#Clean Code Practice

REVIEW: MODERN AND CLEAN CODE

READABLE CODE

- Write code so that **others** can understand it
- Write code so that **you** can understand it in 1 year
- Avoid too "clever" and overcomplicated solutions
- Use descriptive variable names: **what they contain**
- Use descriptive function names: **what they do**

GENERAL

- Use DRY principle (refactor your code)
- Don't pollute global namespace, encapsulate instead
- Don't use `var`
- Use strong type checks (`===` and `!==`)

FUNCTIONS

- Generally, functions should do **only one thing**
- Don't use more than 3 function parameters
- Use default parameters whenever possible
- Generally, return same data type as received
- Use arrow functions when they make code more readable

OOP

- Use ES6 classes
- Encapsulate data and **don't mutate** it from outside the class
- Implement method chaining
- Do **not** use arrow functions as methods (in regular objects)

ASYNCHRONOUS CODE

- Consume promises with async/await for best readability
- Whenever possible, run promises in **parallel** (`Promise.all`)
- Handle errors and promise rejections

Two fundamentally different ways of writing code (paradigms)

IMPERATIVE

- Programmer explains **"HOW to do things"**
- We explain the computer *every single step* it has to follow to achieve a result
- **Example:** Step-by-step recipe of a cake

```
const arr = [2, 4, 6, 8];
const doubled = [];
for (let i = 0; i < arr.length; i++)
  doubled[i] = arr[i] * 2;
```

DECLARATIVE

- Programmer tells **"WHAT do do"**
- We simply *describe* the way the computer should achiev the result
- The **HOW** (step-by-step instructions) gets abstracted av
- **Example:** Description of a cake

```
const arr = [2, 4, 6, 8];
const doubled = arr.map(n => n * 2);
```

FUNCTIONAL PROGRAMMING TECHNIQUES

- Try to avoid data mutations
- Use built-in methods that don't produce side effects
- Do data transformations with methods such as `.map()`, `.filter()` and `.reduce()`
- Try to avoid side effects in functions: this is of course not always possible!

DECLARATIVE SYNTAX

- Use array and object destructuring
- Use the spread operator (`...`)
- Use the ternary (conditional) operator
- Use template literals

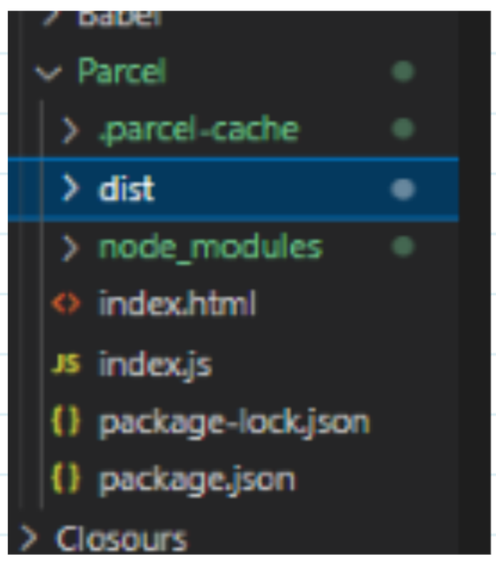
"Parcel" a built tool for JS

```
> npm i parcel --save-dev
```

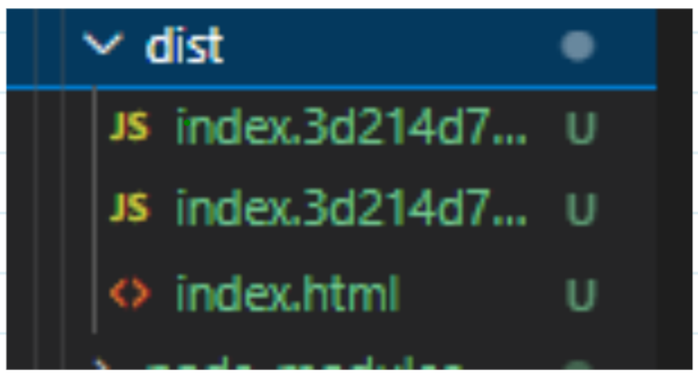
because it is a dev-dependency

it is a dev-dependency → means use for development

Npm is a tool that use to install packages. Npx is a tool that use to execute packages. Packages used by



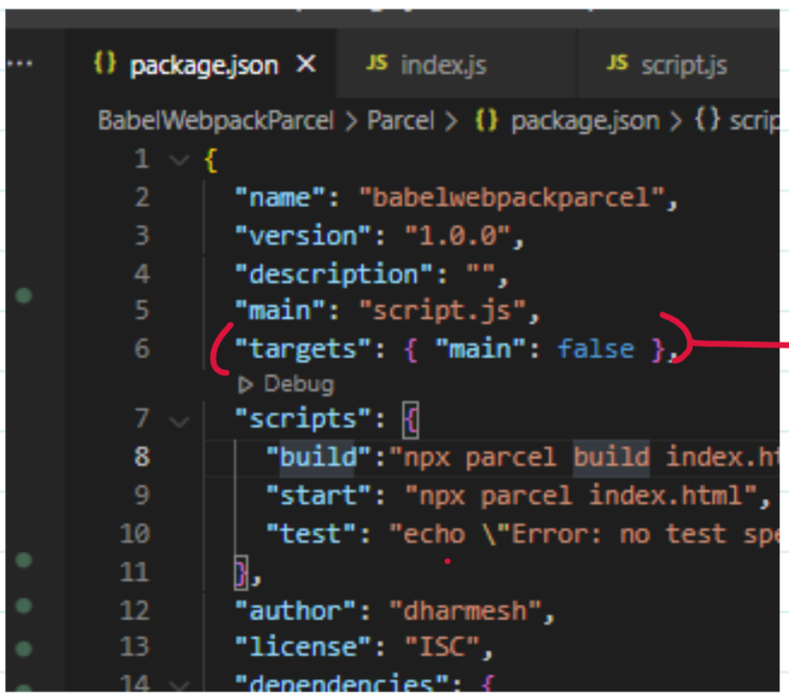
our build come here and dist folder will go in Production




```
Debug
"scripts": {
  "build":"npx parcel build index.html",
  "start": "npx parcel index.html",
```

→ for building

→ for running



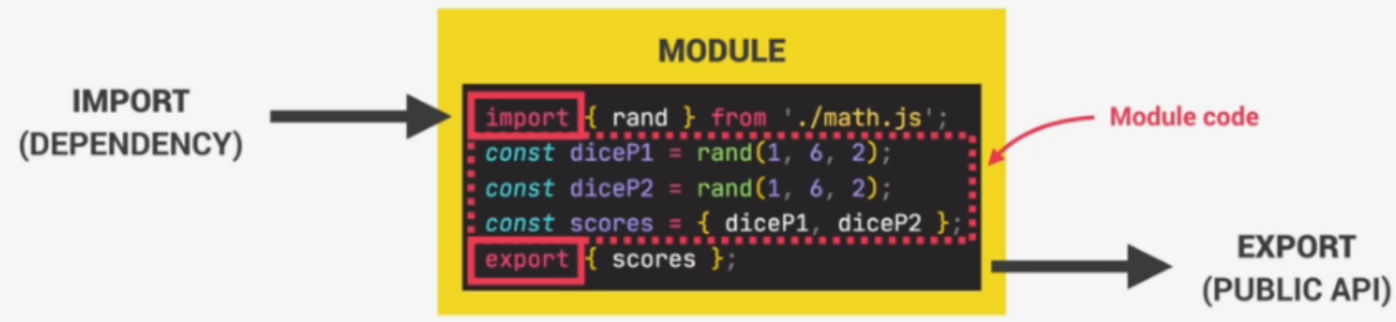
→ add this line

because babel look your file as a module and to counter this we do it

AN OVERVIEW OF MODULES

MODULE

- Reusable piece of code that **encapsulates** implementation details;
- Usually a **standalone file**, but it doesn't have to be.



NATIVE JAVASCRIPT (ES6) MODULES

ES6 MODULES

Modules stored in files, **exactly one module per file.**

```
import { rand } from './math.js';
const diceP1 = rand(1, 6, 2);
const diceP2 = rand(1, 6, 2);
const scores = { diceP1, diceP2 };
export { scores };
```

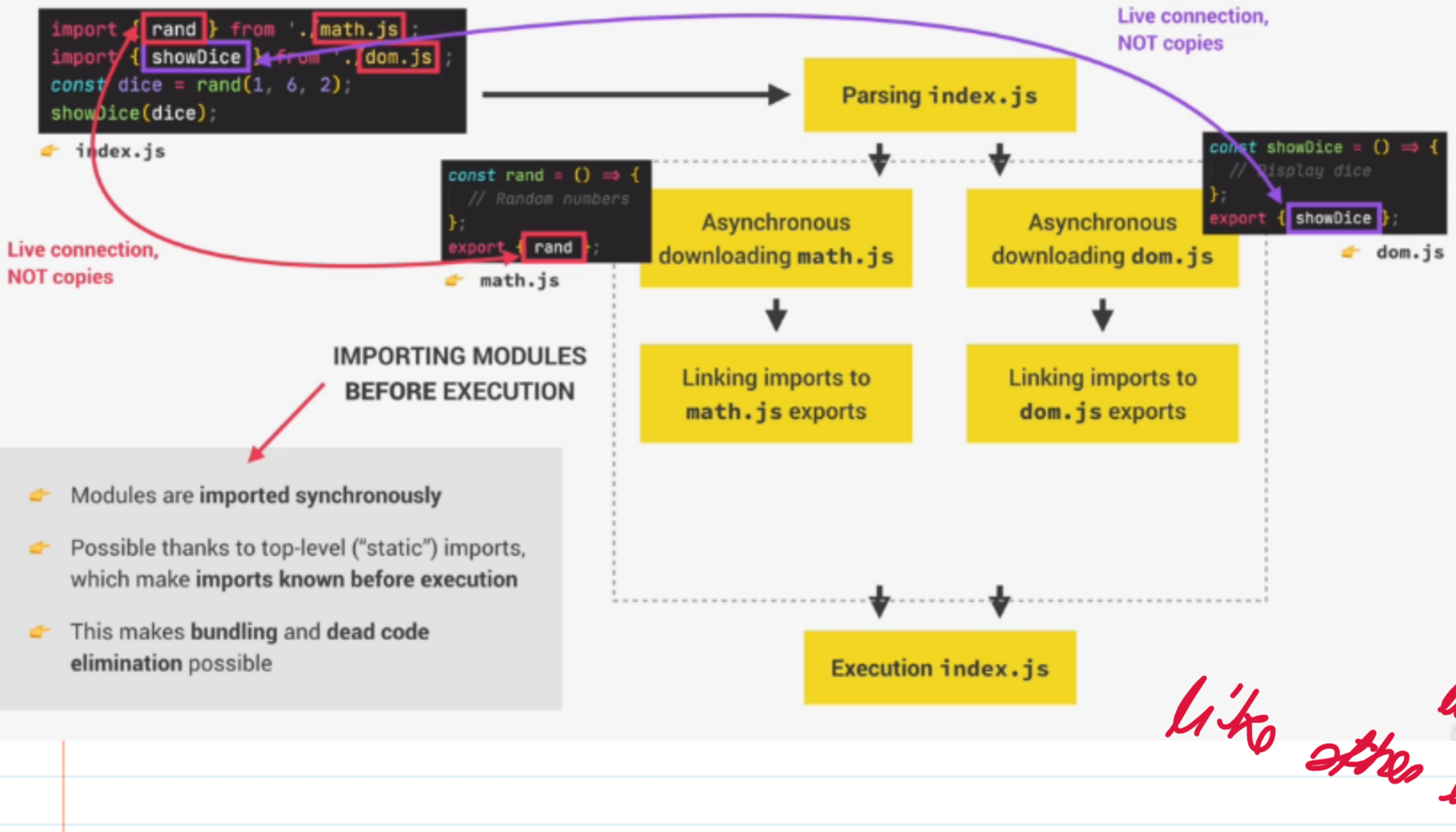
`import` and `export` syntax

	ES6 MODULE	SCRIPT
Top-level variables	Scoped to module	Global
Default mode	Strict mode	"Sloppy" mode
Top-level `this`	`undefined`	`window`
Imports and exports	✅ YES	⛔ NO
HTML linking	`<script type="module">`	`<script>`
File downloading	Asynchronous	Synchronous

☝ Need to happen at top-level

Imports are hoisted!

HOW ES6 MODULES ARE IMPORTED



- Modules are **imported synchronously**
- Possible thanks to top-level ("static") imports, which make **imports known before execution**
- This makes **bundling** and **dead code elimination** possible

In JS modules if value changes in parent module then its value change in child module also # JS modules work like this which is not like other languages

HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run

WEB APIs

fetch()

setTimeout()

DOM()

...

```
el = document.querySelector('img');
el.src = 'dog.jpg';
el.addEventListener('load', () => {
  el.classList.add('fadeIn');
});

fetch('https://someurl.com/api')
  .then(res => console.log(res));

// More code ...
```

Promises

```
res => console.log(res)
```

MICROTASKS QUEUE

Like callback queue, but for callbacks related to **promises**. Has **priority** over callback queue!

Priority

Global Execution context

CALL STACK

EVENT LOOP

Decides when each callback is executed: orchestration

CALLBACK QUEUE

🤔 How can **asynchronous** code be executed in a **non-blocking way**, if there is **only one thread** of execution in the engine?

Call Back Queue for "Promises" have Priority over callback queue.

#Modern JS

MODERN JAVASCRIPT DEVELOPMENT

MODULE

MODULE

3RD-PARTY PACKAGE

DEVELOPMENT

Join all modules into one file

BABEL

Convert modern JavaScript back to ES5

BUNDLING

TRANSPILING/ POLYFILLING

JAVASCRIPT BUNDLE

BUILD PROCESS

PRODUCTION

webpack or PARCEL

3rd party modules

NODE PACKAGE MANAGER

npm

node

Both repository and software

👉 Contains open-source packages to **include 3rd-party code in our own code** (e.g React, jQuery, Leaflet, etc.)

#CallBack Hell

Callback Hell: Callback Hell is essentially nested callbacks stacked below one another forming a pyramid structure. Every callback depends/waits for the previous callback, thereby making a pyramid structure that affects the readability and maintainability of the code.

Example: In the below example we have split the word GeeksForGeeks into three separate words and are trying to animate each word after one after another.

```
<script>
    let words = document.querySelectorAll(".word");

    const animateAll = (animate) => {
        setTimeout(() => {
            animate(words[0]);
            setTimeout(() => {
                animate(words[1]);
                setTimeout(() => {
                    animate(words[2]);
                }, 1000)
            }, 1000)
        }, 1000)
    }

    const animate = (word) => {
        word.classList.add("animate");
    }

    animateAll(animate);
```

#Promises

WHAT ARE PROMISES?

PROMISE

- **Promise:** An object that is used as a placeholder for the future result of an asynchronous operation.

  ↓ Less formal

- **Promise:** A container for an asynchronously delivered value.

