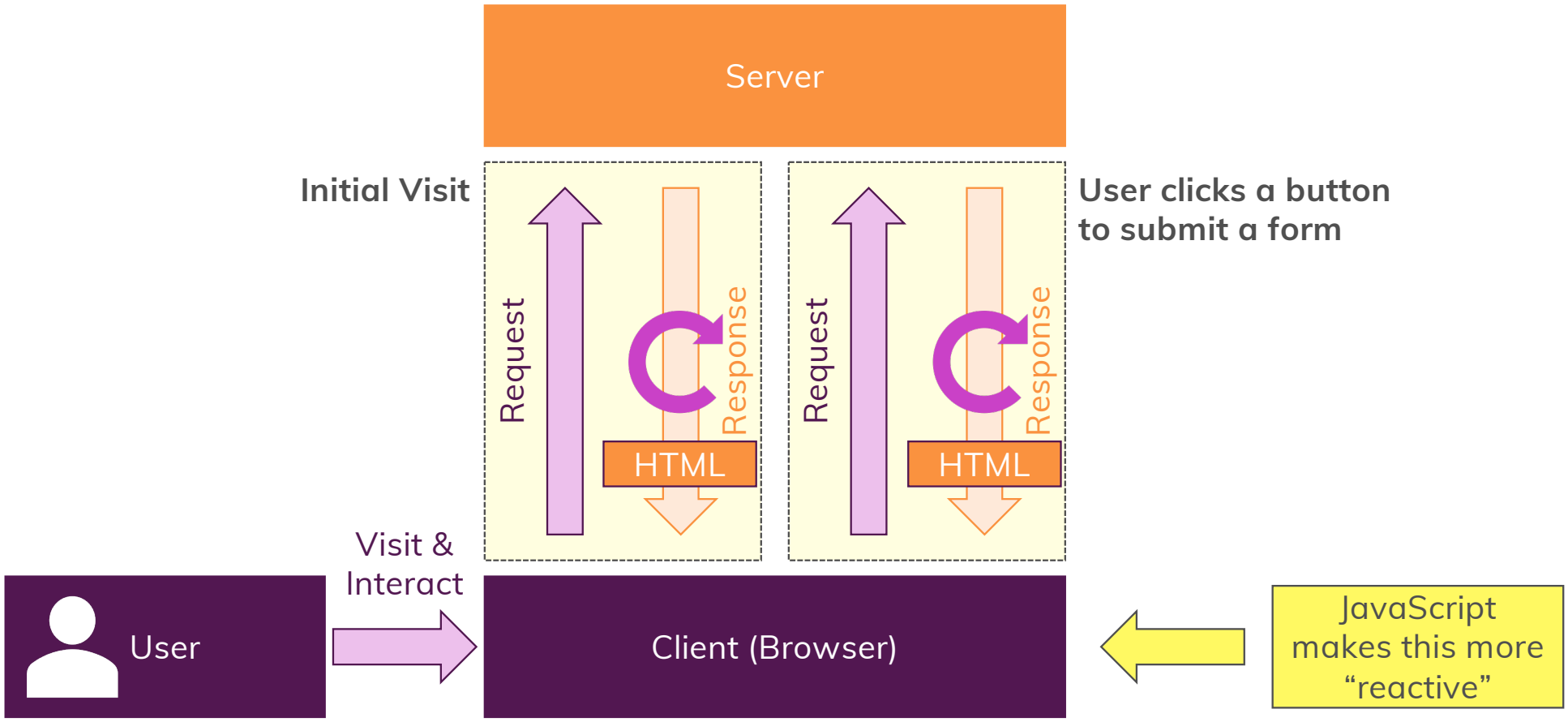
What is JavaScript?

JavaScript is a **dynamic**, **weakly typed** programming language which is **compiled at runtime**. It can be executed as part of a webpage in a browser or directly on any machine (**"host environment"**).

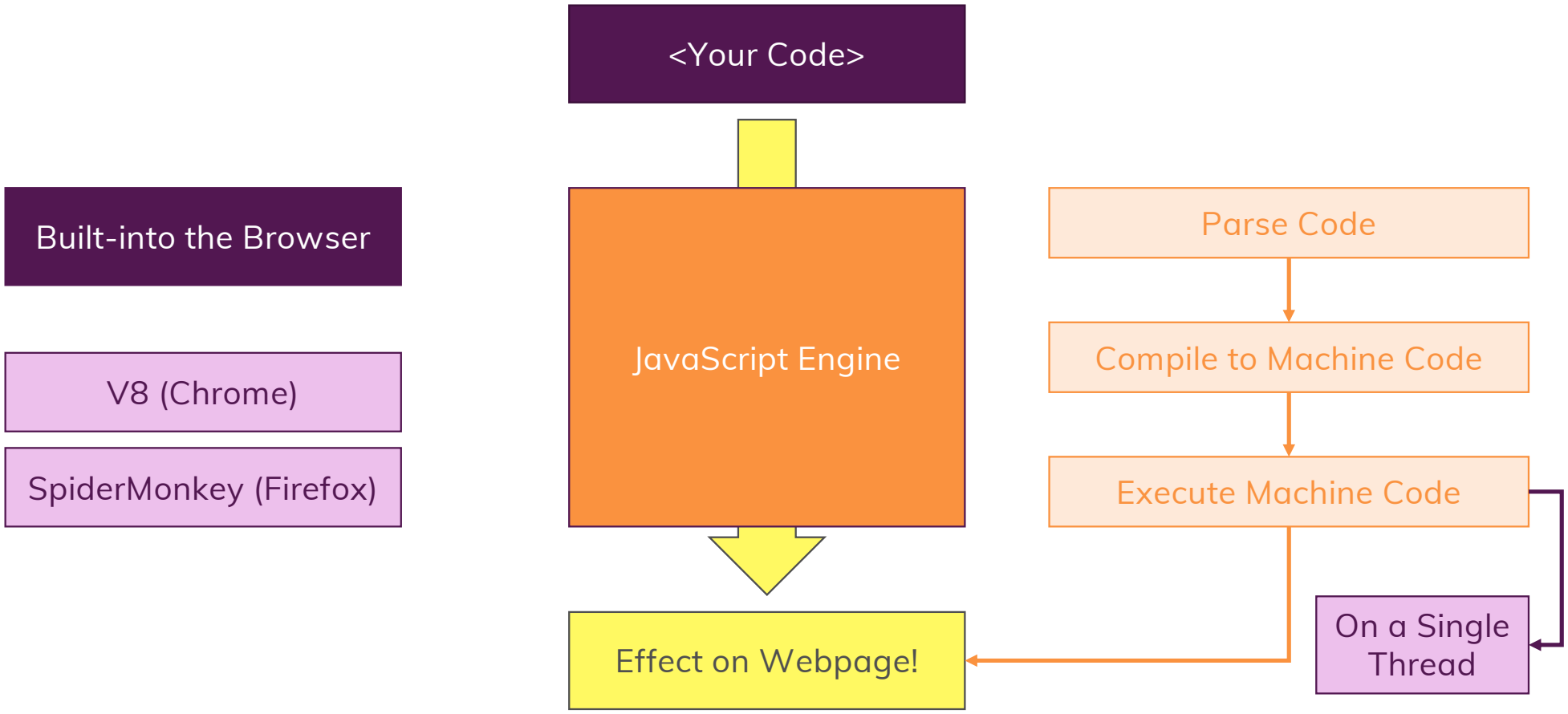
JavaScript was created **to make webpages more dynamic** (e.g. change content on a page directly from inside the browser). Originally, it was called LiveScript but due to the popularity of Java, it was renamed to JavaScript.

JavaScript is totally **independent** from Java and has **nothing in common with Java**!

How Do Webpages Work?



How Is JavaScript Executed?



Dynamic? Weakly Typed?

Dynamic, interpreted Programming Language		Weakly Typed Programming Language
Not pre-compiled, instead parsed and compiled “on the fly” (e.g. in the browser)		Data types are assumed (e.g. assigned to variables) automatically
Code evaluated and executed at runtime		You don’t define that some variable has to hold a certain value (e.g. a number)
Code can change at runtime (e.g. type of a variable)		Data types are not set in stone but can change

JavaScript Runs On A Host Environment

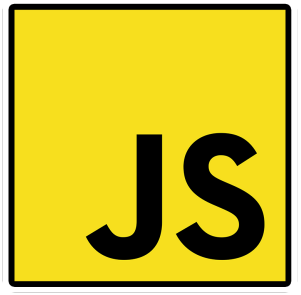
Browser-side	"Other" (e.g. Server-side)
JavaScript was invented to create more dynamic websites by executing in the browser!	Google's JavaScript Engine (V8) was extracted to run JavaScript anywhere (called "Node.js")
JavaScript can manipulate the HTML code, CSS, send background Http requests & much more	Node.js can be executed on any machine and is therefore often used to build web backends (server-side JavaScript)
JavaScript CAN'T access the local filesystem, interact with the operating system etc	Node.js CAN access the local filesystem, interact with the operating system etc. It CAN'T manipulate HTML or CSS

Separate Module!

What's In This Course?

Core Basics	Building a Strong Foundation	Advanced Concepts
Getting Started Language Basics, Base Syntax Efficient Development & Debugging Control Structures (if, Loops, ...) Behind the Scenes of JS A Closer Look at Functions DOM Basics Arrays & Iterables Objects	Classes & OOP Constructor Functions & Prototypes More about DOM & Browser APIs Events Functions Deep Dive More about Numbers & Strings Asynchronous Code Background Http (Ajax)	3rd Party Libraries JavaScript Modules Tooling (Webpack, Babel, ...) Working with Browser Storage Browser Support JavaScript Frameworks Meta-programming NodeJS Introduction Security Deployment Performance Optimizations & Memory Leaks

JavaScript and Java



Object-oriented, strongly types

Totally independent programming languages with different syntax and principles

Flexible, weakly types

JavaScript was named JavaScript to ”sound cool”

Java does NOT run in the browser, JavaScript does

A Brief Overview Of The JavaScript History

1995	Netscape introduces “LiveScript” / “JavaScript”
1996	Microsoft releases its own version for IE
Late 1996	JavaScript submitted to ECMA International to start standardization
1997 - 2005	Standardization efforts, Microsoft didn’t really join and support the standardized JS version though
2006 - 2011	Huge progress in JavaScript ecosystem, Microsoft eventually joined forces

Variables & Constants

```
let userName = 'Max';
```

A “data container” / “data storage”

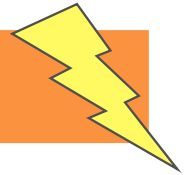
```
userName = 'Manu';
```

...where the value can change!

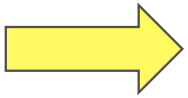
```
const totalUsers = 15;
```

A “data container” / “data storage”

```
totalUsers = 20;
```



...where the value must not change!



Use **constants as often as possible** (i.e. whenever you actually got data that never changes) to be clear about your intentions (in your code).

Variable Naming

Allowed		Not Allowed / Not Recommended	
`let userName`	Best Practice: **camelCase**	`let user_name`	Allowed but bad practice!
`let ageGroup5`	Only letters and digits	`let 21Players`	Starting digits not allowed
`let $kindOfSpecial`	Starting with $ is allowed	`let user-b`	No special characters!
`let _internalValue`	Starting with _ is allowed	`let let`	Keywords not allowed

Operators

+	Add two numbers
-	Subtract two numbers
*	Multiply two numbers
/	Divide two numbers
%	Divide two numbers, yield remainder
**	Exponentiation (e.g. 2 ** 3 = 8)

=	Assign value to variable

Functions

"Code on Demand"

(1) Define Function

```
function greetUser(name) {
  alert('Hi ' + name);
}
```

You can (but don't have to) use ***parameters*** (e.g. *name*) and you can (but don't have to) return values (via ***return***).

(2) Call Function

```
greetUser('Max');
```

As often as you want, passing in (different) parameter values!

Every function execution then runs independent from (possible) other executions.

Operators

+	Add two numbers
-	Subtract two numbers
*	Multiply two numbers
/	Divide two numbers
%	Divide two numbers, yield remainder
**	Exponentiation (e.g. 2 ** 3 = 8)

=	Assign value to variable
+=, -=, ...	Perform calculation and re-assign result to variable
++, --	Increment / Decrement variable value + re-assign

ACADEMIND

Data Types

Numbers	2, -3, 22.956	Important for calculations and code where you need to “work with a number”
Strings (Text)	'Hi', "Hi", `Hi`	Important for outputting results, gathering input
Booleans	true / false	Important for conditional code and situations where you only have 2 options
Objects	{ name: 'Max', age: 31 }	Important for grouped/ related data, helps you with organizing data
Arrays	[1, 3, 5]	Important for list data, unknown amounts of data

null / undefined / NaN

Special Values

undefined	null	NaN (Not a type!)
Default value of uninitialized variables	Never assumed by default	Technically, it's of type number and can therefore be used in calculations
You shouldn't assign undefined as a value manually	You can assign this is a value if you want to "reset" / "clear" a variable	It yields a new NaN and it's the result of invalid calculations (e.g. 3 * 'hi')

NOT entirely equal!

Different Ways of Adding JavaScript to a Page

Use this approach!

