WHICH ARRAY METHOD TO USE? 🤔

"I WANT...:"

To *mutate* original

👉 Add to original:

- `.push` *(end)*
- `.unshift` *(start)*

👉 Remove from original:

- `.pop` *(end)*
- `.shift` *(start)*
- `.splice` *(any)*

👉 Others:

- `.reverse`
- `.sort`
- `.fill`

☝️ **These should usually be avoided!**

A *new* array based on original

👉 Same length as original:

- `.map` *(loop)*

👉 Filtered using condition:

- `.filter`

👉 Taking portion of original:

- `.slice`

👉 With one item replaced:

- `.with`

👉 Flattened:

- `.flat`
- `.flatMap`

👉 Reversed:

- `.toReversed`

👉 Sorted:

- `.toSorted`

👉 With deleted items:

- `.toSpliced`

👉 Joining two arrays:

- `.concat`

An array index

👉 Based on value:

- `.indexOf`

👉 Based on test condition:

- `.findIndex`
- `.findLastIndex`

An array element

👉 Based on test condition:

- `.find`
- `.findLast`

👉 Based on position:

- `.at`

Know if array includes

👉 Based on value:

- `.includes`

👉 Based on test condition:

- `.some`
- `.every`

A new string

👉 Based on separator:

- `.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)

MORE ARRAY TOOLS AND TECHNIQUES

- **Grouping** an array by categories:

```
Object.groupBy
```

- Creating a new array **from scratch**:

```
Array.from
```

- Creating a new array **from scratch** with n empty positions (use together with `.fill` method):

```
new Array(n)
```

- **Joining** 2 or more arrays:

```
[...arr1, ...arr2]
```

- Creating a new array containing **unique** values from `arr`

```
[...new Set(arr)]
```

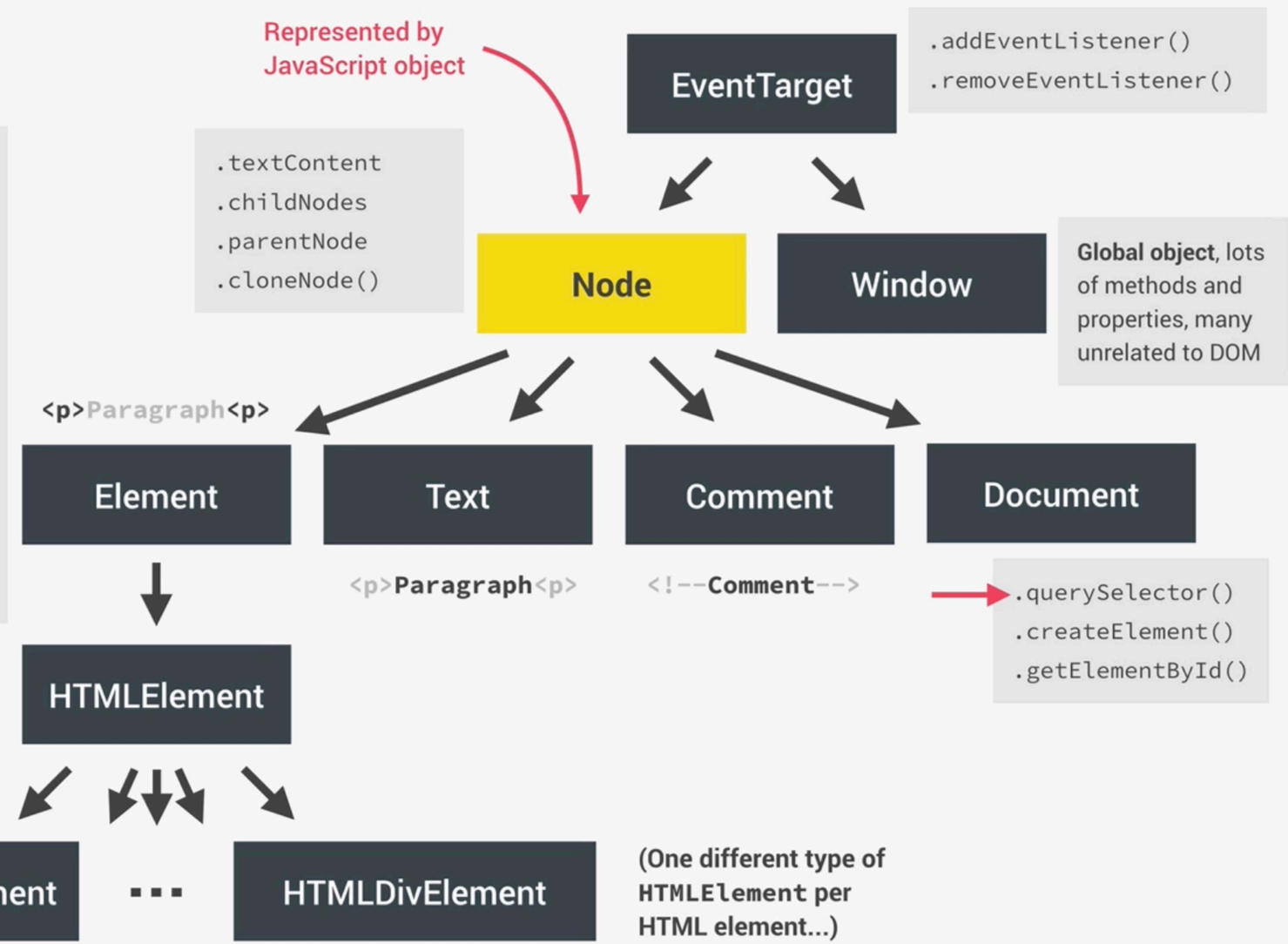
- Creating a new array containing unique elements that are present **in both** `arr1` and `arr2`

```
[...new Set(arr1).intersection(new Set(arr2))]
```

HOW THE DOM API IS ORGANIZED BEHIND THE SCENES

MDN web docs
moz://a

Represented by JavaScript object



.innerHTML
.classList
.children
.parentElement
.append()
.remove()
.insertAdjacentHTML()
.querySelector()
.closest()
.matches()
.scrollIntoView()
.setAttribute()

INHERITANCE OF METHODS AND PROPERTIES

Example:

Any `HTMLElement` will have access to `.addEventListener()`, `.cloneNode()` or `.closest()` methods.

