JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ADVANCED DOM AND EVENTS

LECTURE

EFFICIENT SCRIPT LOADING: DEFER AND ASYNC

JS

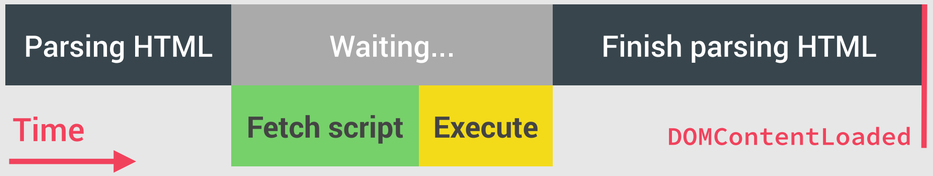
DEFER AND ASYNC SCRIPT LOADING

HEAD

BODY END

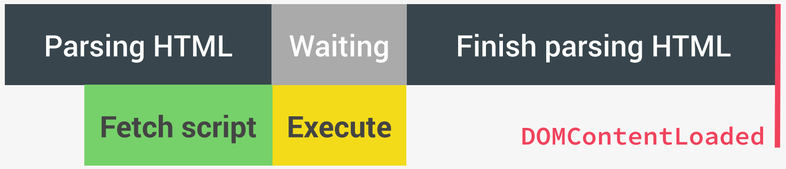
REGULAR

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



ASYNC

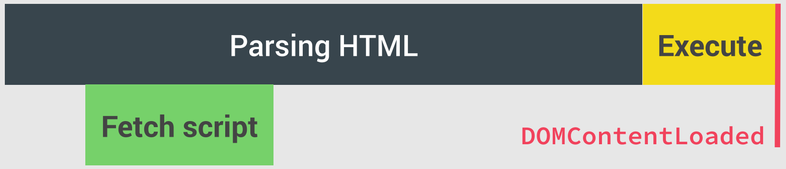
```
<script async src="script.js">
```



👉 Makes no sense 🤷

DEFER

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



👉 Makes no sense 🤷

REGULAR VS. ASYNC VS. DEFER

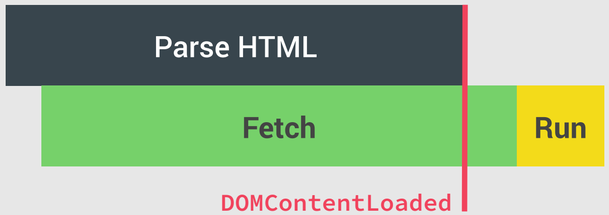
END OF BODY

- 👉 Scripts are fetched and executed ***after the HTML is completely parsed***
- 👉 **Use if you need to support old browsers**

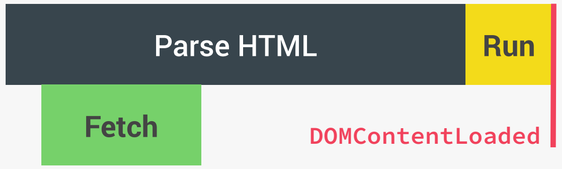
You can, of course, use **different strategies for different scripts**. Usually a complete web applications includes more than just one script

ASYNC IN HEAD

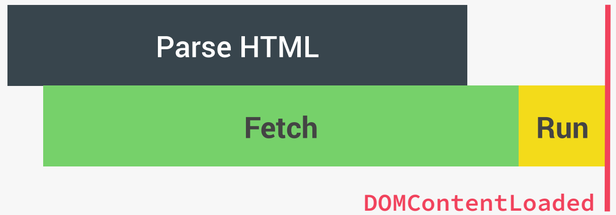
- 👉 Scripts are fetched ***asynchronously*** and executed ***immediately***
- 👉 Usually the `DOMContentLoaded` event waits for ***all*** scripts to execute, except for `async` scripts. So, `DOMContentLoaded` does ***not*** wait for an `async` script
- 👉 Scripts ***not*** guaranteed to execute in order
- 👉 **Use for 3rd-party scripts where order doesn't matter (e.g. Google Analytics)**



DEFER IN HEAD



- 👉 Scripts are fetched ***asynchronously*** and executed ***after the HTML is completely parsed***
- 👉 `DOMContentLoaded` event fires ***after*** `defer` script is executed
- 👉 Scripts are executed ***in order***
- 👉 **This is overall the best solution! Use for your own scripts, and when order matters (e.g. including a library)**



OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

WHAT IS OBJECT-ORIENTED PROGRAMMING?

JS

WHAT IS OBJECT-ORIENTED PROGRAMMING? (OOP)

OOP

Style of code, “how” we write and organize code

- Object-oriented programming (OOP) is a programming paradigm based on the concept of objects;
- We use objects to **model** (describe) real-world or abstract features;
- Objects may contain data (properties) and code (methods). By using objects, we pack **data and the corresponding behavior** into one block;
- In OOP, objects are **self-contained** pieces/blocks of code;
- Objects are **building blocks** of applications, and **interact** with one another;
- Interactions happen through a **public interface** (API): methods that the code **outside** of the object can access and use to communicate with the object;
- OOP was developed with the goal of **organizing** code, to make it **more flexible and easier to maintain** (avoid “spaghetti code”).

E.g. user or todo list item

E.g. HTML component or data structure

Data

```
const user = {
  user: 'jonas',
  password: 'dk23s',

  login(password) {
    // Login logic
  },
  sendMessage(str) {
    // Sending logic
  }
}
```

Behaviour

CLASSES AND INSTANCES (TRADITIONAL OOP)

Like a blueprint from which we can create new objects

CLASS

```
User {
  user
  password
  email

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

Just a representation, NOT actual JavaScript syntax!

JavaScript does NOT support *real* classes like represented here

new User('jonas')

new User('mary')

new User('steven')

Instance

```
{
  user = 'jonas'
  password = 'dk23s'
  email = 'hello@jonas.io'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

New object created from the class. Like a *real* house created from an *abstract* blueprint

Instance

```
{
  user = 'mary'
  password = 'qwerty23'
  email = 'mary@test.com'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

Instance

```
{
  user = 'steven'
  password = '5p8dz32dd'
  email = 'steven@tes.co'

  login(password) {
    // Login logic
  }
  sendMessage(str) {
    // Sending logic
  }
}
```

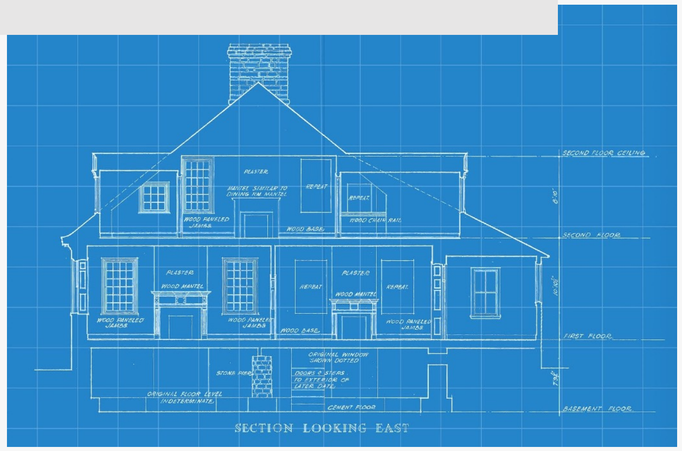
☝ Conceptual overview: it works a bit **differently** in JavaScript. Still important to understand!

THE 4 FUNDAMENTAL OOP PRINCIPLES

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

The 4 fundamental principles of Object-Oriented Programming

🤔 *"How do we actually design classes? How do we model real-world data into classes?"*



PRINCIPLE 1: ABSTRACTION

- **Abstraction**
- Encapsulation
- Inheritance
- Polymorphism

```
Phone {
  charge
  volume
  voltage
  temperature

  homeBtn() {}
  volumeBtn() {}
  screen() {}
  verifyVolt() {}
  verifyTemp() {}
  vibrate() {}
  soundSpeaker() {}
  soundEar() {}
  frontCamOn() {}
  frontCamOff() {}
  rearCamOn() {}
  rearCamOff() {}
}
```

Real phone

Abstracted phone

```
Phone {
  charge
  volume

  homeBtn() {}
  volumeBtn() {}
  screen() {}
}
```

Details have been abstracted away

Do we *really need* all these low-level details?

👉 **Abstraction:** Ignoring or hiding details that **don't matter**, allowing us to get an **overview** perspective of the *thing* we're implementing, instead of messing with details that don't really matter to our implementation.

PRINCIPLE 2: ENCAPSULATION

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

NOT accessible from **outside the class!**

STILL accessible from **within the class!**

STILL accessible from **within the class!**

NOT accessible from **outside the class!**

```
User {
  user
  private password
  private email

  login(word) {
    this.password === word
  }
  comment(text) {
    this.checkSPAM(text)
  }
  private checkSPAM(text) {
    // Verify logic
  }
}
```

Again, **NOT** actually JavaScript syntax (the `private` keyword doesn't exist)

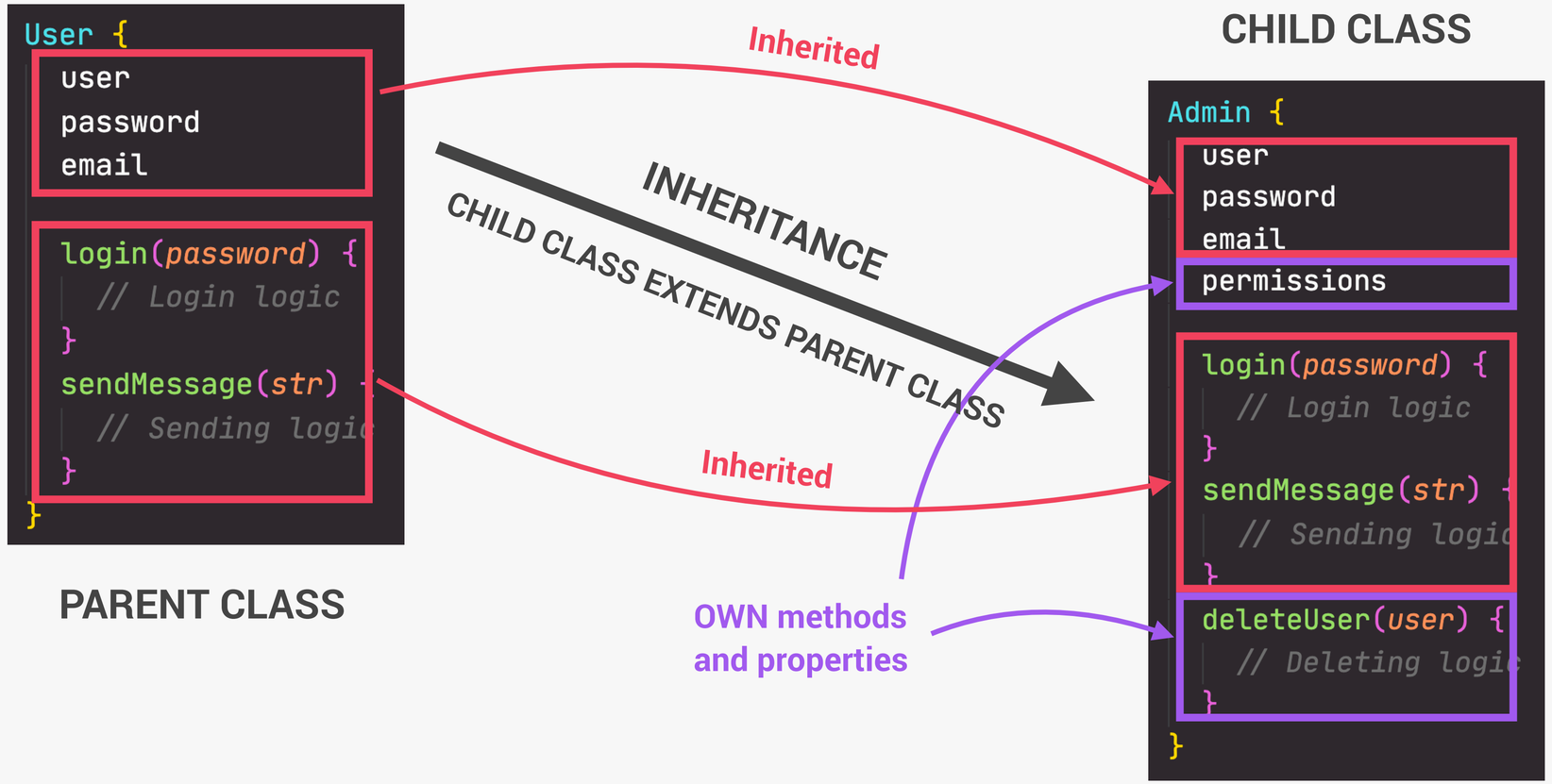
WHY?

- 👉 Prevents external code from accidentally manipulating internal properties/state
- 👉 Allows to change internal implementation without the risk of breaking external code

👉 **Encapsulation:** Keeping properties and methods **private** inside the class, so they are **not accessible from outside the class**. Some methods can be **exposed** as a public interface (API).

PRINCIPLE 3: INHERITANCE

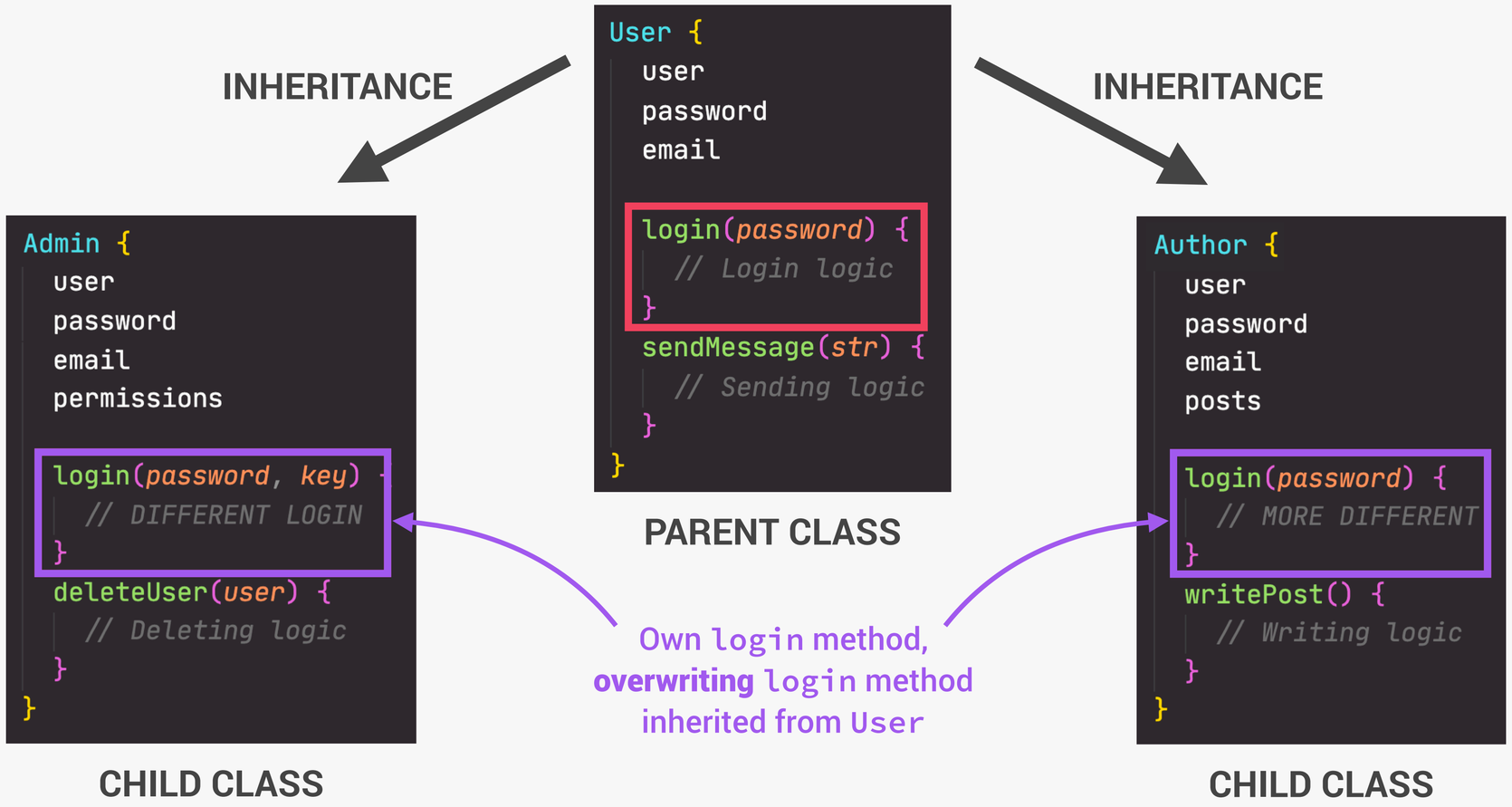
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



👉 **Inheritance:** Making all properties and methods of a certain class **available to a child class**, forming a hierarchical relationship between classes. This allows us to **reuse common logic** and to model real-world relationships.

PRINCIPLE 4: POLYMORPHISM

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



👉 **Polymorphism:** A child class can **overwrite** a method it inherited from a parent class [it's more complex that that, but enough for our purposes].

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

OOP IN JAVASCRIPT

OOP IN JAVASCRIPT: PROTOTYPES

"CLASSICAL OOP": CLASSES

Class

INSTANTIATION

Instance

- Objects (instances) are **instantiated** from a class, which functions like a blueprint;
- Behavior (methods) is **copied** from class to all instances.

OOP IN JS: PROTOTYPES

Prototype ← Contains methods

PROTOTYPAL INHERITANCE/ DELEGATION

Object ← Can access methods

- 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.

👉 **Example: Array**