Inline JavaScript Code	External / Imported JavaScript Files

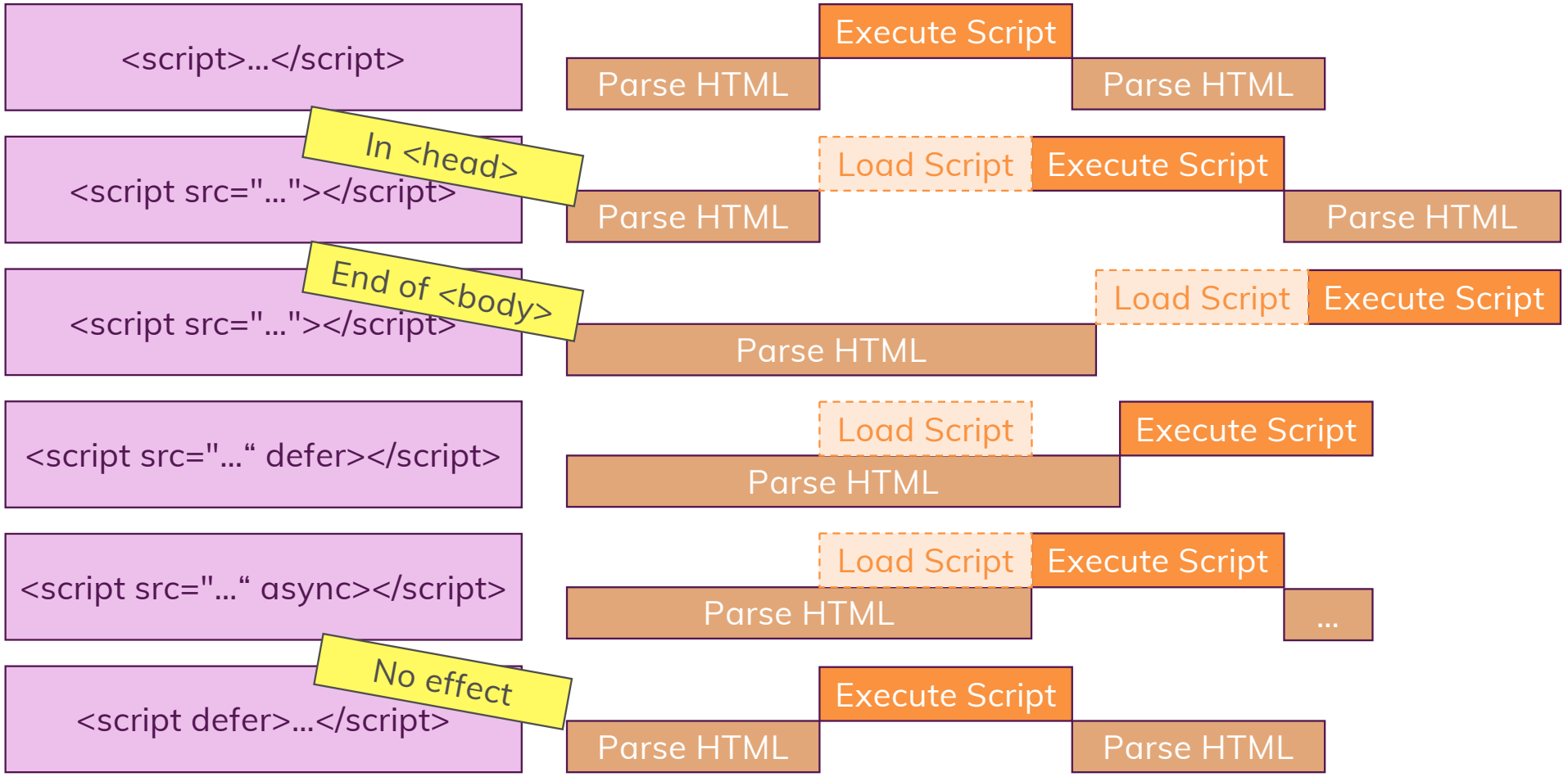
`<script>`
`alert('Hi there!');`
`</script>`
	`<script src="someFile.js"></script>`
	`<script src="otherFile.js"></script>`
	`<script src="otherFile.js"></script>`
	Multiple imports (even of the same file) are possible and normal!

Executed directly when the browser (HTML parser) reaches this tag	Requested & loaded when browser (HTML parser) reaches this tag
Typically only used for very trivial sites/ scripts (you'd create a huge HTML file otherwise)	Execution behavior depends on configuration: **async** and **defer** allows HTML parser to continue

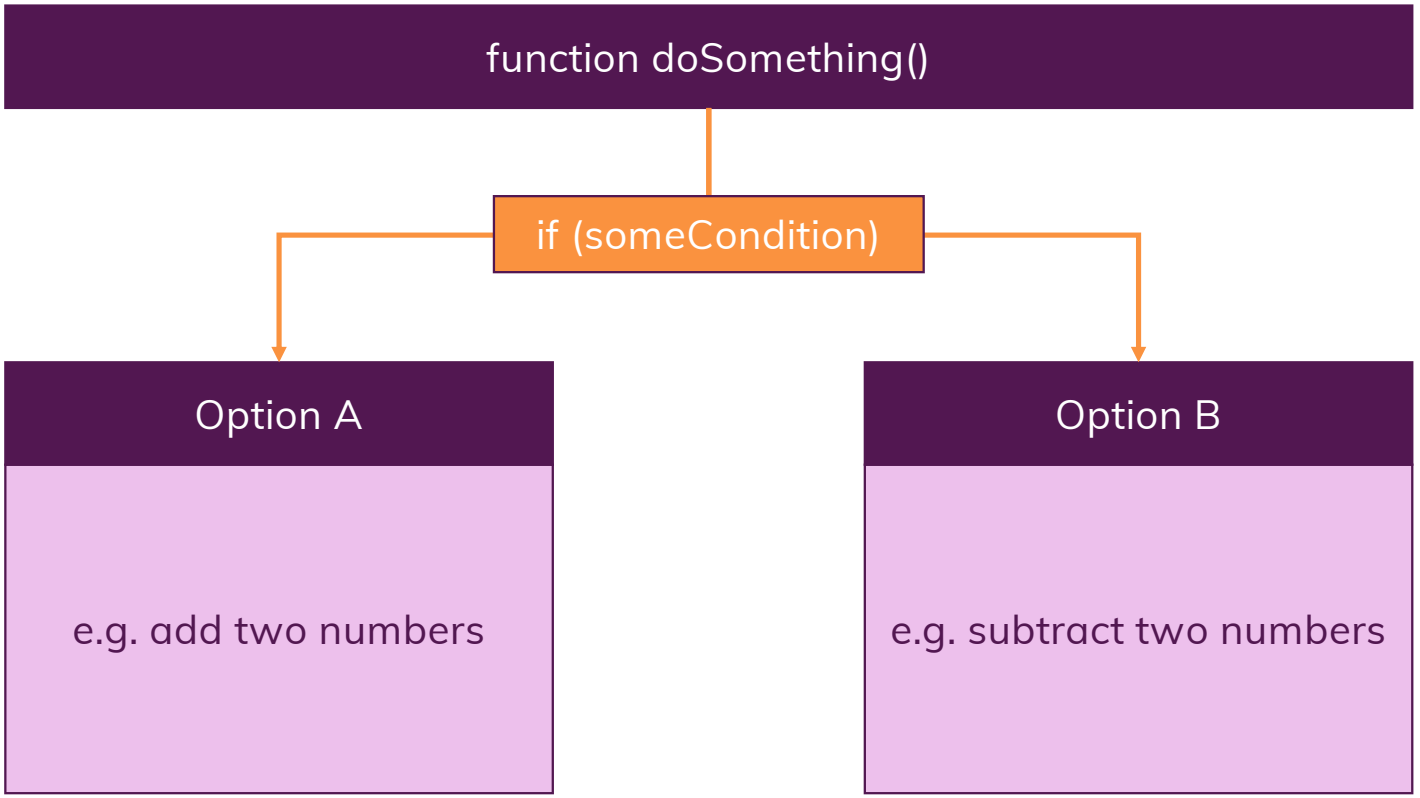
How To Import Your Scripts

Inline	<script> alert('Hi!'); </script>	Immediatley executed, blocks HTML parsing & rendering
External File	<script src="file.js"> </script>	Immediatley loaded & executed, blocks HTML parsing & rendering
External File (async)	<script src="file.js" async> </script>	Immediatley loaded & executed thereafter, blocks HTML parsing & rendering
External File (defer)	<script src="file.js" defer> </script>	Immediatley loaded & executed after HTML parsing & rendering

Timeline Summary



Conditional Code Execution

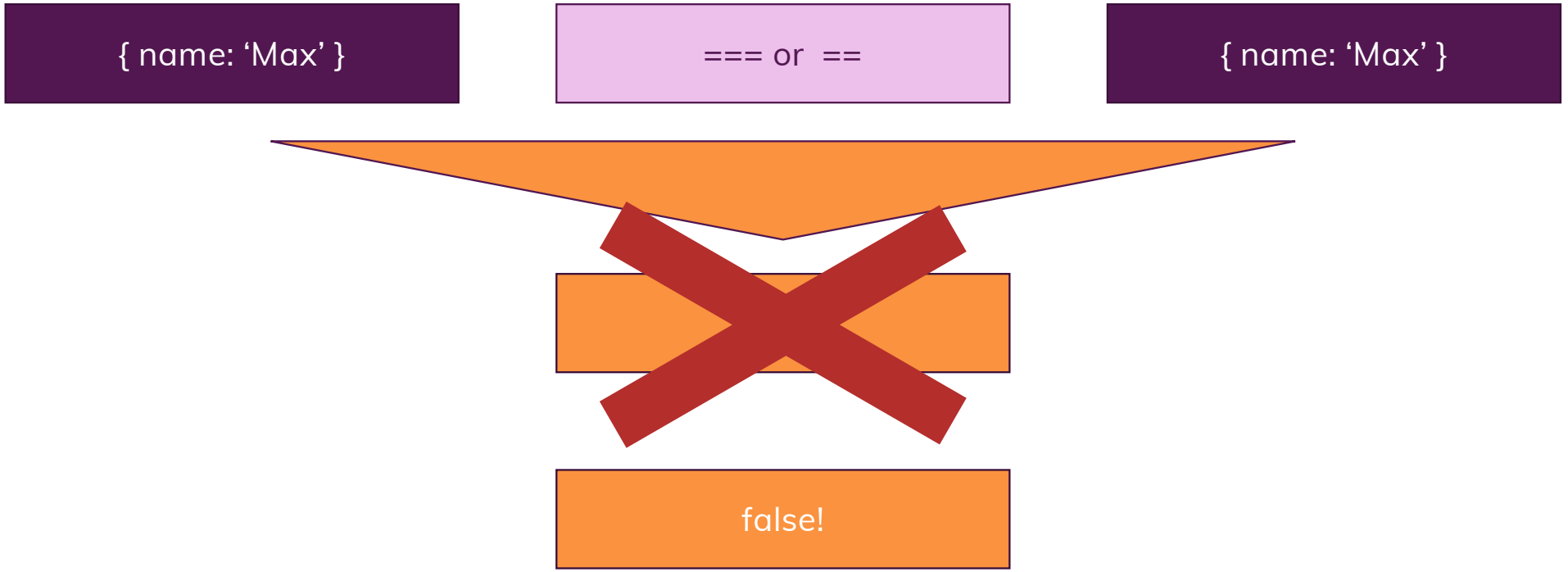


Boolean Operators

Important For Conditional Code: Return **true** or **false**

==	Check for value equality	e.g. a == b
!=	Check for value inequality	e.g. a != b
=== and !== (Prefer over ==)	Check for value AND type (in)equality	e.g. a === b / a !== b
> & <	Check for value being greater / smaller	e.g. a > b / a < b
>= & <=	Check for value being greater or equal / smaller or equal	e.g. a >= b / a <= b
!	Check if NOT true	e.g. !a

Beware of Objects & Arrays in Comparisons!



Objects and arrays are kind of special in JavaScript!

Combining Conditions

Condition A	AND	Condition B	OR	Condition C
name === 'Max'	&&	age === 30	||	isAdmin

Part 1

Evaluated together (yields true if each condition yields true)