(THIS IS NOT A DOM TREE)

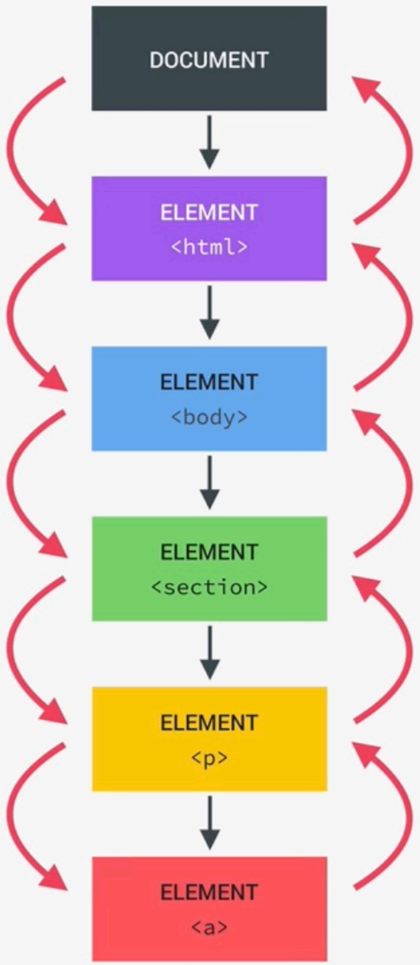
BUBBLING AND CAPTURING

```
<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>
```

(THIS DOES NOT HAPPEN ON *ALL* EVENTS)

1

CAPTURING PHASE

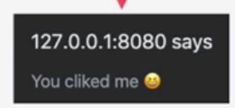


Click event

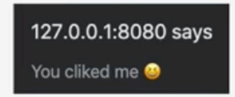
3

BUBBLING PHASE

```
document
  .querySelector('section')
  .addEventListener('click', () => {
  alert('You cliked me 😀');
});
```




```
document
  .querySelector('a')
  .addEventListener('click', () => {
  alert('You cliked me 😀');
});
```



2 TARGET PHASE

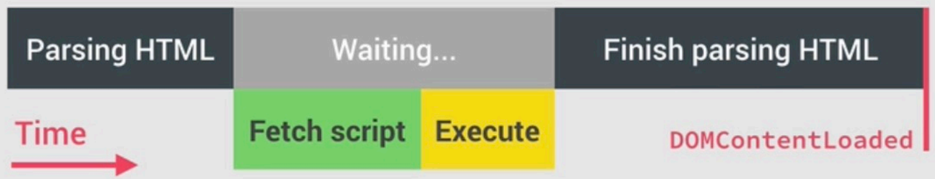
DEFER AND ASYNC SCRIPT LOADING

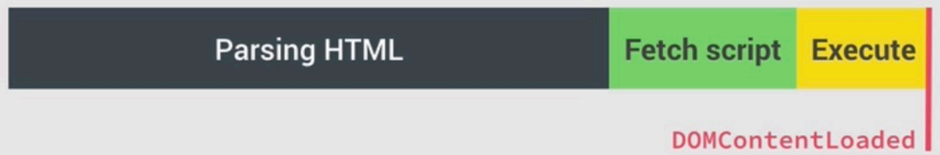
HEAD

BODY END

REGULAR

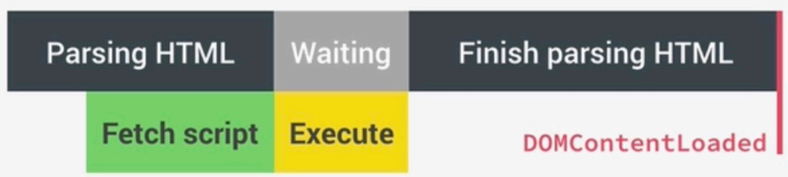
```
<script src="script.js">
```





ASYNC

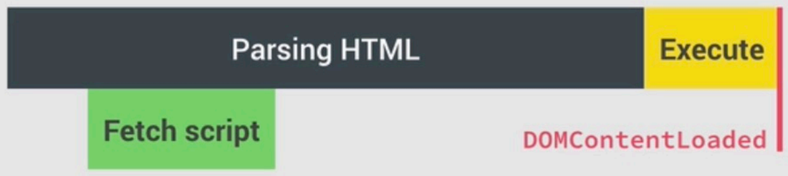
```
<script async src="script.js">
```



👉 Makes no sense 🤷‍♂️

DEFER

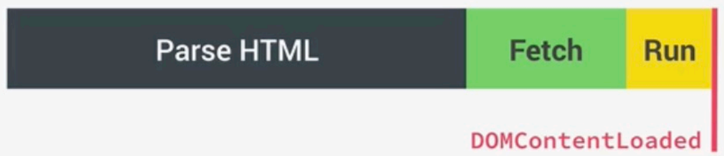
```
<script defer src="script.js">
```



👉 Makes no sense 🤷‍♂️

REGULAR VS. ASYNC VS. DEFER

END OF BODY

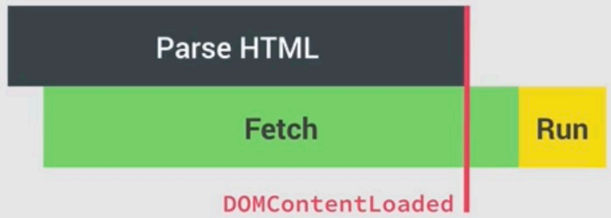


- Scripts are fetched and executed ***after the HTML is completely parsed***
- **Use if you need to support old browsers**

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)**

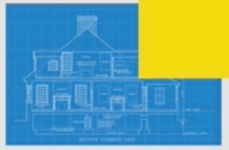


DEFER IN HEAD

- 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)**

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?"*

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.