```
const num = [1, 2, 3];
num.map(v => v * 2);
```

MDN web docs
moz://a

```
Array.prototype.keys()
Array.prototype.lastIndexOf()
Array.prototype.map()
```

`Array.prototype` is the **prototype** of all array objects we create in JavaScript

Therefore, **all** arrays have access to the `map` method!

```
ƒ Array()
  arguments: (...)
  caller: (...)
  length: 1
  name: "Array"
  prototype: Array(0)
    unique: ƒ ()
    length: 0
    constructor: ƒ Array()
    concat: ƒ concat()
    map: ƒ map()
```

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?”*

☝️ **The 4 pillars of OOP are still valid!**

- 👉 Abstraction
- 👉 Encapsulation
- 👉 Inheritance
- 👉 Polymorphism

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()`

- 👉 The easiest and most straightforward way of linking an object to a prototype object.

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

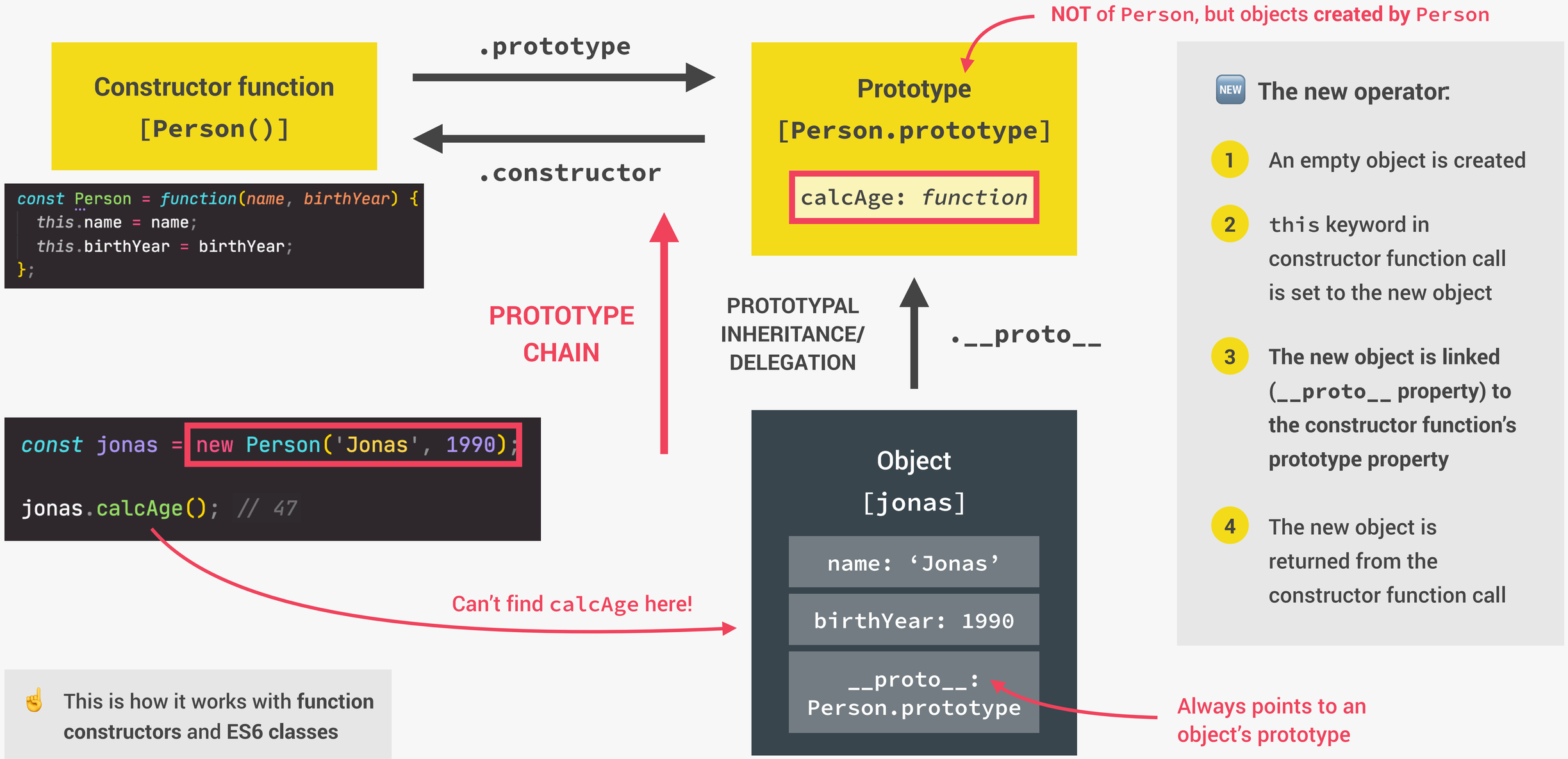
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

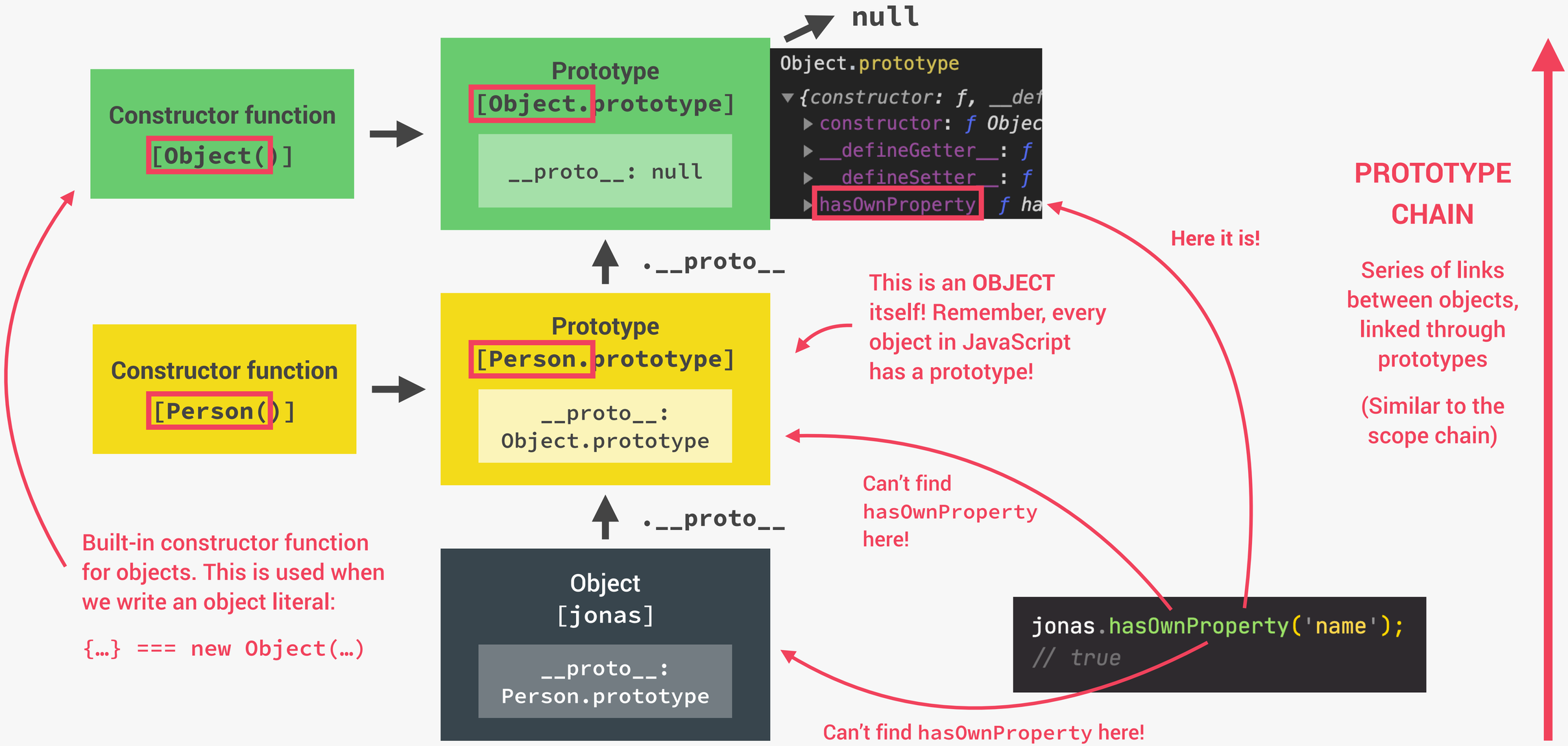
PROTOTYPAL INHERITANCE AND THE PROTOTYPE CHAIN

JS

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS



THE PROTOTYPE CHAIN



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

OBJECT.CREATE

HOW OBJECT.CREATE WORKS

Prototype
[PersonProto]

calcAge: *function*

```
const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },
};
```

.__proto__

```
const steven = Object.create(PersonProto);
```

Object
[steven]

name: 'Steven'

birthYear: 2002

__proto__: PersonProto

OBJECT.CREATE

CONSTRUCTOR FUNCTIONS

~~Constructor function~~
~~[Person()]~~

~~.prototype~~

Prototype
[Person.prototype]

calcAge: *function*