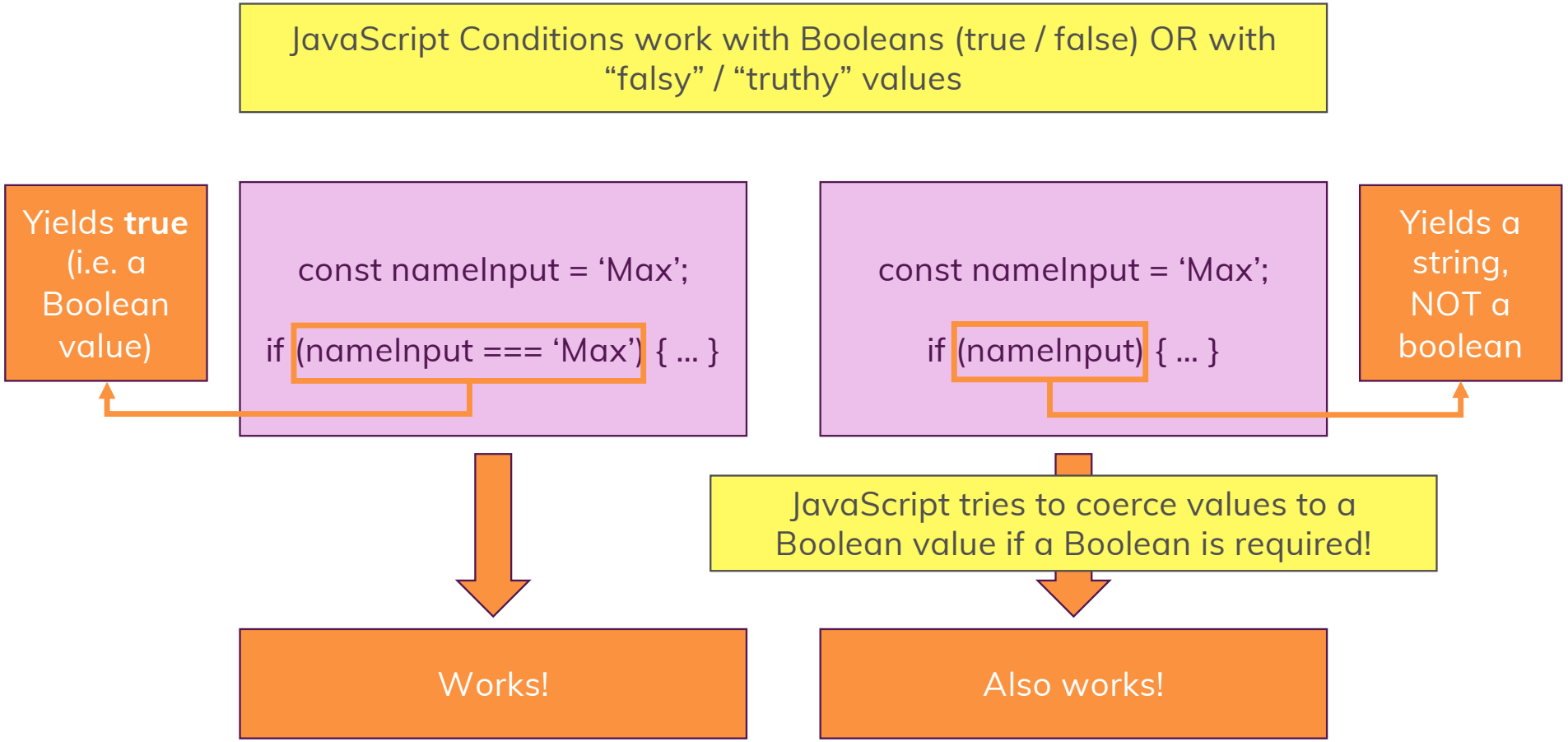
Part 2

Evaluated as an alternative

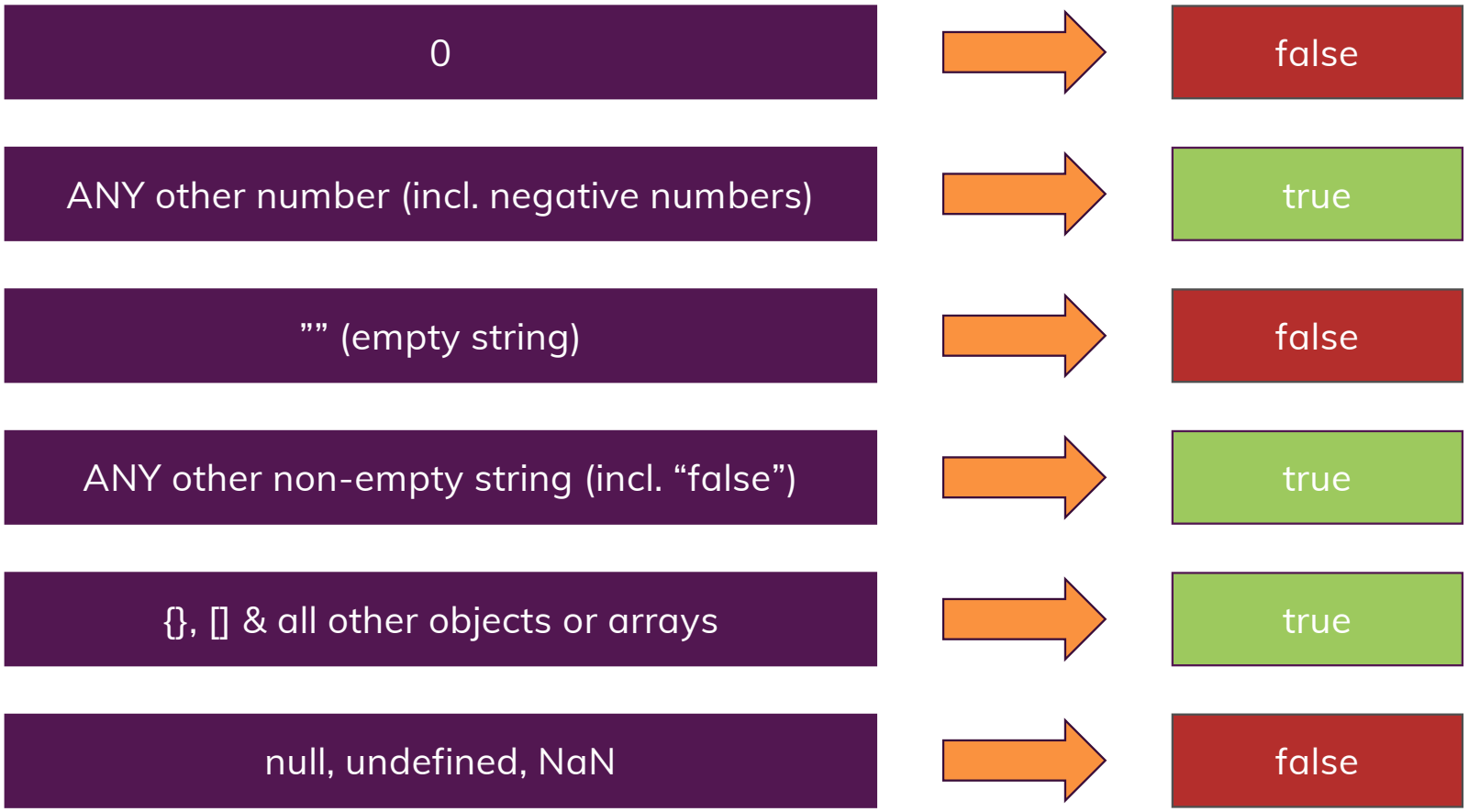
Yields true if Part 1 OR Part 2 yields true

You can use parentheses to control what's evaluated together!

Falsy and Truthy Values



Falsy and Truthy Values



Conditional Expressions / Ternary Operator

if statements return no values!

This will NOT work!

```
const userName = if (isLogin) {
  return 'Max';
} else {
  return null;
}
```

Use the ternary operator in such cases

```
const userName = isLogin ? 'Max' : null
```

- Condition (can be written exactly like in if statements)
- Value if condition is true / truthy
- Value if condition is false / falsy

ACADEMIND

“Boolean Tricks” with Logical Operators

Boolean Coercion via double NOT (double bang) operator	!!	e.g. !!“”, e.g. !!1	false, true
Default value assignment via OR operator	||	e.g. const name = someInput || ‘Max’	someInput if not falsy, ‘Max’ otherwise
Use value if condition is true via AND operator	&&	e.g. const name = isLoggedIn && ‘Max’	‘Max’ is set if isLoggedIn is true, false otherwise

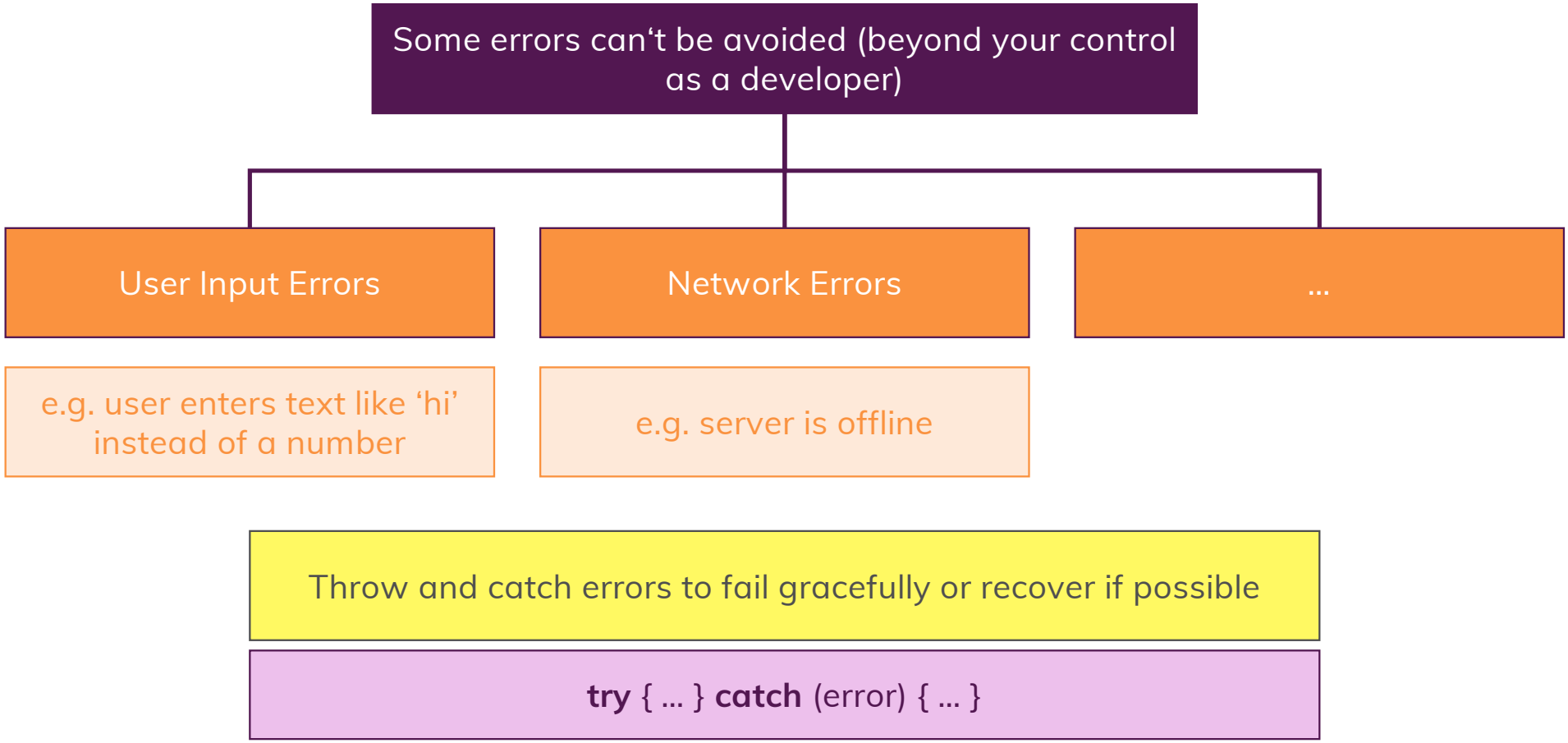
Loops

Execute code multiple times

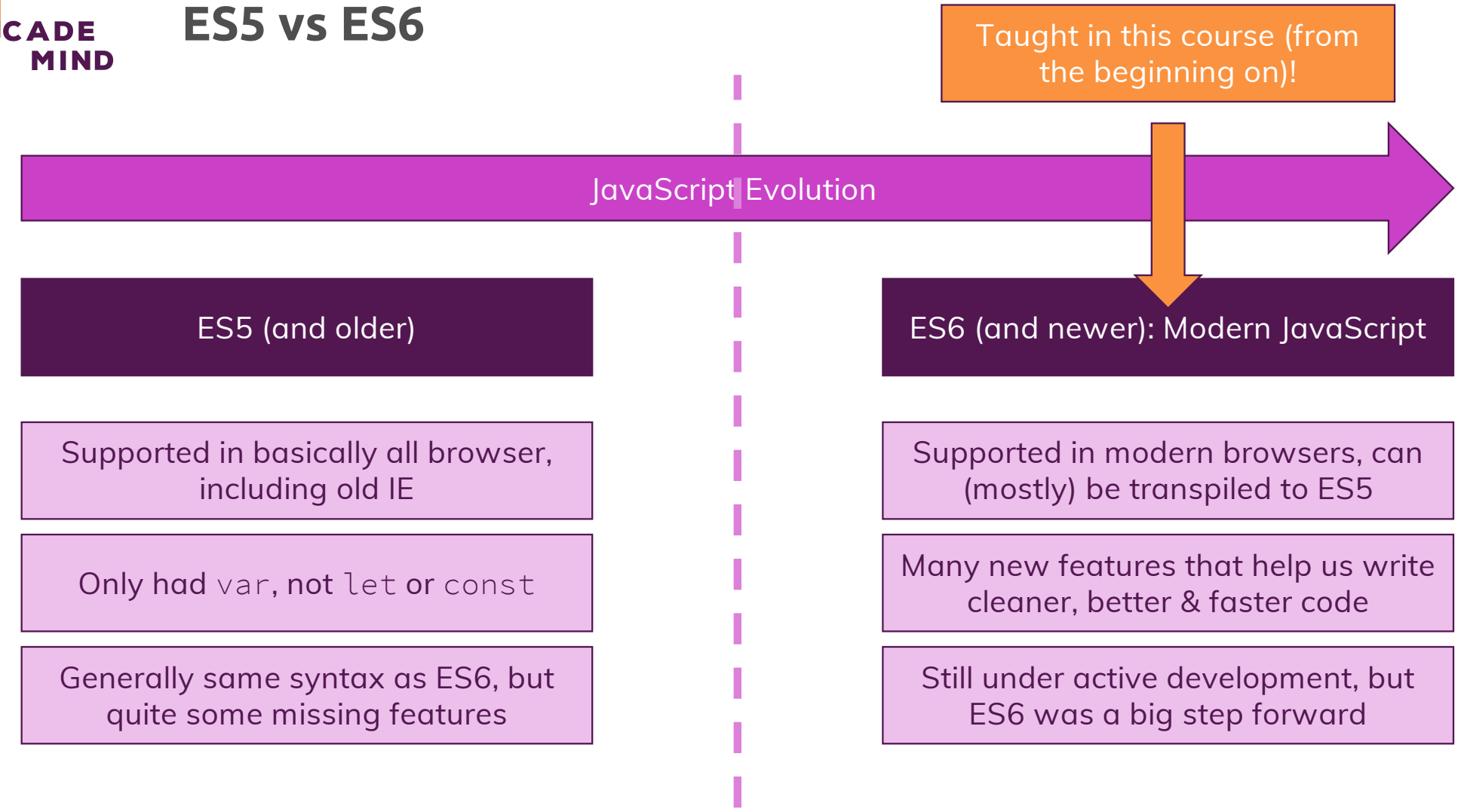
for loop	for-of loop	for-in loop	while loop
Execute code a certain amount of times (with counter variable)	Execute for every element in an array	Execute for every key in an object	Execute code as long as a condition is true

for (let i = 0; i < 3; i++)
{
  console.log(i);
}
	for (const el of array)
	{
	  console.log(el);
	}
		for (const key in obj) {
		  console.log(key);
		  console.log(obj[key]);
		}
			while (isLoggedIn) {
			  ...
			}