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

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

HOW PROTOTYPAL INHERITANCE / DELEGATION WORKS

Constructor function
[Person()]

```
const Person = function(name, birthYear) {
  this.name = name;
  this.birthYear = birthYear;
};
```

.prototype

.constructor

NOT of Person, but objects created by Person

Prototype
[Person.prototype]

```
calcAge: function
```

PROTOTYPE CHAIN

PROTOTYPAL INHERITANCE/ DELEGATION

.__proto__

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

jonas.calcAge(); // 47
```

Can't find calcAge here!

Object
[jonas]

```
name: 'Jonas'
birthYear: 1990
__proto__:
Person.prototype
```

Always points to an object's prototype

☝ This is how it works with **function constructors** and **ES6 classes**

🆕 **The new operator:**

1. An empty object is created
2. `this` keyword in constructor function call is set to the new object
3. The new object is linked (`__proto__` property) to the constructor function's prototype property
4. The new object is returned from the constructor function call

THE PROTOTYPE CHAIN

Constructor function
[Object()]

Prototype
[Object.prototype]
__proto__: null

null

Object.prototype
{constructor: ƒ, __def
constructor: ƒ Objec
__defineGetter__: ƒ
__defineSetter__: ƒ
hasOwnProperty ƒ ha

.__proto__

Here it is!

This is an OBJECT itself! Remember, every object in JavaScript has a prototype!

Constructor function
[Person()]

Prototype
[Person.prototype]
__proto__:
Object.prototype

.__proto__

Can't find hasOwnProperty here!

Built-in constructor function for objects. This is used when we write an object literal:

{…} === new Object(…)

Object
[jonas]
__proto__:
Person.prototype