```
const jonas = new Person('Jonas', 1990);
```

AUTOMATIC
.__proto__

Object
[jonas]

name: 'Jonas'

birthYear: 1990

__proto__: Person.prototype

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

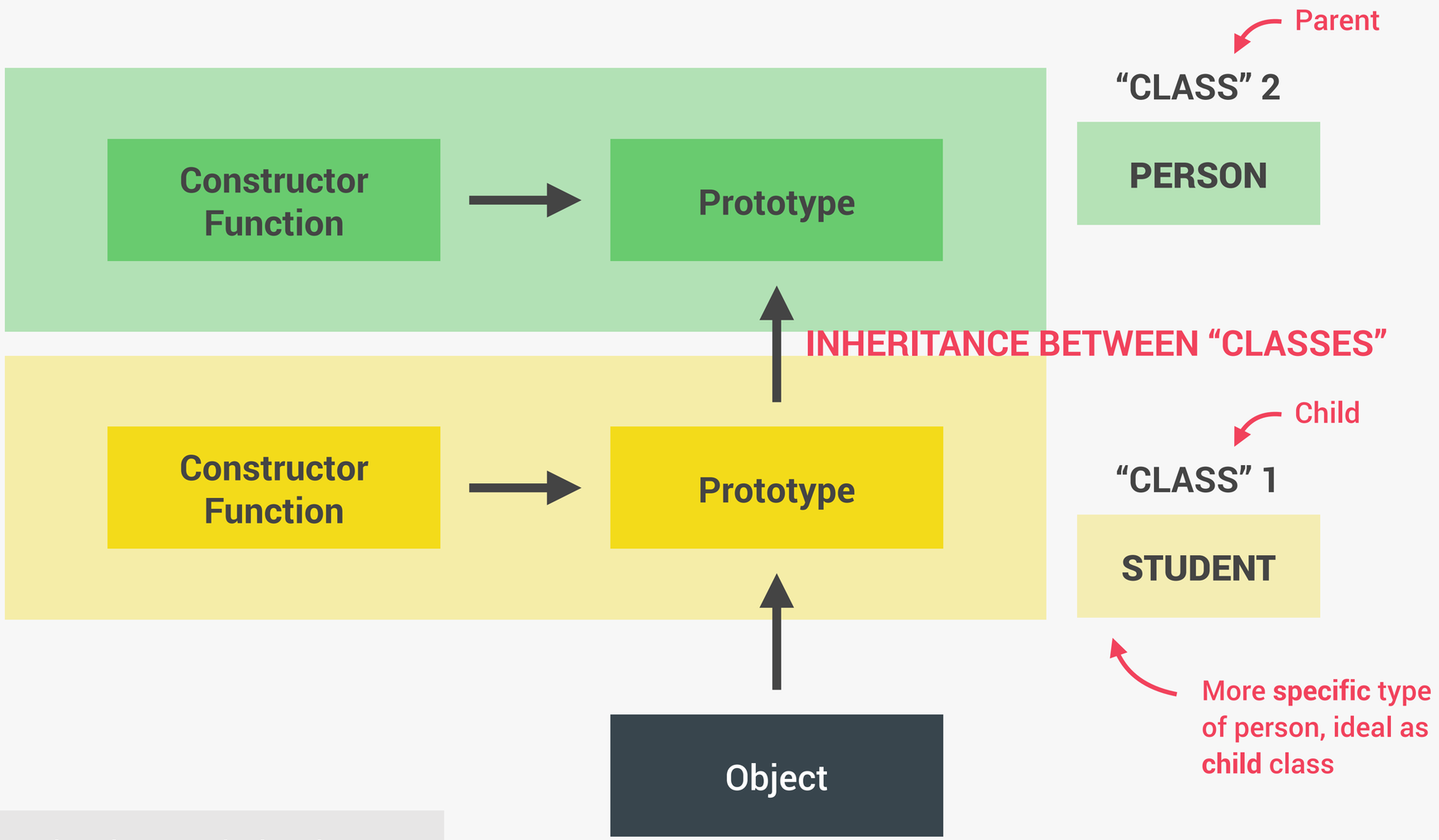
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES": CONSTRUCTOR FUNCTIONS

JS

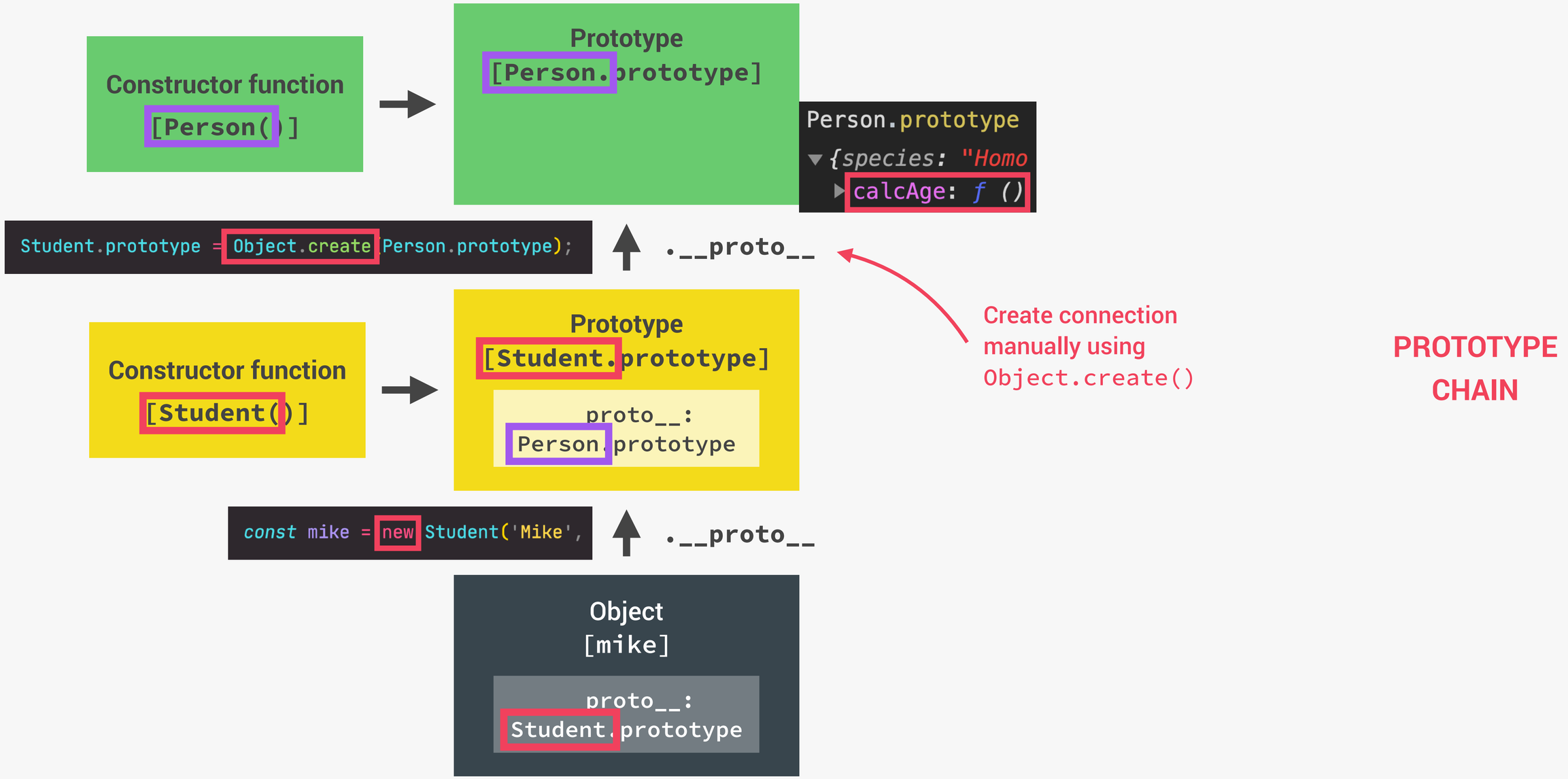
INHERITANCE BETWEEN "CLASSES"



1. Constructor functions
2. ES6 Classes
3. `Object.create()`

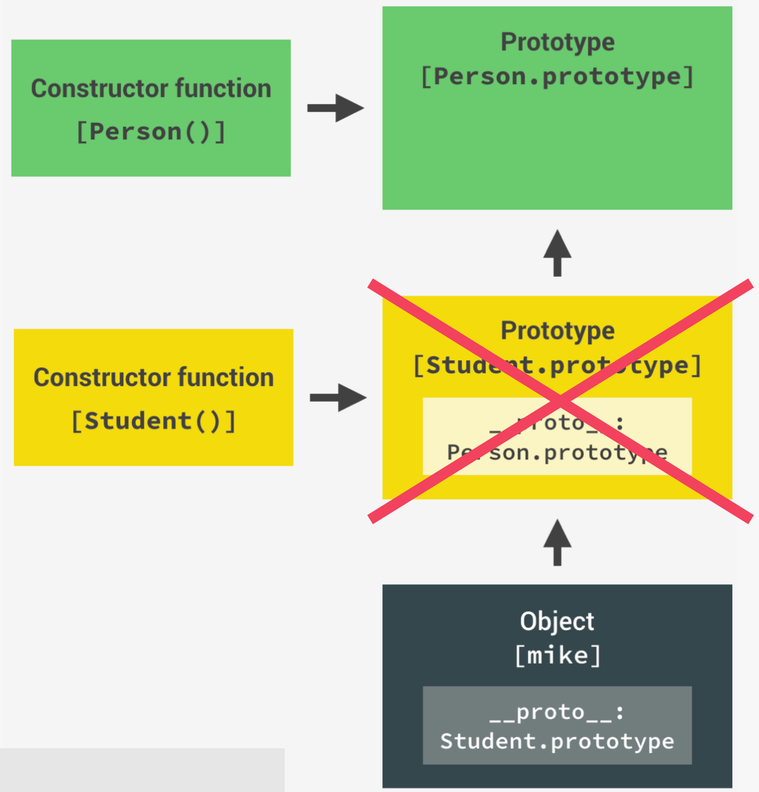
☝ Using class terminology here to make it easier to understand.

INHERITANCE BETWEEN "CLASSES"



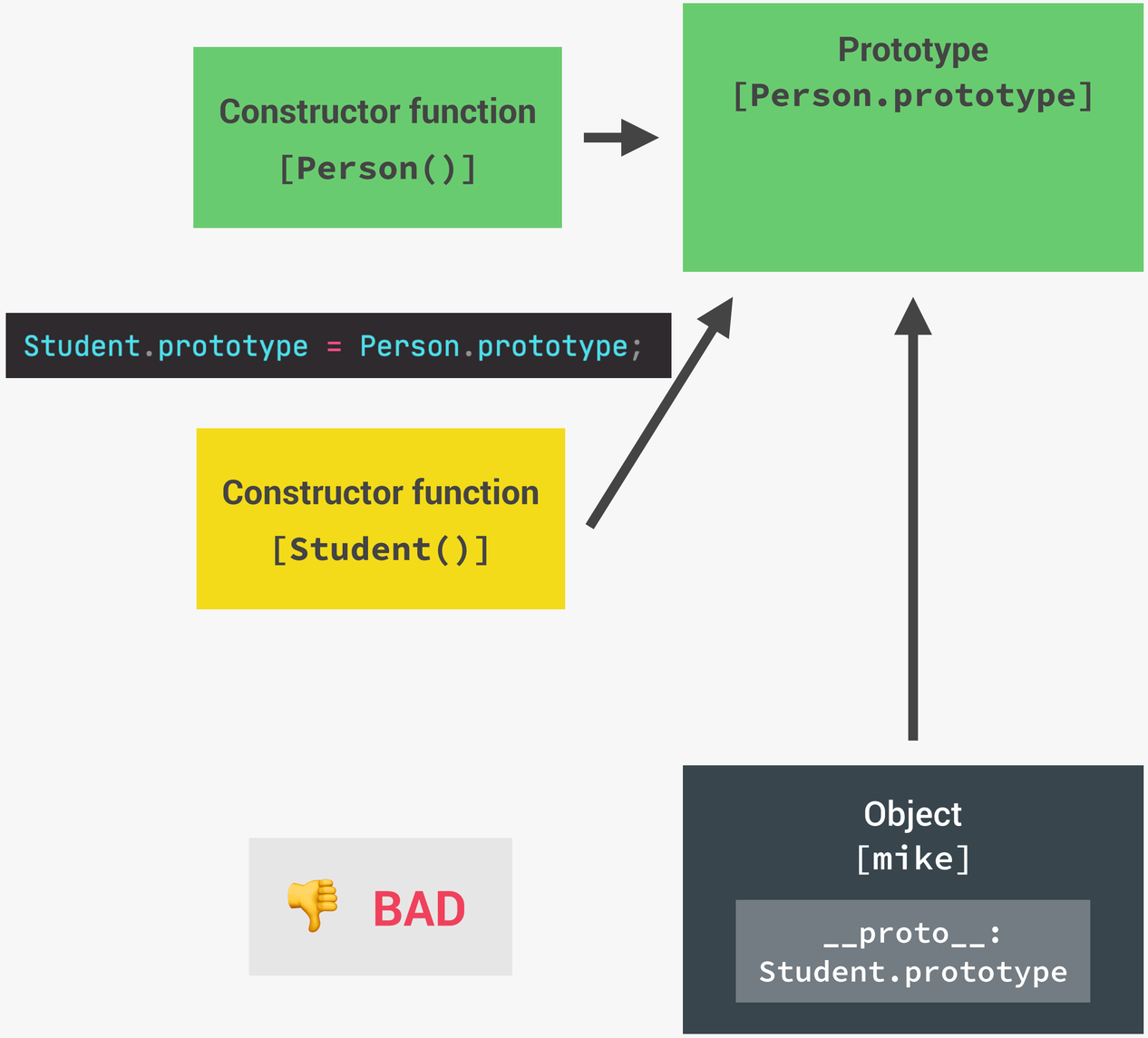
INHERITANCE BETWEEN "CLASSES"

```
Student.prototype = Object.create(Person.prototype);
```



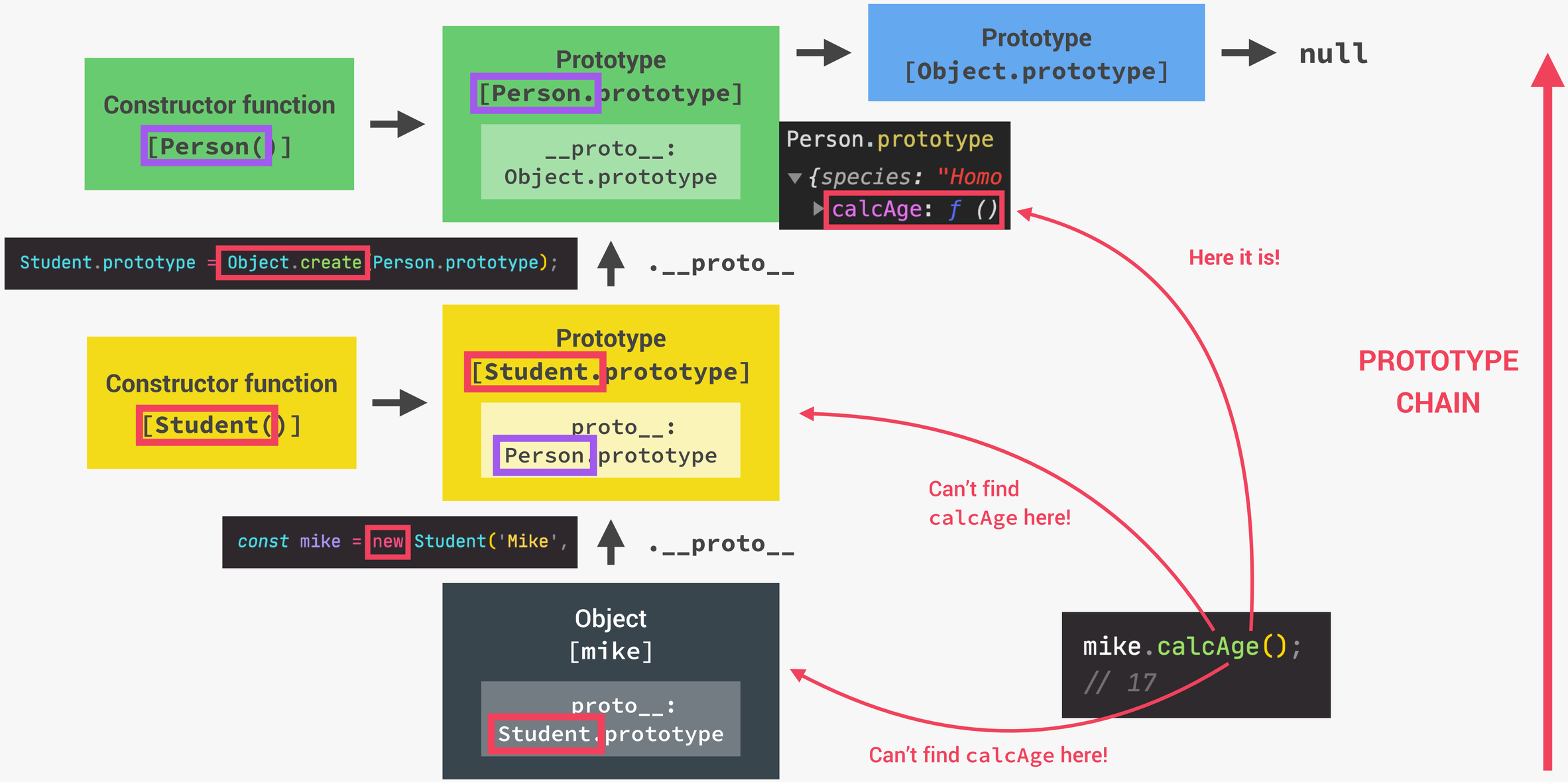
👍 GOOD