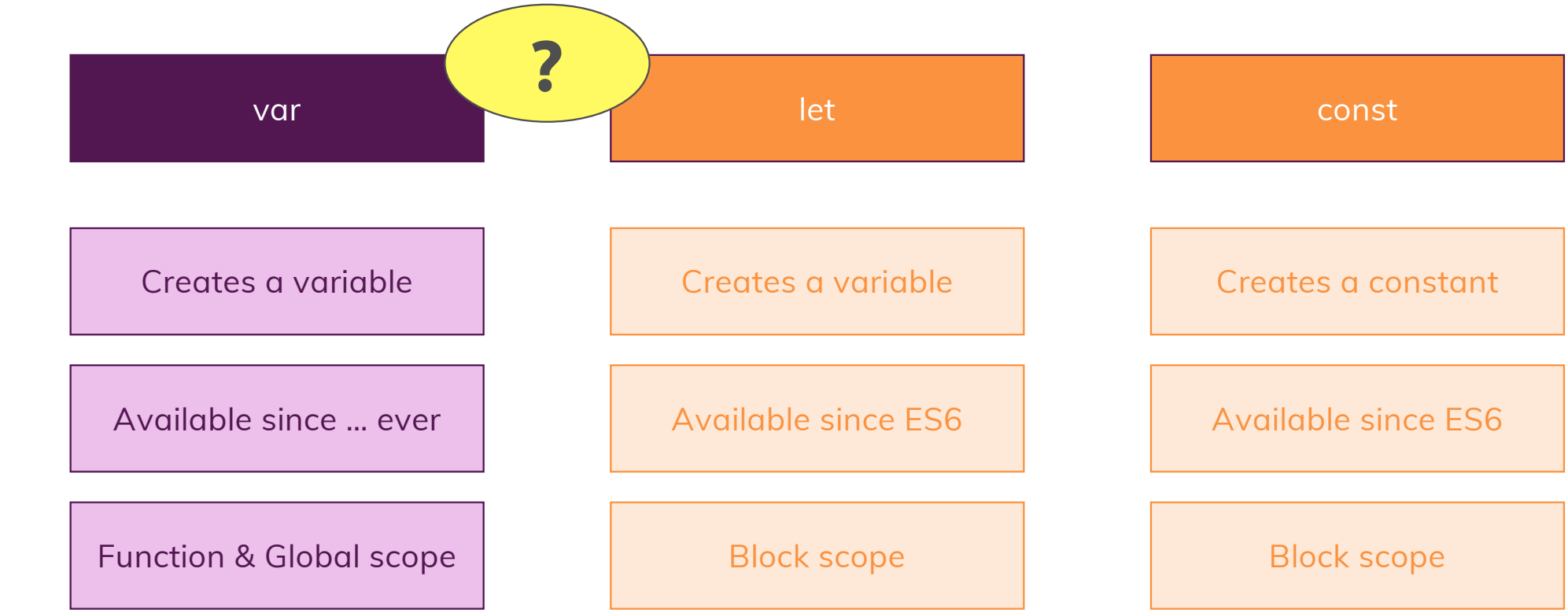
Error Handling



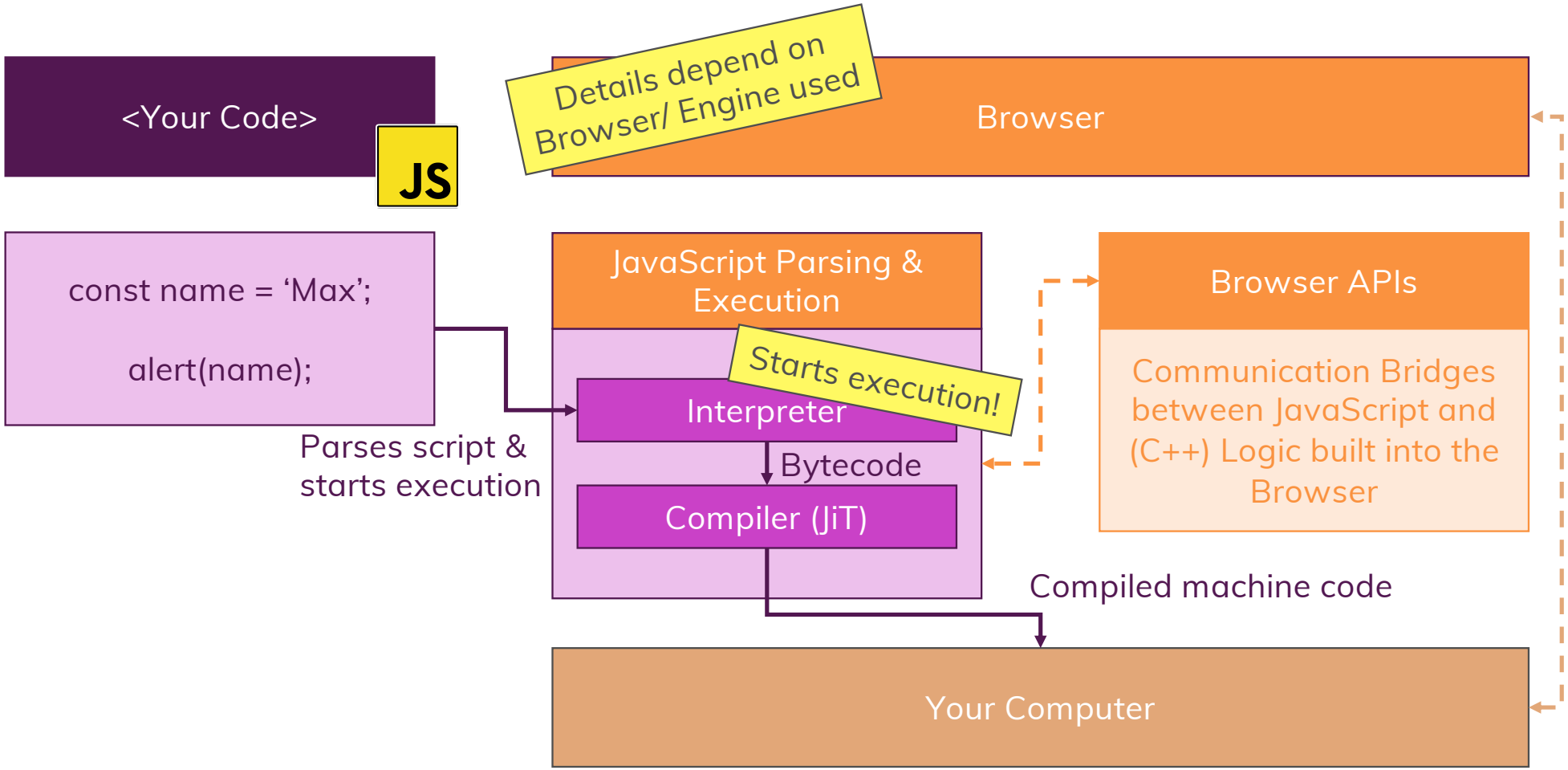
ES5 vs ES6


Taught in this course (from the beginning on)!
JavaScript Evolution
ES5 (and older)
ES6 (and newer): Modern JavaScript
Supported in basically all browser, including old IE
Only had var, not let or const
Generally same syntax as ES6, but quite some missing features
Supported in modern browsers, can (mostly) be transpiled to ES5
Many new features that help us write cleaner, better & faster code
Still under active development, but ES6 was a big step forward

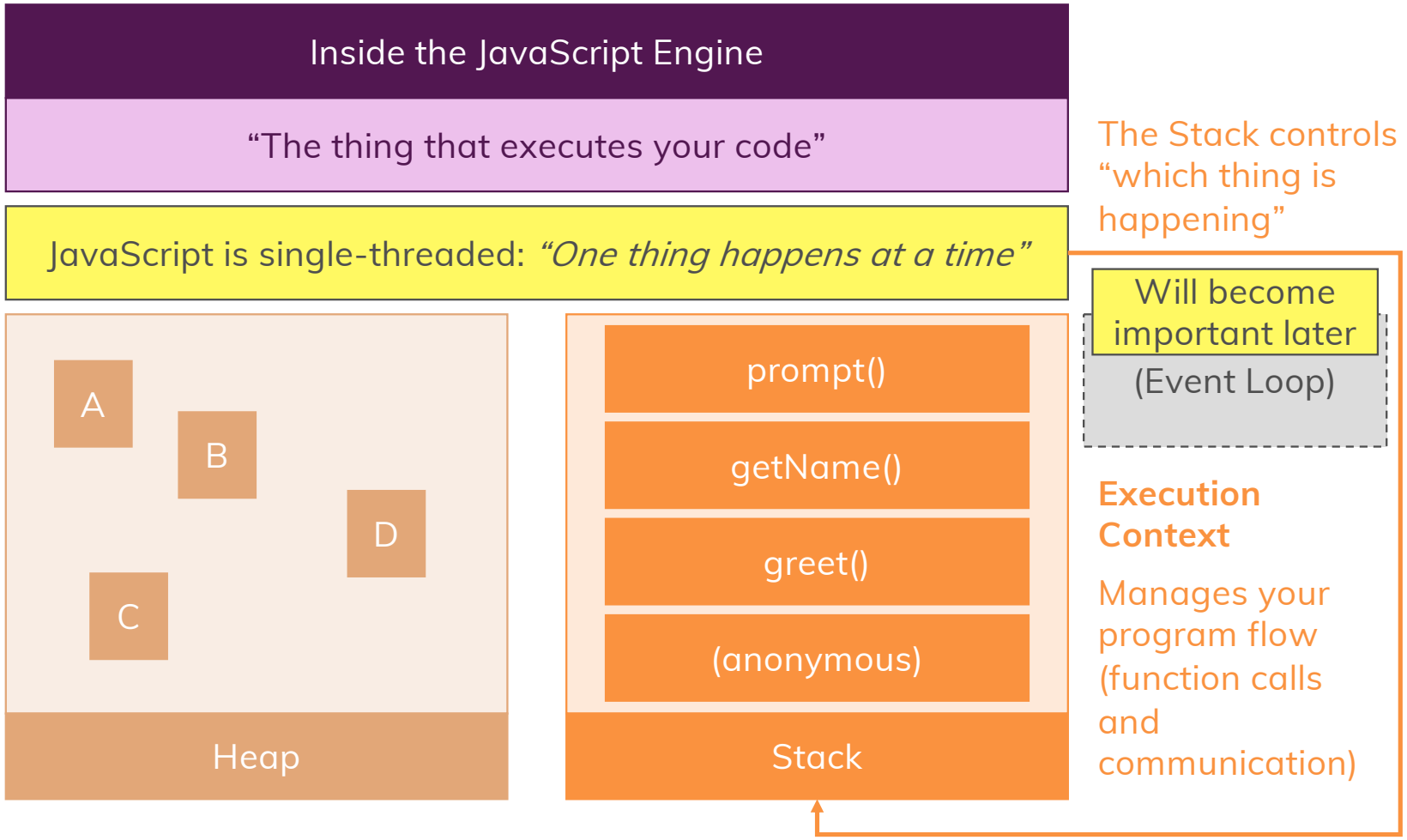
var vs let vs const



JavaScript Engines & What They Do



How Code Gets Executed

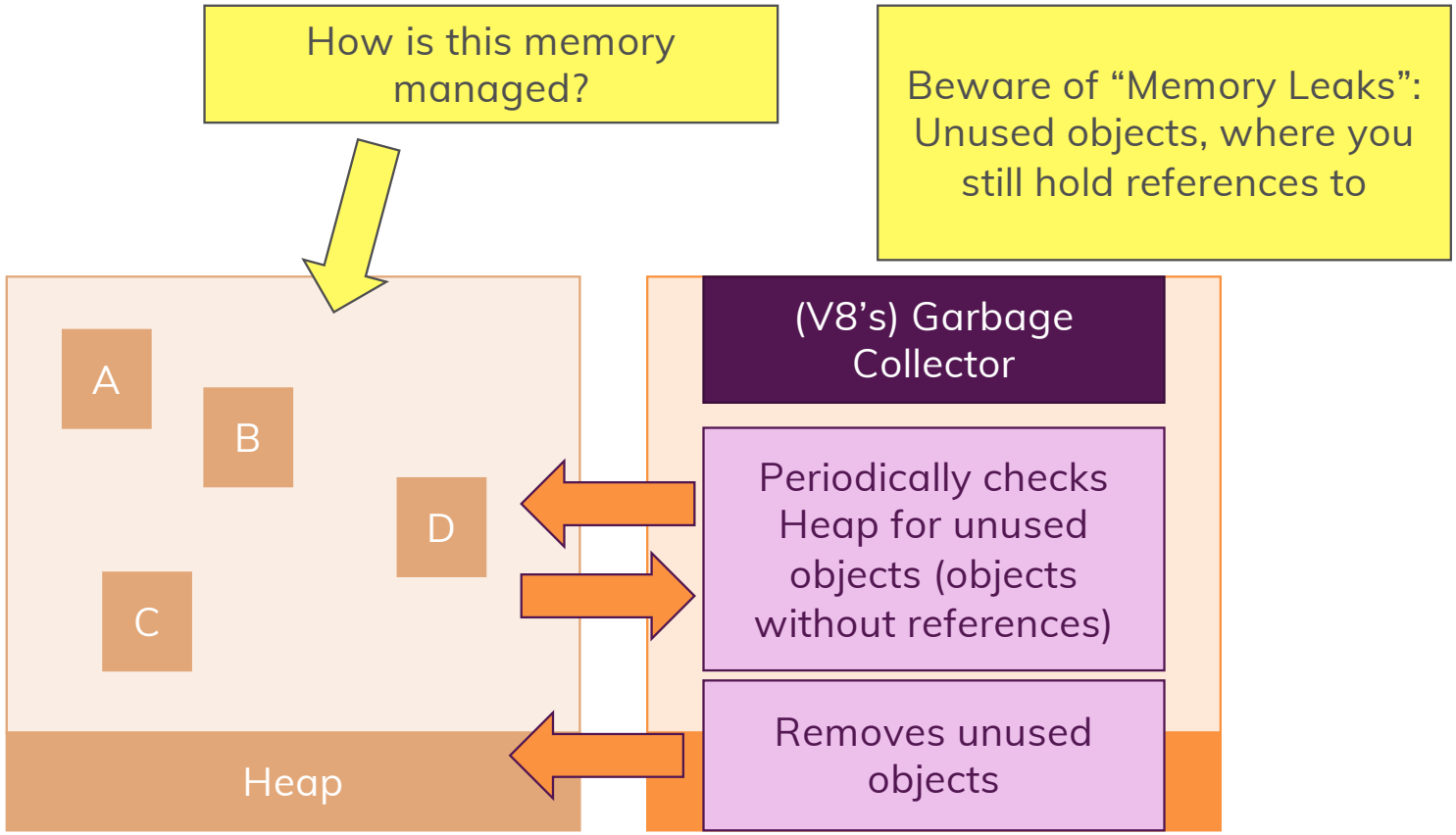


Primitive vs Reference Values

Two Categories of Types/ Values in JavaScript

Primitive Values	Reference Values
Strings, Numbers, Booleans, null, undefined, Symbol	All other objects ("more expensive to create")
Stored in memory (normally on Stack), variable stores value itself	Stored in memory (Heap), variable stores a pointer (address) to location in memory
Copying a variable (= assign to different variable) **copies the value**	Copying a variable (= assign to different variable) **copies the pointer/ reference**

Garbage Collection



A Different Way of Defining Functions

Function Declaration / Function Statement	Function Expression

`function multiply(a, b) {`
`  return a * b;`
`}`
	`const multiply = function(a, b) {`
	`  return a * b;`
	`}`