```
jonas.hasOwnProperty('name');
// true
```

Can't find hasOwnProperty here!

PROTOTYPE CHAIN

Series of links between objects, linked through prototypes

(Similar to the scope chain)

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*

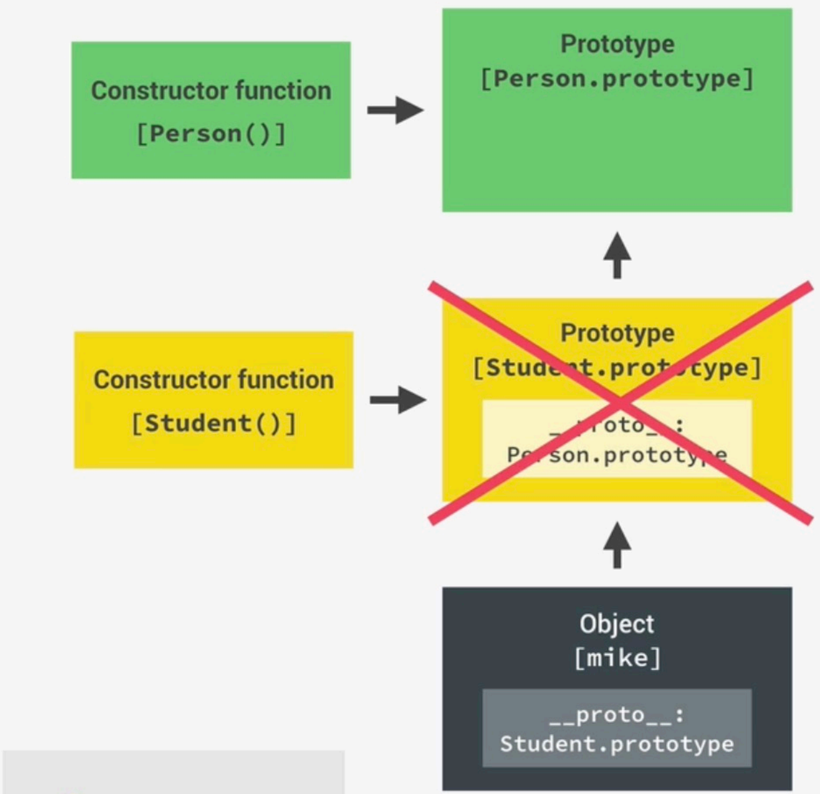
AUTOMATIC
.__proto__

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

Object
[jonas]
name: ‘Jonas’
birthYear: 1990
__proto__: Person.prototype

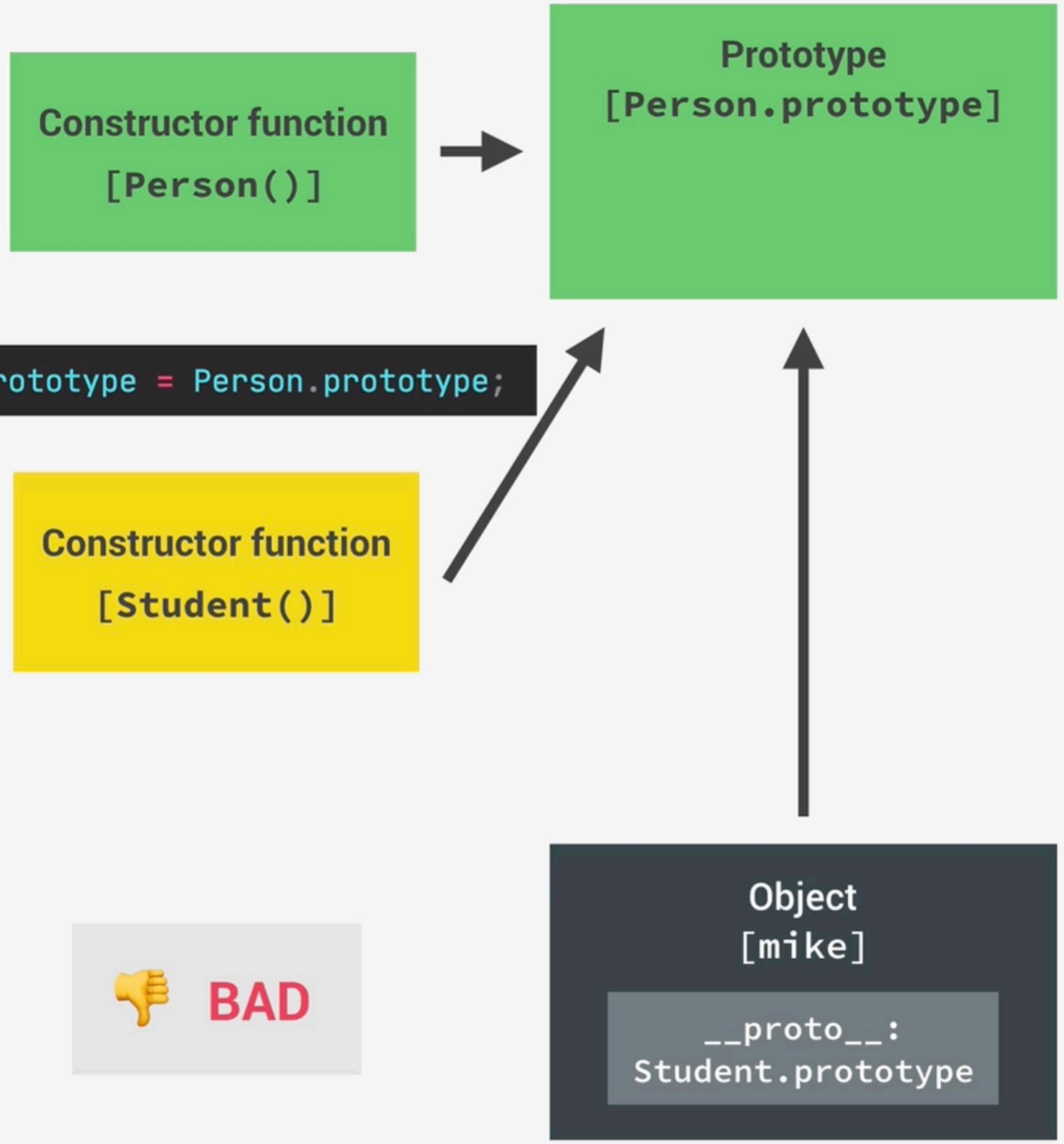
INHERITANCE BETWEEN "CLASSES"

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



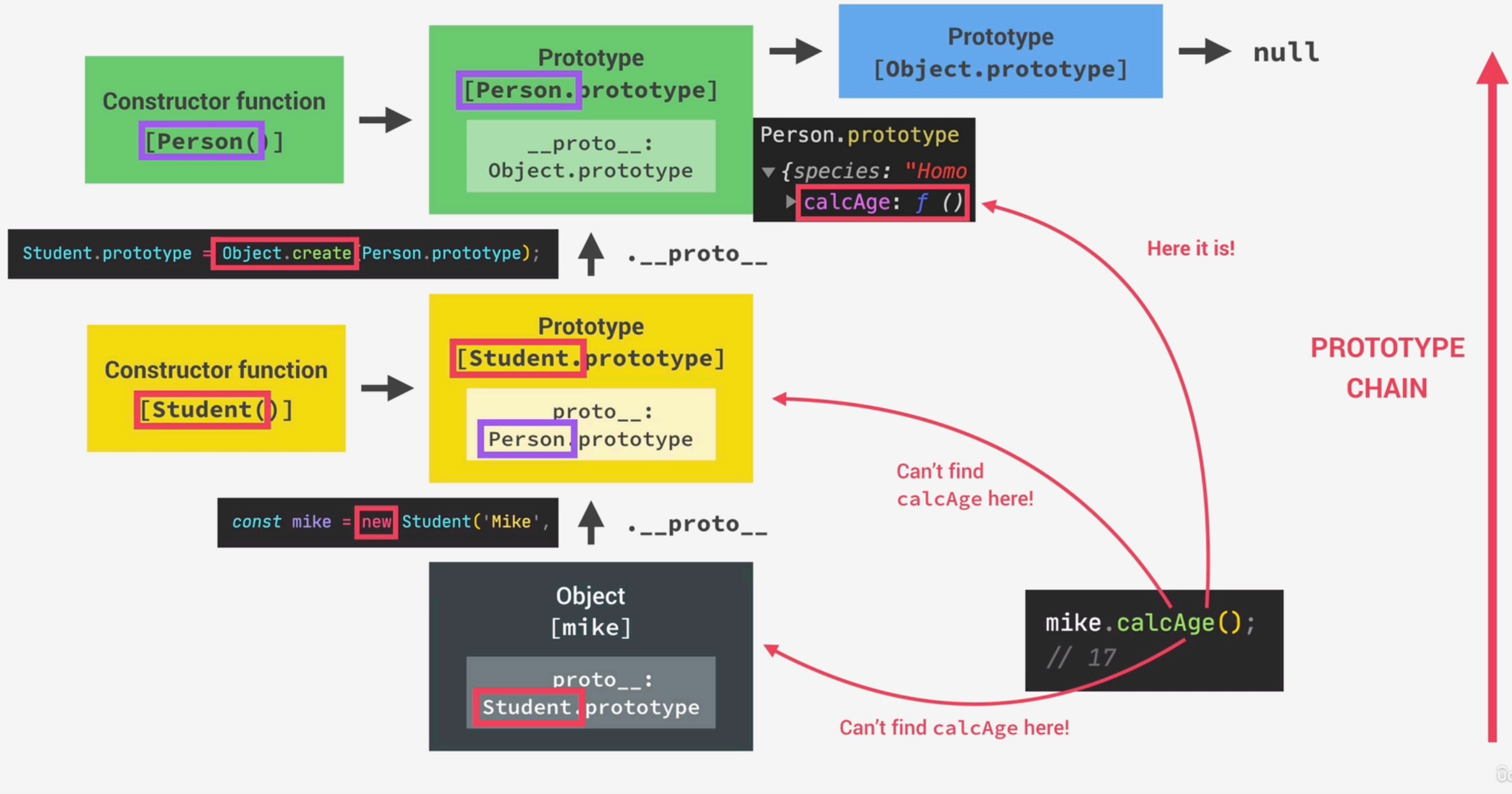
👍 GOOD

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



👎 BAD

INHERITANCE BETWEEN "CLASSES"



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

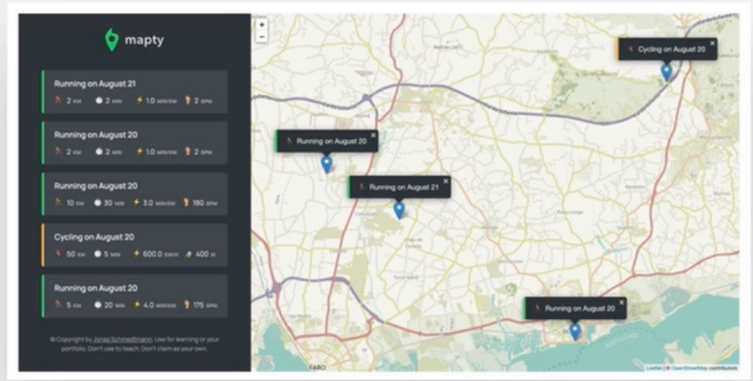
Constructor method, called by new operator. Mandatory in regular class, might be omitted in a **child** class

- 👉 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**