  ↓ Less formal

- **Promise:** A container for a future value. ← Example: Response from AJAX call

- We no longer need to rely on events and callbacks passed into asynchronous functions to handle asynchronous results;

- Instead of nesting callbacks, we can **chain promises** for a sequence of asynchronous operations: **escaping callback hell** 🎉

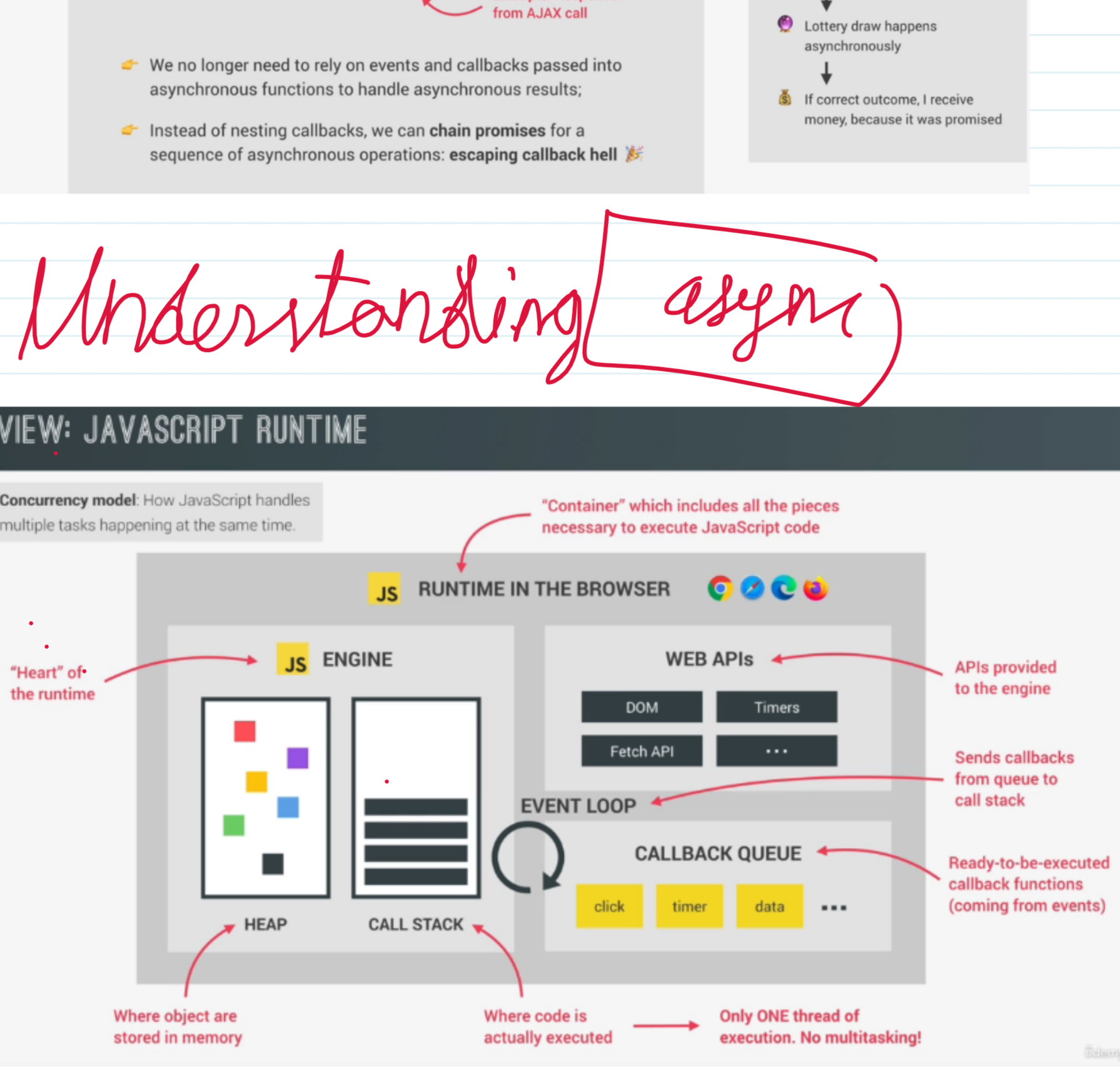
Promise that I will receive money if I guess correct outcome

- I buy lottery ticket (promise) right now
- ↓
- Lottery draw happens asynchronously
- ↓
- If correct outcome, I receive money, because it was promised

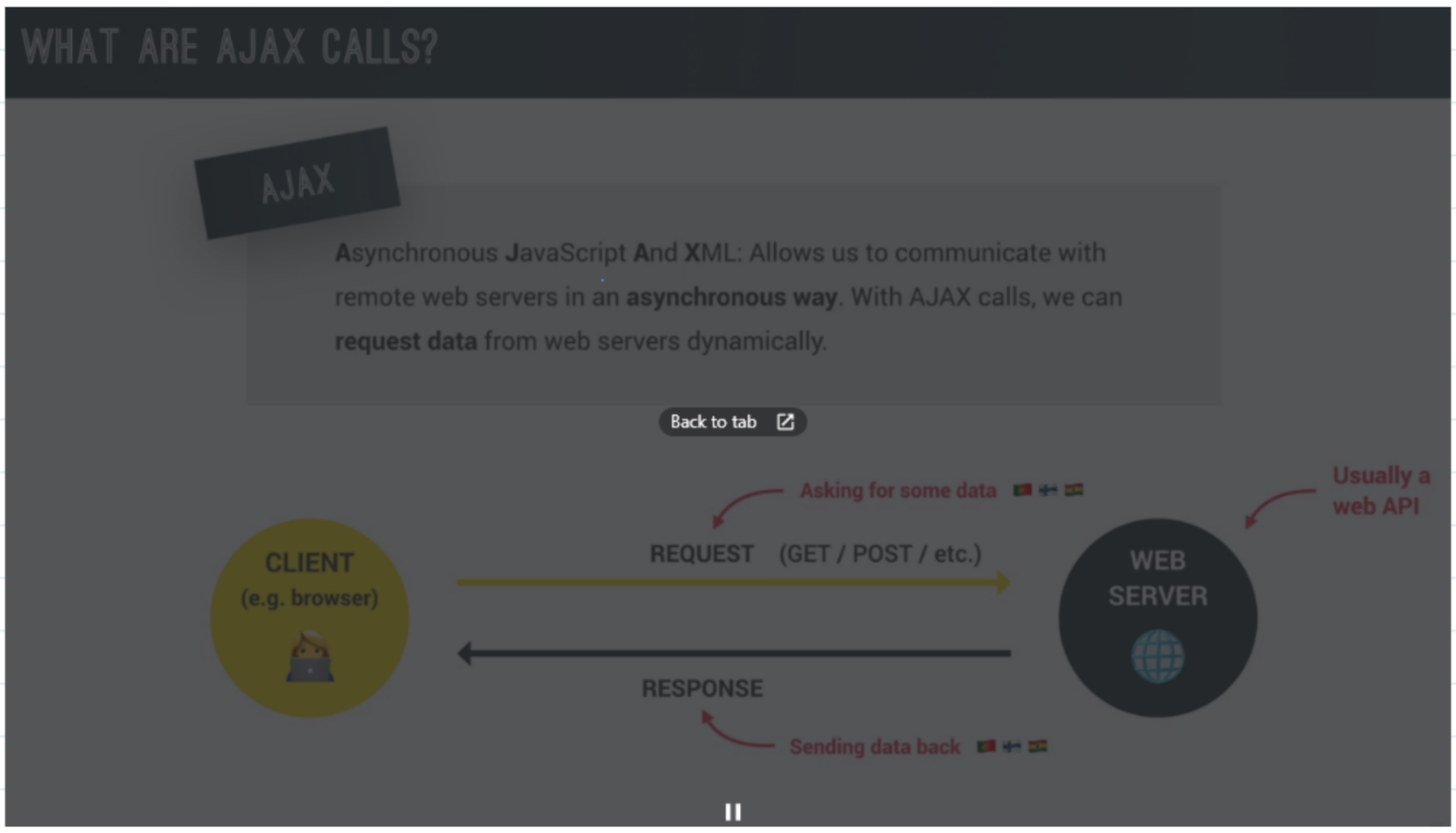
Understanding async

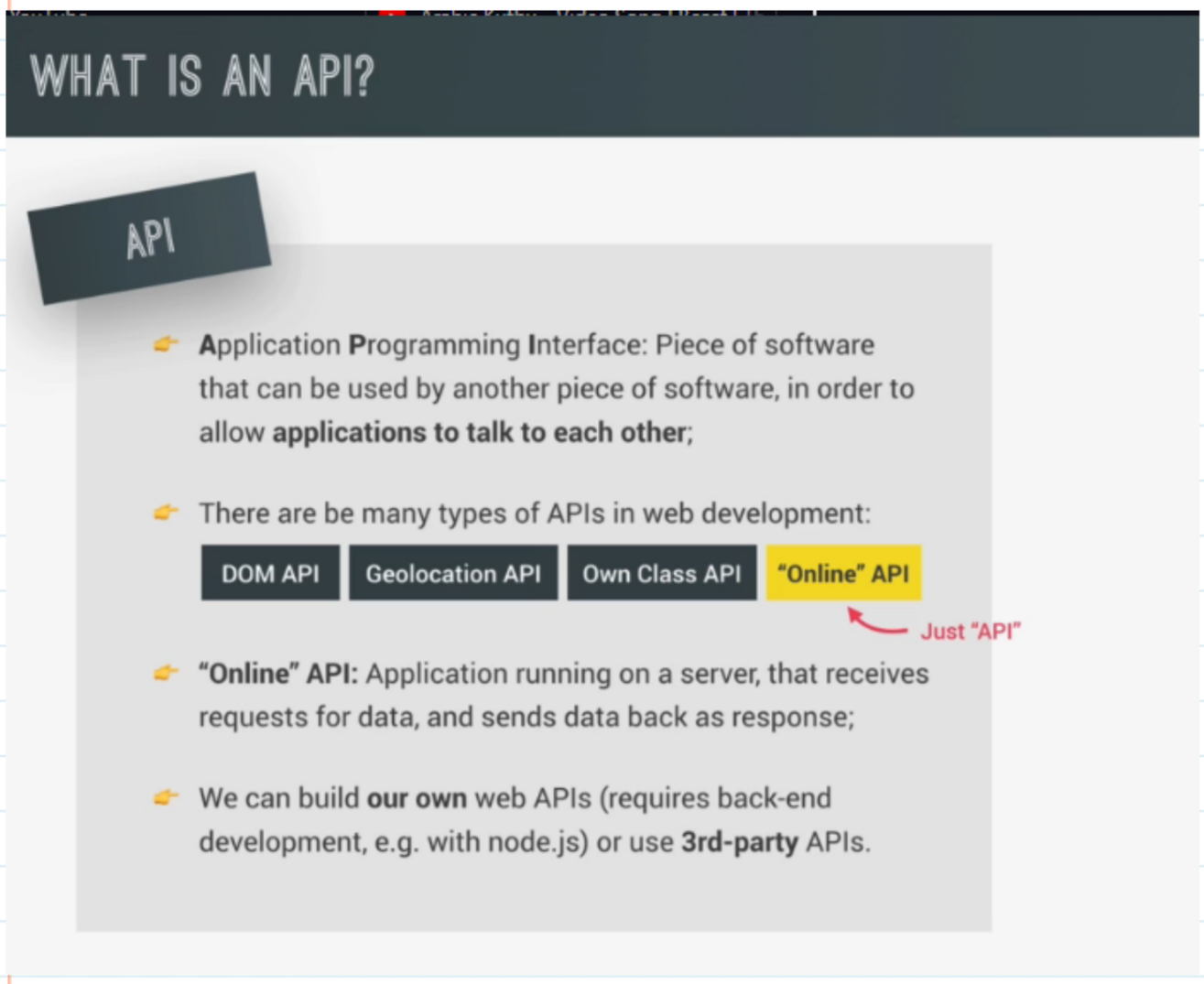
REVIEW: JAVASCRIPT RUNTIME

- **Concurrency model**: How JavaScript handles multiple tasks happening at the same time.



AJAX


WHAT ARE AJAX CALLS?
AJAX
Asynchronous JavaScript And XML: Allows us to communicate with remote web servers in an asynchronous way. With AJAX calls, we can request data from web servers dynamically.
Back to tab
Asking for some data
Usually a web API
CLIENT
(e.g. browser)
REQUEST (GET / POST / etc.)
WEB SERVER
RESPONSE
Sending data back


WHAT IS AN API?
API
Application Programming Interface: Piece of software that can be used by another piece of software, in order to allow applications to talk to each other;
There are be many types of APIs in web development:
DOM API
Geolocation API
Own Class API
"Online" API
Just "API"
"Online" API: Application running on a server, that receives requests for data, and sends data back as response;
We can build our own web APIs (requires back-end development, e.g. with node.js) or use 3rd-party APIs.

Public field (similar to property, available on created object)

Private fields (not accessible **outside** of class)

Static public field (available **only** on class)

Call to parent (super) class (necessary with `extend`). Needs to happen before accessing `this`

Instance property (available on created object)

Redefining private field

Public method

Referencing private field and method

Private method (⚠️ Might not yet work in your browser. “Fake” alternative: _ instead of #)

Getter method

Setter method (use _ to set property with same name as method, and also add getter)

Static method (available **only** on class. Can **not** access instance properties nor methods, only static ones)

Creating new object with new operator

```
class Student extends Person {
  university = 'University of Lisbon';
  #studyHours = 0;
  #course;
  static numSubjects = 10;

  constructor(fullName, birthYear, startYear, course) {
    super(fullName, birthYear);

    this.startYear = startYear;

    this.#course = course;
  }

  introduce() {
    console.log(`I study ${this.#course} at ${this.university}`);
  }

  study(h) {
    this.#makeCoffe();
    this.#studyHours += h;
  }

  #makeCoffe() {
    return 'Here is a coffe for you 😁';
  }

  get testScore() {
    return this._testScore;
  }

  set testScore(score) {
    this._testScore = score <= 20 ? score : 0;
  }

  static printCurriculum() {
    console.log(`There are ${this.numSubjects} subjects`);
  }
}

const student = new Student('Jonas', 2020, 2037, 'Medicine');
```

Parent class

Inheritance between classes, automatically sets prototype

Child class

Constructor method, called by new operator. Mandatory in regular class, might be omitted in a child class

- Classes are just “**syntactic sugar**” over constructor functions
- Classes are **not** hoisted
- Classes are **first-class** citizens
- Class body is always executed in **strict mode**