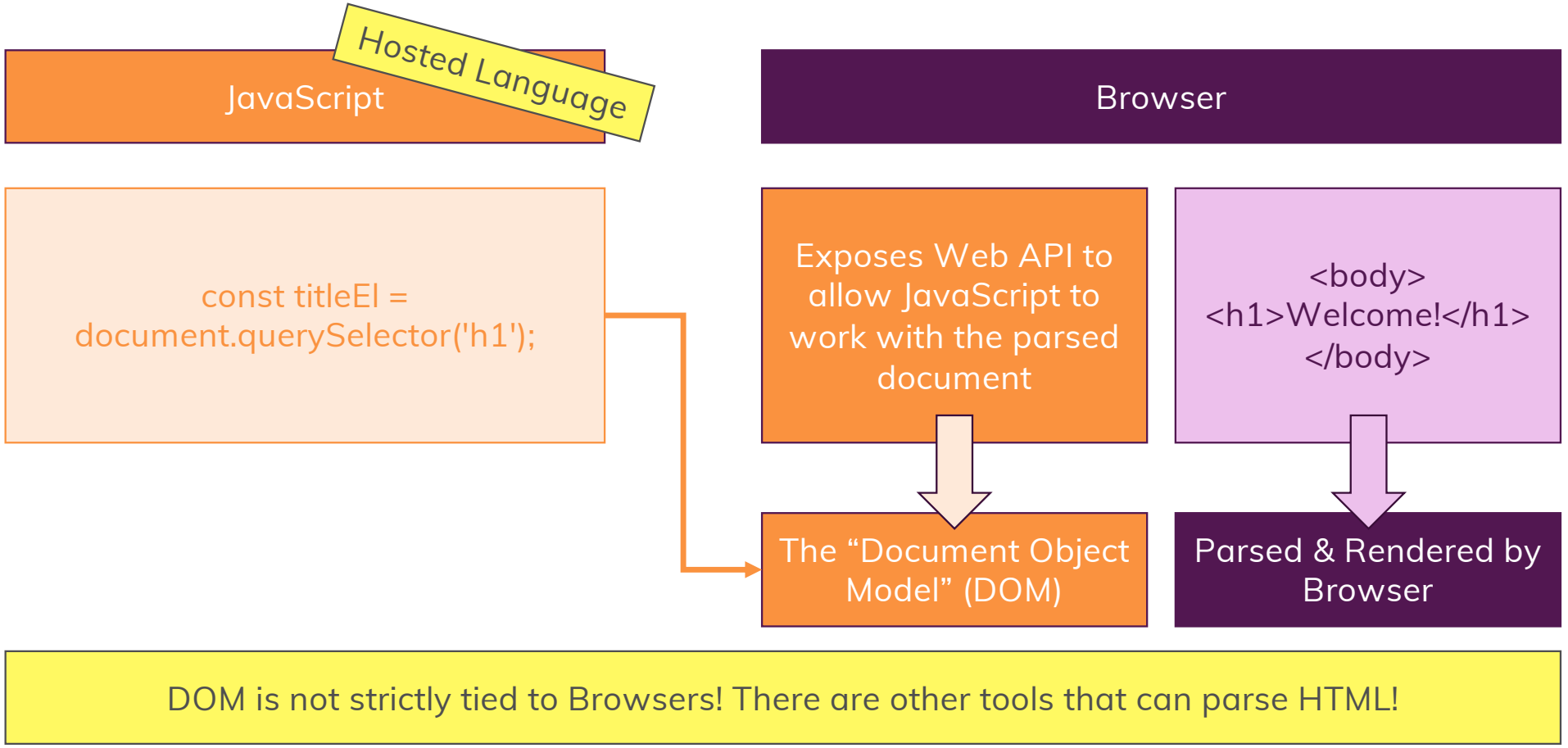
Hoisted to top, can be declared anywhere in the file (i.e. also after it's used)	Hoisted to top but **not initialized**/ defined, **can't** be declared anywhere in the file (i.e. not after it's used)

Arrow Functions

General Syntax: `(arg1, arg2) => { … }`

No Arguments / Parameters	`() => { … }`	Empty pair of parentheses is required
Exactly one (1) Argument / Parameter	`arg => { … }`	Parentheses can be omitted
Exactly one expression in function body	`(a, b) => a + b`	Curly braces can be omitted, result is returned
More than one expression in function body	`(a, b) => { a *= 2; return a + b; }`	Curly braces and return statement required

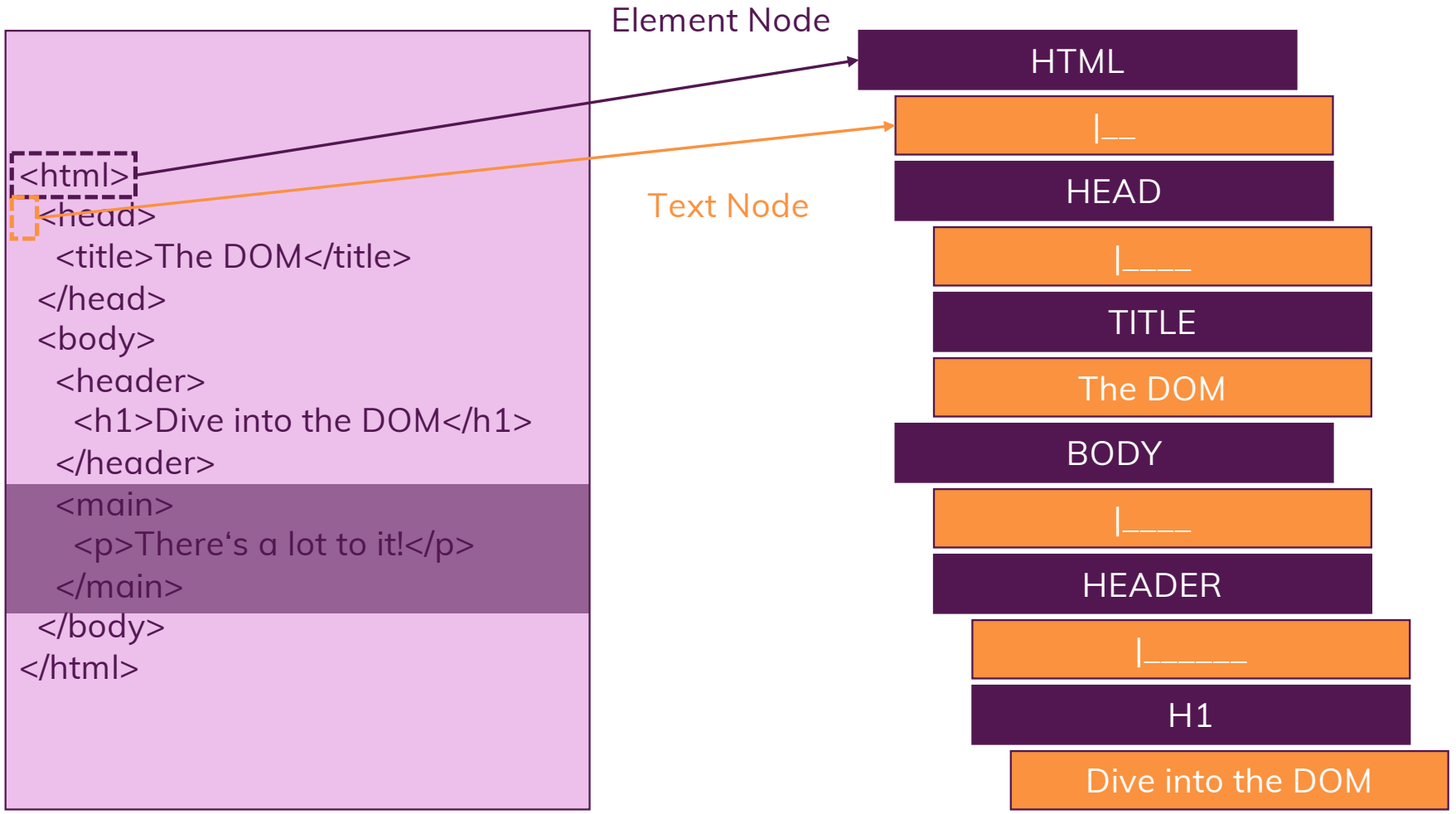
The Document Object Model (DOM)



document & window

document	window
Root DOM Node	The active Browser Window / Tab
Provides access to element querying, DOM content etc	Acts as global storage for script, also provides access to window-specific properties and methods

The Document Object Model (DOM)



Nodes & Elements

Nodes

- The objects that make up the DOM
- HTML tags are “element nodes” (or just “elements”)
- Text creates “text nodes”
- Attributes create “attribute nodes”

Elements are one type of nodes

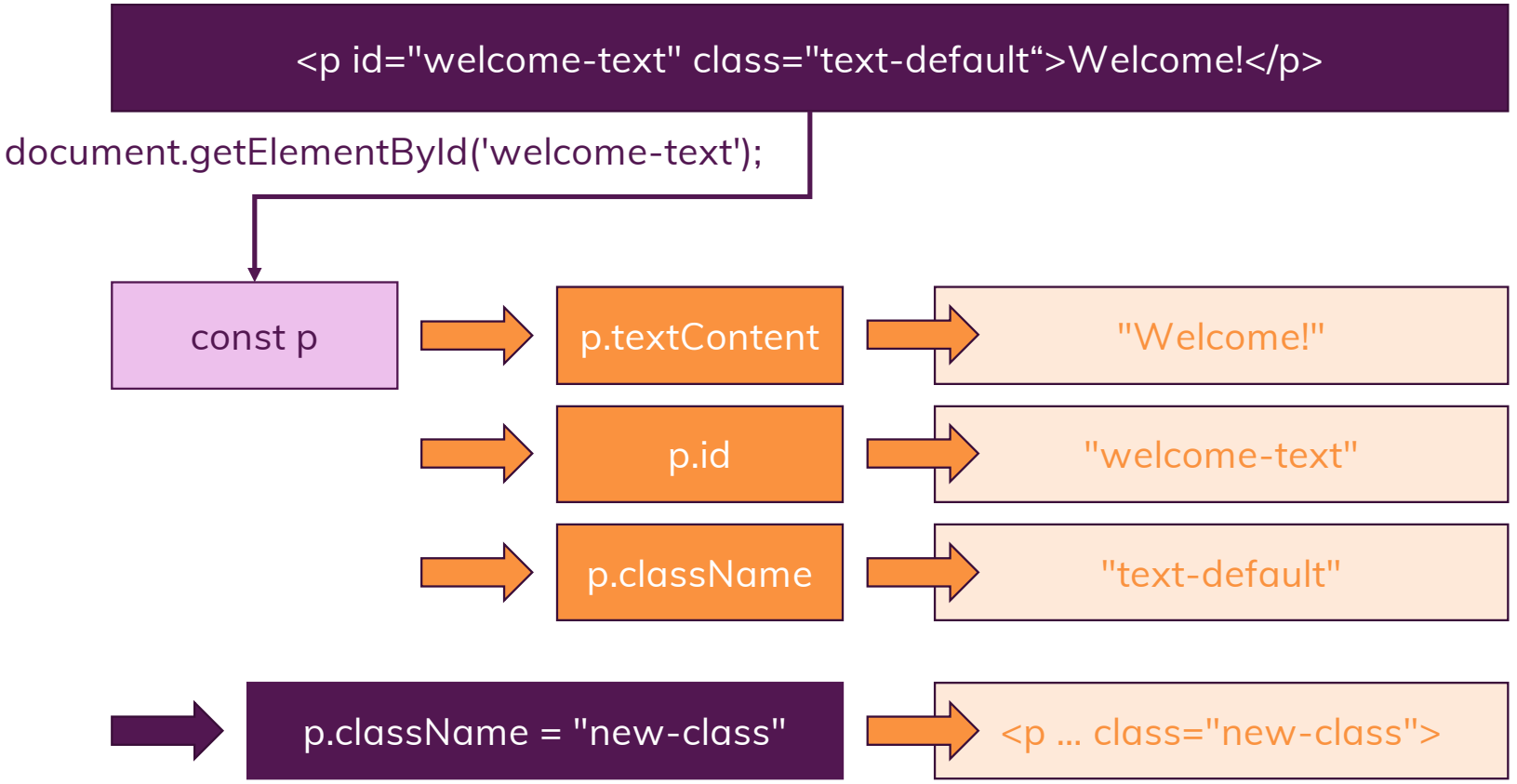
Elements

- Special properties and methods to interact with the elements
- Available methods and properties depend on the kind of element
- Can be selected in various different ways (via JavaScript)
- Can be created and removed via JavaScript

Querying Elements

querySelector(), getElementById()	querySelectorAll(), getElementsByTagName(), ...
Return single elements	Return collections of elements (array-like objects): NodeList
Different ways of querying elements (by CSS selector, by ID)	Different ways of querying elements (by CSS selector, by tag name, by CSS class)
Direct reference to DOM element is returned	querySelectorAll() returns a non-live NodeList, getXByY return live NodeLists

Evaluating & Manipulating Elements



Attributes vs Properties

Often (but not always!), attributes are mapped to properties and "live synchronization" is set up.

```
<input id="input-1" class="input-default" value="Enter text...">
```

Attributes (placed in HTML code, on element tags)