```
Student.prototype = Person.prototype;
```



👎 BAD

INHERITANCE BETWEEN "CLASSES"



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

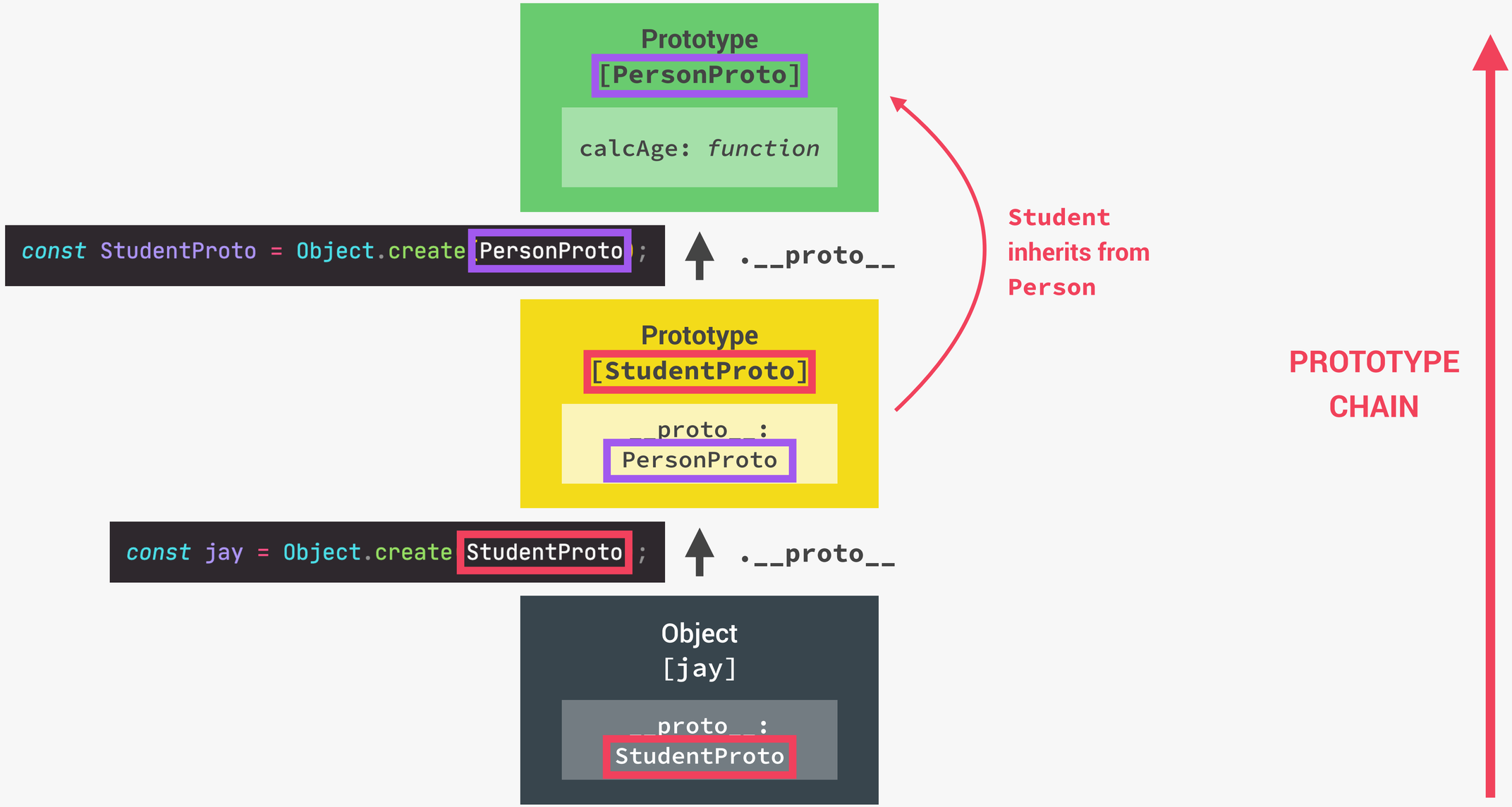
OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE

JS

INHERITANCE BETWEEN "CLASSES": OBJECT.CREATE



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

OBJECT ORIENTED PROGRAMMING (OOP) WITH JAVASCRIPT

LECTURE

ES6 CLASSES SUMMARY

JS


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

MAPTY APP: OOP, GEOLOCATION, EXTERNAL LIBRARIES, AND MORE!

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

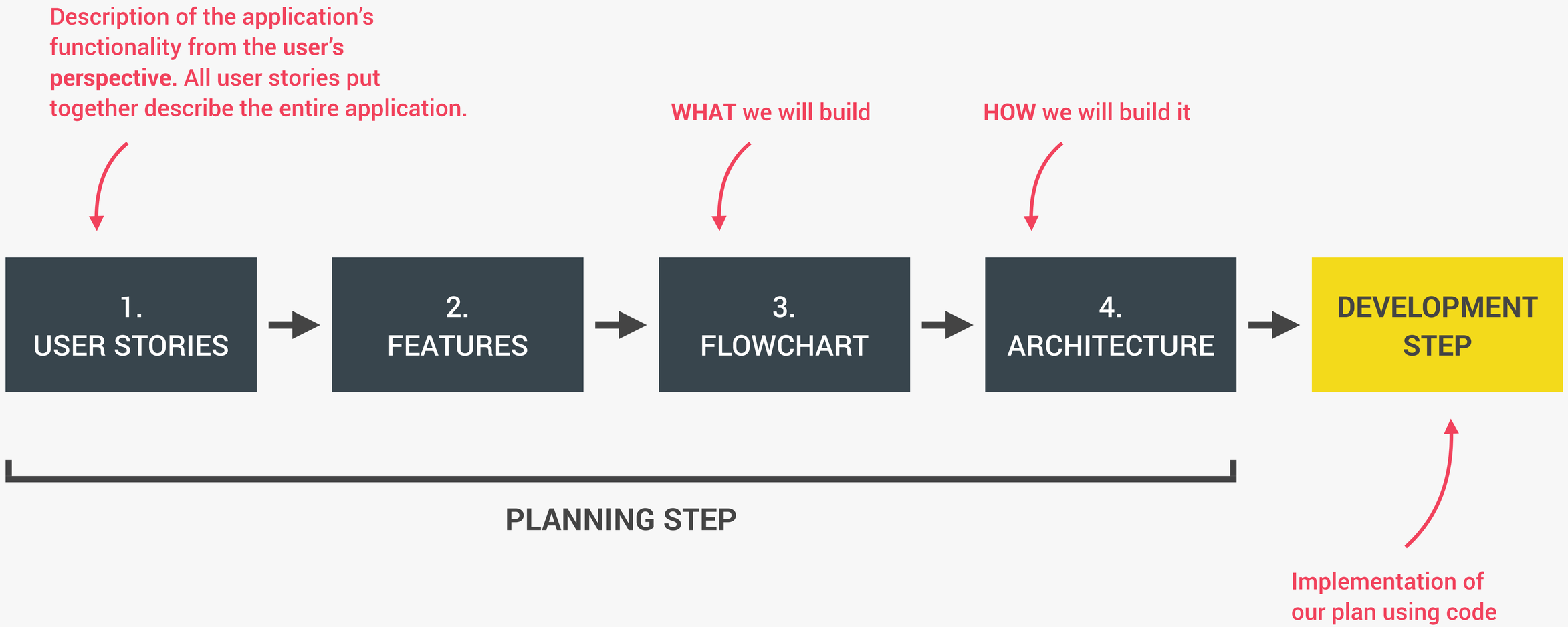
MAPTY APP: OOP, GEOLOCATION, EXTERNAL LIBRARIES, AND MORE!

LECTURE

HOW TO PLAN A WEB PROJECT

JS

PROJECT PLANNING



1. USER STORIES

mapty

👉 **User story:** Description of the application's functionality from the user's perspective.

👉 **Common format:** As a *[type of user]*, I want *[an action]* so that *[a benefit]*

Who? Example: user, admin, etc.

What?

Why?

1. As a user, I want to **log my running workouts with location, distance, time, pace and steps/minute**, so I can keep a log of all my running

2. As a user, I want to **log my cycling workouts with location, distance, time, speed and elevation gain**, so I can keep a log of all my cycling

3. As a user, I want to **see all my workouts at a glance**, so I can easily track my progress over time

4. As a user, I want to **also see my workouts on a map**, so I can easily check where I work out the most

5. As a user, I want to **see all my workouts when I leave the app and come back later**, so that I can keep using there app over time

2. FEATURES

mapty

USER STORIES	FEATURES
1 Log my **running** workouts with location, distance, time, pace and steps/minute	👉 Map where user clicks to add new workout (best way to get location coordinates) 👉 Geolocation to display map at current location (more user friendly) 👉 Form to input distance, time, pace, steps/minute
2 Log my **cycling** workouts with location, distance, time, speed and elevation gain	👉 Form to input distance, time, speed, elevation gain
3 See all my workouts at a glance	👉 Display all workouts in a list
4 See my workouts on a map	👉 Display all workouts on the map
5 See all my workouts when I leave the app and come back later	👉 Store workout data in the browser using local storage API 👉 On page load, read the saved data from local storage and display

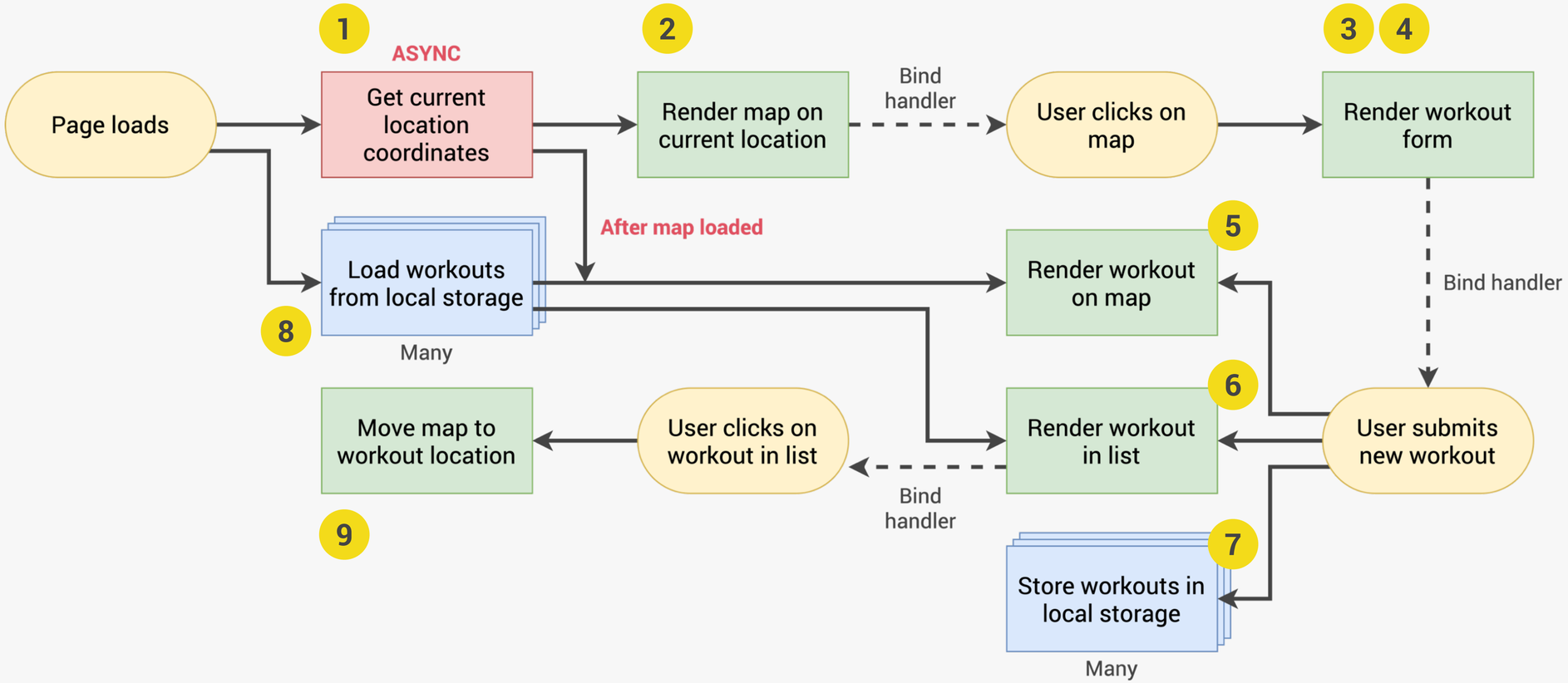
3. FLOWCHART

mapty

FEATURES

1. Geolocation to display map at current location
2. Map where user clicks to add new workout
3. Form to input distance, time, pace, steps/minute
4. Form to input distance, time, speed, elevation gain
5. Display workouts in a list
6. Display workouts on the map
7. Store workout data in the browser
8. On page load, read the saved data and display
9. Move map to workout location on click

Added later



☝ In the real-world, you don't have to come with the final flowchart right in the planning phase. It's normal that it changes throughout implementation!

mapty

FOR NOW, LET'S JUST START CODING 😍

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MAPTY APP: OOP, GEOLOCATION, EXTERNAL LIBRARIES, AND MORE!

LECTURE

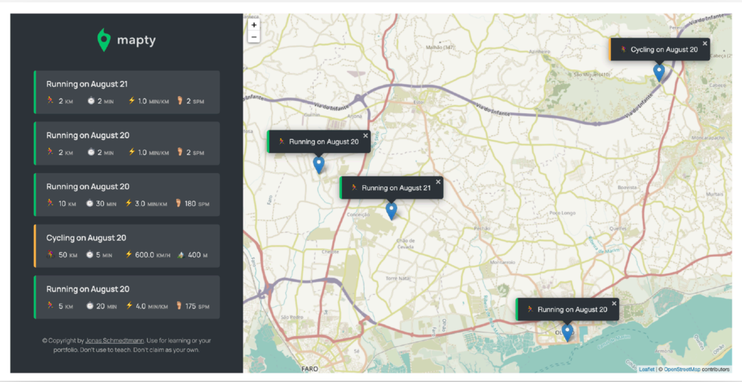
FINAL CONSIDERATIONS

JS

10 ADDITIONAL FEATURE IDEAS: CHALLENGES 🤓

mapty

- 👉 Ability to **edit** a workout;
- 👉 Ability to **delete** a workout;
- 👉 Ability to **delete all** workouts;
- 👉 Ability to **sort** workouts by a certain field (e.g. distance);
- 👉 **Re-build** `Running` and `Cycling` objects coming from Local Storage;
- 👉 More realistic error and confirmation **messages**;
- 👉 Ability to position the map to **show all workouts** [very hard];
- 👉 Ability to **draw lines and shapes** instead of just points [very hard];
- 👉 **Geocode location** from coordinates (“Run in Faro, Portugal”) [only after asynchronous JavaScript section];
- 👉 **Display weather** data for workout time and place [only after asynchronous JavaScript section].



ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

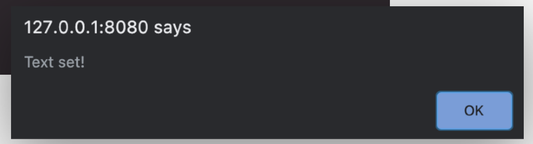
ASYNCHRONOUS JAVASCRIPT, AJAX AND APIS

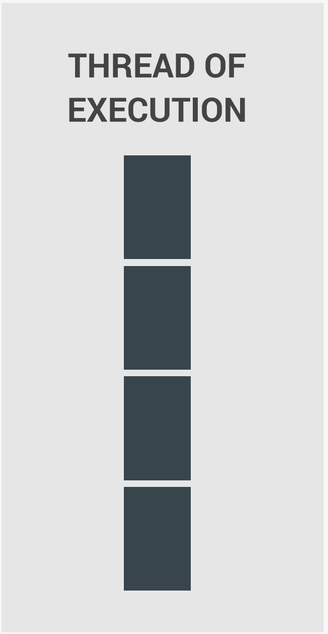
JS

SYNCHRONOUS CODE

BLOCKING