Asynchronous

ASYNCHRONOUS CODE

Asynchronous

CALLBACK WILL RUN AFTER IMAGE LOADS

```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

THREAD OF EXECUTION

“BACKGROUND”

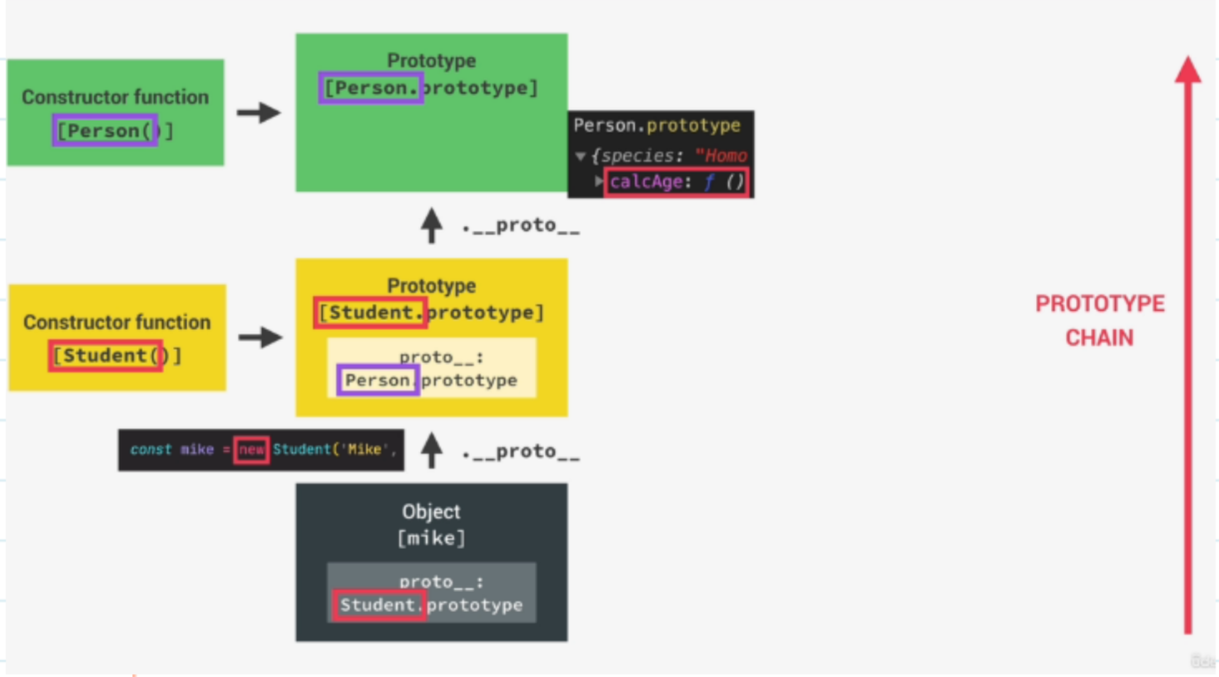
(More on this in the lecture on Event Loop)

- **Example:** Asynchronous image loading with event and callback
- Asynchronous code is executed **after a task that runs in the “background” finishes**;
- Asynchronous code is **non-blocking**;
- Execution doesn’t wait for an asynchronous task to finish its work;
- Callback functions alone do **NOT** make code asynchronous!

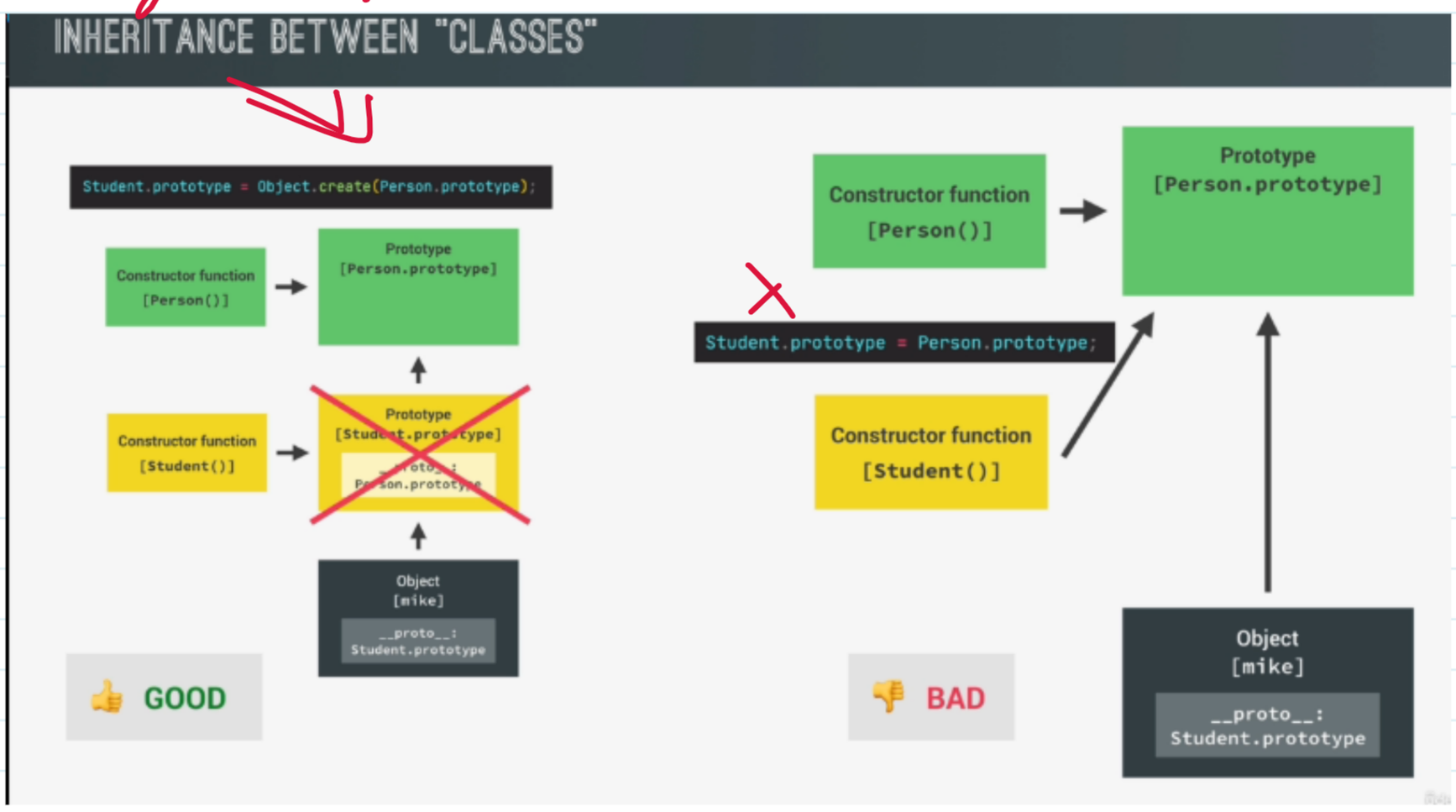
ASYNCHRONOUS

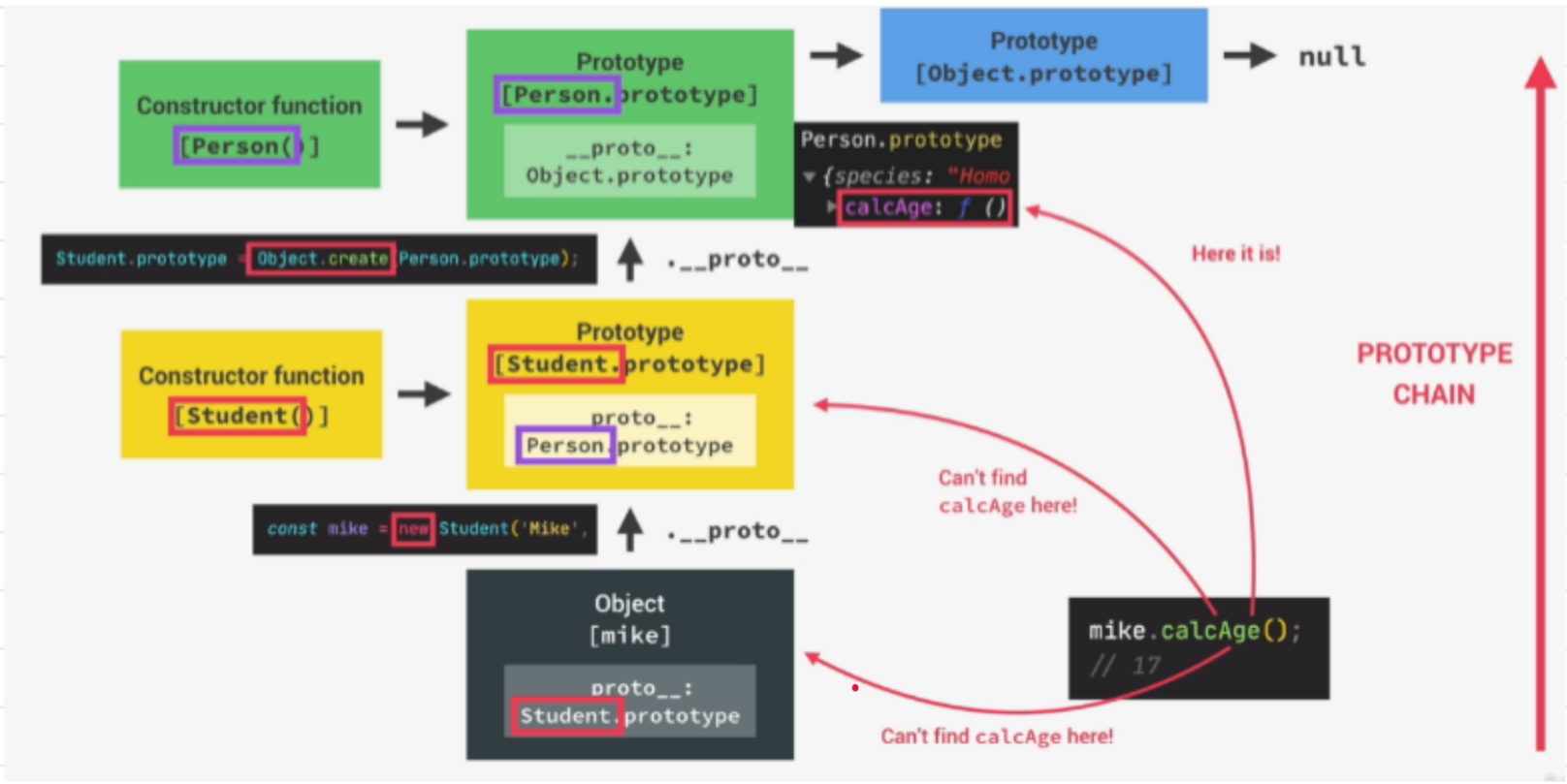
Coordinating behavior of a program ... period of time

2x 8:50 / 17:57



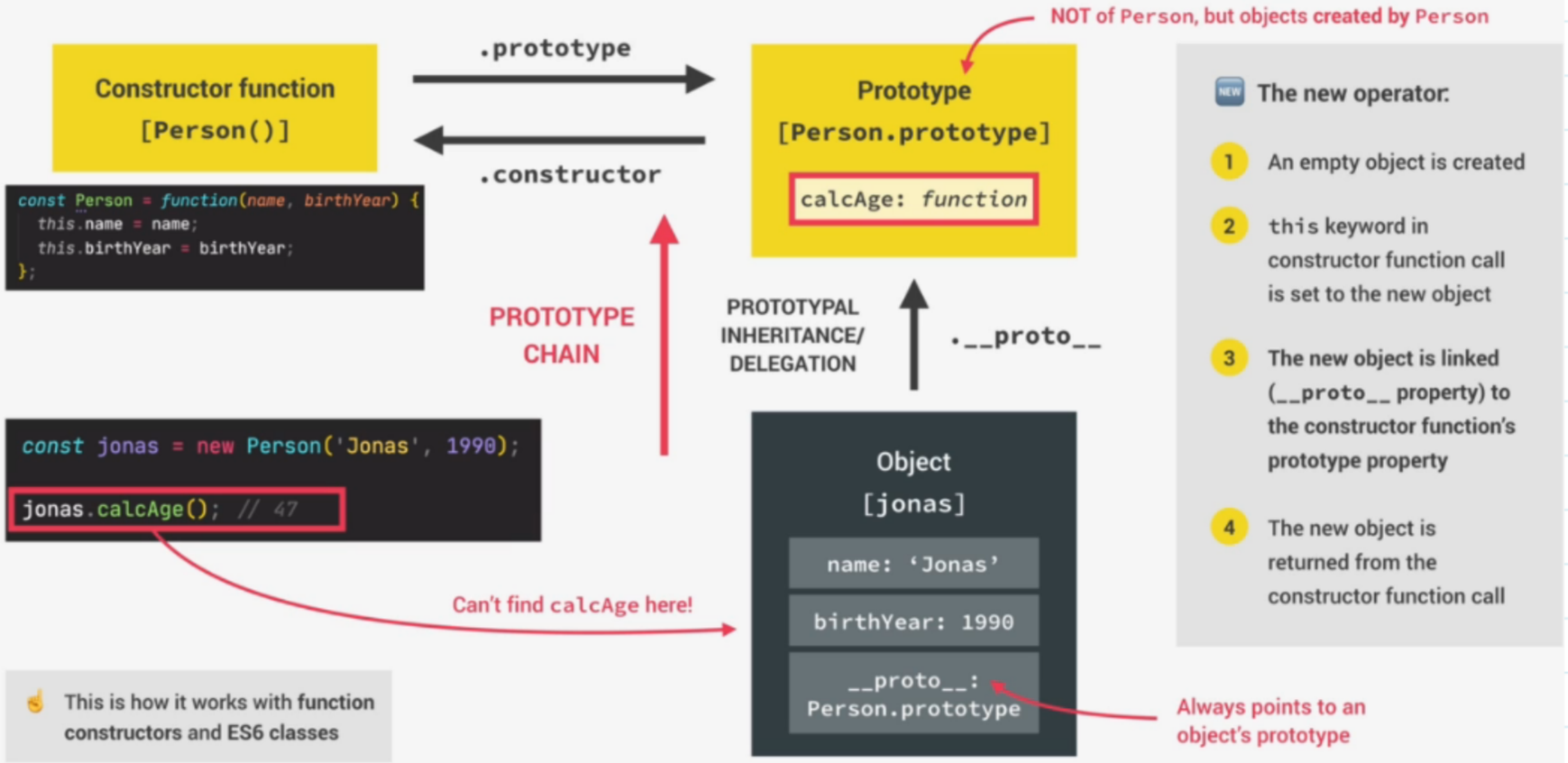
why Object.create( ) should be use for functional class



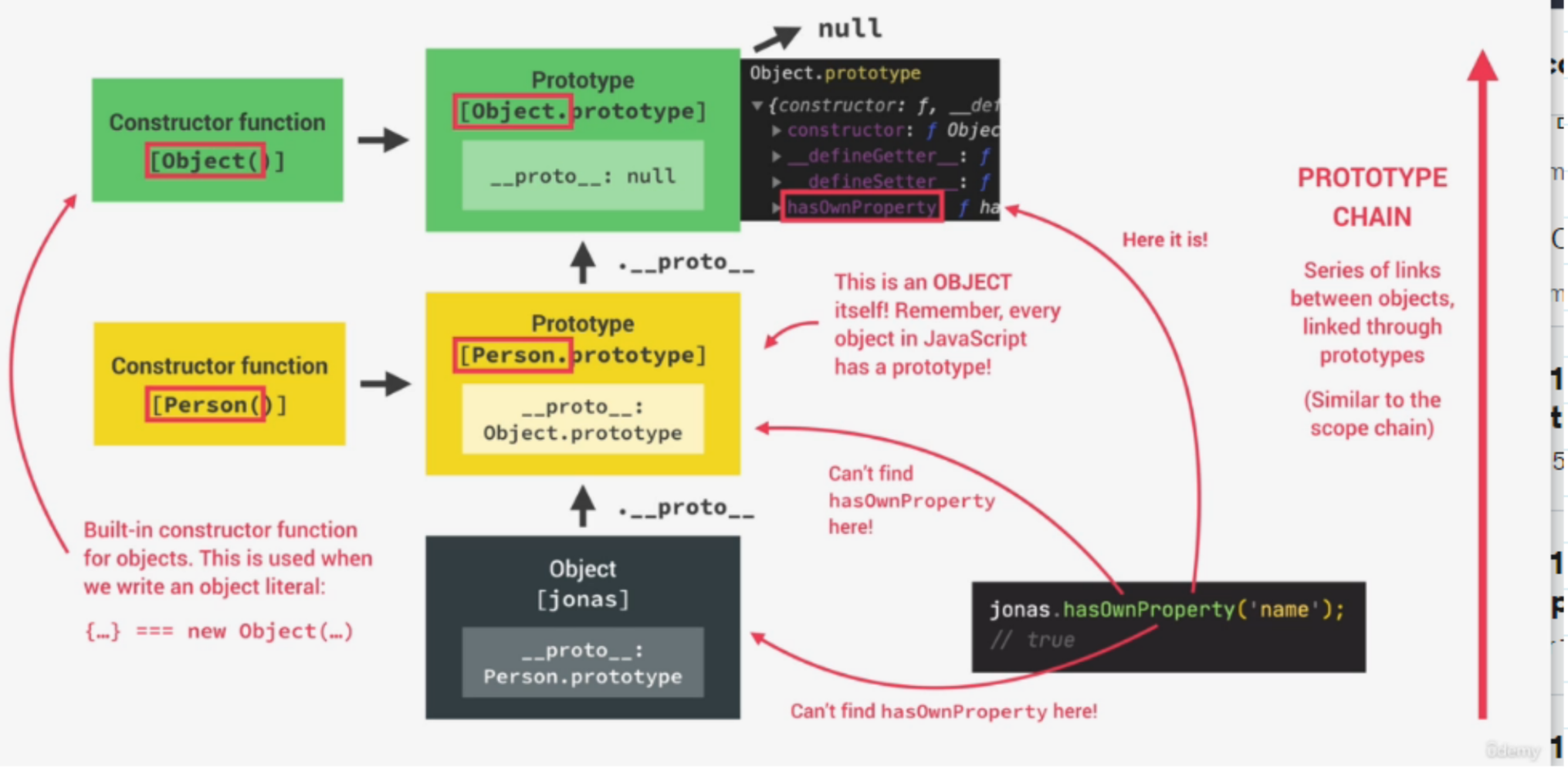


Prototype Chain

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS

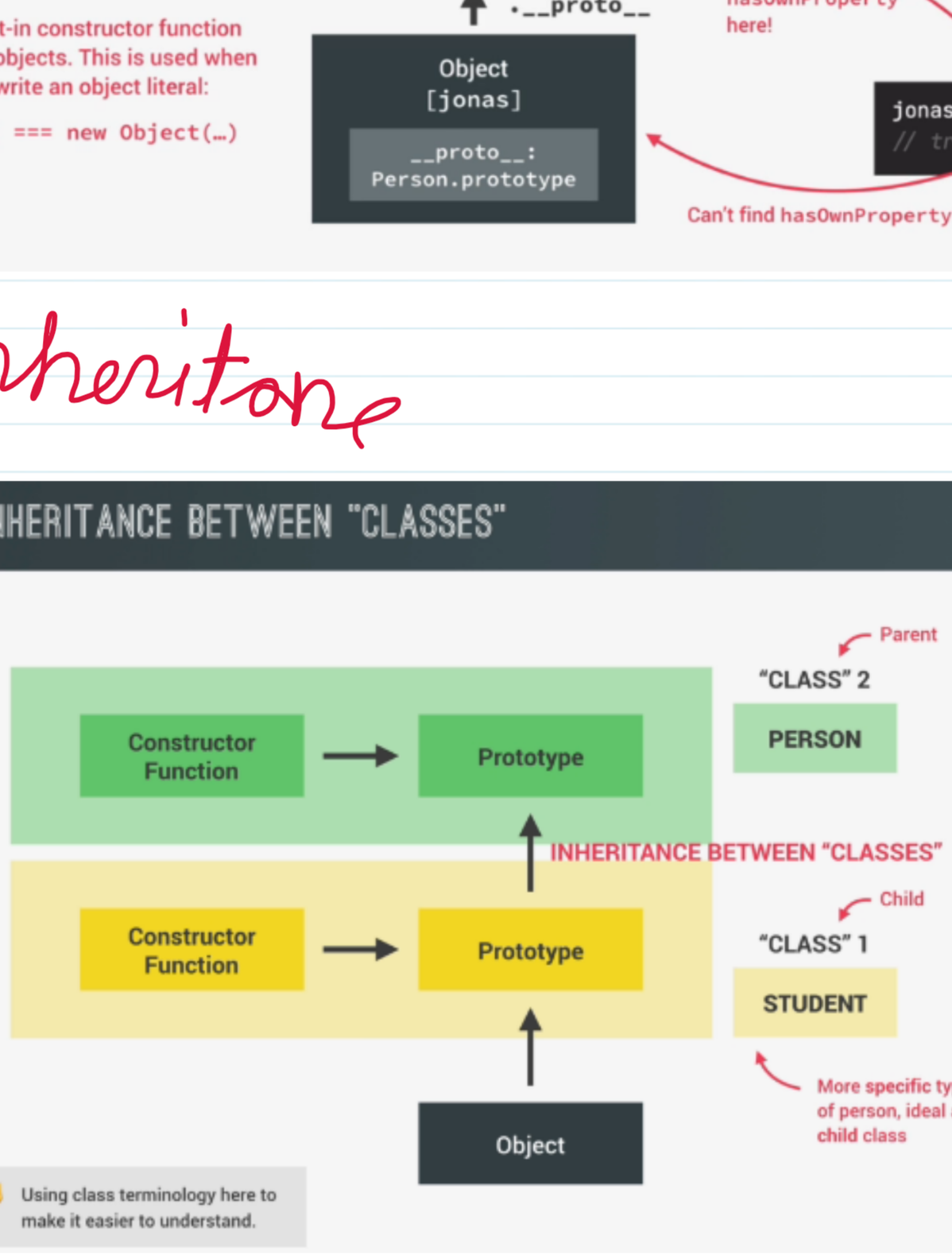


THE PROTOTYPE CHAIN



Inheritance

INHERITANCE BETWEEN "CLASSES"



JS achille Inheritance

#0 OOPs

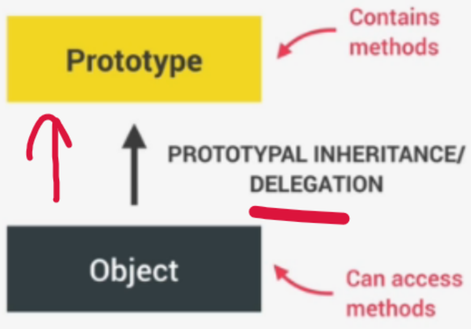
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

OOP IN JAVASCRIPT: PROTOTYPES

"CLASSICAL OOP": CLASSES

- Objects (instances) are **instantiated** from a class, which functions like a blueprint;

OOP IN JS: PROTOTYPES



- Objects are **linked** to a prototype object;
- **Prototypal inheritance:** The prototype contains methods (behavior) that are **accessible to all objects linked to that prototype;**
- Behavior is **delegated** to the linked prototype object.

JS has Prototypes

3 WAYS OF IMPLEMENTING PROTOTYPAL INHERITANCE IN JAVASCRIPT

🤔 *"How do we actually create prototypes? And how do we link objects to prototypes? How can we create new objects, without having classes?"*

1 Constructor functions

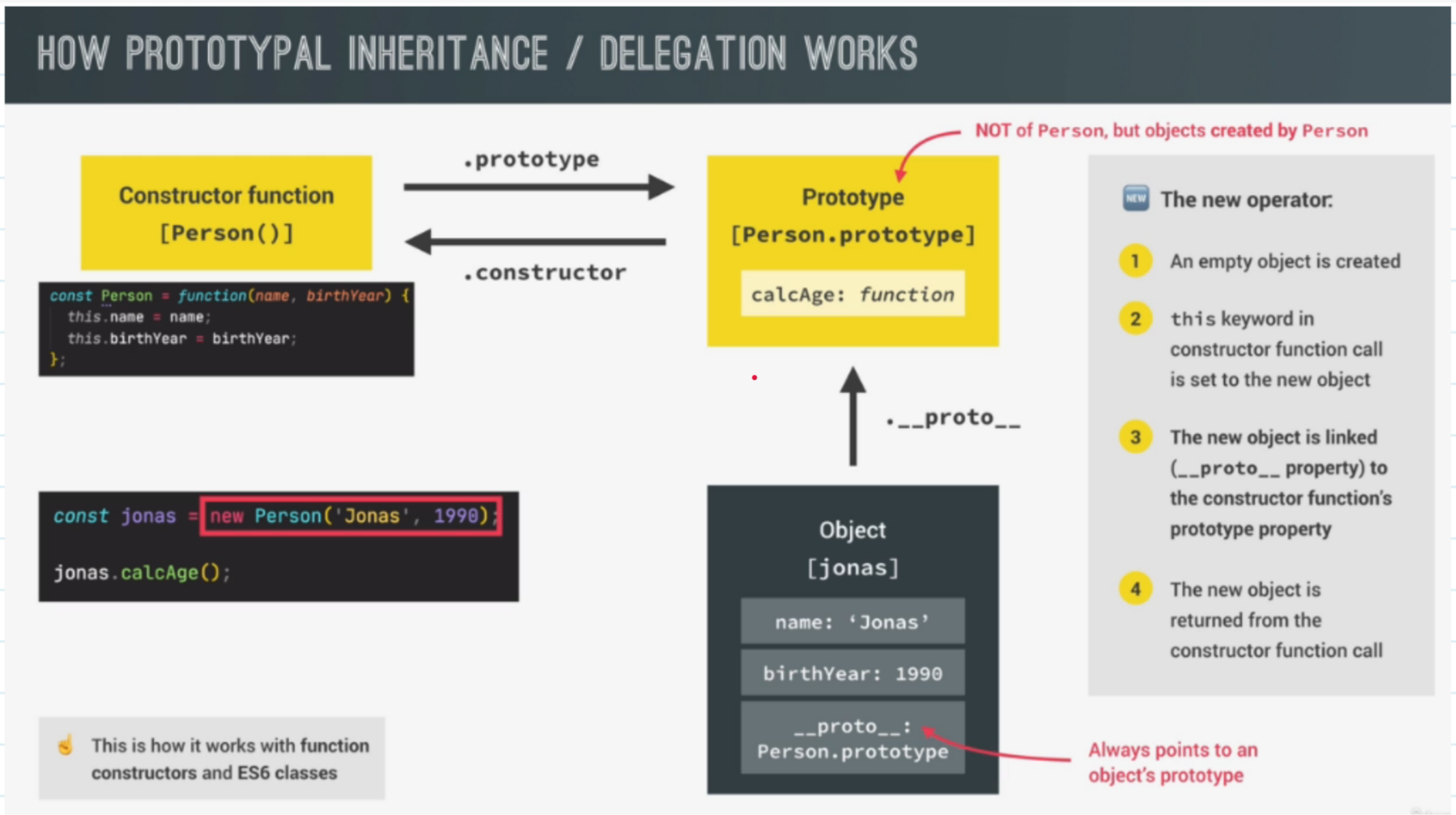
- Technique to create objects from a function;
- This is how built-in objects like Arrays, Maps or Sets are actually implemented.

2 ES6 Classes

- Modern alternative to constructor function syntax;
- "Syntactic sugar": behind the scenes, ES6 classes work **exactly** like constructor functions;
- ES6 classes do **NOT** behave like classes in "classical OOP" (last lecture).

3 `Object.create()`

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS



Best Way to load JS file

Inside head with defer keyword