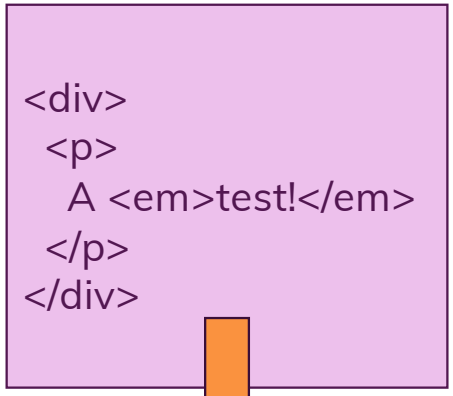
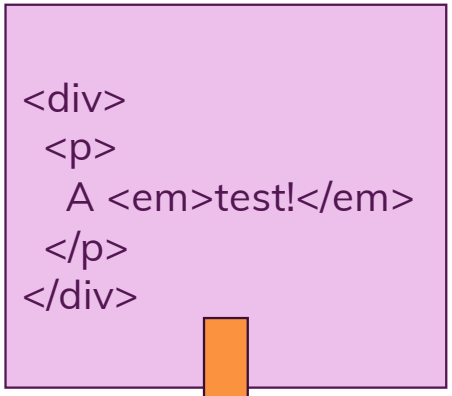
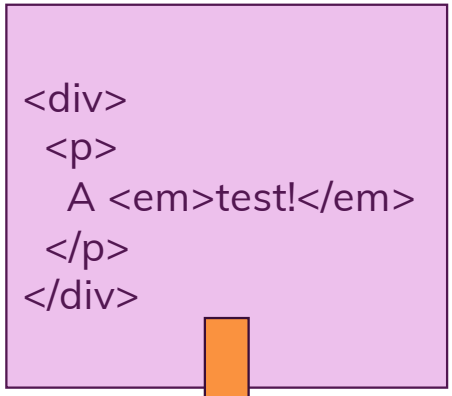
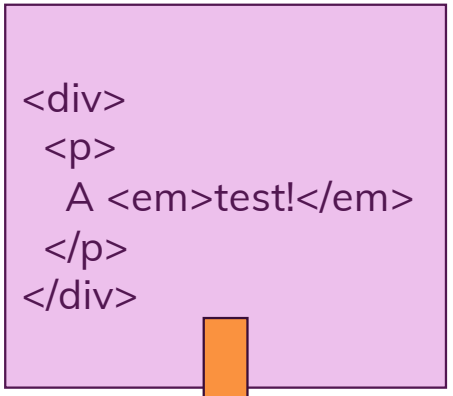
const input

1:1 mapping (+ live-sync)	input.id
Different names (but live-sync)	input.className
1:1 mapping (1-way live-sync)	input.value

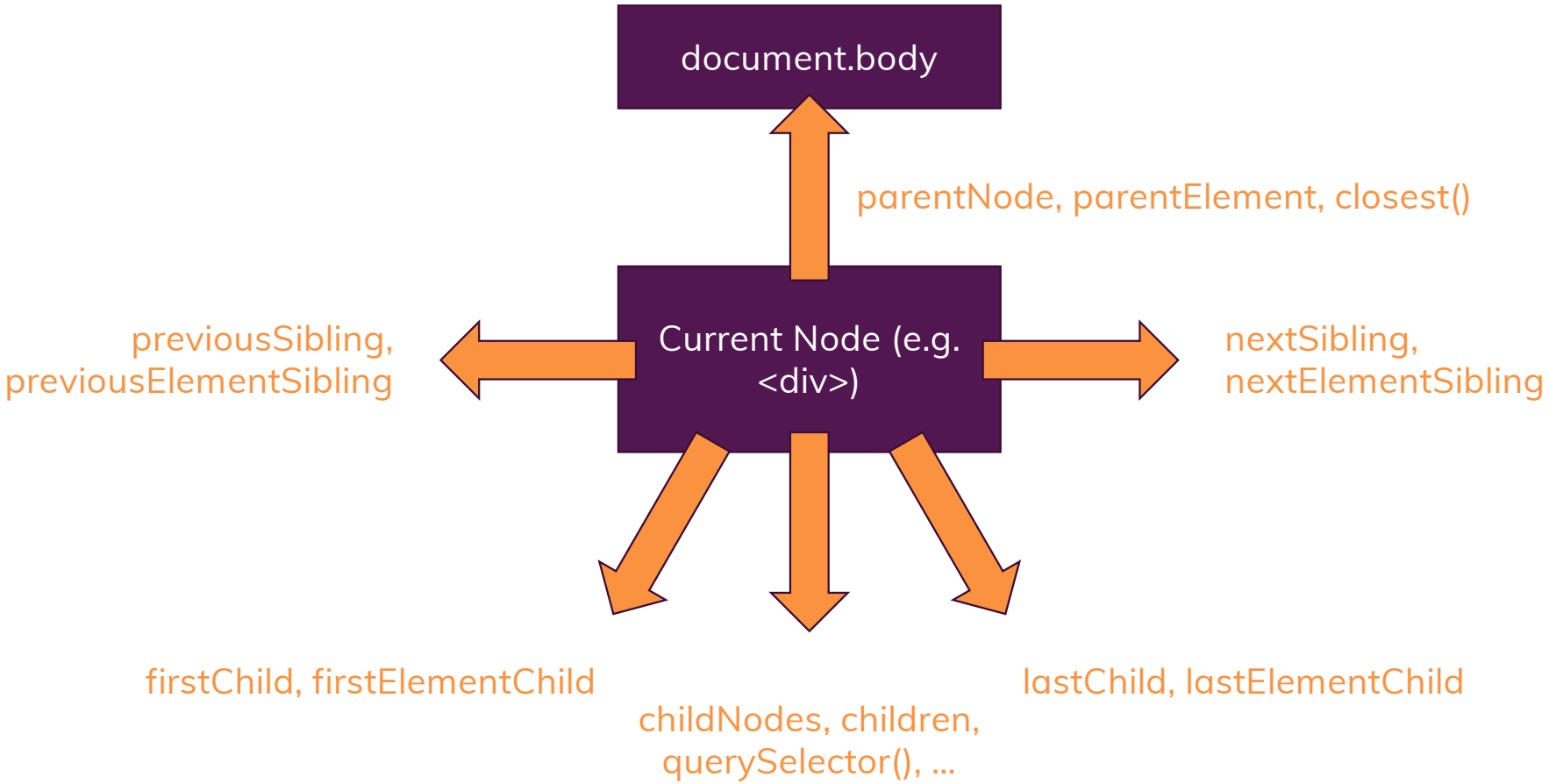
MDN for full reference of available properties + their behavior!

Properties (automatically added on created DOM objects)

Children, Descendants, Parent & Ancestors

Child	Descendant	Parent	Ancestor
Direct child node or element	Direct or indirect child node or element	Direct parent node or element	Direct or indirect parent node/ element
			
<p> is a child of . isn't!	<p> is a descendant of . So is !	is a parent of <p> but ot of !	is an ancestor of <p> and of !

Traversing the DOM



Styling DOM Elements

Via style Property	Via className	Via classList
Directly target individual CSS styles (on the element)	Directly set the CSS classes assigned to the element	Conveniently add, remove or toggle CSS classes
Controls styles as inline styles on the element	Set/ Control all classes at once	Fine-grained control over classes that are added
Style property names are based on CSS properties but have adjusted names (e.g. backgroundColor)	You can also control the id or other properties	Can be used with className (with care)

Creating & Inserting Elements

HTML string	innerHTML	Add (render) HTML string
	insertAdjacentHTML()	Add (render) HTML string in specific position
createElement()	appendChild() / append()	Append new DOM element/ node
	prepend(), before(), after(), insertBefore()	Insert new DOM element/ node in specific position
	replaceChild(), replaceWith()	Replace existing DOM element/ node with new one

Insertion & Removal Methods

appendChild()

insertAdjacentElement()

replaceChild()

removeChild()

Broad browser support

Plenty of resources on the internet

NOT supported in all browsers

Seen in some tutorials but not most common option

append()

prepend()

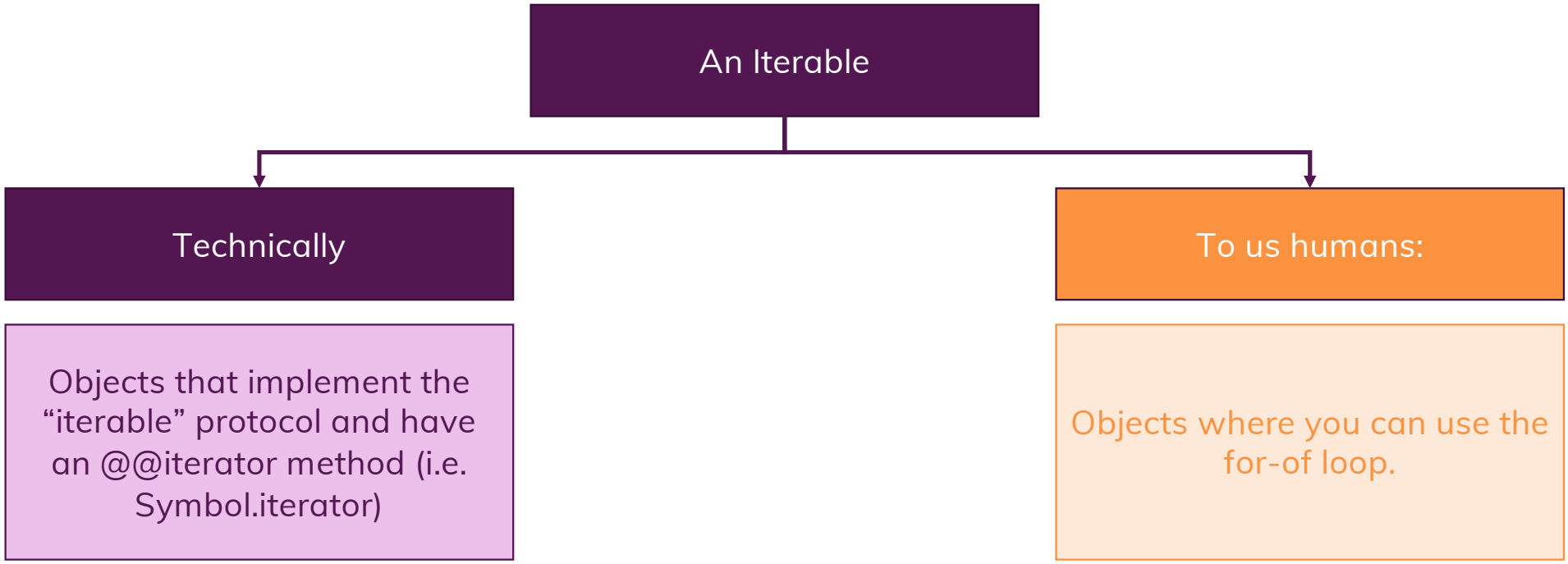
before()

after()

replaceWith()

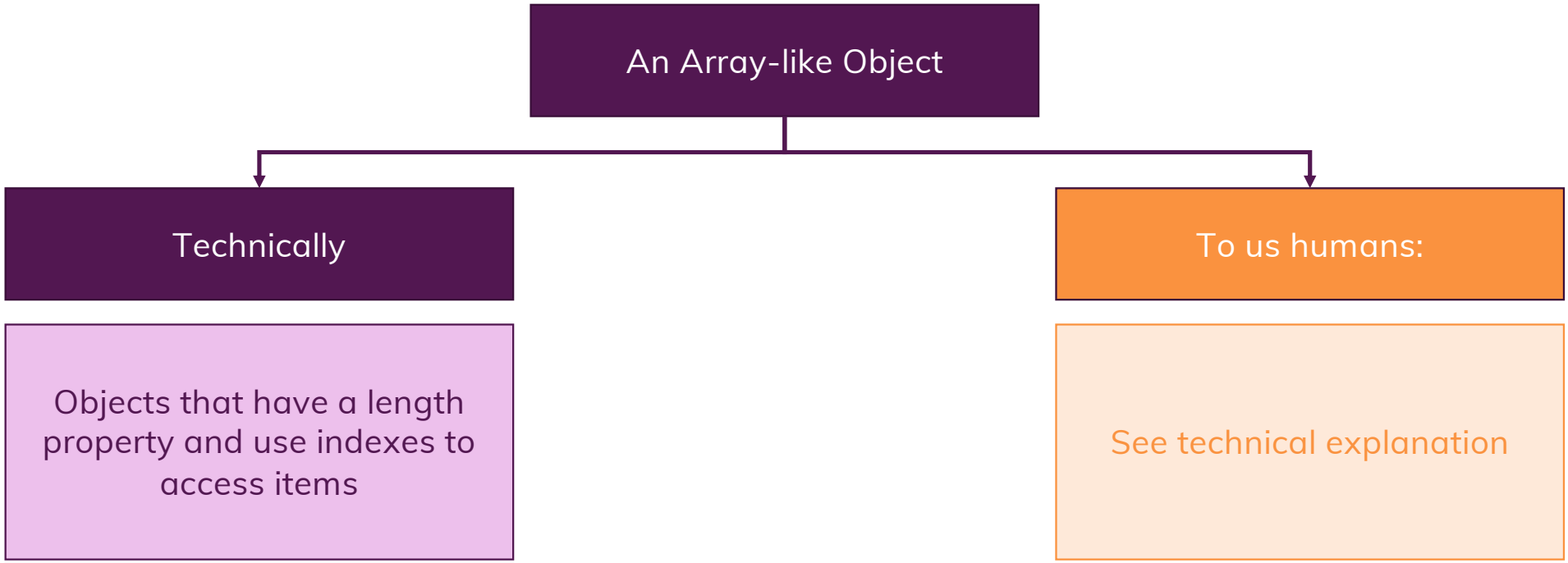
remove()

What's an "Iterable"?



Not every iterable is an array! Other iterables are (for example): NodeList, String, Map, Set

What's an "Array-like Object"?