```
const p = document.querySelector('.p');
p.textContent = 'My name is Jonas!';
alert('Text set!');
p.style.color = 'red';
```





Part of execution context that actually executes the code in computer's CPU

SYNCHRONOUS

- 👉 Most code is **synchronous**;
- 👉 Synchronous code is **executed line by line**;
- 👉 Each line of code **waits** for previous line to finish;
- 👎 Long-running operations **block** code execution.

ASYNCHRONOUS CODE

Asynchronous

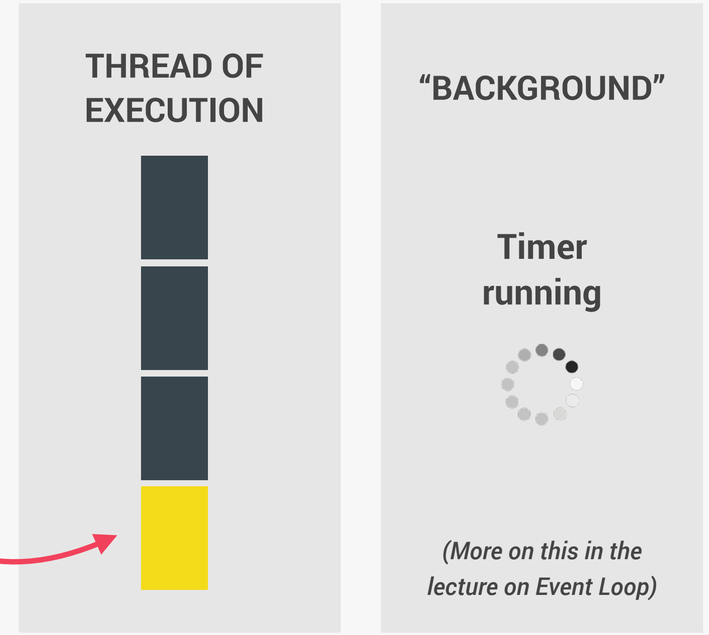
```
const p = document.querySelector('.p');
setTimeout(function () {
  p.textContent = 'My name is Jonas!';
}, 5000);
p.style.color = 'red';
```

CALLBACK WILL RUN AFTER TIMER

👉 **Example:** Timer with callback

```
[1, 2, 3].map(v => v * 2);
```

Callback does NOT automatically make code asynchronous!



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

- 👉 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 CODE

Asynchronous

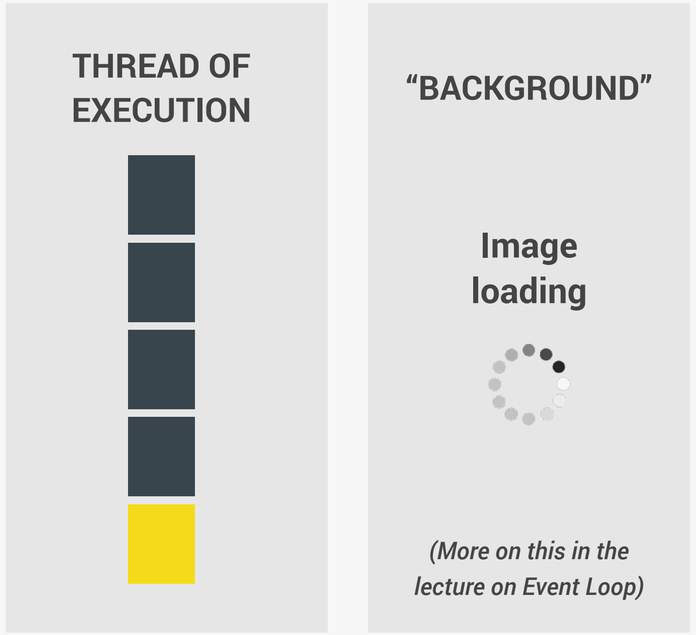
```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

CALLBACK WILL RUN AFTER IMAGE LOADS

`addEventListener` does NOT automatically make code asynchronous!

👉 **Example:** Asynchronous image loading with event and callback

👉 **Other examples:** Geolocation API or **AJAX calls**



ASYNCHRONOUS

Coordinating behavior of a program over a period of time

👉 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!

WHAT ARE AJAX CALLS?

AJAX

Asynchronous **J**avaScript **A**nd **X**ML: Allows us to communicate with remote web servers in an **asynchronous way**. With AJAX calls, we can **request data** from web servers dynamically.

WHAT IS AN API?

API

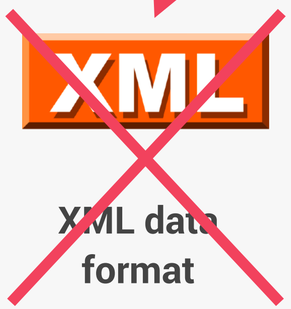
- **A**pplication **P**rogramming **I**nterface: Piece of software that can be used by another piece of software, in order to allow **applications to talk to each other**;
- There are be many types of APIs in web development:

  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.

There is an API for everything

AJAX



XML data format

JSON data format

Most popular API data format

- Weather data
- Data about countries
- Flights data
- Currency conversion data
- APIs for sending email or SMS
- Google Maps
- *Millions of possibilities...*

✈️ 🌍

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

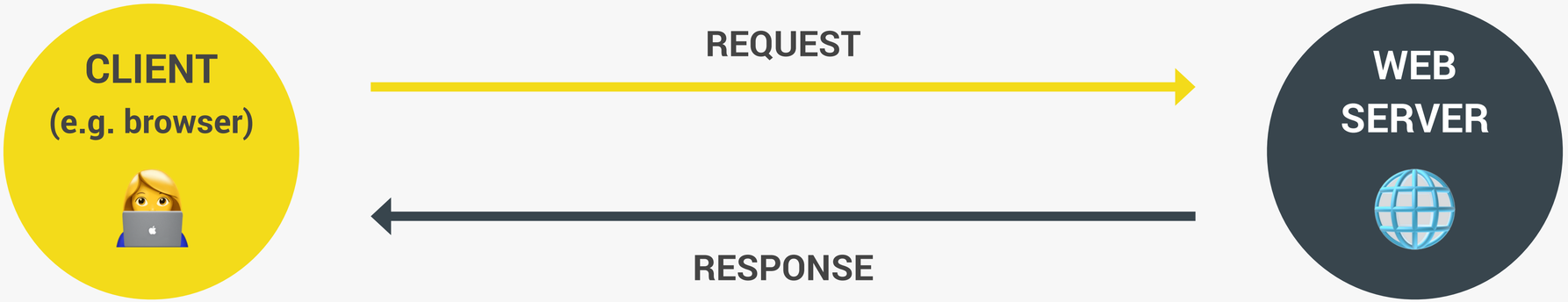
LECTURE

HOW THE WEB WORKS: REQUESTS AND RESPONSES

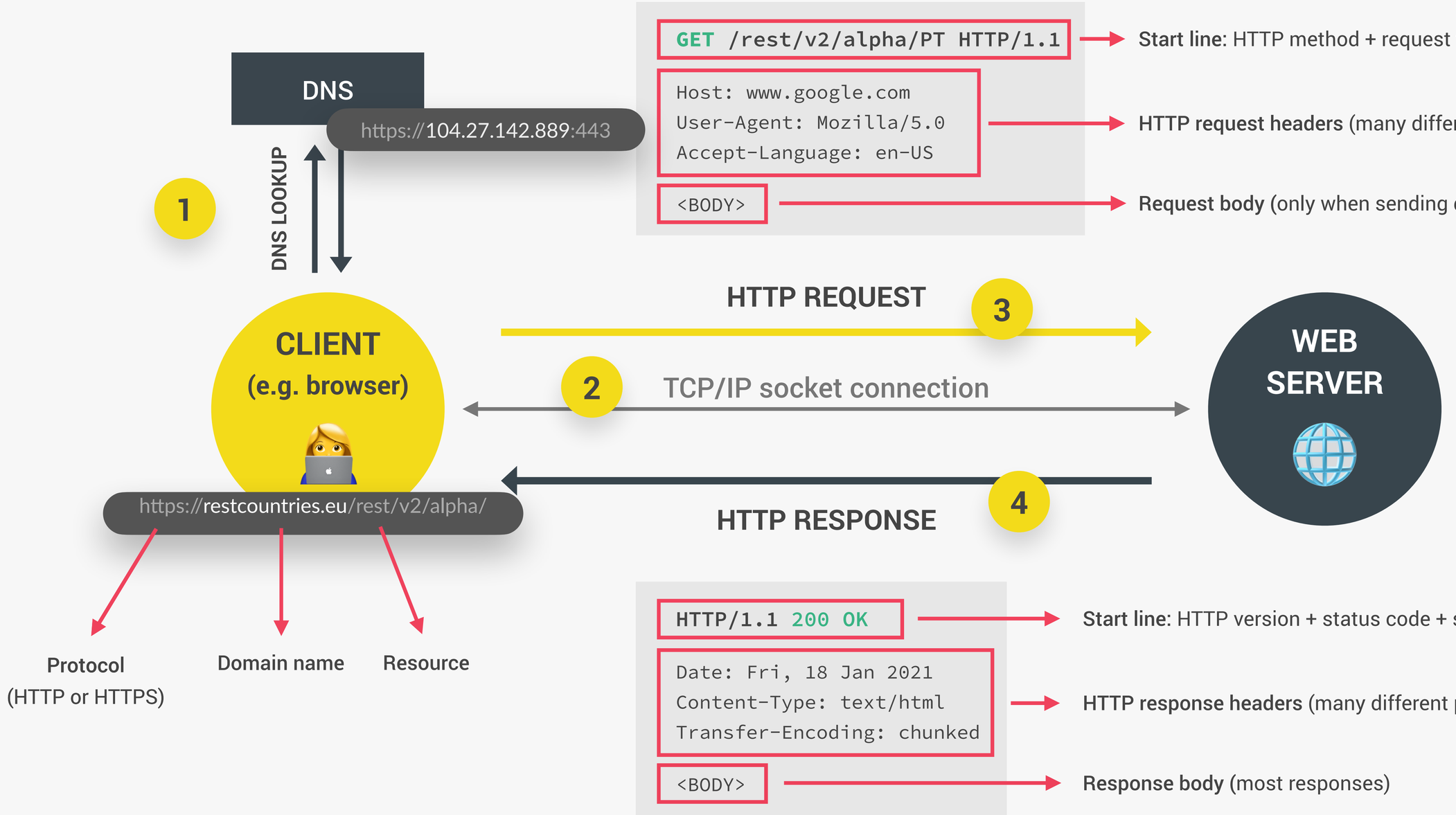
JS

WHAT HAPPENS WHEN WE ACCESS A WEB SERVER

👉 **Request-response model** or **Client-server architecture**



WHAT HAPPENS WHEN WE ACCESS A WEB SERVER



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

LECTURE

PROMISES AND THE FETCH API

JS

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.

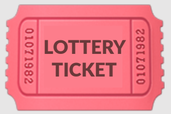
↓ Less formal

👉 **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

THE PROMISE LIFECYCLE

Before the future value is available

Asynchronous task **has finished**

PENDING

ASYNC TASK

SETTLED

FULFILLED

REJECTED

Success! The value is now **available**

An **error** happened

👉 We are able **handle** these different states in our code!