```
<head>

<script defer src="script.js"></script>

</head>
```

Just tell me the best way

The best thing to do to speed up your page loading when using scripts is to put them in the `head`, and add a `defer` attribute to your `script` tag:

```
<script defer src="script.js"></script>
```

This is the scenario that triggers the faster `domInteractive` event.

Considering the pros of `defer`, is seems a better choice over `async` in a variety of scenarios.

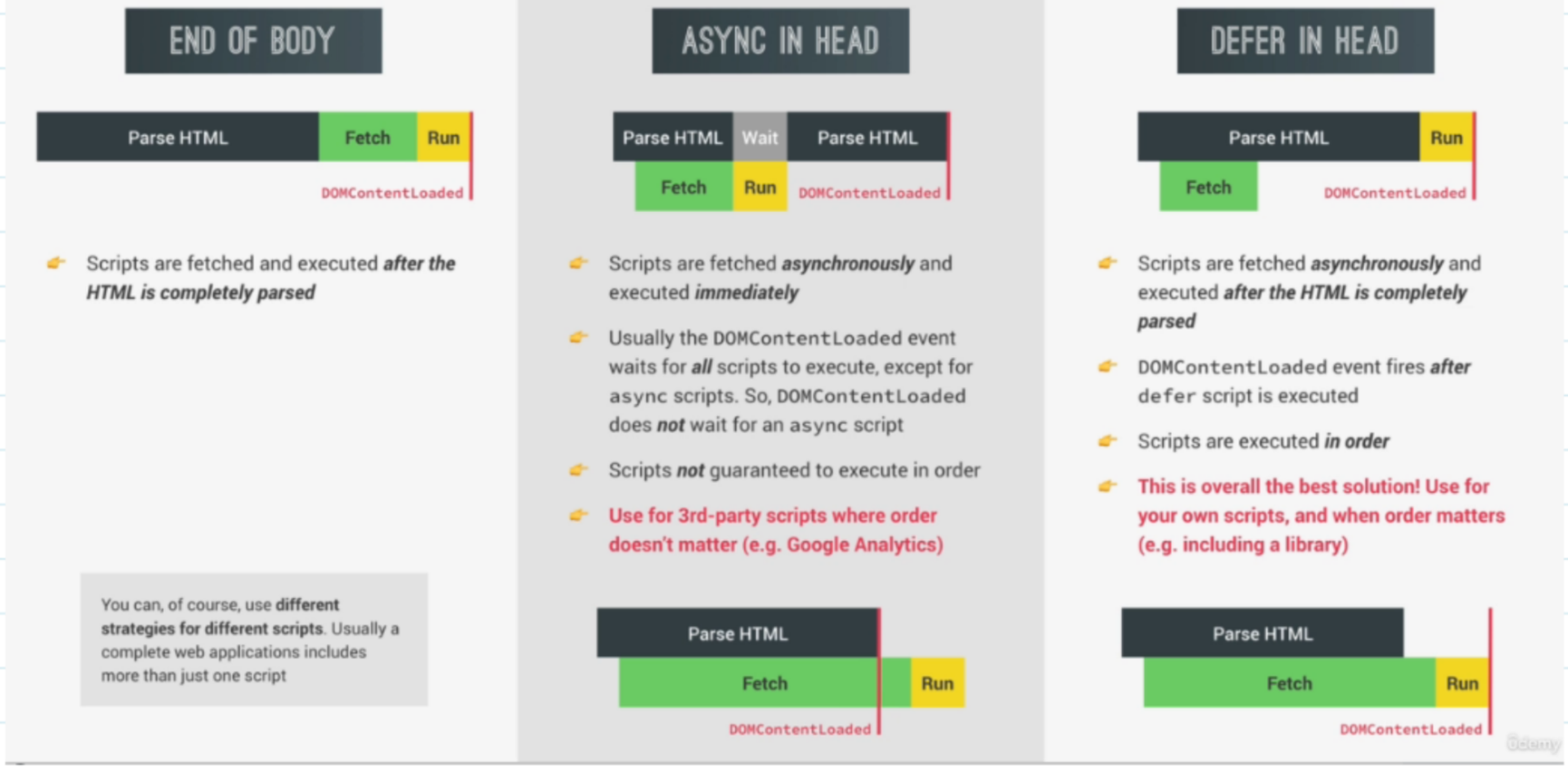
Unless you are fine with delaying the first render of the page, make sure that when the page is parsed the JavaScript you want is already executed.

DEFER AND ASYNC SCRIPT LOADING

Parsing of HTML should done first and in a single go

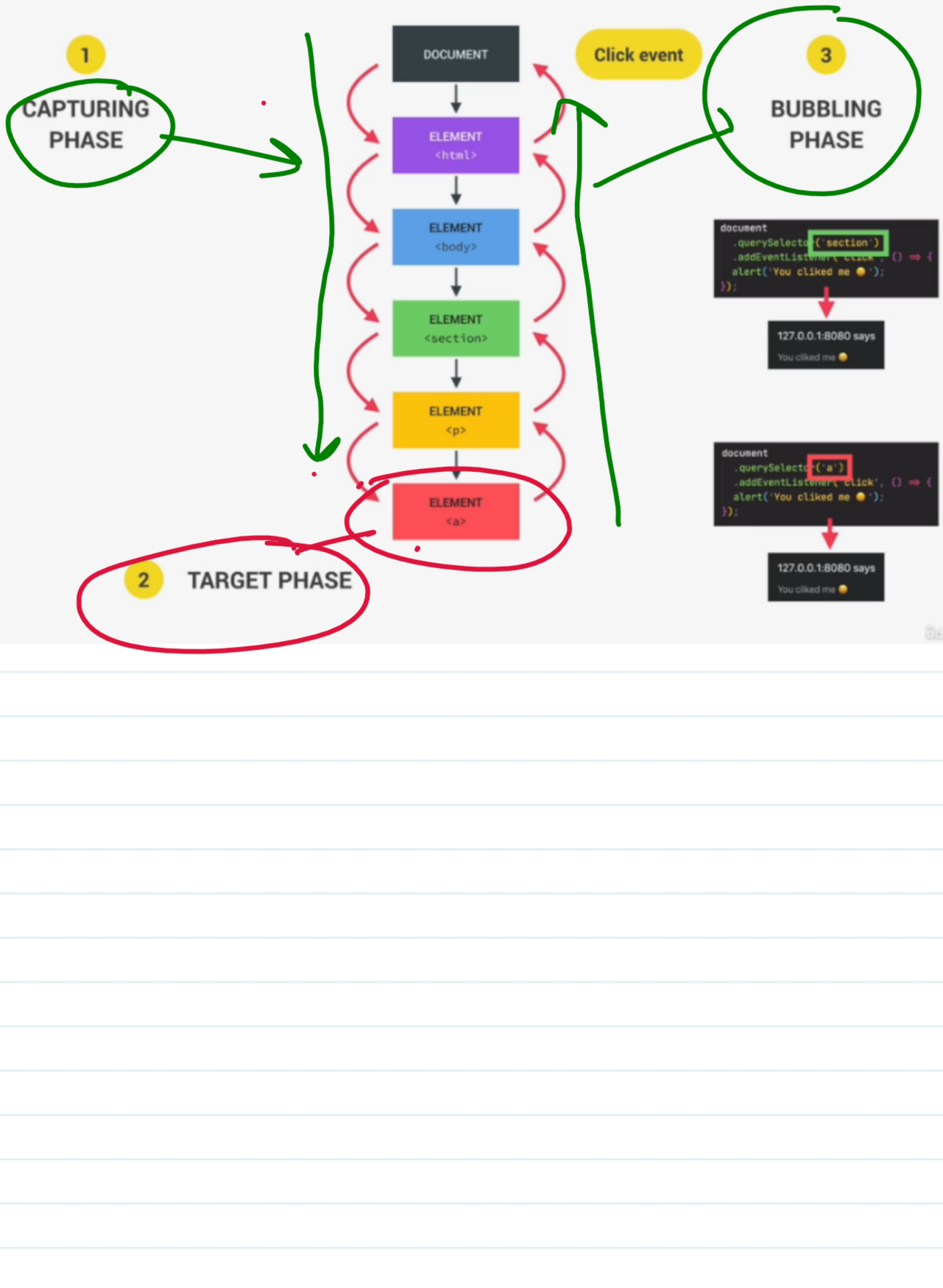
REGULAR VS. ASYNC VS. DEFER

REGULAR VS. ASYNC VS. DEFER

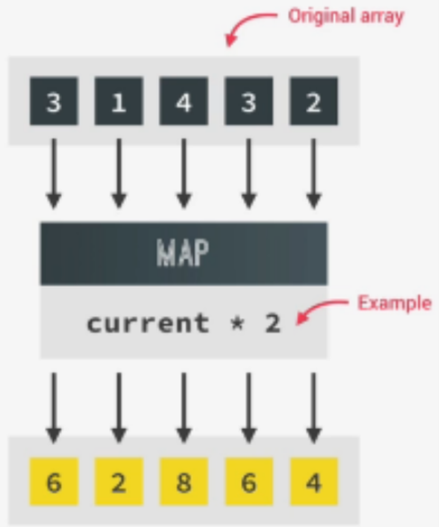


BUBBLING AND CAPTURING