Not every array-like object is an array! Other array-likes are (for example): NodeList, String

ACADEMIND

Maps & Sets

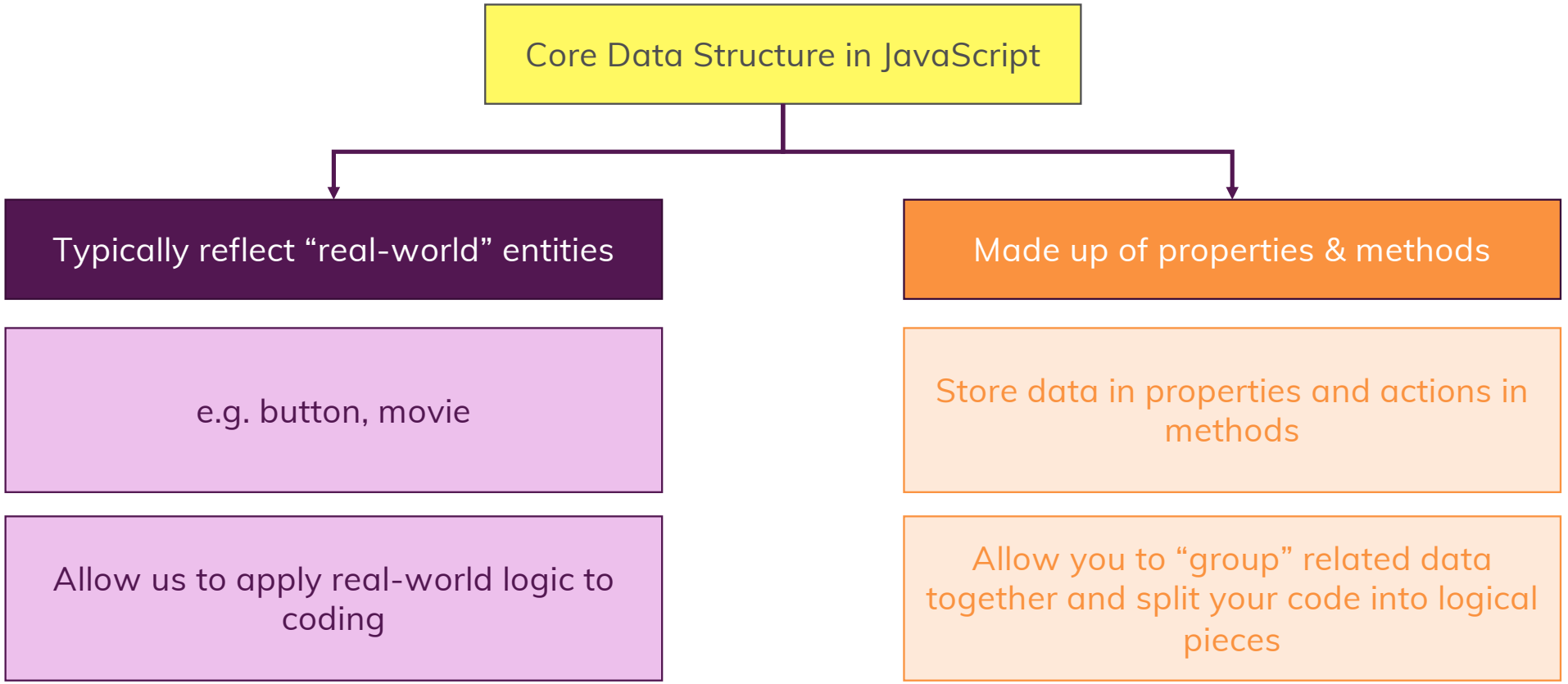
Arrays	Sets	Maps
Store (nested) data of any kind and length	Store (nested) data of any kind and length	Store key-value data of any kind and length, any key values are allowed
Iterable, also many special array methods available	Iterable, also some special set methods available	Iterable, also some special map methods available
Order is guaranteed, duplicates are allowed, zero-based index to access elements	Order is NOT guaranteed, duplicates are NOT allowed, no index-based access	Order is guaranteed, duplicate keys are NOT allowed, key-based access

ACADEMIND

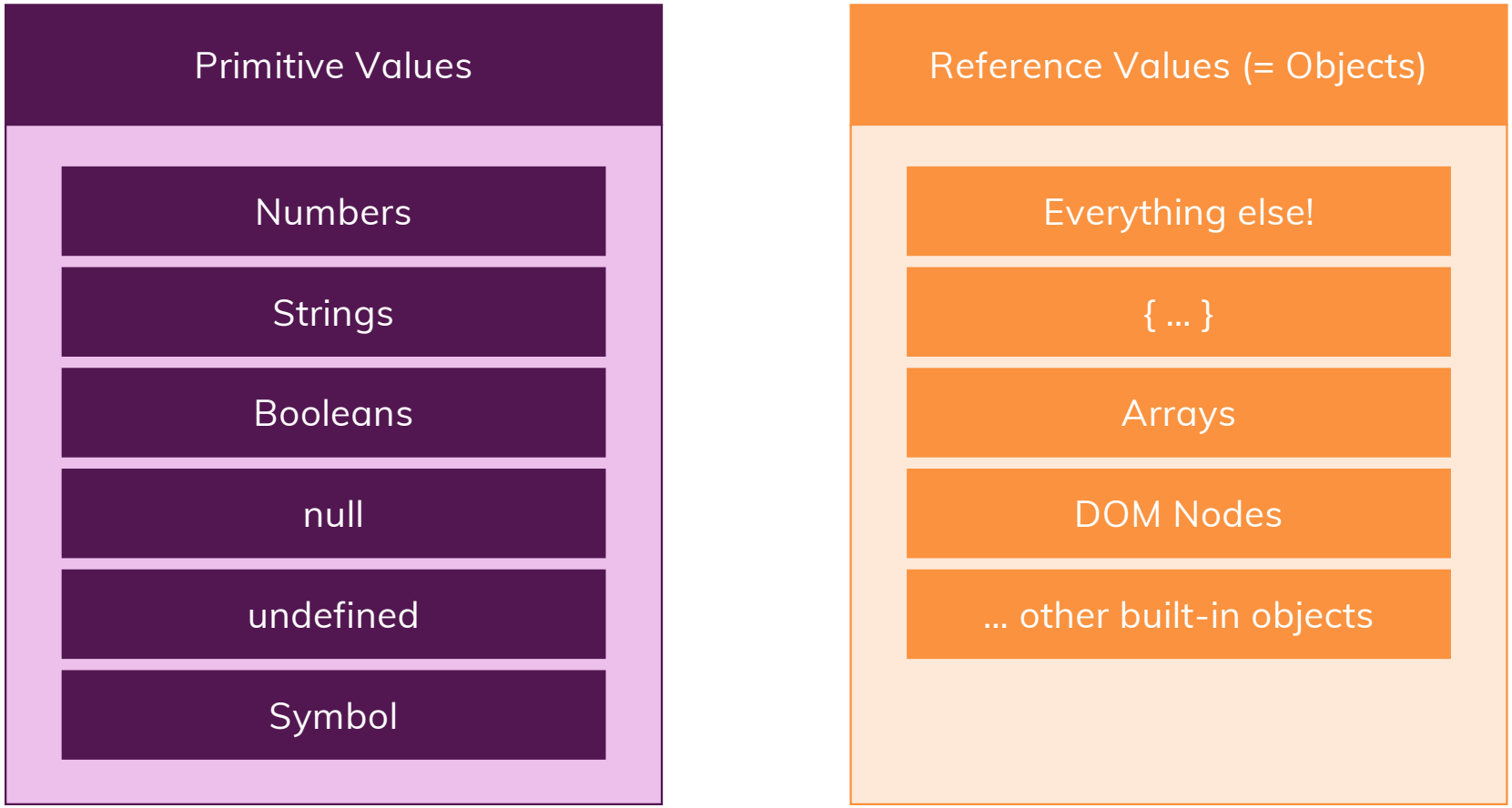
Maps vs Objects

Maps	Objects
Can use ANY values (and types) as keys	Only may use strings, numbers or symbols as keys
Better performance for large quantities of data	Perfect for small/ medium-sized sets of data
Better performance when adding + removing data frequently	Easier/ quicker to create (typically also with better performance)

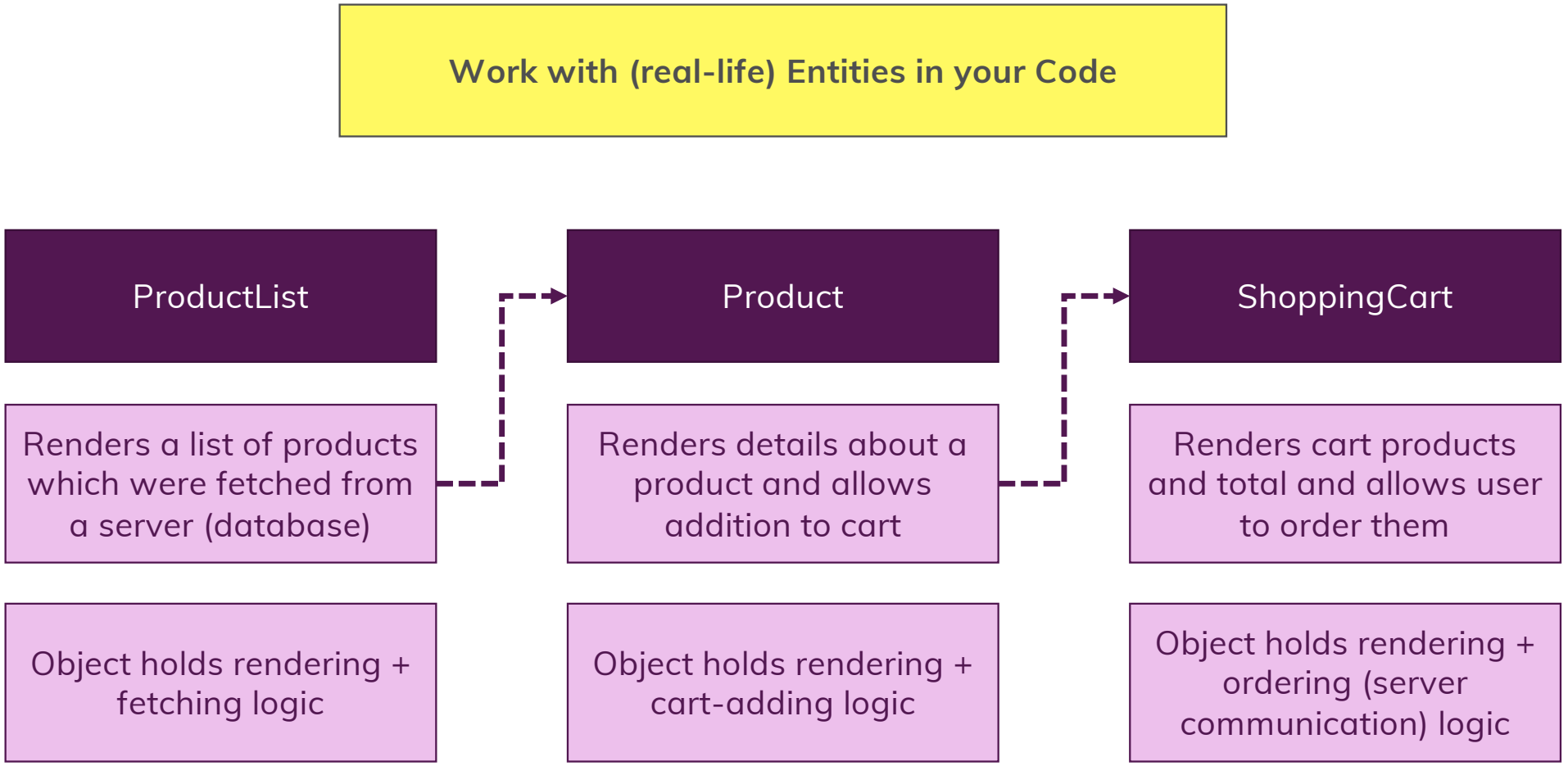
What are Objects?



Objects in JavaScript

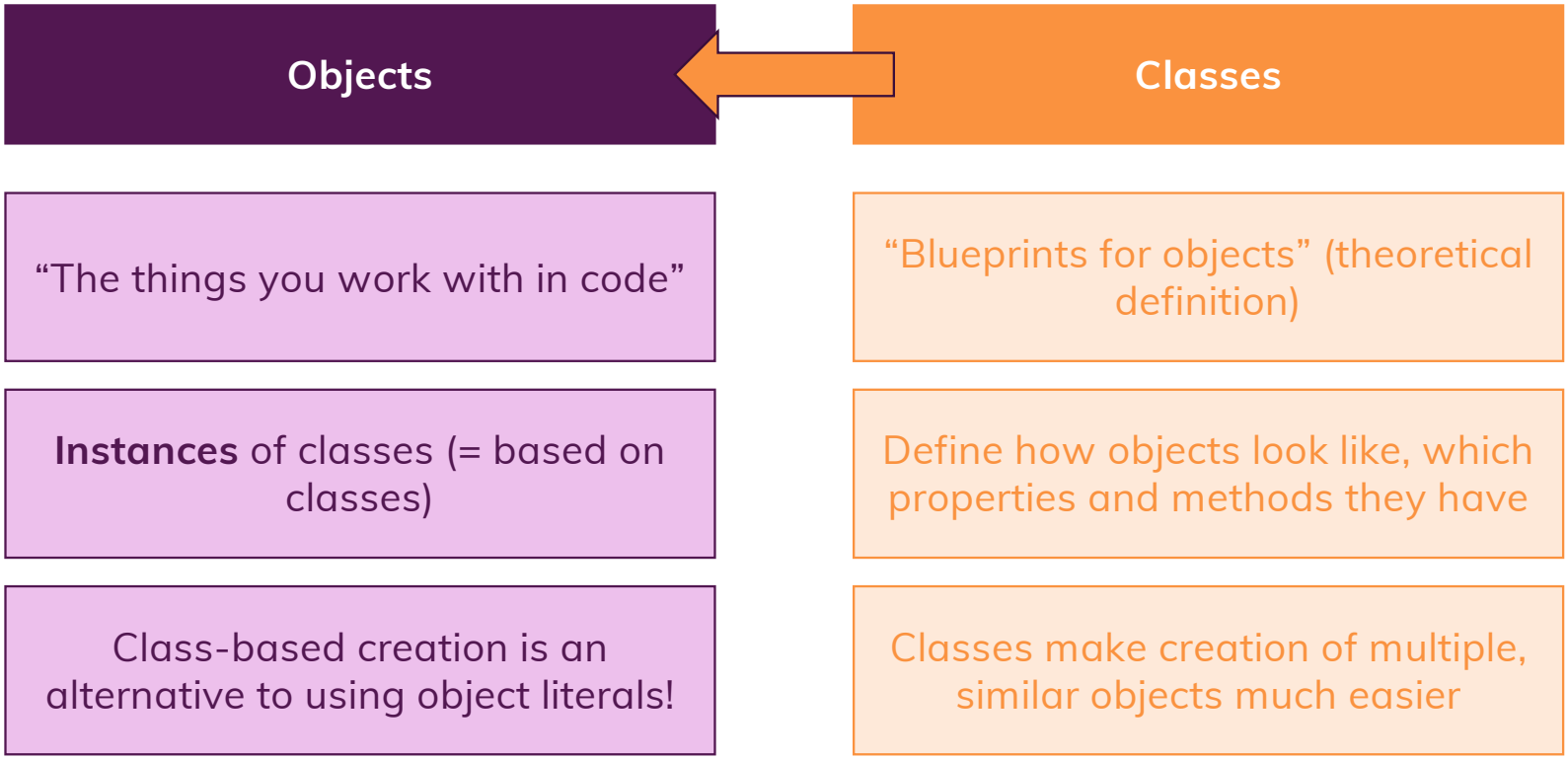


What's Object-oriented Programming (OOP)?

