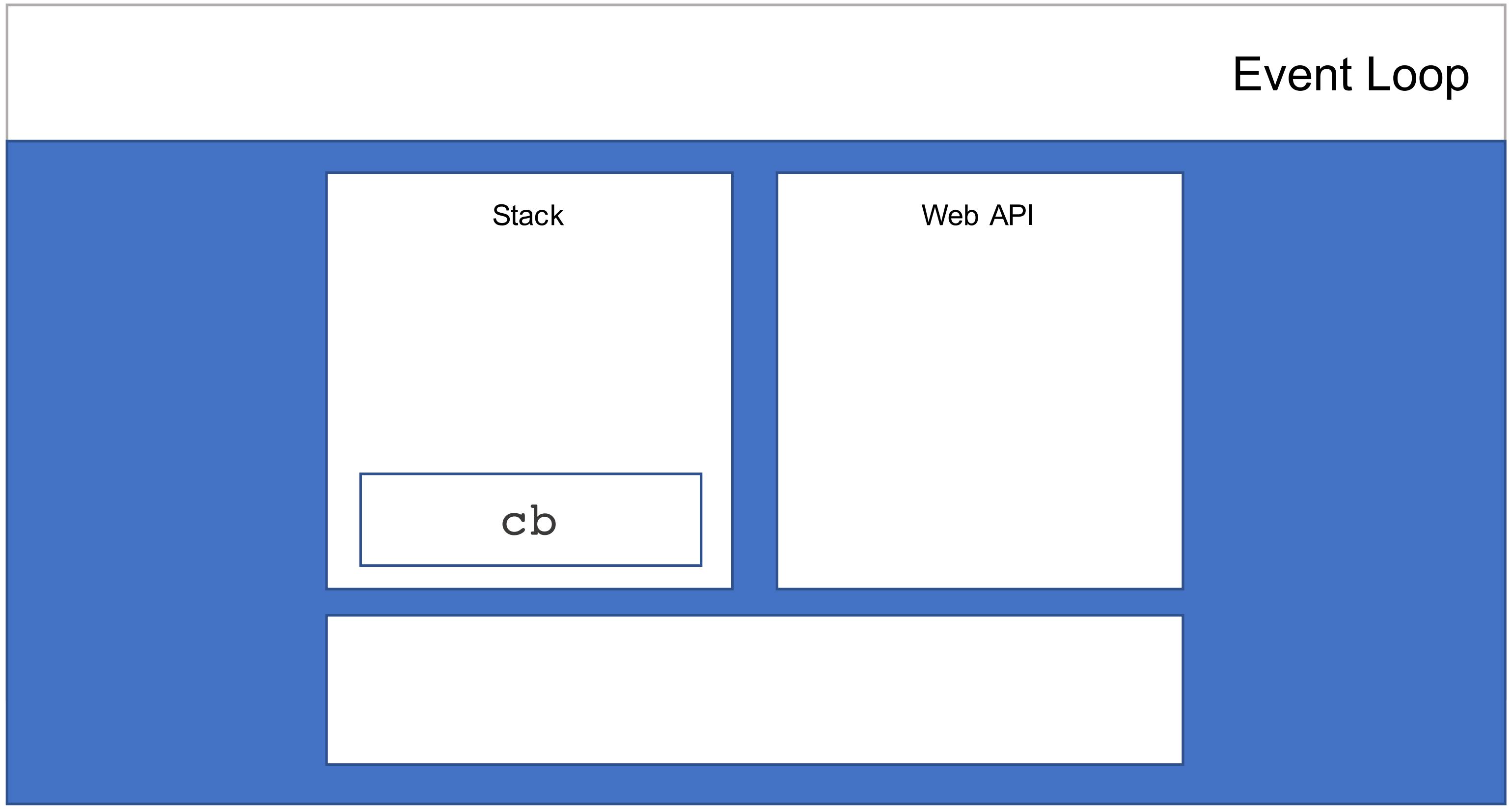
JavaScript Event Loop



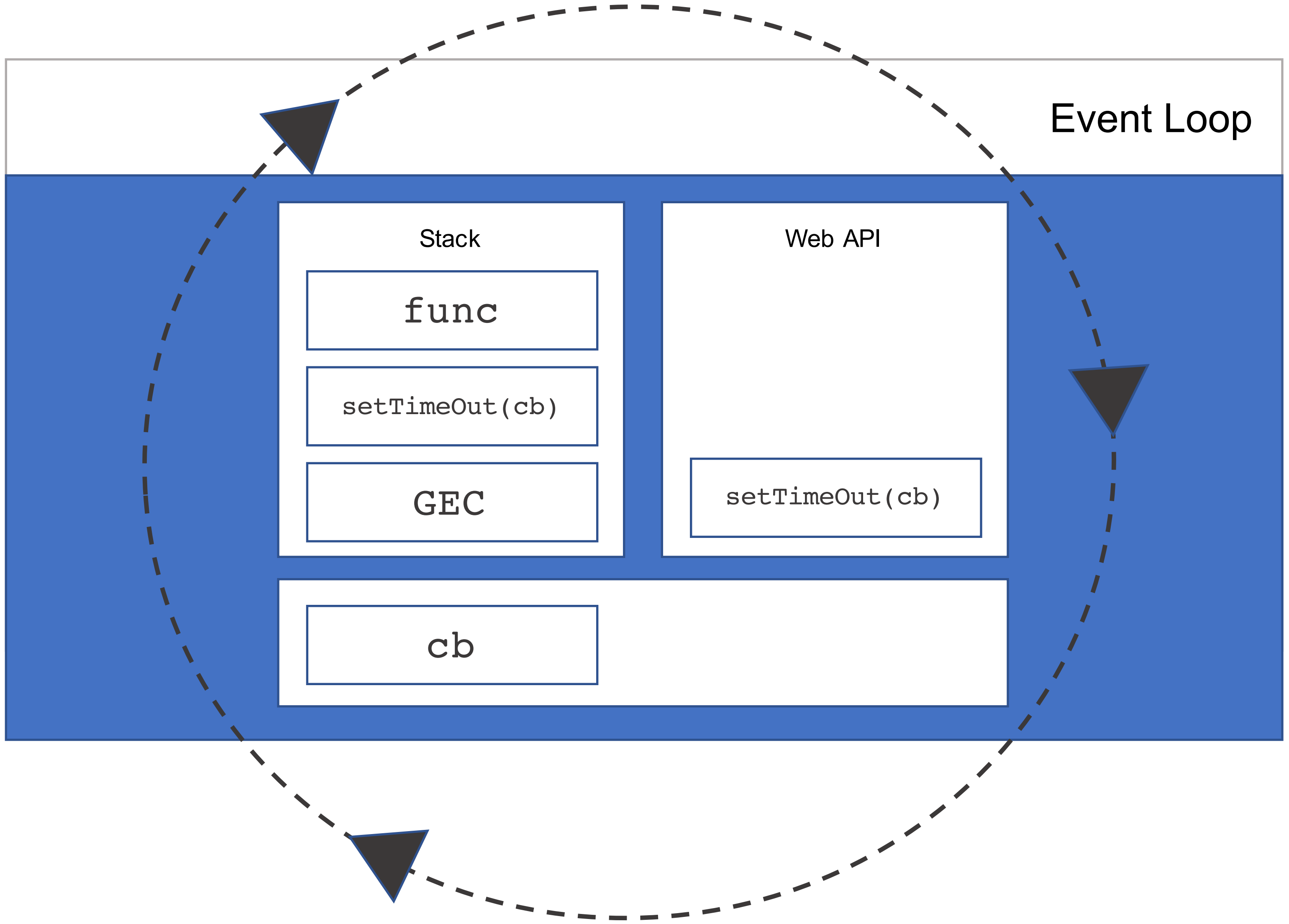
JavaScript Event Loop



JavaScript Event Loop



JavaScript Event Loop



Let’s code…

7.2

Summary

- ✓ Objects, Hoisting and Execution
- ✓ Functions and IIFEs
- ✓ Closure, Apply/Call/Bind
- ✓ ES6 Syntax Refresher
- ✓ Classes & Inheritance
- ✓ Design Patterns
- ✓ Async Programming

Contact Information

@sahilkhosla

/in/sahilkhosla/

sahilkhosla@gmail.com

Please fill out the survey!

Thank You!

Resources

- https://github.com/airbnb/javascript
- https://devdocs.io/javascript/
- https://developer.mozilla.org/en-US/docs/Web/JavaScript
- https://www.codecademy.com/catalog/language/javascript