```
<html>
  <head>
    <title>A Simple Page</title>
  </head>
  <body>
    <section>
      <p>A paragraph with a <a>link</a></p>
      <p>A second paragraph</p>
    </section>
    <section>
      <img src="dom.png" alt="The DOM" />
    </section>
  </body>
</html>
```

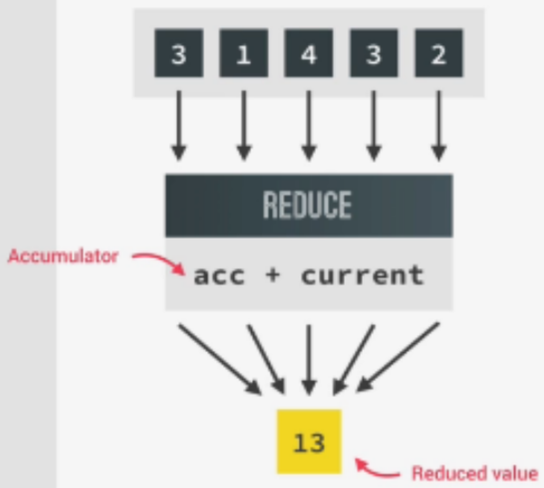


DATA TRANSFORMATIONS WITH MAP, FILTER AND REDUCE



- `map` returns a **new array** containing the results of applying an operation on all original array elements

- `filter` returns a **new array** containing the array elements that passed a specified **test condition**



- `reduce` boils ("reduces") all array elements down to one single value (e.g. adding all elements together)

WHICH ARRAY METHOD TO USE? 🤔

"I WANT...:"

To mutate original array

- Add to original:
  - `.push` *(end)*
  - `.unshift` *(start)*
- Remove from original:
  - `.pop` *(end)*
  - `.shift` *(start)*
  - `.splice` *(any)*
- Others:
  - `.reverse`
  - `.sort`
  - `.fill`

A new array

- Computed from original:
  - `.map` *(loop)*
- Filtered using condition:
  - `.filter`
- Portion of original:
  - `.slice`
- Adding original to other:
  - `.concat`
- Flattening the original:
  - `.flat`
  - `.flatMap`

An array index

- Based on value:
  - `.indexOf`
- Based on test condition:
  - `.findIndex`

An array element

- Based on test condition:
  - `.find`

Know if array includes

- Based on value:
  - `.includes`
- Based on test condition:
  - `.some`
  - `.every`

A new string

- Based on separator string:
  - `.join`

To transform to value

- Based on accumulator:
  - `.reduce`

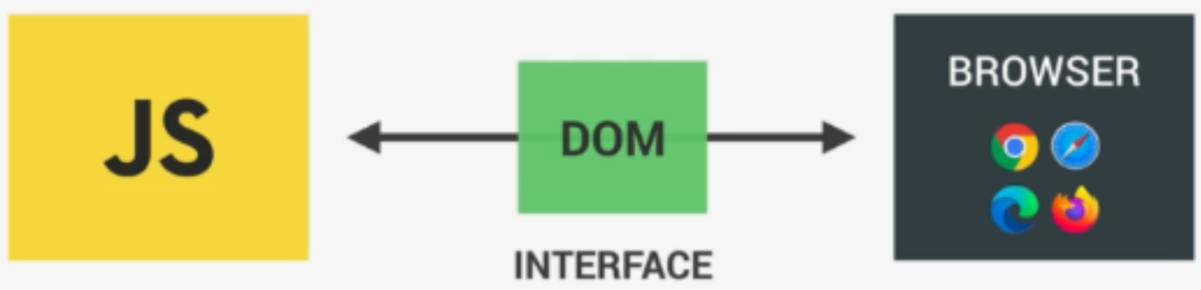
(Boil down array to single value of any type: number, string, boolean, or even new array or object)

To just loop array

- Based on callback:
  - `.forEach`

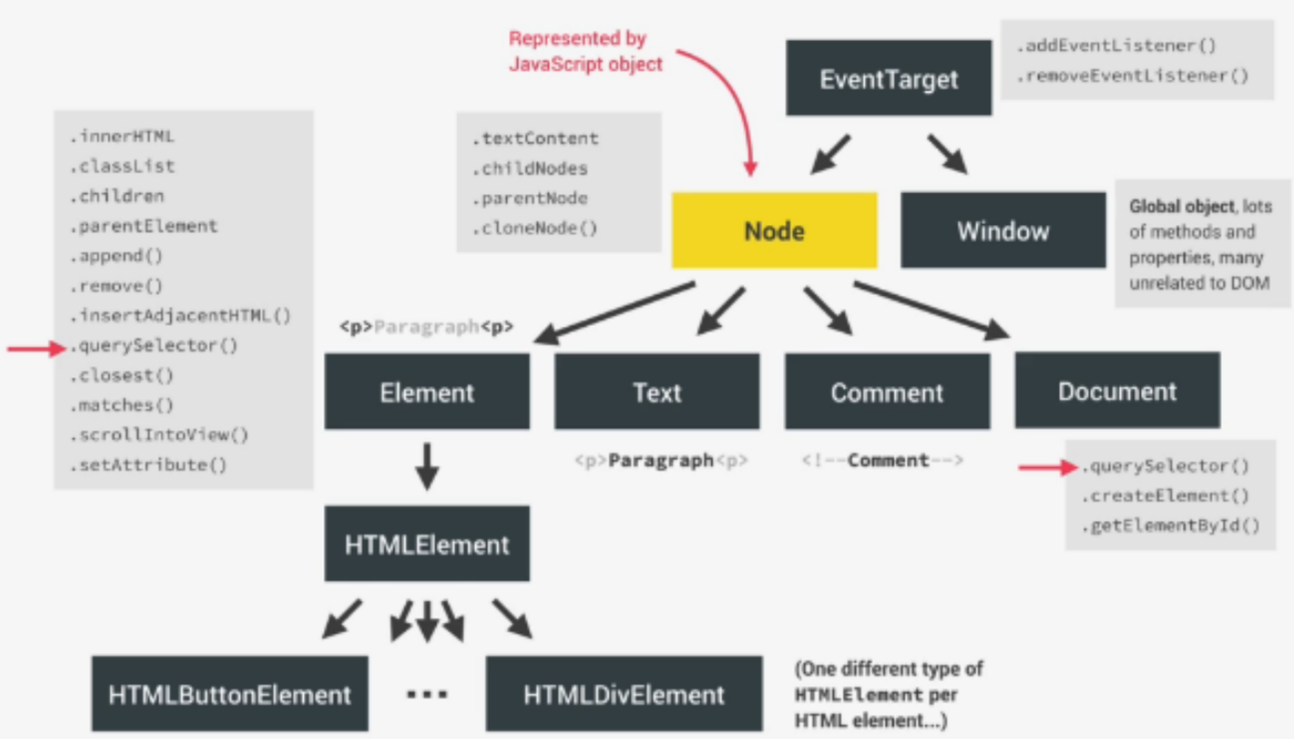
(Does not create a new array, just loops over it)

DOM



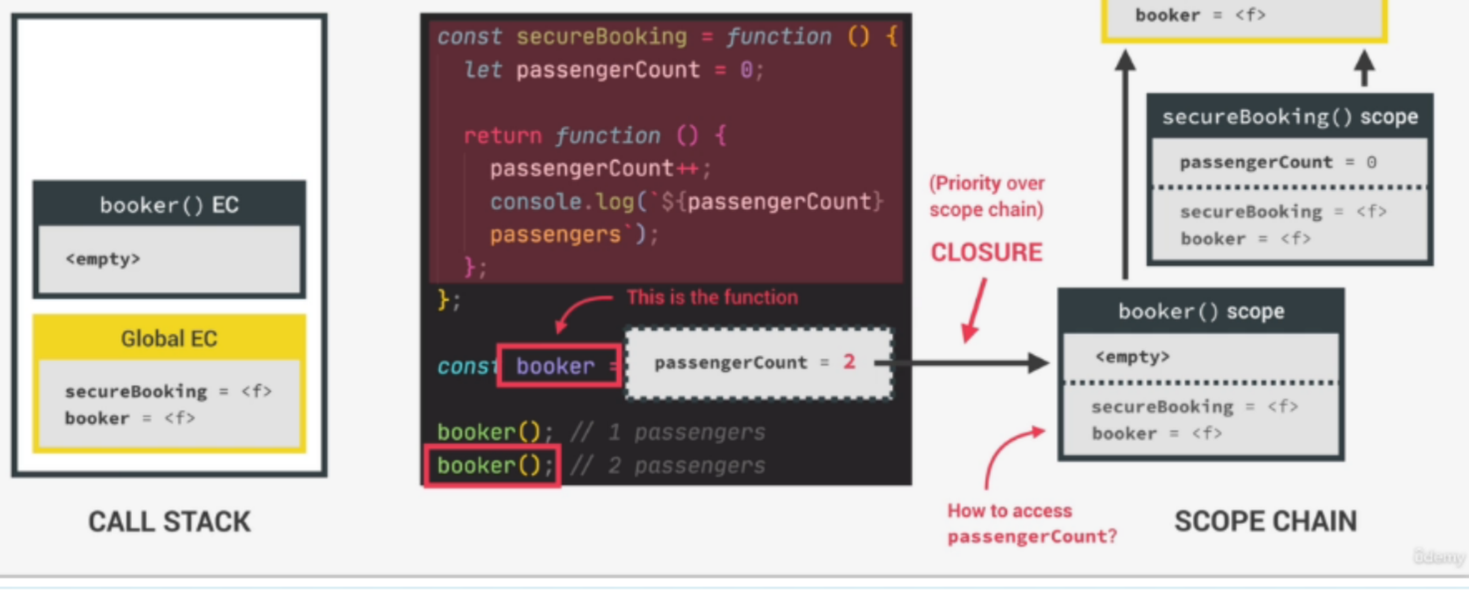
- Allows us to make JavaScript interact with the browser;
- We can write JavaScript to create, modify and delete HTML elements; set styles, classes and attributes; and listen and respond to events;
- DOM tree is generated from an HTML document, which we can then interact with;
- DOM is a very complex API that contains lots of methods and properties to interact with the DOM tree

HOW THE DOM API IS ORGANIZED BEHIND THE SCENES



UNDERSTANDING CLOSURES

- A function has access to the variable environment (VE) of the execution context in which it was created
- **Closure:** VE attached to the function, exactly as it was at the time and place the function was created



CLOSURES SUMMARY 🥳

- A closure is the closed-over **variable environment** of the execution context **in which a function was created**, even ***after*** that execution context is gone;

  Less formal

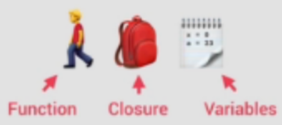
- A closure gives a function access to all the variables **of its parent function**, even ***after*** that parent function has returned. The function keeps a **reference** to its outer scope, which ***preserves*** the scope chain throughout time.

  Less formal

- A closure makes sure that a function doesn't loose connection to **variables that existed at the function's birth place**;

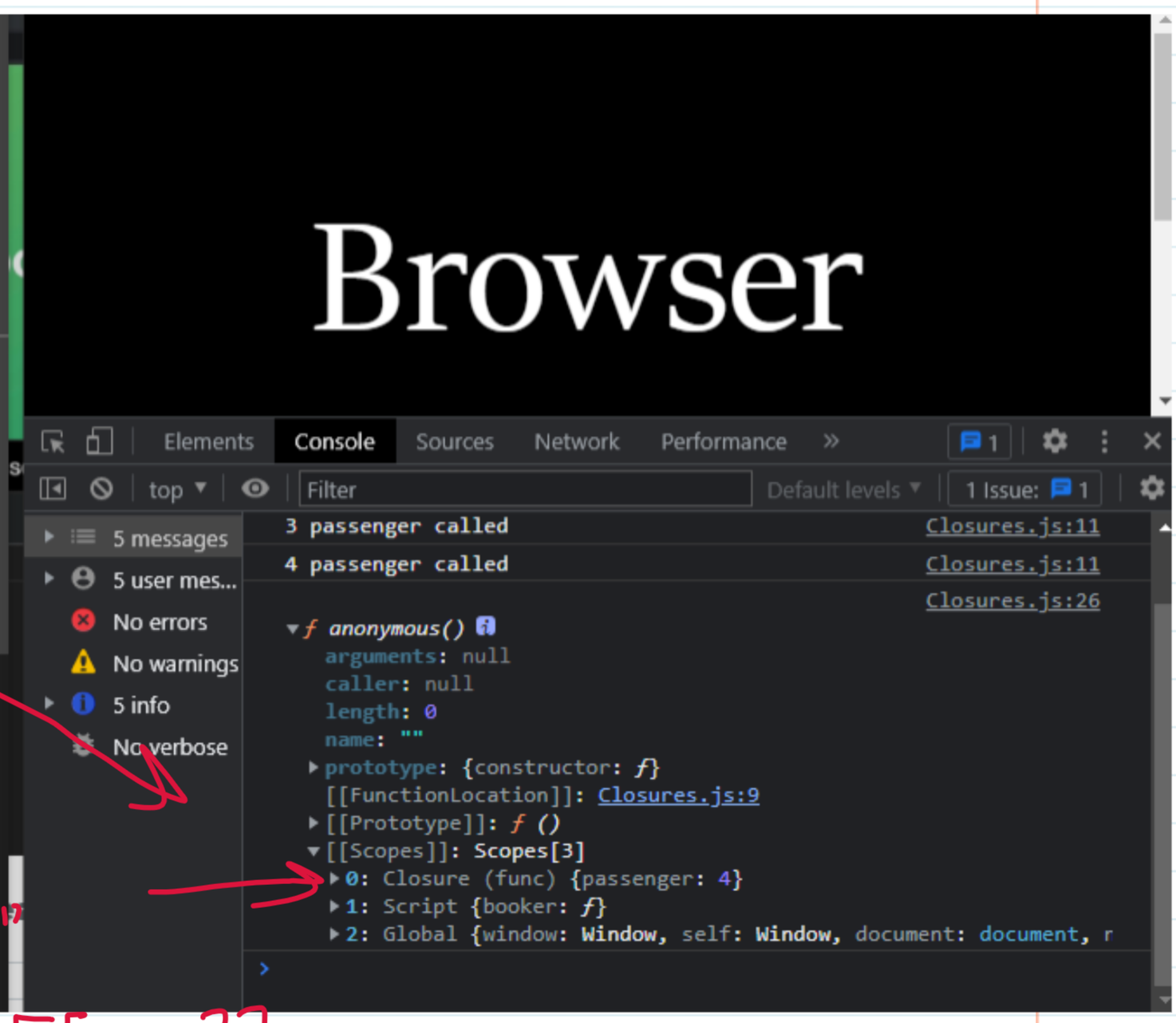
  Less formal

- A closure is like a **backpack** that a function carries around wherever it goes. This backpack has all the **variables that were present in the environment where the function was created**.