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 CODE

Asynchronous

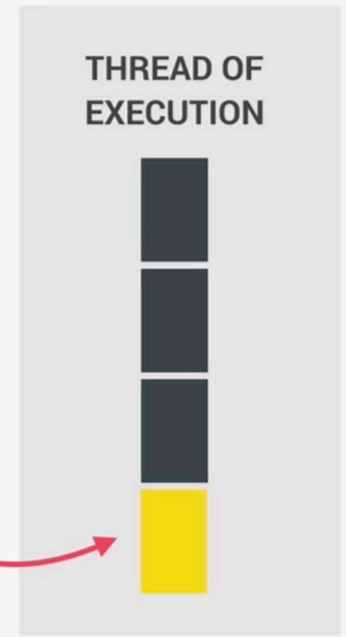
CALLBACK WILL RUN AFTER TIMER

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

👉 **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

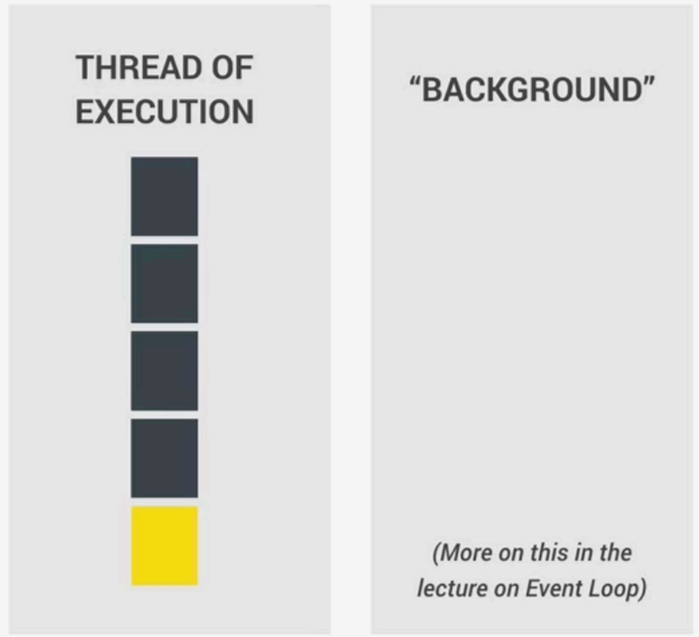
```
const img = document.querySelector('.dog');
img.src = 'dog.jpg';
img.addEventListener('load', function () {
  img.classList.add('fadeIn');
});
p.style.width = '300px';
```

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 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.

👉 **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

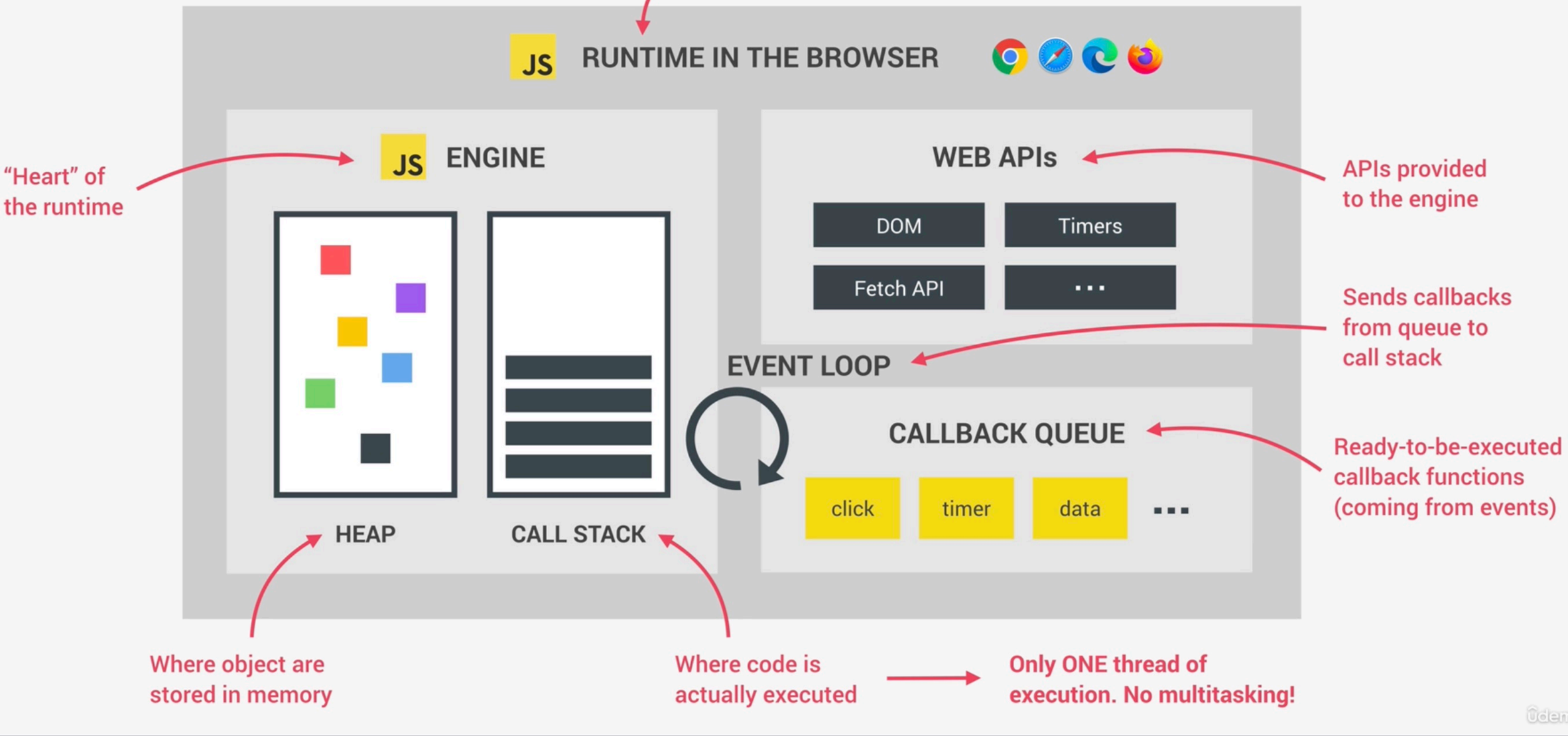
BUILD PROMISE

CONSUME PROMISE

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

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

CALL STACK

EVENT LOOP

Decides when each callback is executed: orchestration

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 ...
```