E.g. Fetch API returns promise

BUILD PROMISE

CONSUME PROMISE

When we already have a promise. E.g. promise returned from Fetch API

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

ASYNCHRONOUS JAVASCRIPT: PROMISES, ASYNC/AWAIT AND AJAX

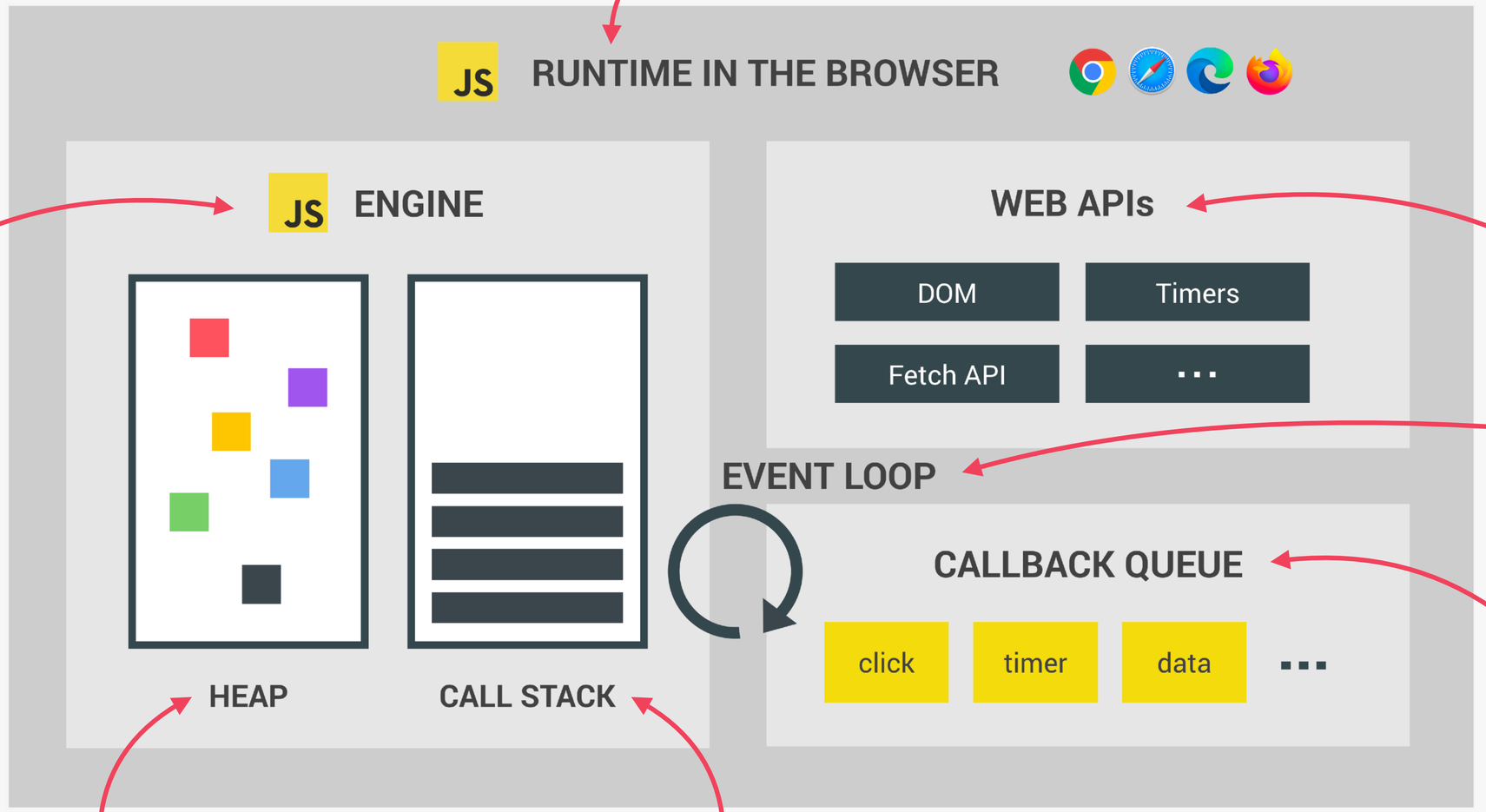
LECTURE

ASYNCHRONOUS BEHIND THE SCENES: THE EVENT LOOP

JS

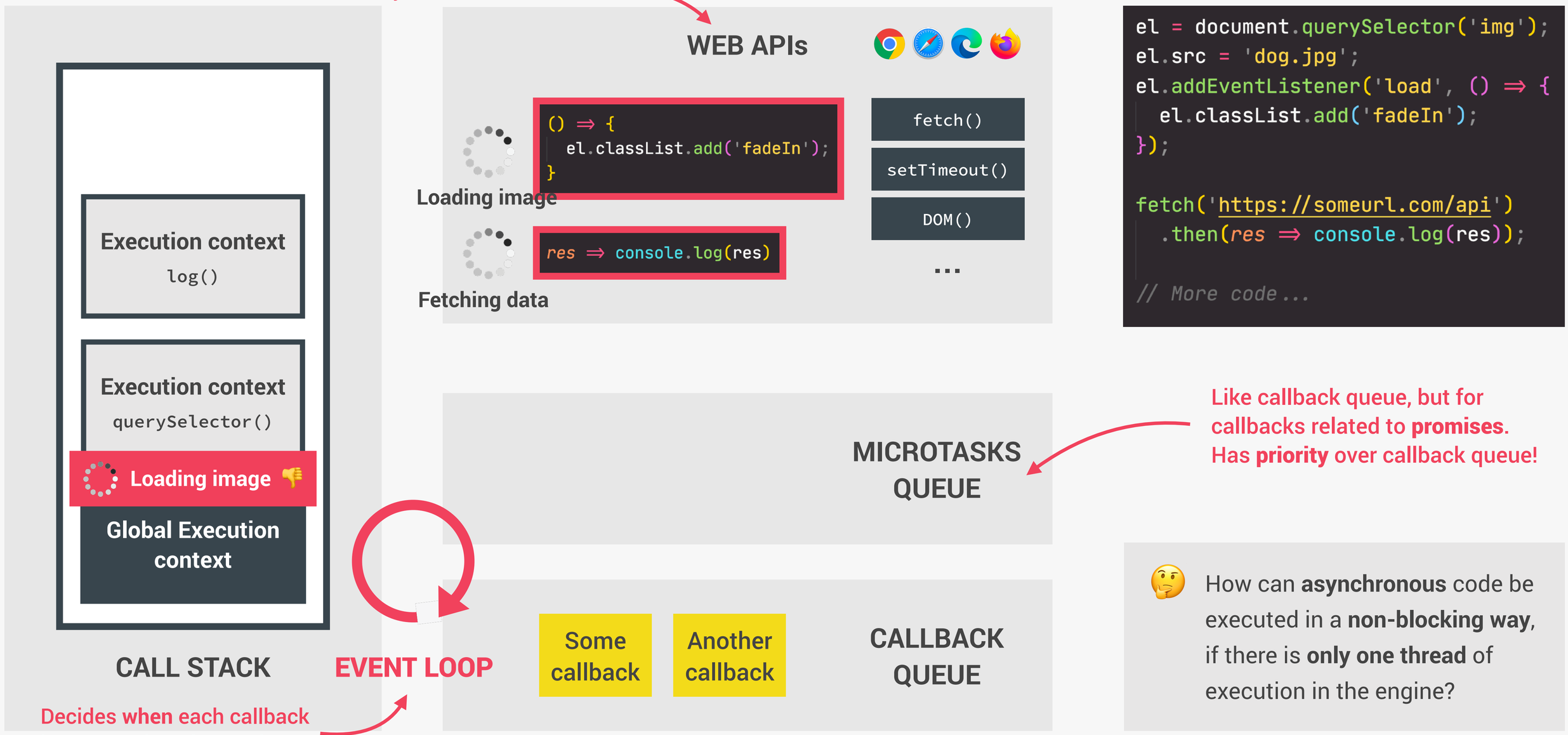
REVIEW: JAVASCRIPT RUNTIME

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



HOW ASYNCHRONOUS JAVASCRIPT WORKS BEHIND THE SCENES

Where asynchronous tasks run




```
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...
```

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

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

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

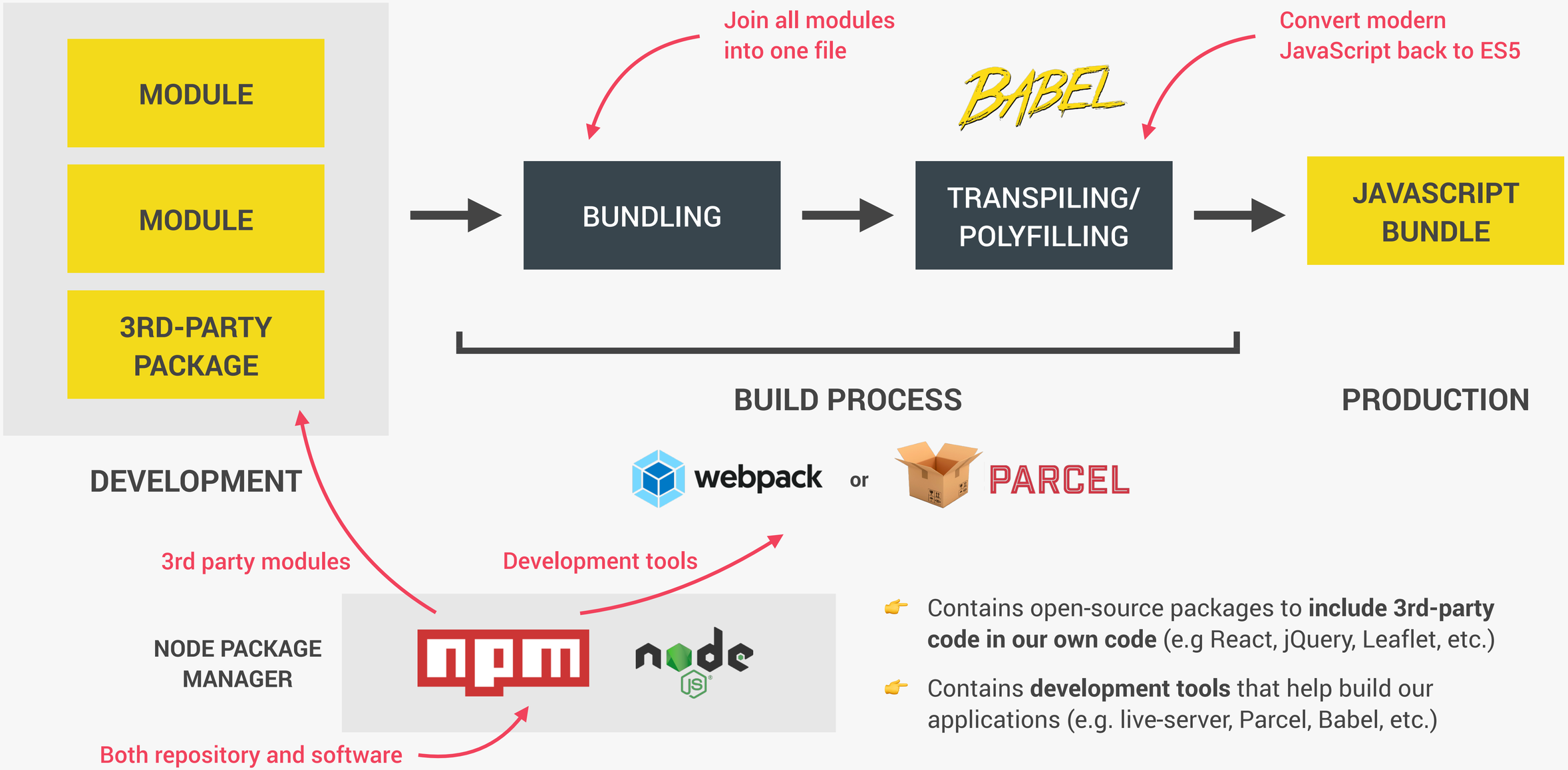
MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODERN JAVASCRIPT DEVELOPMENT

JS

MODERN JAVASCRIPT DEVELOPMENT



- 👉 Contains open-source packages to **include 3rd-party code in our own code** (e.g React, jQuery, Leaflet, etc.)
- 👉 Contains **development tools** that help build our applications (e.g. live-server, Parcel, Babel, etc.)

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

AN OVERVIEW OF MODULES IN JAVASCRIPT

JS

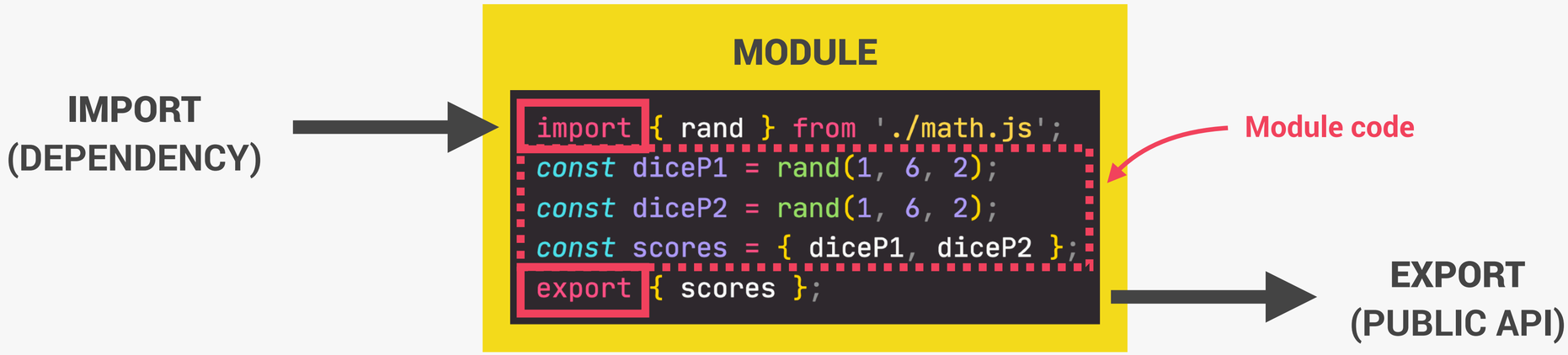
AN OVERVIEW OF MODULES

MODULE

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

WHY MODULES?

- 👉 **Compose software:** Modules are small building blocks that we put together to build complex applications;
- 👉 **Isolate components:** Modules can be developed in isolation without thinking about the entire codebase;
- 👉 **Abstract code:** Implement low-level code in modules and import these abstractions into other modules;
- 👉 **Organized code:** Modules naturally lead to a more organized codebase;
- 👉 **Reuse code:** Modules allow us to easily reuse the same code, even across multiple projects.