```
*/


function func(){
    let  passenger=0;
   return function(){
    passenger++;
    console.log(passenger+" passenger called")
}
}


/**
 * Function remenber variable which were present in their birth place
 */

let booker=func()
booker();
booker();
booker();
booker();

console.dir(booker)
```

to see



"closer in scope"

[[ ]]

we cannot access variable in this but closure can

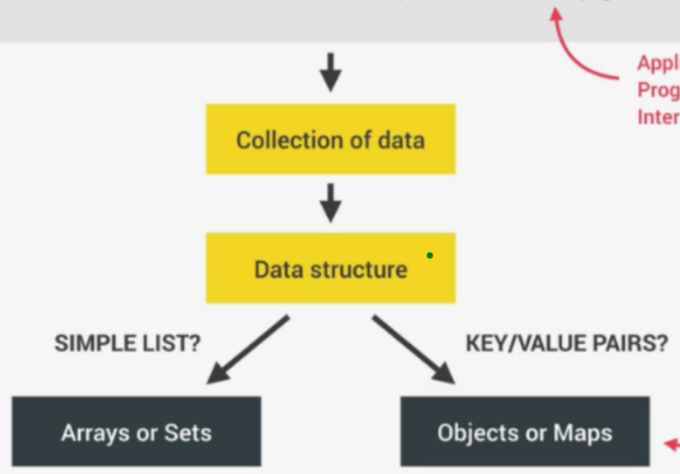
DATA STRUCTURES OVERVIEW

SOURCES OF DATA

1. **From the program itself:** Data written directly in source code (e.g. status messages)
2. **From the UI:** Data input from the user or data written in DOM (e.g tasks in todo app)
3. **From external sources:** Data fetched for example from web API (e.g. recipe objects)

Application Programming Interface

JSON data format example



Keys allow us to *describe* values

ARRAYS vs. SETS

```
tasks = ['Code', 'Eat', 'Code'];
// ["Code", "Eat", "Code"]
```

- Use when you need **ordered** list of values (might contain duplicates)
- Use when you need to **manipulate** data

```
tasks = new Set(['Code', 'Eat', 'Code']);
// {"Code", "Eat"}
```

- Use when you need to work with **unique** values
- Use when **high-performance** is *really* important
- Use to **remove duplicates** from arrays

OBJECTS vs. MAPS

```
task = {
  task: 'Code',
  date: 'today',
  repeat: true
};
```

- More "traditional" key/value store ("abused" objects)
- Easier to write and access values with `.` and `[]`

```
task = new Map([
  ['task', 'Code'],
  ['date', 'today'],
  [false, 'Start coding!']
]);
```

- Better performance
- Keys can have **any** data type
- Easy to iterate
- Easy to compute size

FIRST-CLASS VS. HIGHER-ORDER FUNCTIONS

FIRST-CLASS FUNCTIONS

- JavaScript treats functions as **first-class citizens**
- This means that functions are **simply values**
- Functions are just another **"type" of object**

- Store functions in variables or properties:

```
const add = (a, b) => a + b;

const counter = {
  value: 23,
  inc: function() { this.value++; }
};
```

- Pass functions as arguments to OTHER functions:

```
const greet = () => console.log('Hey Jonas');
btnClose.addEventListener('click', greet)
```

- Return functions FROM functions
- Call methods on functions:

```
counter.inc.bind(someOtherObject);
```

HIGHER-ORDER FUNCTIONS

- A function that **receives** another function as an argument, that **returns** a new function, or **both**
- This is only possible because of first-class functions

1. Function that receives another function

```
const greet = () => console.log('Hey Jonas');
btnClose.addEventListener('click', greet)
```

Higher-order function — Callback function

2. Function that returns new function

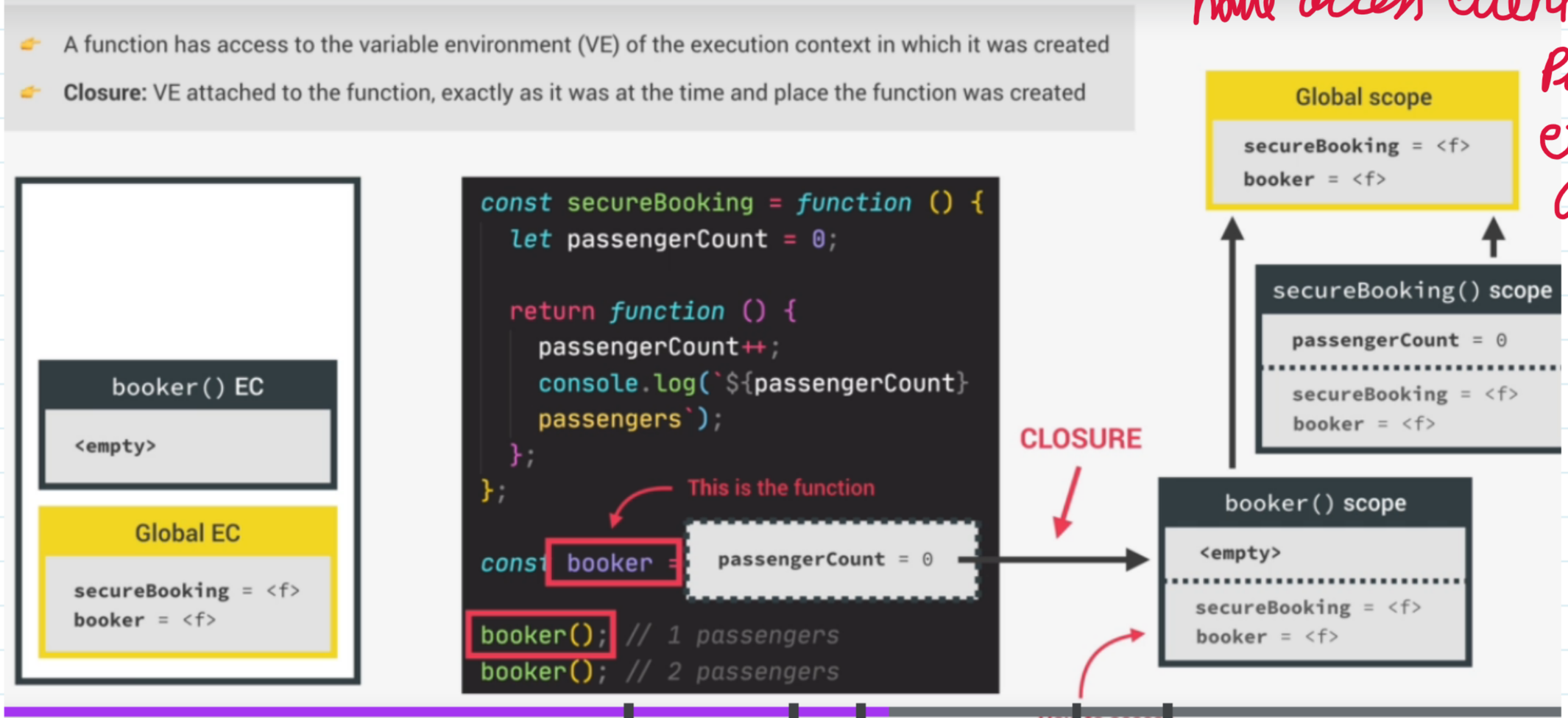
```
function count() {
  let counter = 0;
  return function() {
    counter++;
  };
}
```

Higher-order function — Returned function

how does event

- A function has access to the variable environment (VE) of the execution context in which it was created
- **Closure:** VE attached to the function, exactly as it was at the time and place the function was created

Parent execution context destroyed



TEMPORAL DEAD ZONE, LET AND CONST

```
const myName = 'Jonas';

if (myName === 'Jonas') {
  console.log(`Jonas is a ${job}`);
  const age = 2037 - 1989;
  console.log(age);
  const job = 'teacher';
  console.log(x);
}
```

TEMPORAL DEAD ZONE FOR `job` VARIABLE

👉 Different kinds of error messages:

`ReferenceError: Cannot access 'job' before initialization`

`ReferenceError: x is not defined`

WHY TDZ?

👉 **Makes it easier to avoid and catch errors:** accessing variables before declaration is bad practice and should be avoided;

👉 **`this` keyword/variable:** Special variable that is created for every execution context (every function). Takes the value of (points to) the "owner" of the function in which the `this` keyword is used.

☝ `this` is **NOT** static. It depends on **how** the function is called, and its value is only assigned when the function **is actually called**.

EXECUTION CONTEXT
- ✅ Variable environment
- ✅ Scope chain
- ✅ this keyword

Method 👉 `this` = <Object that is calling the method>

Simple function call 👉 `this` = `undefined` (In strict mode! Otherwise: window (in the browser))

Arrow functions 👉 `this` = <`this` of surrounding function (lexical `this`)> (Don't get own this)

Event listener 👉 `this` = <DOM element that the handler is attached to>

☝ `this` does **NOT** point to the function itself, and also **NOT** the its variable environment!

👉 Method example:

```
const jonas = {
  name: 'Jonas',
  year: 1989,
  calcAge: function() {
    return 2037 - this.year
  }
};
jonas.calcAge(); // 48
```

calcAge is method — jonas — 1989

Way better than using `jonas.year`!

PRIMITIVES

- Number
- String
- Boolean
- Undefined
- Null
- Symbol
- BigInt

PRIMITIVE TYPES

JS ENGINE

CALL STACK — HEAP

STORED IN

OBJECTS

- Object literal
- Arrays
- Functions
- Many more...

REFERENCE TYPES

PRIMITIVE VS. REFERENCE VALUES

👉 **Primitive** values example:

```
let age = 30;
let oldAge = age;
age = 31;
console.log(age); // 31
console.log(oldAge); // 30
```

Identifier	Address	Value
age	0001	30
oldAge	0002	31
me	0003	D30F
friend		

CALL STACK

Address	Value
D30F	{ name: 'Jonas'; age: 30; }

Reference to memory address in Heap

HEAP

👉 **Reference** values example:

```
const me = {
  name: 'Jonas',
  age: 30
};
const friend = me;
friend.age = 27;

console.log('Friend:', friend);
// { name: 'Jonas', age: 27 }

console.log('Me:', me);
// { name: 'Jonas', age: 27 }
```

age → 30, oldAge → age, but when age occurs both value were different so JS will create new variable because value in stack change

JS will not create new value as D30F (address) remain same

EXECUTION CONTEXT IN DETAIL

WHAT'S INSIDE EXECUTION CONTEXT?

1. Variable Environment
   - `let`, `const` and `var` declarations
   - Functions
   - ~~arguments~~ object
2. Scope chain
3. ~~this~~ keyword

NOT in arrow functions!

Generated during "creation phase", right before execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```

Global

```
name = 'Jonas'
first = <function>
second = <function>
x = <unknown>
```

Literally the function code

Need to run `first()` first

first()

```
a = 1
b = <unknown>
```

Need to run `second()` first

second()

```
c = 2
arguments = [7, 9]
```

Array of passed arguments. Available in all "regular" functions (not arrow)

(Technically, values only become known during execution)

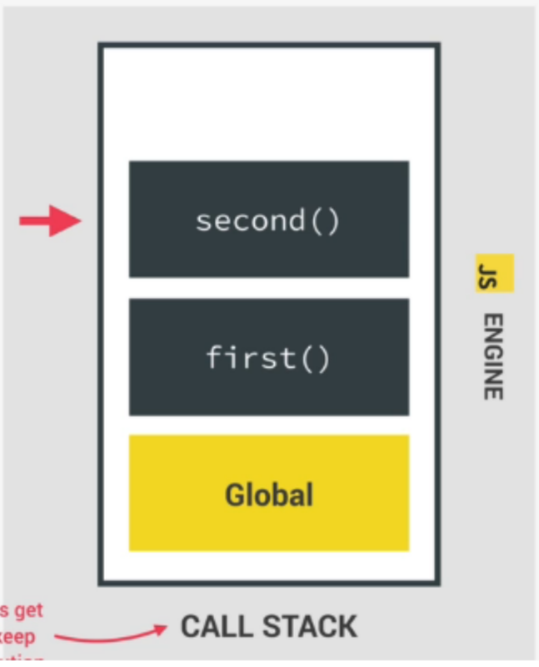
"Place" where execution contexts get stacked on top of each other, to keep track of where we are in the execution

CALL STACK

THE CALL STACK

Compiled code starts execution

```
const name = 'Jonas';

const first = () => {
  let a = 1;
  const b = second(7, 9);
  a = a + b;
  return a;
};