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

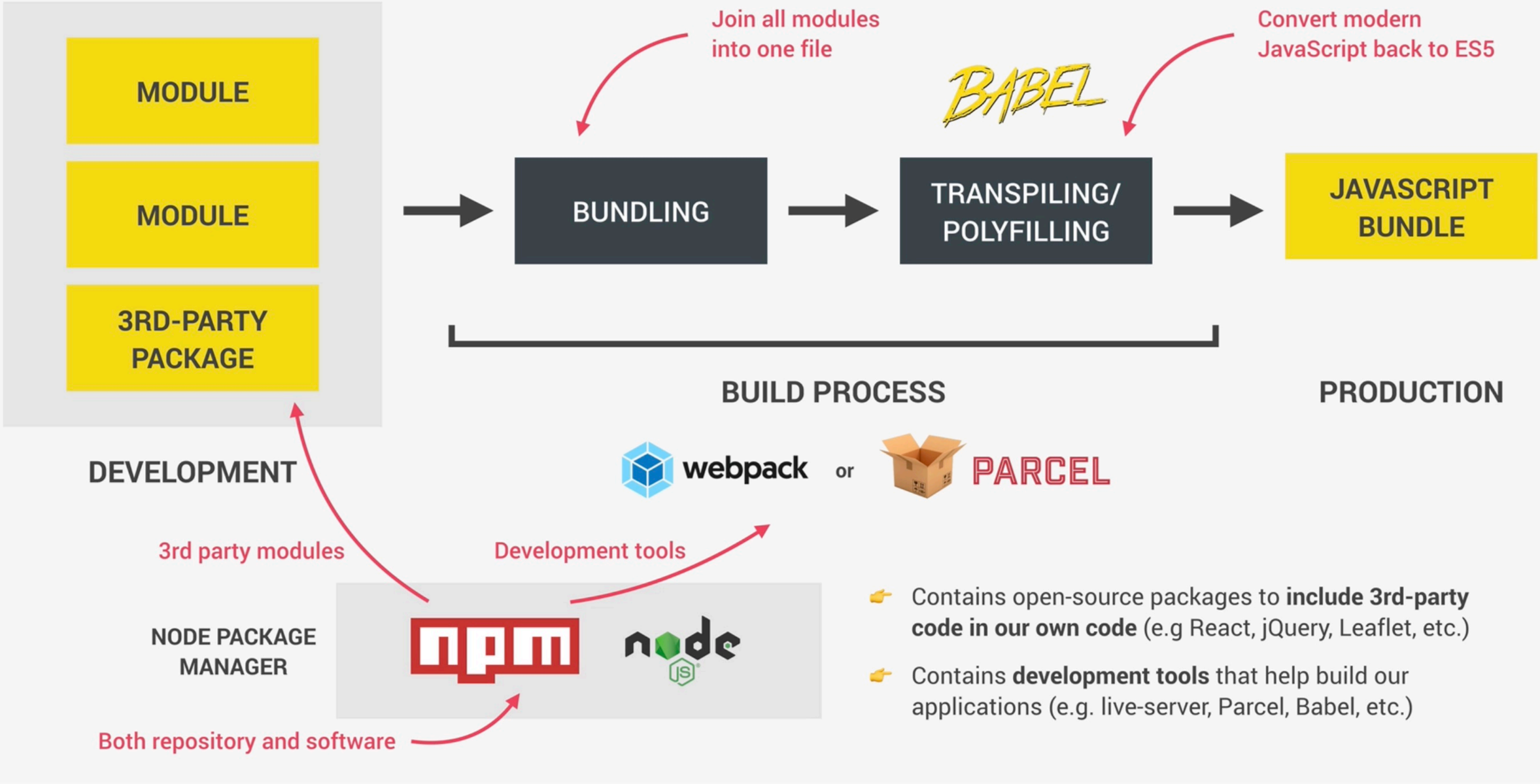
MICROTASKS QUEUE

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

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



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

```
import { rand } from './math.js';
import { showDice } from './dom.js';
const dice = rand(1, 6, 2);
showDice(dice);
```