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 };
```

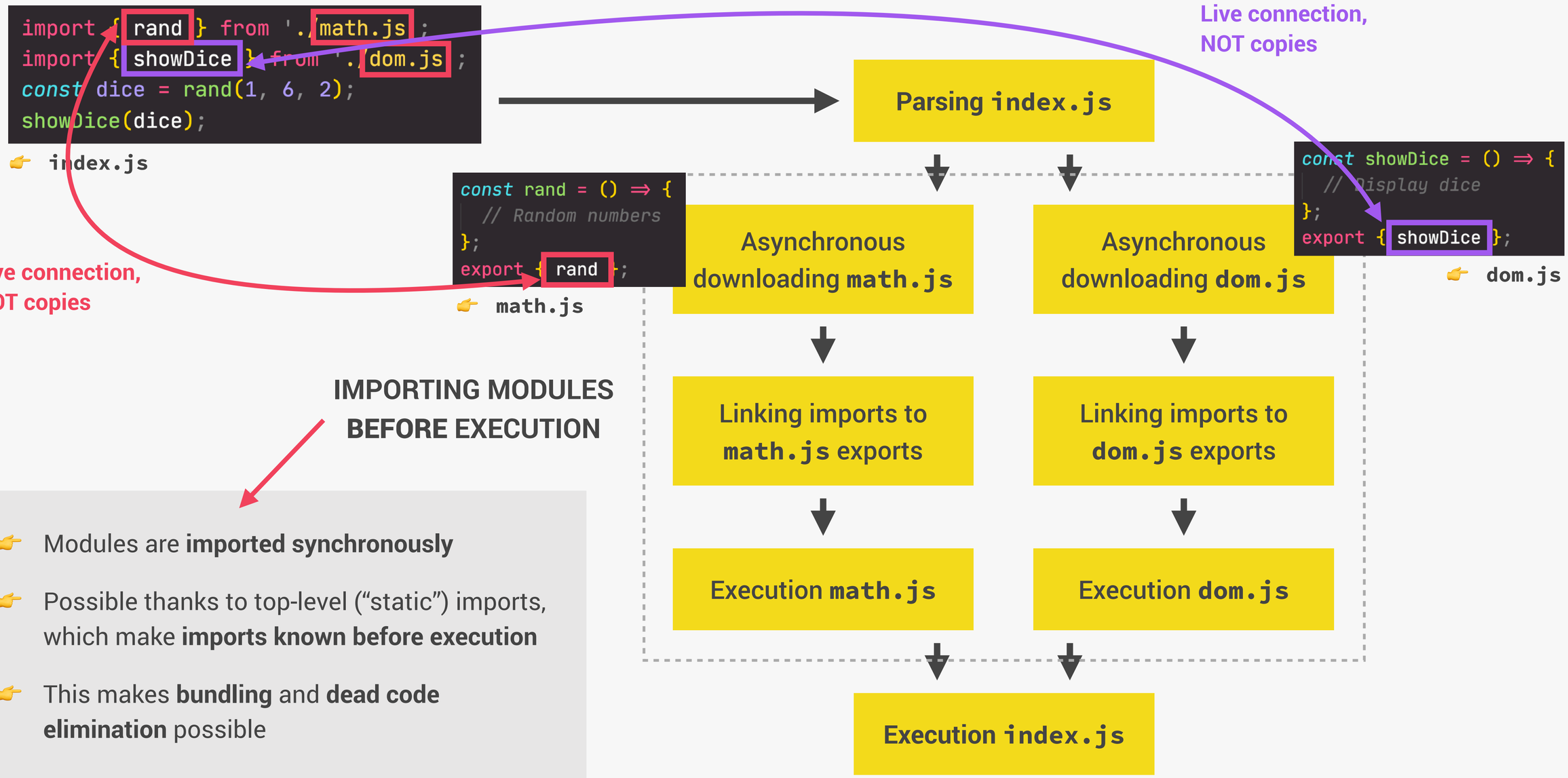
`import` and `export` syntax

	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



IMPORTING MODULES BEFORE EXECUTION

- 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

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

REVIEW: WRITING CLEAN AND MODERN JAVASCRIPT

JS

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)

REVIEW: MODERN AND CLEAN CODE

AVOID NESTED CODE

- 👉 Use early `return` (guard clauses)
- 👉 Use ternary (conditional) or logical operators instead of `if`
- 👉 Use multiple `if` instead of `if/else-if`
- 👉 Avoid `for` loops, use array methods instead
- 👉 Avoid callback-based asynchronous APIs

ASYNCHRONOUS CODE

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

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

MODERN JAVASCRIPT DEVELOPMENT: MODULES AND TOOLING

LECTURE

DECLARATIVE AND FUNCTIONAL JAVASCRIPT PRINCIPLES

JS

IMPERATIVE VS. DECLARATIVE CODE

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 achieve the result
- 👉 The **HOW** (step-by-step instructions) gets abstracted away
- 👉 **Example:** Description of a cake

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

FUNCTIONAL PROGRAMMING PRINCIPLES

FUNCTIONAL PROGRAMMING

- 👉 **Declarative** programming paradigm
- 👉 Based on the idea of writing software by combining many **pure functions**, avoiding **side effects** and **mutating** data

- 👉 **Side effect:** Modification (mutation) of any data **outside** of the function (mutating external variables, logging to console, writing to DOM, etc.)
- 👉 **Pure function:** Function without side effects. Does not depend on external variables. **Given the same inputs, always returns the same outputs**.
- 👉 **Immutability:** State (data) is **never** modified! Instead, state is **copied** and the copy is mutated and returned.

- 👉 **Examples:**   Redux

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

FORKIFY APP: BUILDING A MODERN APPLICATION

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

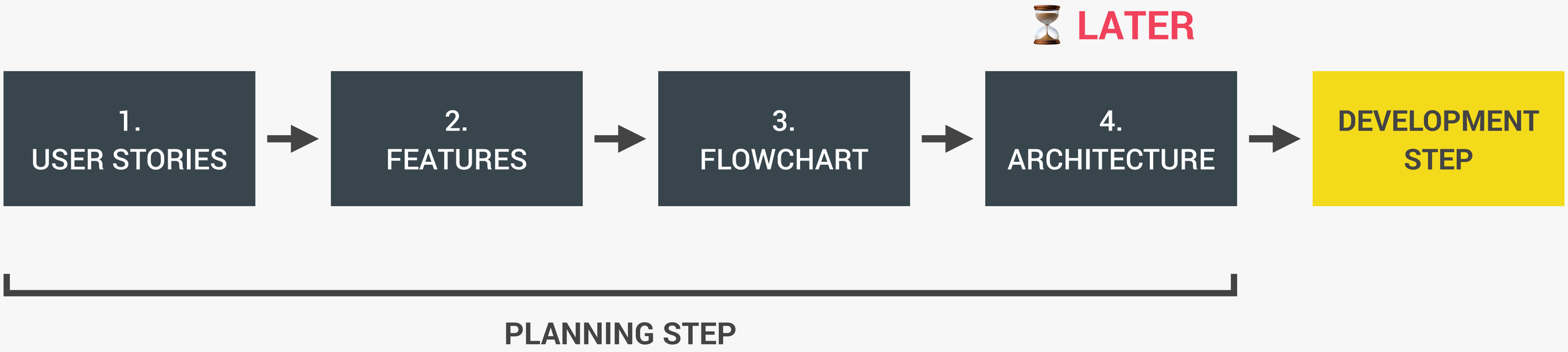
FORKIFY APP: BUILDING A MODERN APPLICATION

LECTURE

PROJECT OVERVIEW AND PLANNING

JS

PROJECT PLANNING



1. USER STORIES

forkify

👉 **User story:** Description of the application's functionality from the user's perspective.

👉 **Common format:** As a *[type of user]*, I want *[an action]* so that *[a benefit]*

1. As a user, I want to **search for recipes**, so that I can find new ideas for meals
2. As a user, I want to be able to **update the number of servings**, so that I can cook a meal for different number of people
3. As a user, I want to **bookmark recipes**, so that I can review them later
4. As a user, I want to be able to **create my own recipes**, so that I have them all organized in the same app
5. As a user, I want to be able to **see my bookmarks and own recipes when I leave the app and come back later**, so that I can close the app safely after cooking

2. FEATURES

forkify

USER STORIES	FEATURES
1 Search for recipes	👉 Search functionality: input field to send request to API with searched keywords 👉 Display results with pagination 👉 Display recipe with cooking time, servings and ingredients
2 Update the number of servings	👉 Change servings functionality: update all ingredients according to current number of servings
3 Bookmark recipes	👉 Bookmarking functionality: display list of all bookmarked recipes
4 Create my own recipes	👉 User can upload own recipes 👉 User recipes will automatically be bookmarked 👉 User can only see their own recipes, not recipes from other users
5 See my bookmarks and own recipes when I leave the app and come back later	👉 Store bookmark data in the browser using local storage 👉 On page load, read saved bookmarks from local storage and display

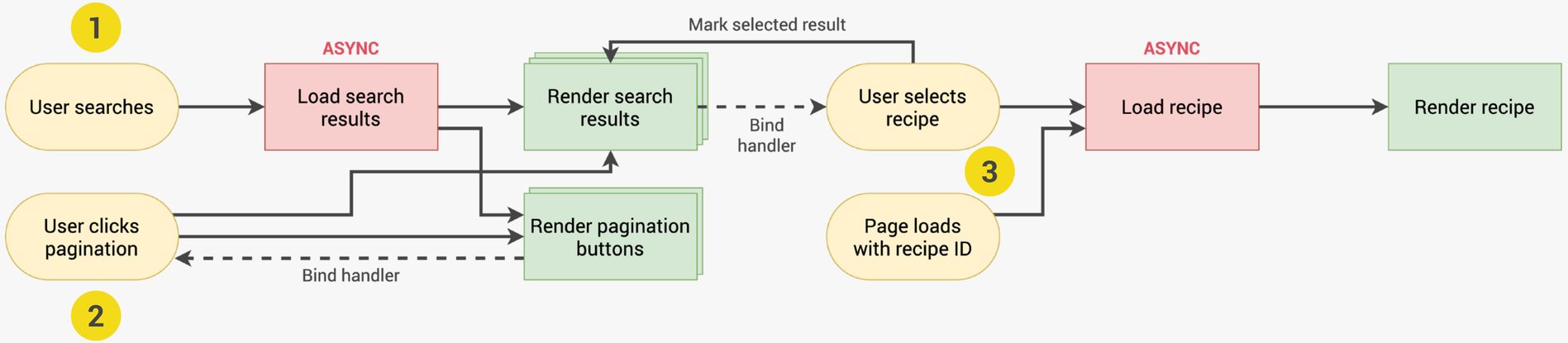
3. FLOWCHART (PART 1)

forkify

FEATURES

1. Search functionality: API search request