function second(x, y) {
  var c = 2;
  return c;
}

const x = first();
```



"Place" where execution contexts get stacked on top of each other, to keep

CALL STACK

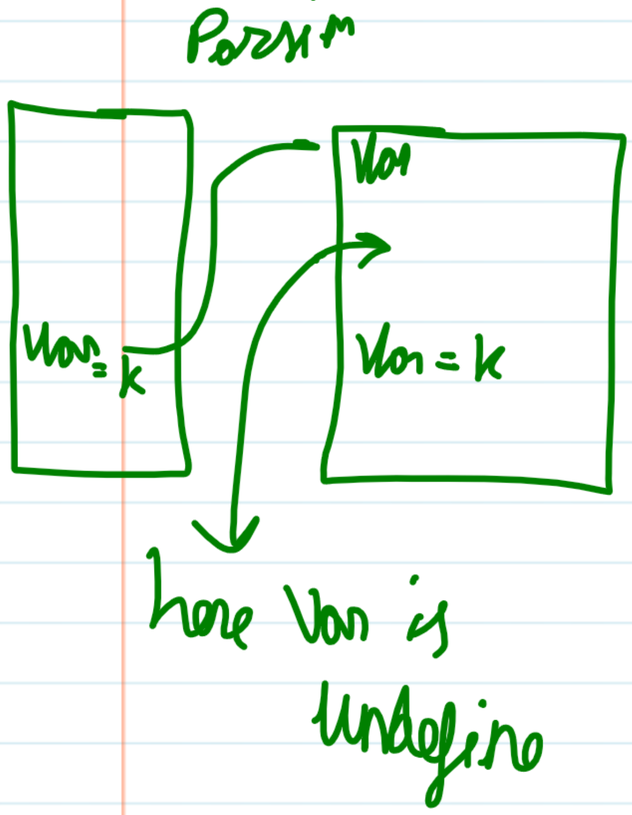
var	let	const
The scope of a var variable is functional scope.	The scope of a let variable is block scope.	The scope of a const variable is block scope.

- Scoping asks the question *"Where do variables live?"* or *"Where can we access a certain variable, and where not?"*;
- There are 3 types of scope in JavaScript: the global scope, scopes defined by functions, and scopes defined by blocks;
- Only `let` and `const` variables are block-scoped. Variables declared with `var` end up in the closest function scope;
- In JavaScript, we have lexical scoping, so the rules of where we can access variables are based on exactly where in the code functions and blocks are written;

Back to tab

- Every scope always has access to all the variables from all its outer scopes. This is the scope chain!
- When a variable is not in the current scope, the engine looks up in the scope chain until it finds the variable it's looking for. This is called variable lookup;
- The scope chain is a one-way street: a scope will never, ever have access to the variables of an inner scope;
- The scope chain in a certain scope is equal to adding together all the variable environments of the all parent scopes;
- The scope chain has nothing to do with the order in which functions were called. It does not affect the scope chain at all!

Posh'm



HOISTING IN JAVASCRIPT

- **Hoisting:** Makes some types of variables accessible/usable in the code before they are actually declared. "Variables lifted to the top of their scope".

BEHIND THE SCENES

Before execution, code is scanned for variable declarations, and for each variable, a new property is created in the **variable environment object**.

EXECUTION CONTEXT
- Variable environment
- Scope chain
- `this` keyword

	HOISTED?	INITIAL VALUE	SCOPE
`function` declarations	✅ YES	Actual function	Block
`var` variables	✅ YES	`undefined`	Function
`let` and `const` variables	⛔ NO	`<uninitialized>`, TDZ	Block
`function` expressions and arrows		Depends if using `var` or `let`/`const`	

In strict mode. Otherwise: function!

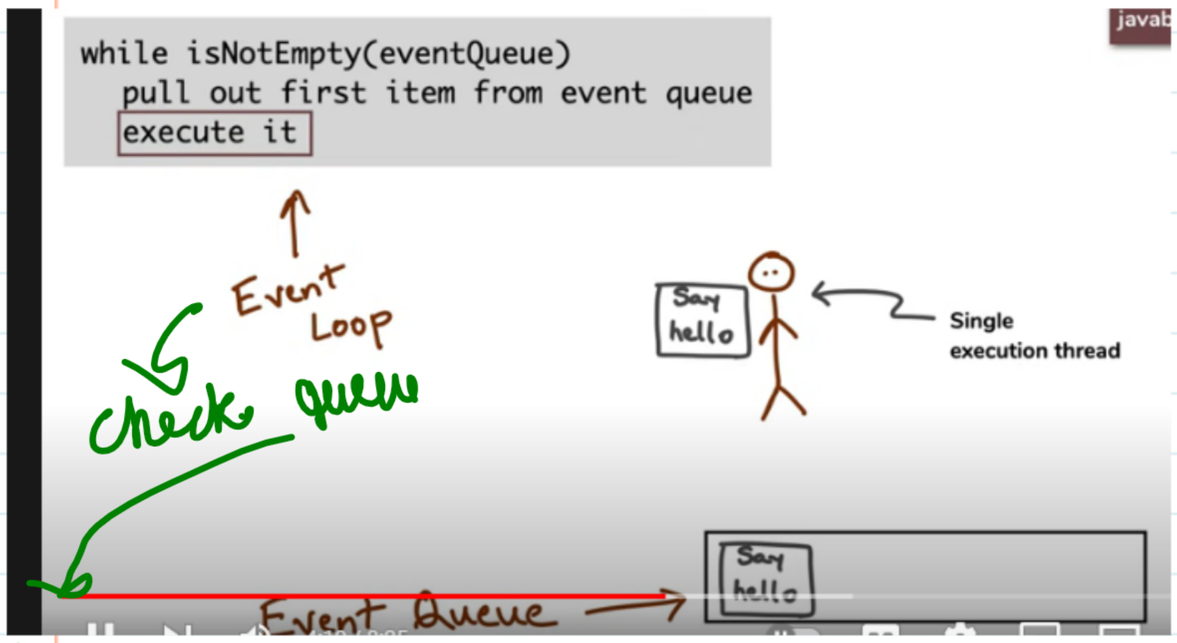
Technically, yes. But not in practice

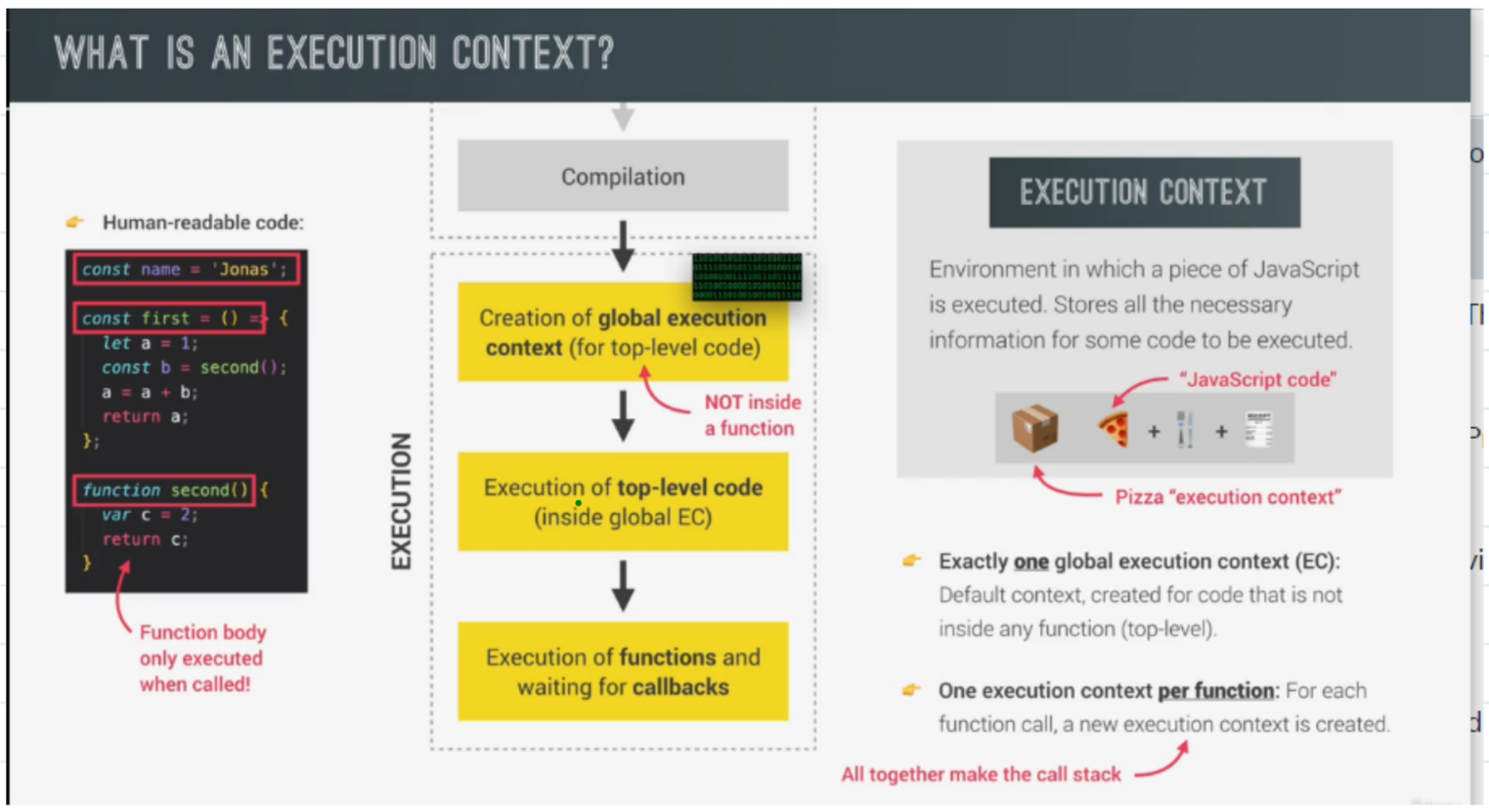
Temporal Dead Zone

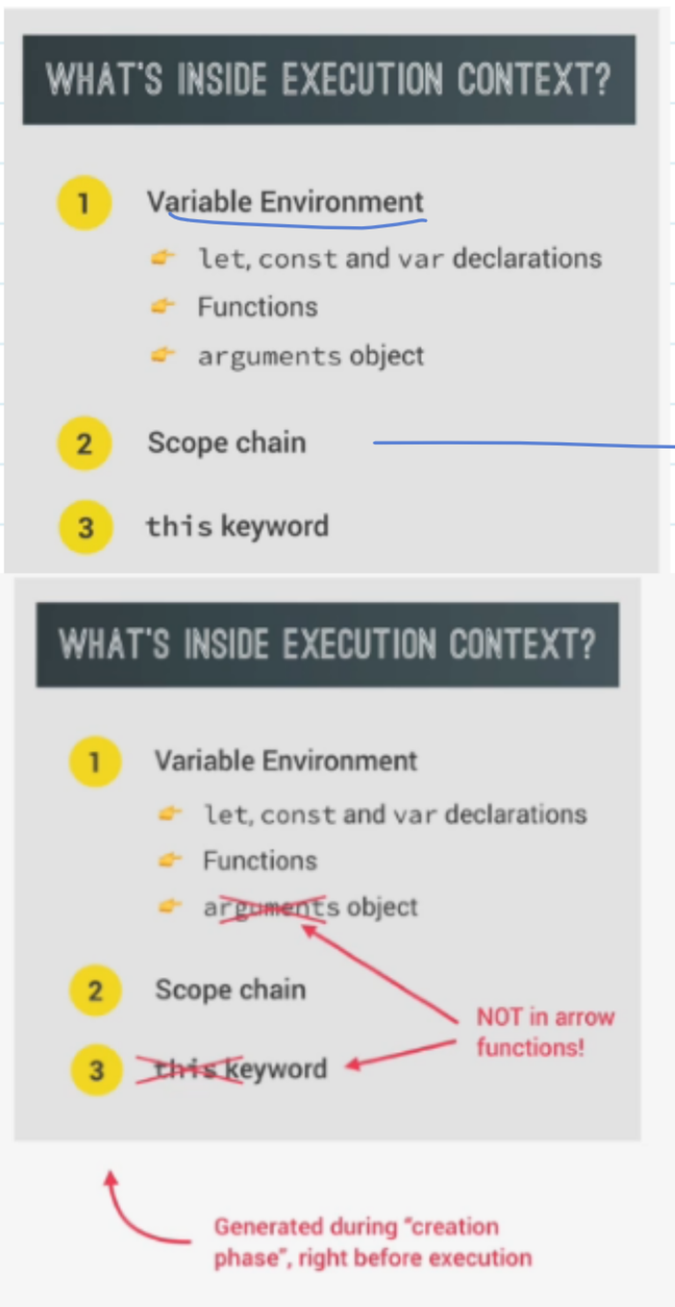
JS have " one execution thread "

(have worker thread also 😐)

Worker threads are **responsible for handling CPU-intensive tasks by transferring ArrayBuffer instances**. They have proven to be the best solution for CPU performance due to the following features: They run a single process with multiple threads. Executing one event loop per thread. 26-May-2021







What is JUST-ITIME compilation?

(Compilation with no "portable file")

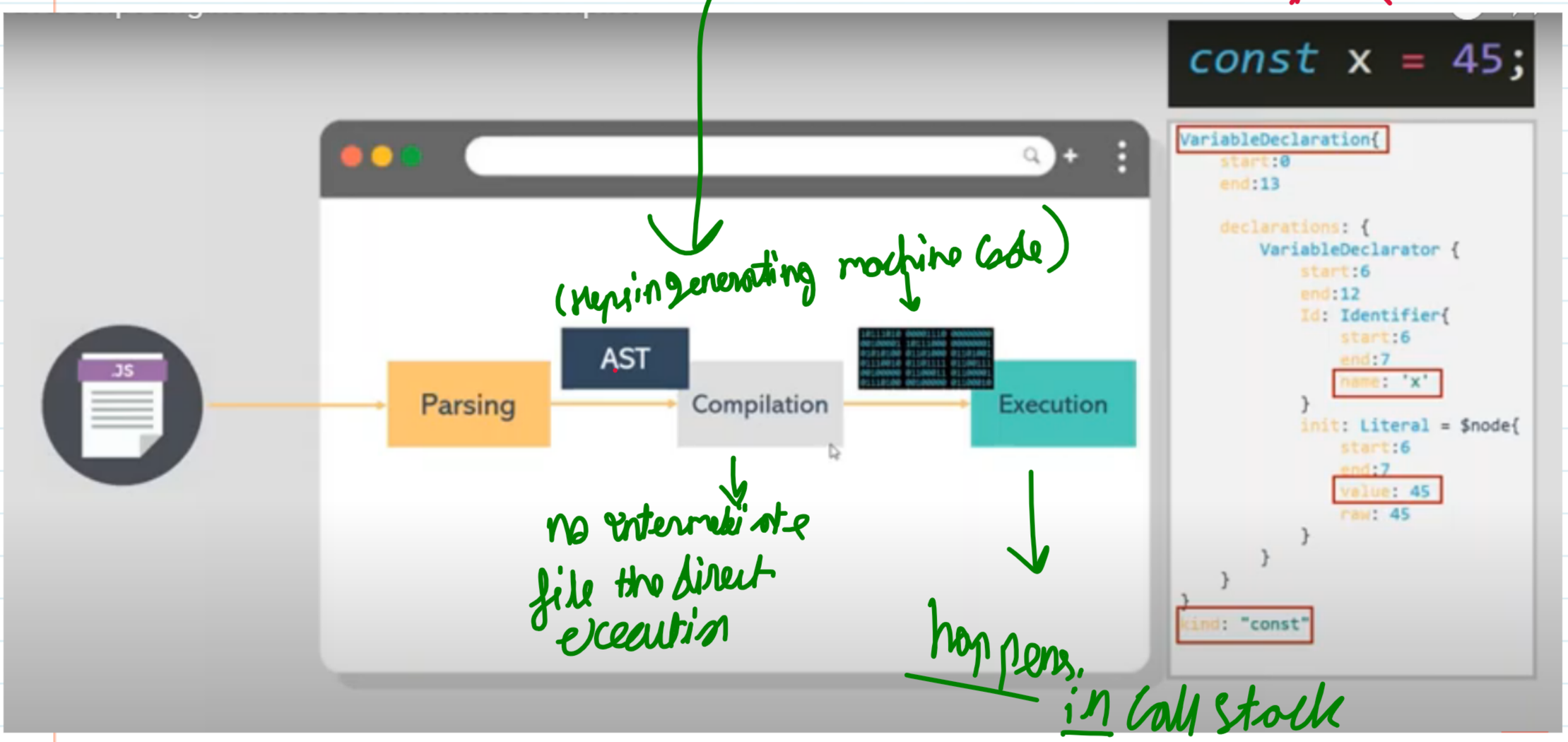
JUST IN TIME Compilation: *Entire code is converted into machine code at once and then executed immediately.*

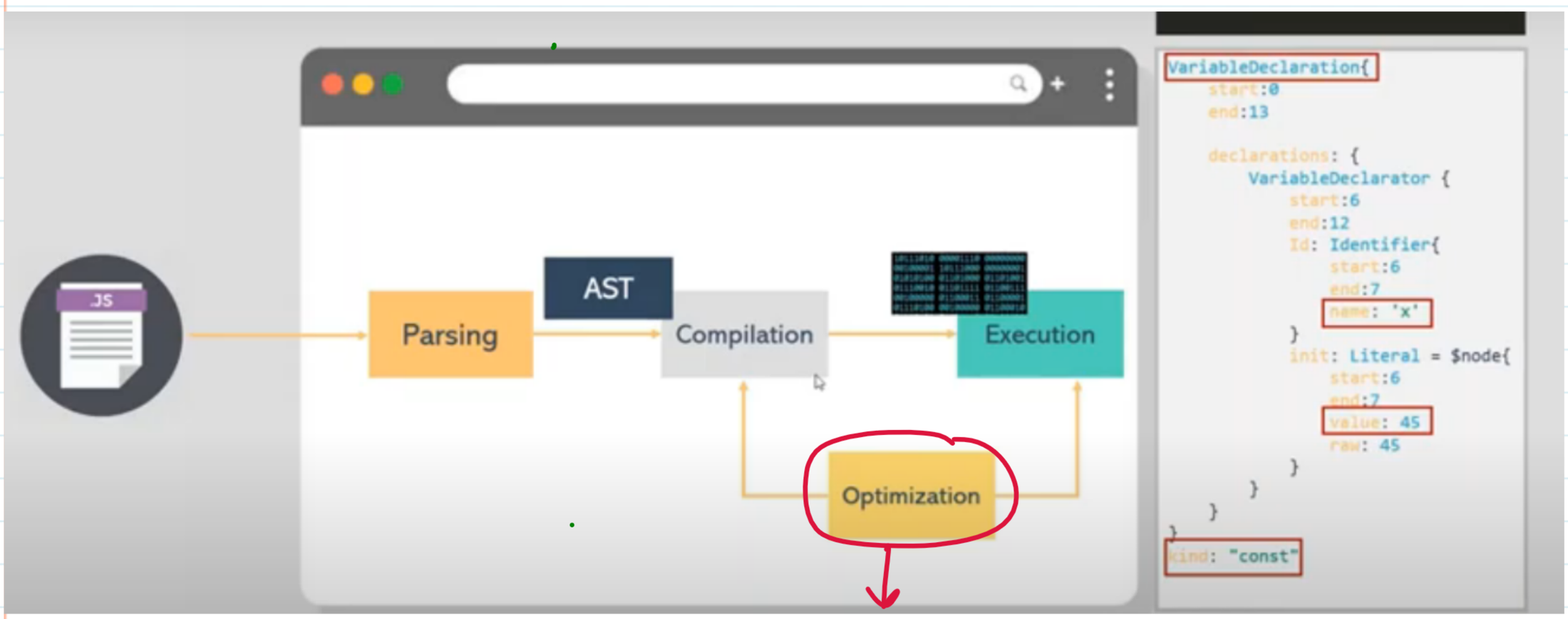
"execute immediately"

What is compilation?

Compilation: *Entire code is converted into machine code at once and written to a binary file that can be executed by a browser*

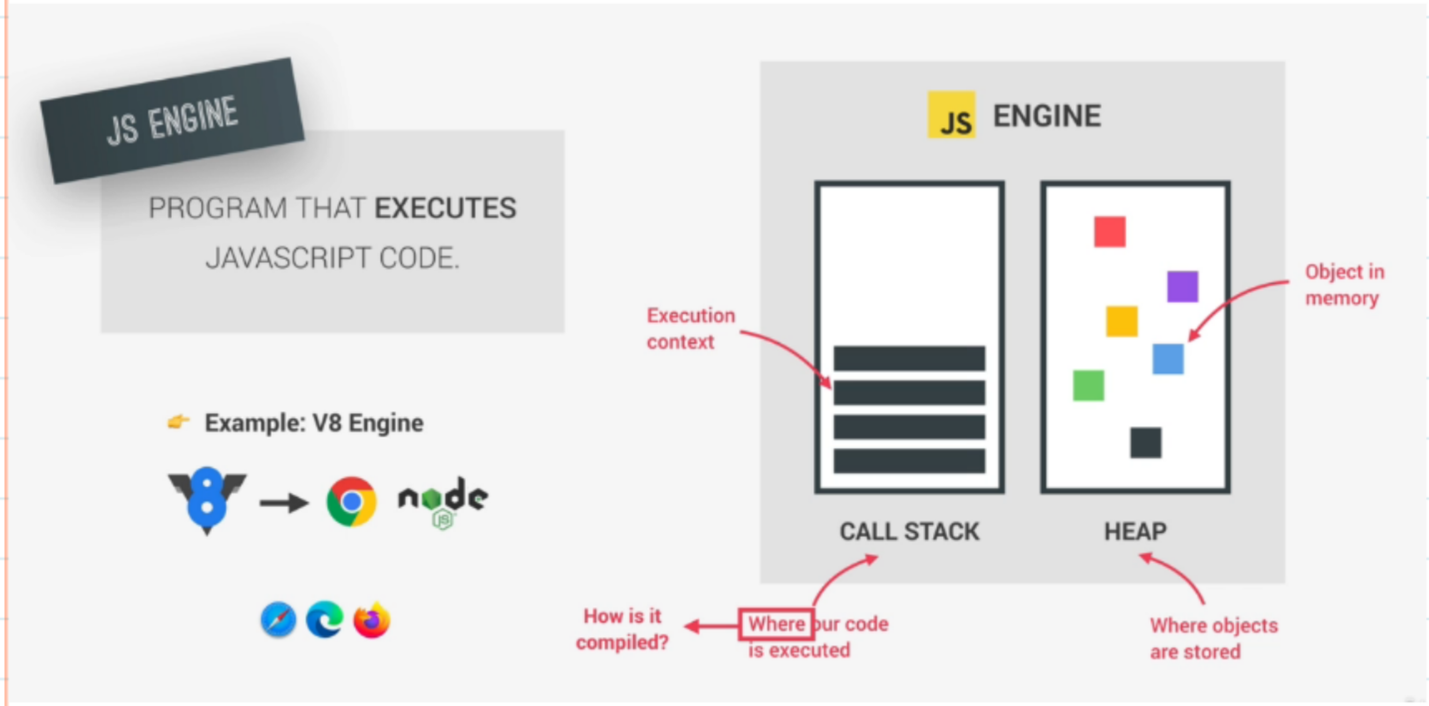
AST = Apstract Syntax tree





at first compilation done with no optimization and when execution starts, Optimized compilation done again and again and make the execution fast

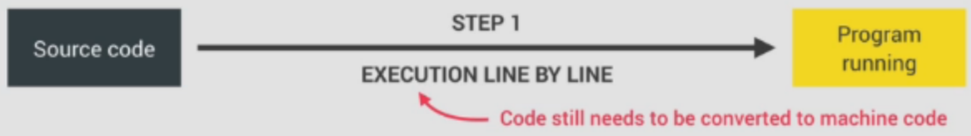
(here optimize compilation is done when program is in execution)



COMPUTER SCIENCE SIDENOTE: COMPILATION VS. INTERPRETATION

Compilation: Entire code is converted into machine code at once, and written to a binary file that can be executed by a computer.

Interpretation: Interpreter runs through the source code and executes it line by line.

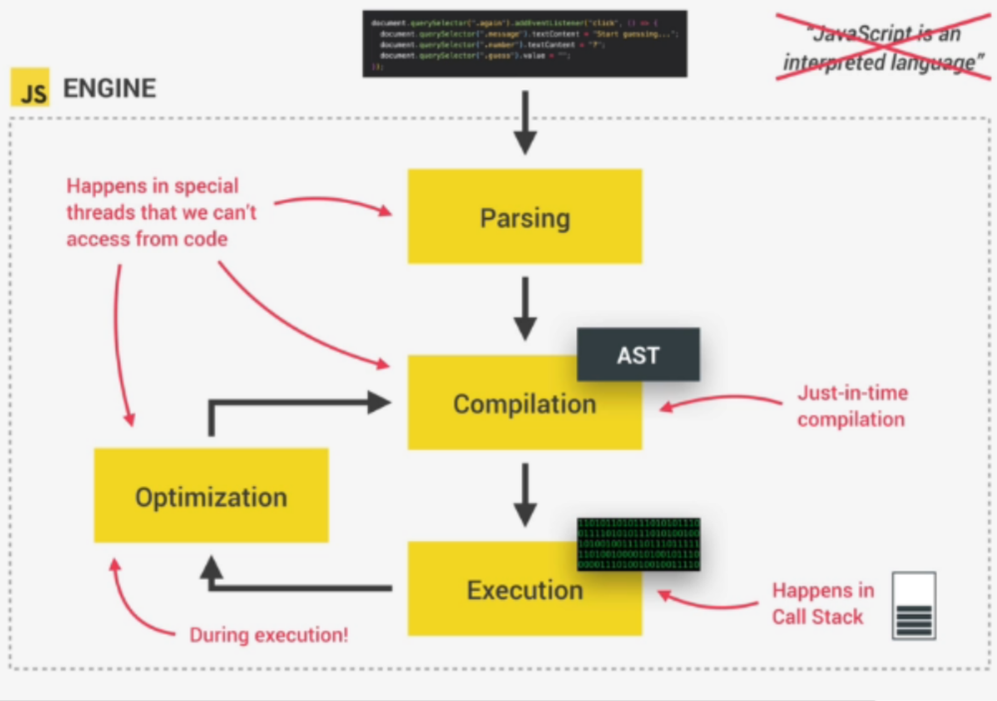


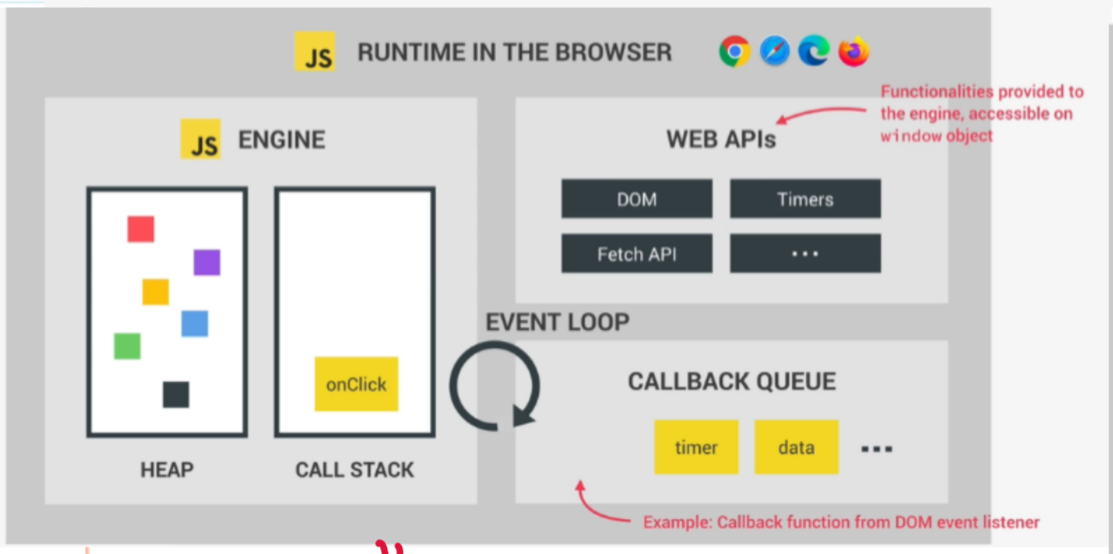
Just-in-time (JIT) compilation: Entire code is converted into machine code at once, then executed immediately.

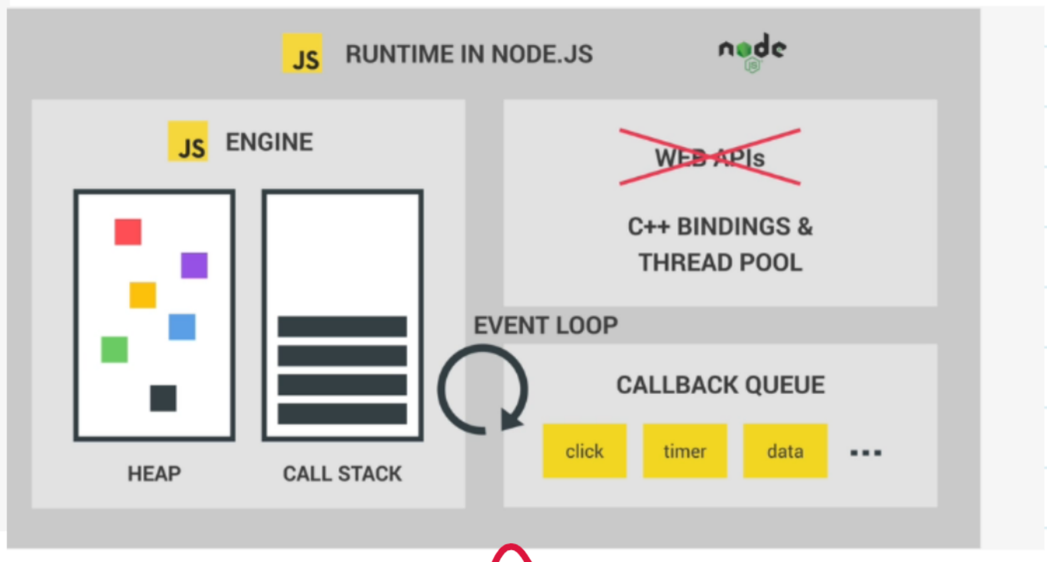
JIT vs Non-JIT comparison:

- In JIT not all the code is converted into machine code first a part of the code that is necessary will be converted into machine code then if a method or functionality called is not in machine then that will be turned into machine code... it reduces burden on the CPU.
- As the machine code will be generated on run time....the JIT compiler will produce machine code that is optimised for running machine's CPU architecture.

MODERN JUST-IN-TIME COMPILATION OF JAVASCRIPT







"event loop" take function from callback queue and put it in call stack

In node it

- **Function declaration**

  Function that can be used before it's declared

- **Function expression**

  Essentially a function *value* stored in a variable

- **Arrow function**

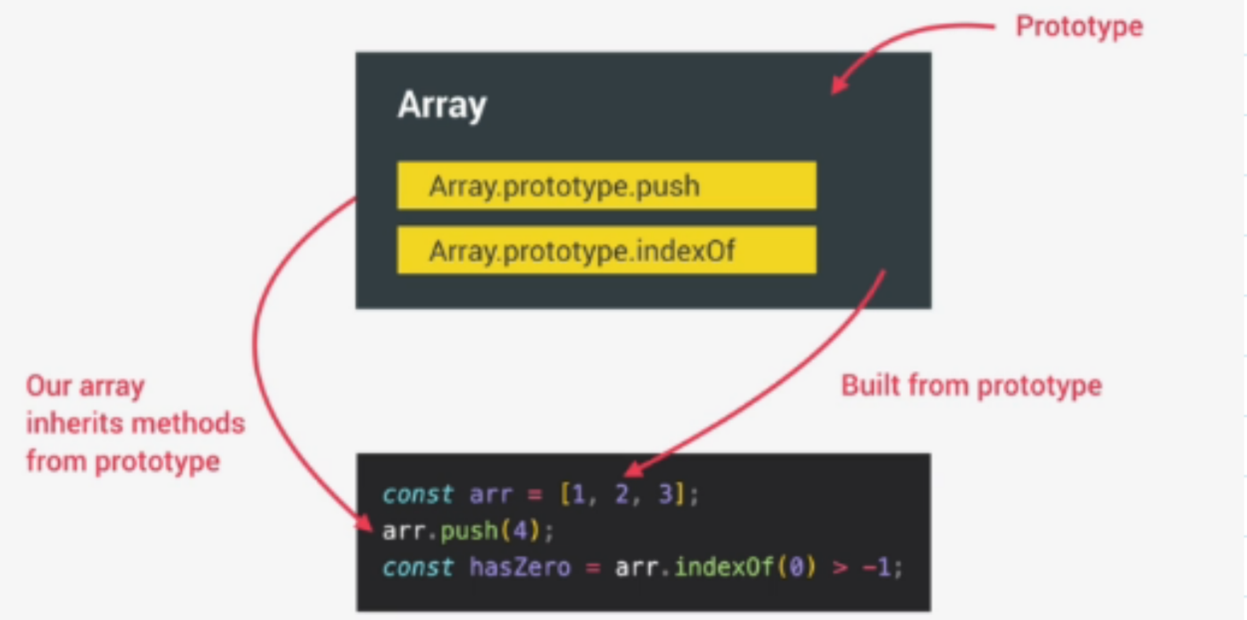
  Great for a quick one-line functions. Has no `this` keyword (more later...)