👉 index.js

Live connection,
NOT copies

Parsing index.js

```
const rand = () => {
  // Random numbers
};
export { rand };
```

👉 math.js

```
const showDice = () => {
  // Display dice
};
export { showDice };
```

👉 dom.js

Asynchronous downloading **math.js**

Asynchronous downloading **dom.js**

Linking imports to **math.js** exports

Linking imports to **dom.js** exports

Execution **math.js**

Execution **dom.js**

Execution **index.js**

Live connection,
NOT copies

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

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

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:** React Redux

FUNCTIONAL PROGRAMMING TECHNIQUES

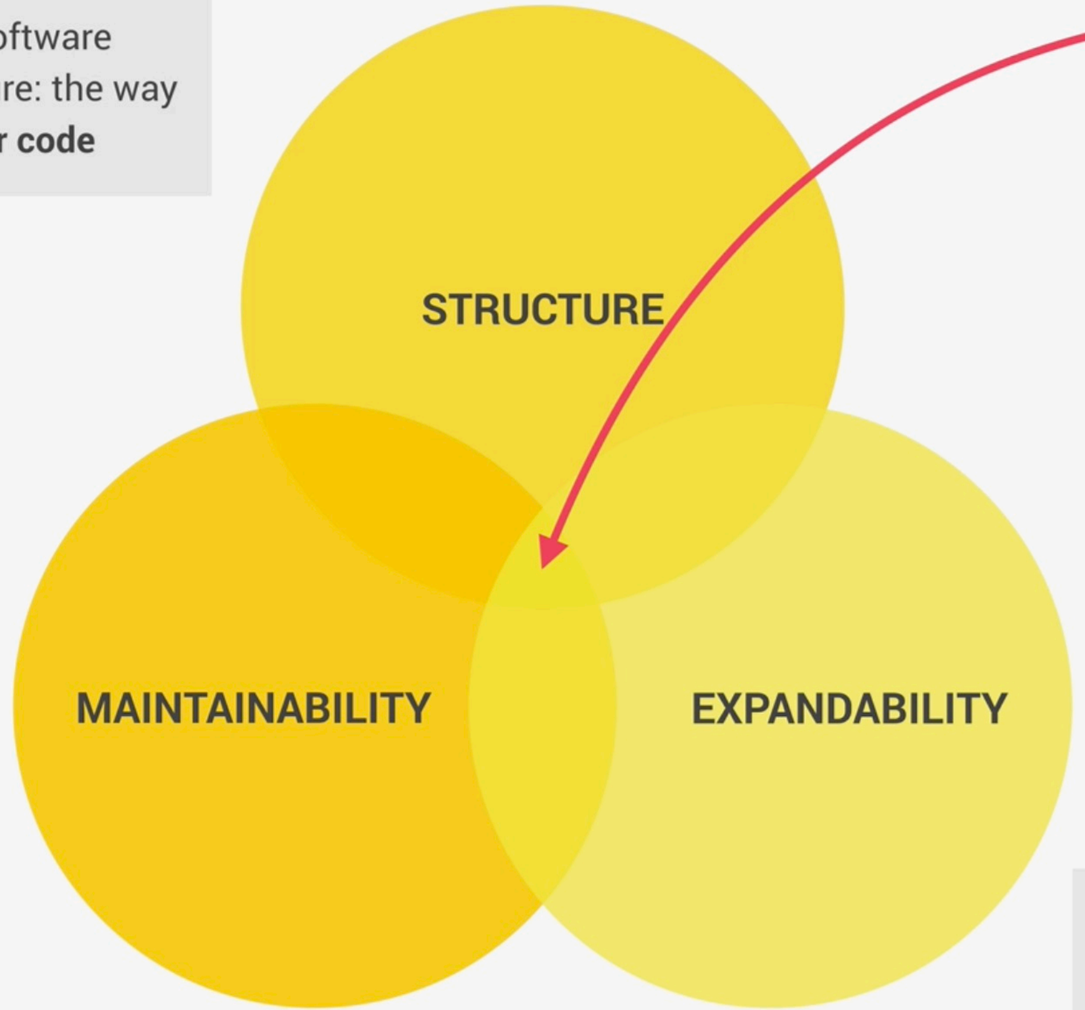
- 👉 Try to avoid data mutations
- 👉 Use built-in methods that don't produce side effects
- 👉 Do data transformations with methods such as `.map()`, `.filter()` and `.reduce()`
- 👉 Try to avoid side effects in functions: this is of course not always possible!

DECLARATIVE SYNTAX

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

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**

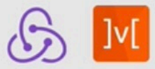
COMPONENTS OF ANY ARCHITECTURE

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

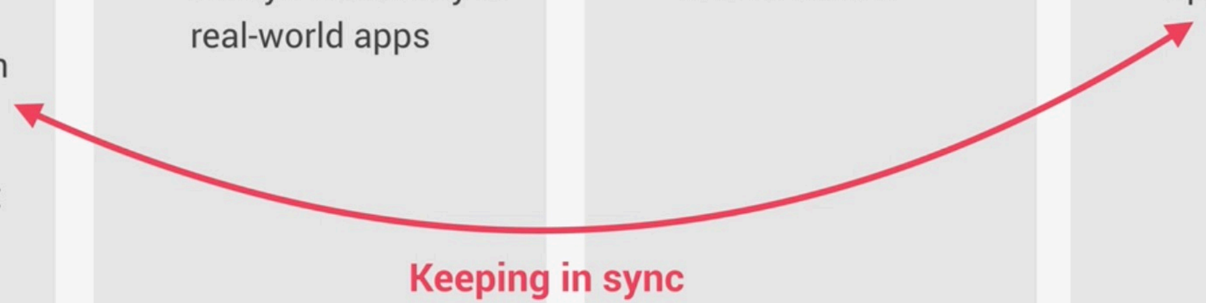
- Responsible for making and receiving AJAX requests
- Optional but almost always necessary in real-world apps