2. Results with pagination
3. Display recipe

Other features later



JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

FORKIFY APP: BUILDING A MODERN APPLICATION

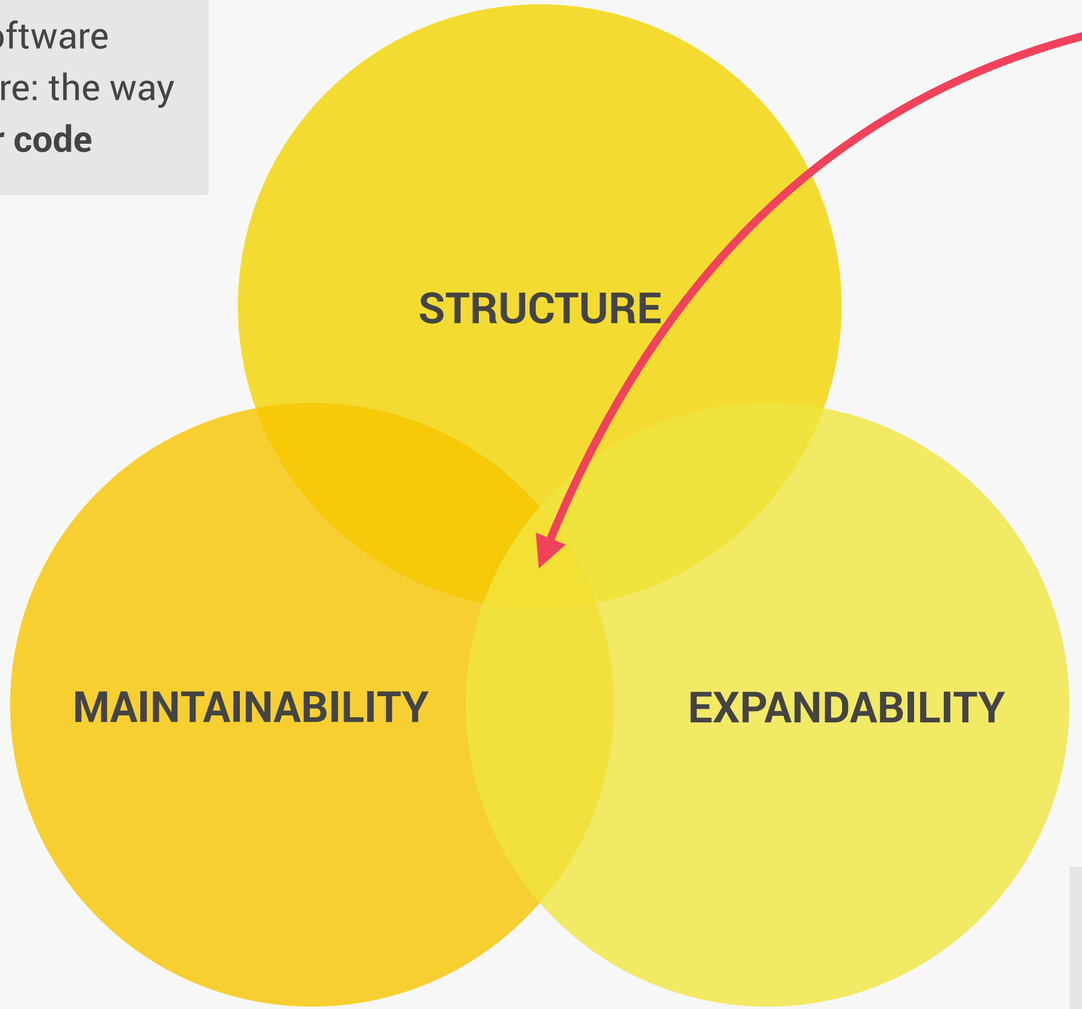
LECTURE

THE MVC ARCHITECTURE

JS

WHY WORRY ABOUT ARCHITECTURE?

👉 Like a house, software needs a structure: the way we **organize our code**



The perfect architecture

👉 We can create our own architecture (Mapty project)

👉 We can use a well-established architecture pattern like MVC, MVP, Flux, etc. (**this project**)

👉 We can use a framework like React, Angular, Vue, Svelte, etc.

👉 A project is never done! We need to be able to easily **change it in the future**

👉 We also need to be able to easily **add new features**

COMPONENTS OF ANY ARCHITECTURE

BUSINESS LOGIC

- 👉 Code that **solves the actual business problem**;
- 👉 Directly related to what business does and what it needs;
- 👉 **Example:** sending messages, storing transactions, calculating taxes, …

STATE

- 👉 Essentially **stores all the data** about the application
- 👉 Should be the “single source of truth”
- 👉 UI should be kept in sync with the state
- 👉 State libraries exist


HTTP LIBRARY

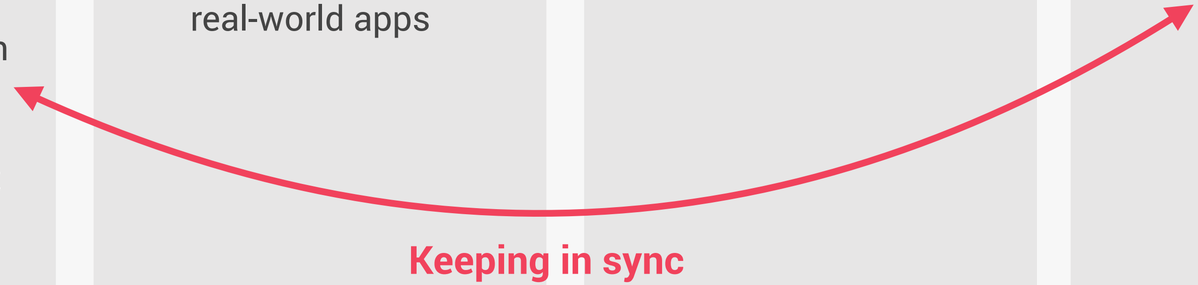
- 👉 Responsible for making and receiving AJAX requests
- 👉 Optional but almost always necessary in real-world apps

APPLICATION LOGIC (ROUTER)

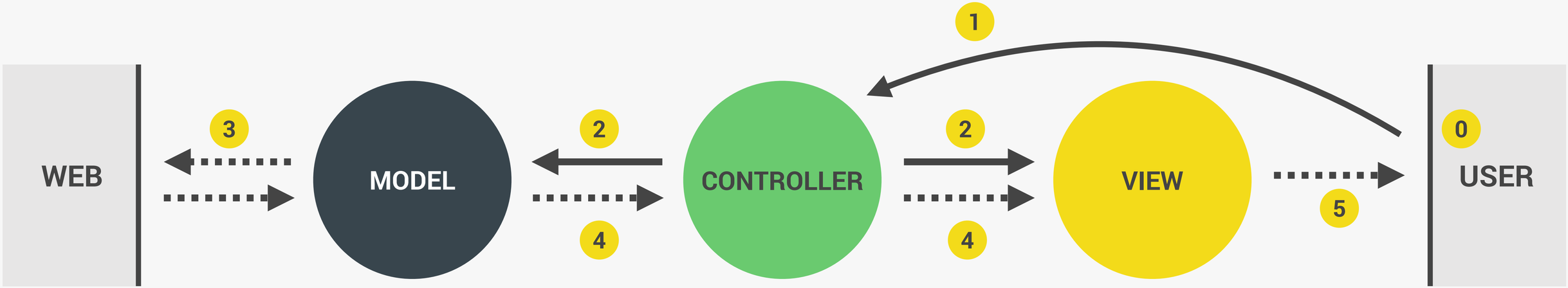
- 👉 Code that is only concerned about the **implementation of application itself**;
- 👉 Handles navigation and UI events

PRESENTATION LOGIC (UI LAYER)

- 👉 Code that is concerned about the **visible part** of the application
- 👉 Essentially displays application state



THE MODEL-VIEW-CONTROLLER (MVC) ARCHITECTURE



BUSINESS LOGIC

STATE

HTTP LIBRARY

APPLICATION LOGIC

- 👉 Bridge between model and views (which don't know about one another)
- 👉 Handles UI events and **dispatches tasks to model and view**

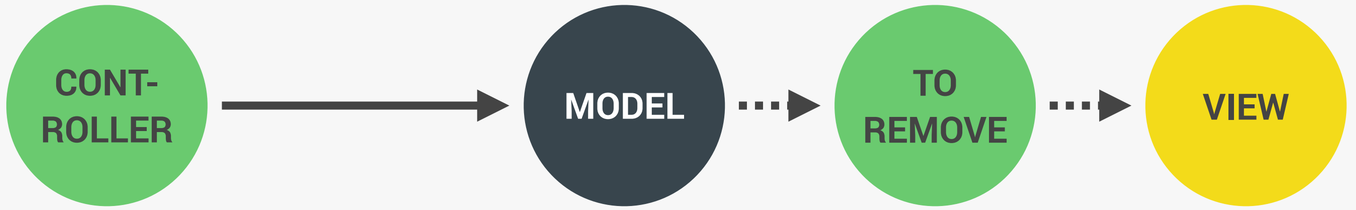
PRESENTATION LOGIC

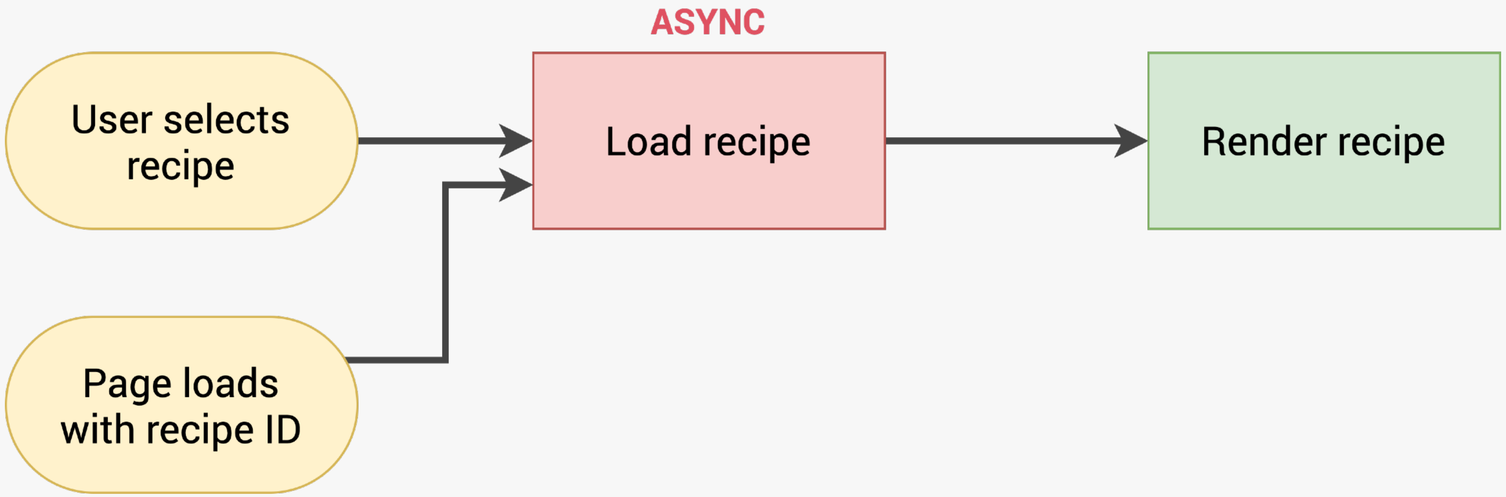
→ Connected by function call and import

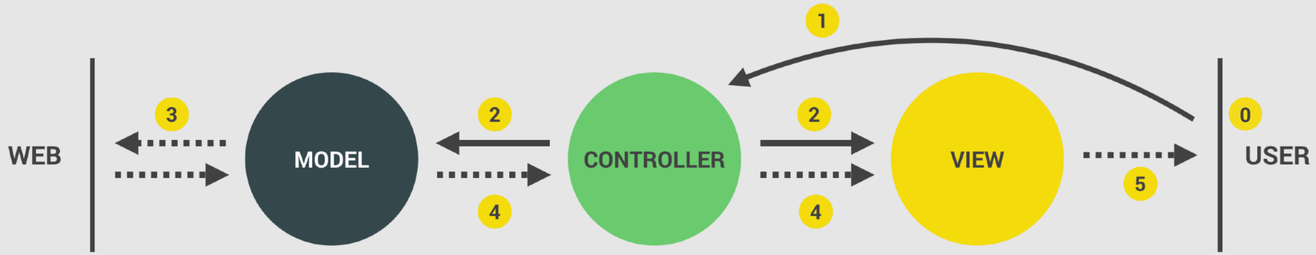
⇢ Data flow

MODEL, VIEW AND CONTROLLER IN FORKIFY (RECIPE DISPLAY ONLY)

forkify

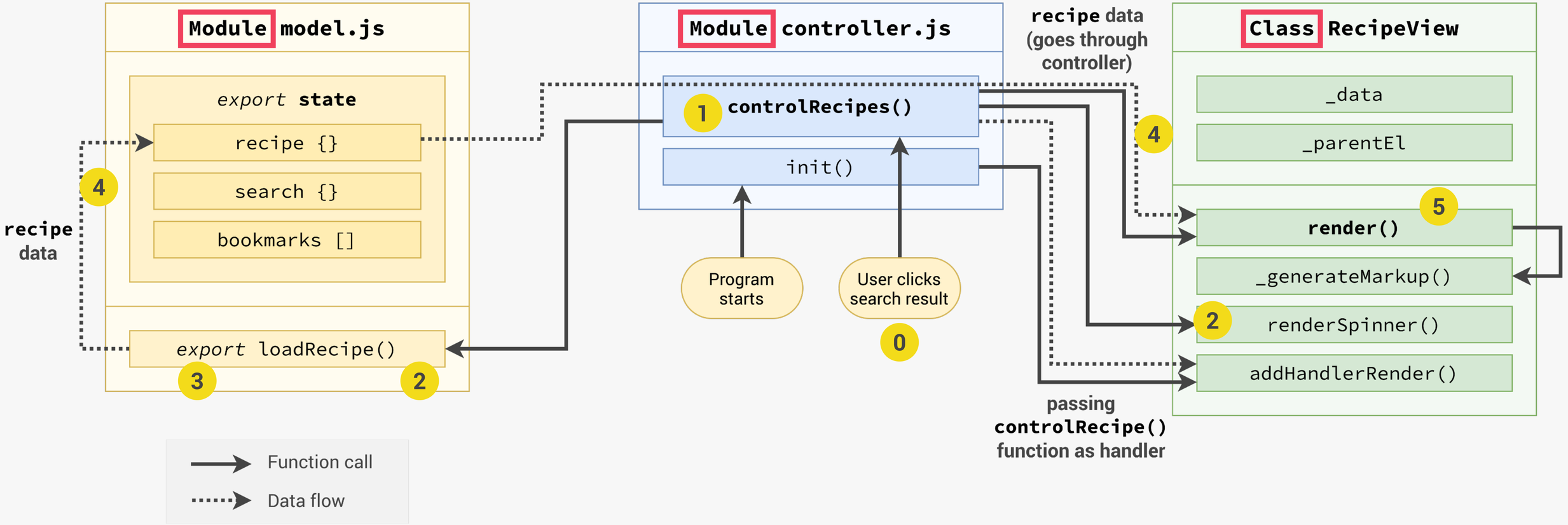


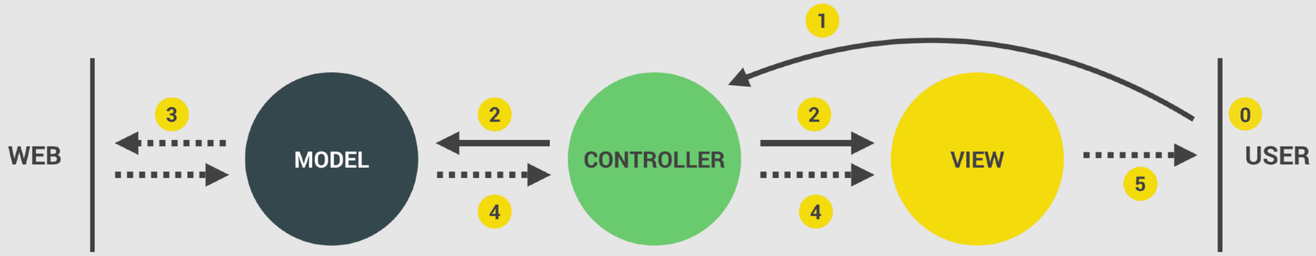




MVC IMPLEMENTATION (RECIPE DISPLAY ONLY)

forkify





JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

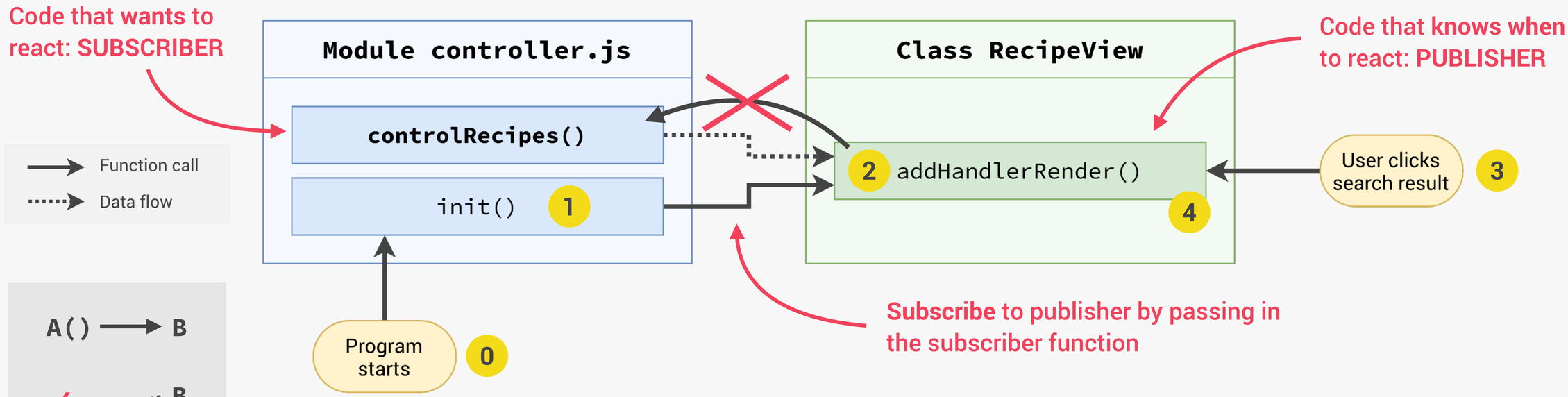
FORKIFY APP: BUILDING A MODERN APPLICATION

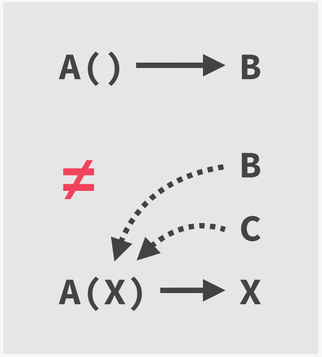
LECTURE

EVENT HANDLERS IN MVC: PUBLISHER-SUBSCRIBER PATTERN

JS

EVENT HANDLING IN MVC: PUBLISHER-SUBSCRIBER PATTERN





- 👉 `controlRecipes` will be passed into `addHandlerRender` when program starts
- 👉 `addHandlerRender` listens for events (`addEventListener`), and uses `controlRecipes` as callback

- 👉 Events should be **handled** in the **controller** (otherwise we would have application logic in the view)
- 👉 Events should be **listened for** in the **view** (otherwise we would need DOM elements in the controller)

JONAS.IO
SCHMEDTMANN

THE COMPLETE JAVASCRIPT COURSE

FROM ZERO TO EXPERT!

@JONASSCHMEDTMAN

SECTION

FORKIFY APP: BUILDING A MODERN APPLICATION

LECTURE

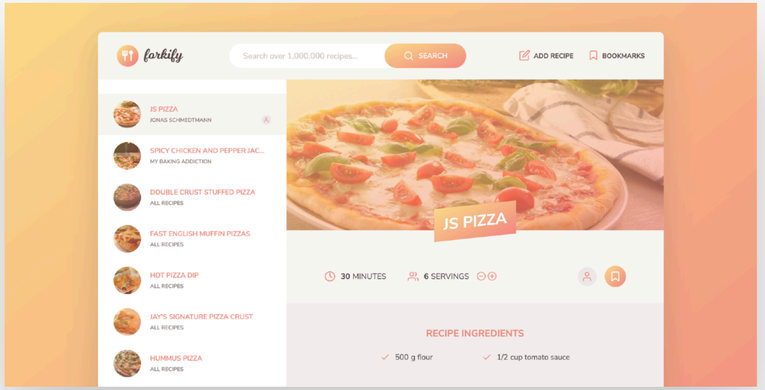
WRAPPING UP: FINAL CONSIDERATIONS

JS

IMPROVEMENT AND FEATURE IDEAS: CHALLENGES 🤓

forkify

- 👉 Display **number of pages** between the pagination buttons;
- 👉 Ability to **sort** search results by duration or number of ingredients;
- 👉 Perform **ingredient validation** in view, before submitting the form;
- 👉 **Improve recipe ingredient input**: separate in multiple fields and allow more than 6 ingredients;



- 👉 **Shopping list feature**: button on recipe to add ingredients to a list;
- 👉 **Weekly meal planning feature**: assign recipes to the next 7 days and show on a weekly calendar;
- 👉 **Get nutrition data** on each ingredient from spoonacular API (https://spoonacular.com/food-api) and calculate total calories of recipe.

END