```
function calcAge(birthYear) {
  return 2037 - birthYear;
}

const calcAge = function (birthYear) {
  return 2037 - birthYear;
};

const calcAge = birthYear => 2037 - birthYear;
```

JAVASCRIPT

JAVASCRIPT IS A HIGH-LEVEL, PROTOTYPE-BASED OBJECT-ORIENTED, MULTI-PARADIGM, INTERPRETED OR JUST-IN-TIME COMPILED, DYNAMIC, SINGLE-THREADED, GARBAGE-COLLECTED PROGRAMMING LANGUAGE WITH FIRST-CLASS FUNCTIONS AND A NON-BLOCKING EVENT LOOP CONCURRENCY MODEL. 🤔 🤯 🤣

Almost everything in JavaScript is an object. In fact, only six things are not objects. They are — null , undefined , strings, numbers, boolean, and symbols. These are called primitive values or primitive types. 02-Jan-2019



- In a language with **first-class functions**, functions are simply **treated as variables**. We can pass them into other functions, and return them from functions.

```
const closeModal = () => {
  modal.classList.add("hidden");
  overlay.classList.add("hidden");
};

overlay.addEventListener("click", closeModal);
```

Passing a function into another function as an argument: First-class functions!

- **Dynamically-typed language:**

No data type definitions. Types becomes known at runtime

Data type of variable is automatically changed

```
let x = 23;
let y = 19;
x = "Jonas";
```

DECONSTRUCTING THE MONSTER DEFINITION

- High-level
- Garbage-collected
- Interpreted or just-in-time compiled
- Multi-paradigm
- Prototype-based object-oriented
- First-class functions
- Dynamic
- Single-threaded
- Non-blocking event loop

- **Concurrency model**: how the JavaScript engine handles multiple tasks happening at the same time.

  ↓ Why do we need that?

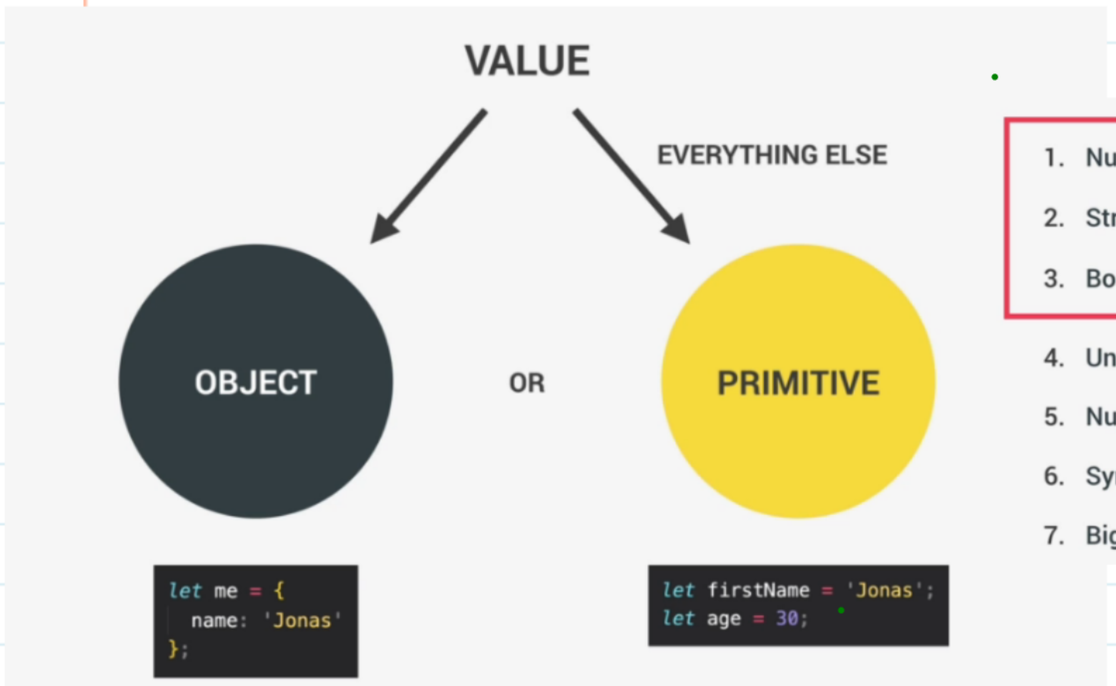
- JavaScript runs in one **single thread**, so it can only do one thing at a time.

  ↓ So what about a long-running task?

- Sounds like it would block the single thread. However, we want non-blocking behavior!

  ↓ How do we achieve that?

- By using an **event loop**: takes long running tasks, executes them in the "background", and puts them back in the main thread once they are finished.



1. **Number:** Floating point numbers 👉 Used for decimals and integers `let age = 23;`
2. **String:** Sequence of characters 👉 Used for text `let firstName = 'Jonas';`
3. **Boolean:** Logical type that can only be `true` or `false` 👉 Used for taking decisions `let fullAge = true;`
4. **Undefined:** Value taken by a variable that is not yet defined ('empty value') `let children;`
5. **Null:** Also means 'empty value'
6. **Symbol (ES2015):** Value that is unique and cannot be changed *[Not useful for now]*
7. **BigInt (ES2020):** Larger integers than the Number type can hold

☝ **JavaScript has dynamic typing:** We do ***not*** have to manually define the data type of the value stored in a variable. Instead, data types are determined **automatically**.

keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES

Is Babel a compiler or transpiler?

Babel enables developers to use cutting-edge Javascript without worrying about browser support. **Babel is a JavaScript transpiler**, meaning it converts a newer version of ECMAScript, such as ES9, to a standard version (ES5).

Babel is **a transpiler i.e. it converts the JSX to vanilla JavaScript**. You can view babel as an intermediate step between your code and "executable" code. React also uses ES6, which is not supported by most of the browsers. Babel converts the ES6 code to a code which is compatible with the browsers. 12-Aug-2016

In brief, Webpack goes through your package and creates what it calls a **dependency graph** which consists of various **modules** which your webapp would require to function as expected. Then, depending on this graph, it creates a new package which consists of the very bare minimum number of files required, often just a single bundle.js file which can be plugged in to the html file easily and used for the application.

Over the next part part of this article I will take you through the step by step setup of webpack. By the end of it, I hope you understand the basics of Webpack. So lets get this rolling...