APPLICATION LOGIC (ROUTER)

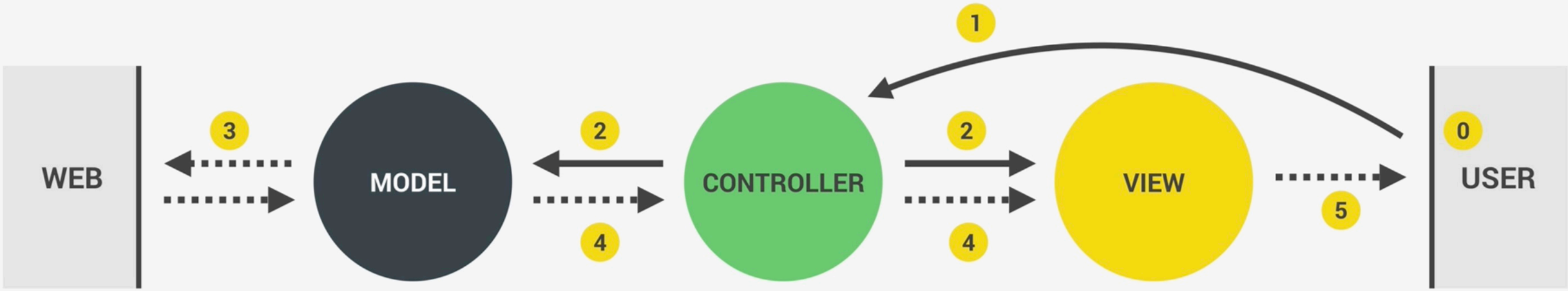
- Code that is only concerned about the **implementation of application itself**;
- Handles navigation and UI events

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**

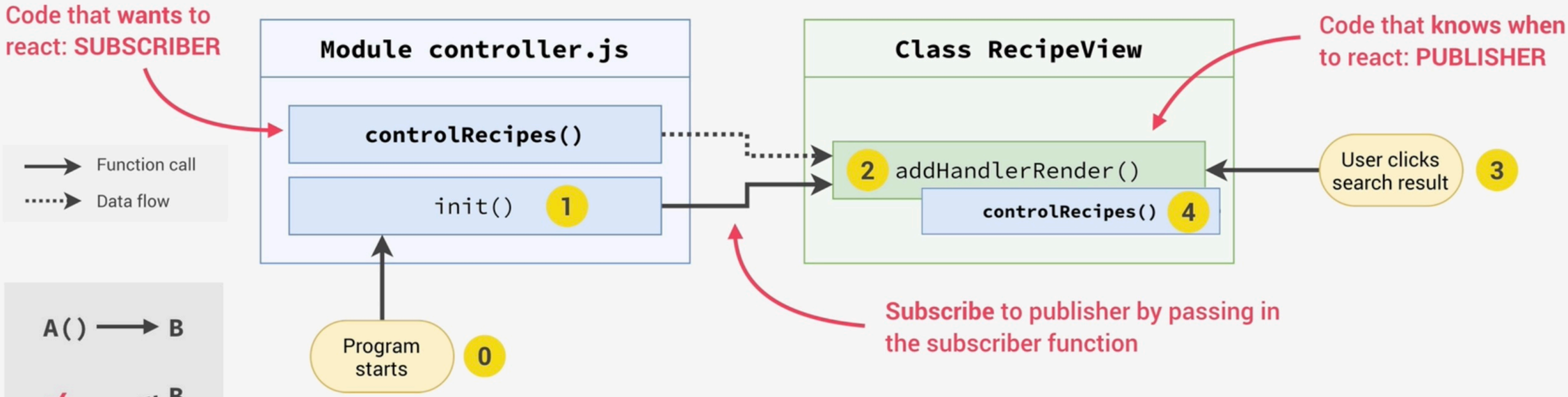
PRESENTATION LOGIC

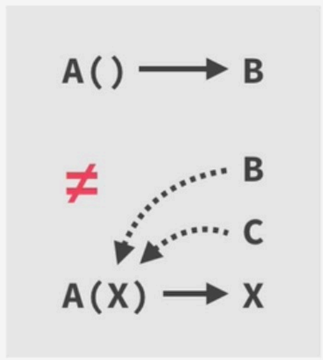
→ Connected by function call and import

⇢ Data flow

EVENT HANDLING IN MVC: PUBLISHER-SUBSCRIBER PATTERN

forkify





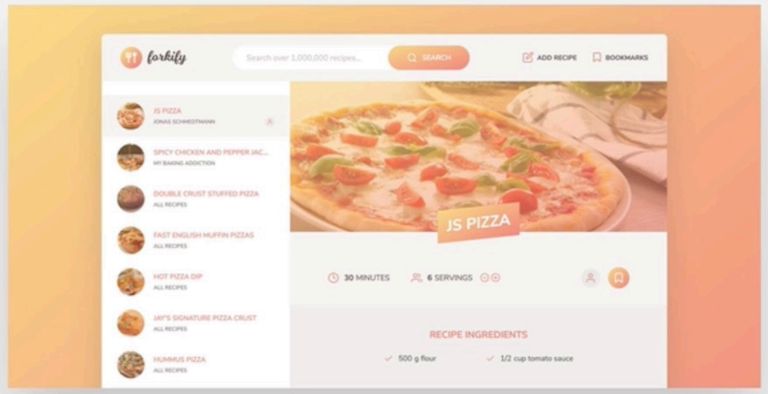
- `controlRecipes` will be passed into `addHandlerRender` when program starts
- `addHandlerRender` listens for events (`addEventListener`), and uses `controlRecipes` as callback

- 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)

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.