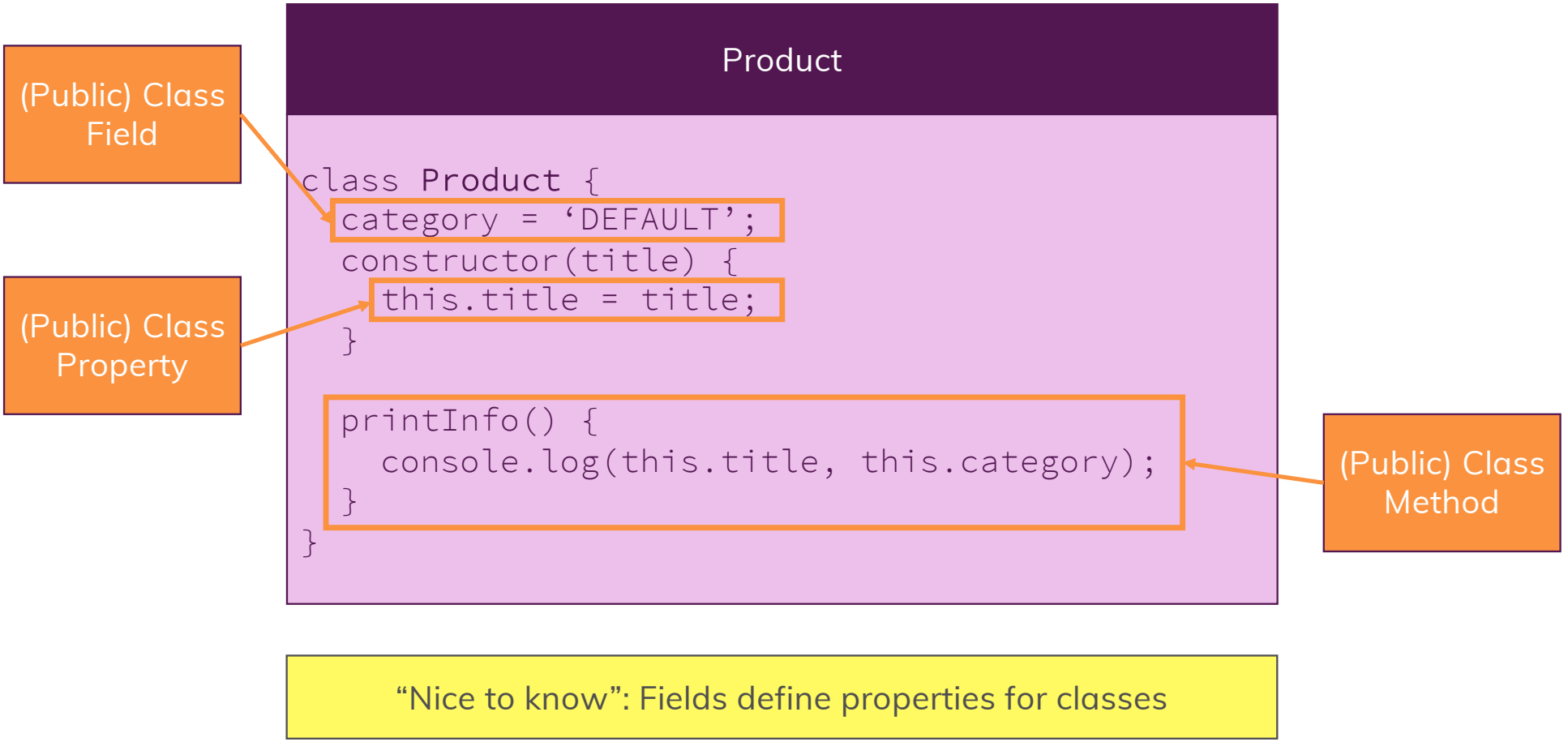


ACADEMIND

Classes & Instances



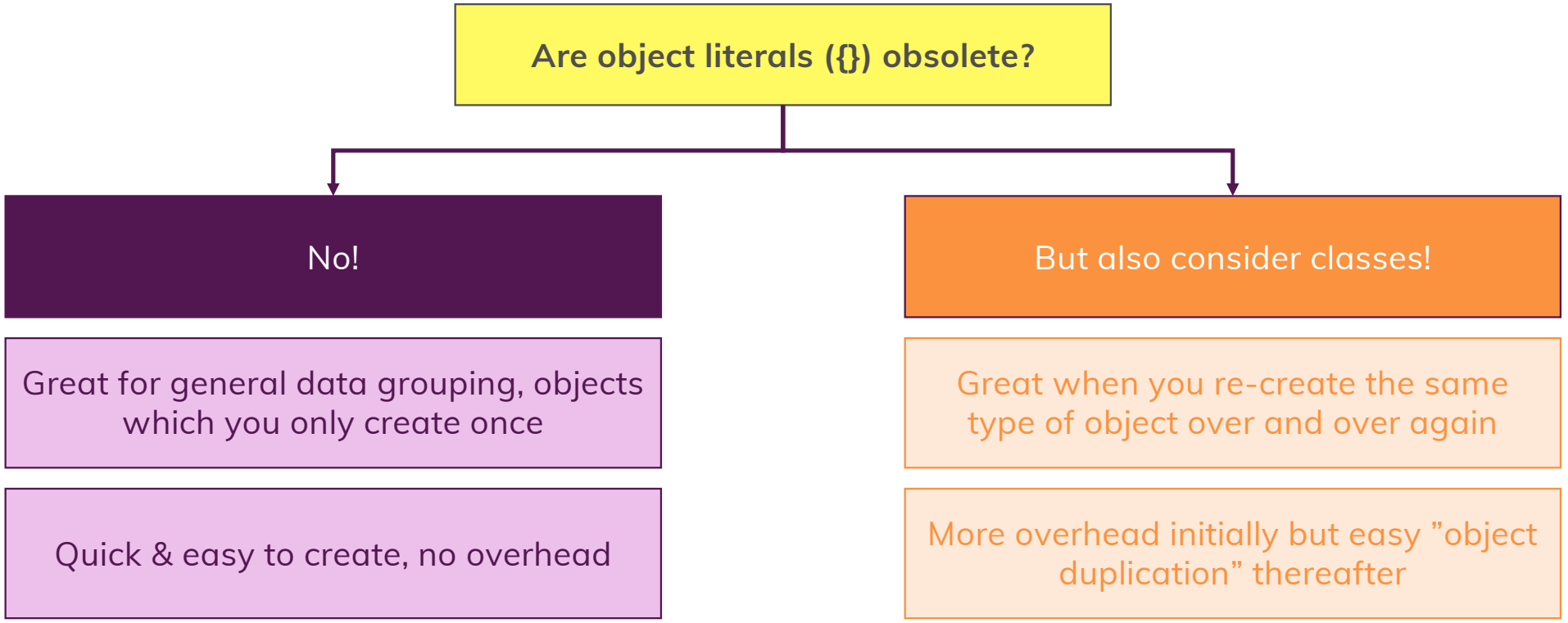
Class Fields (vs Properties)



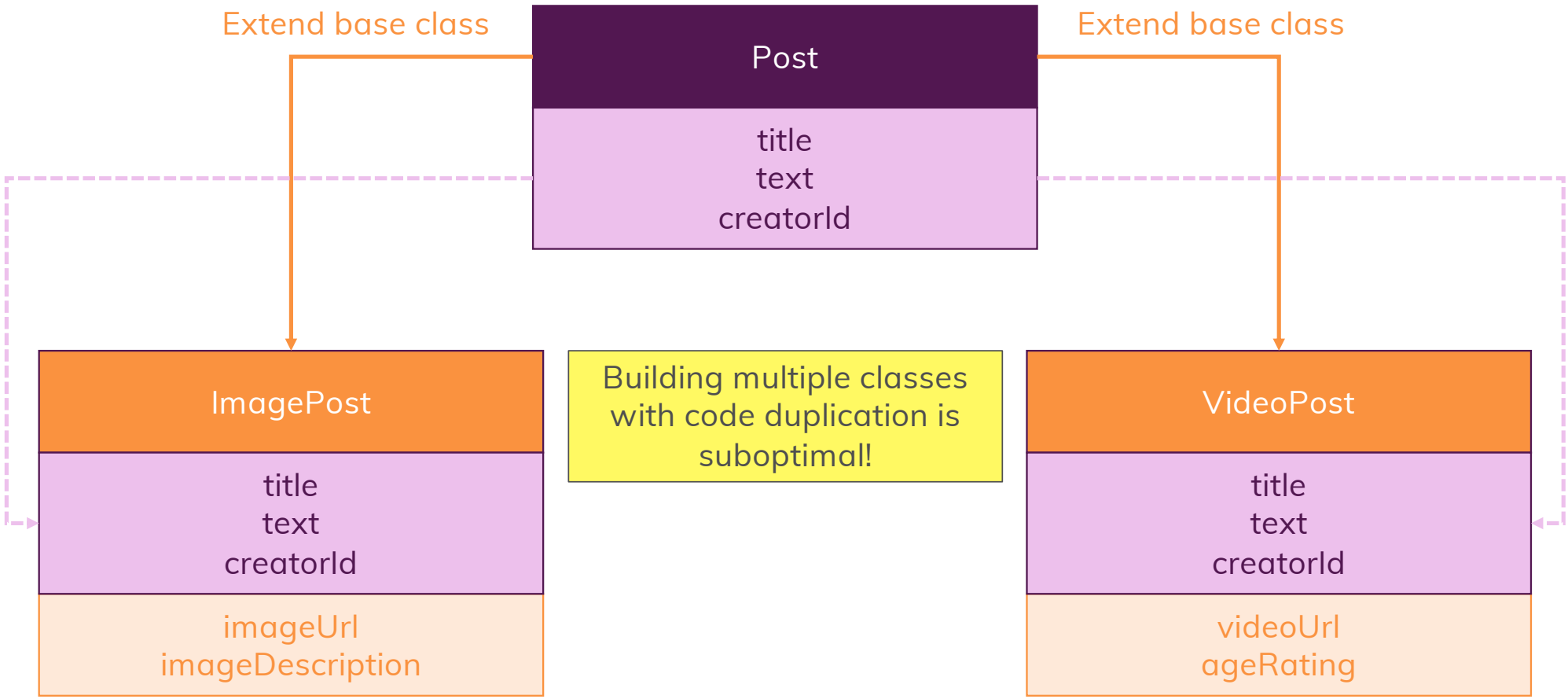
Static Properties, Fields & Methods

Static Field / Property / Method	Instance Field / Property / Method
Defined with `static` keyword	Defined without `static` keyword
Only accessible on class itself, without instantiation (i.e. not on instance)	Only accessible on instances (= objects) based on class
Typically used in helper classes, global configuration etc.	Used for core, re-usable logic

When to use Classes



Inheritance



Private Fields, Properties & Methods

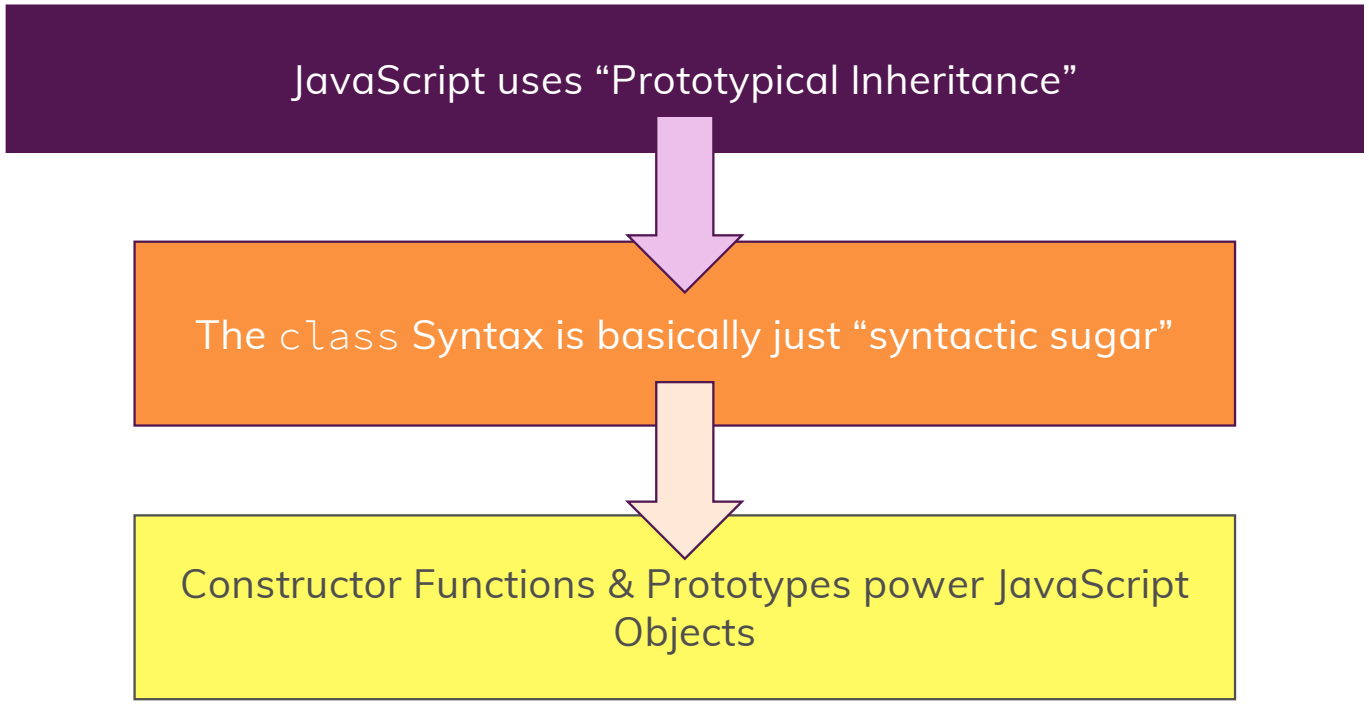
Public	Private
Accessible OUTSIDE of the class/ object	Accessible ONLY INSIDE of the class/ object
The “things” you work with in your other code	The “things” you work with in your class only (to split & re-use code)
Example: product.buy()	Example: Hard-coded (fallback) values, re-used class-specific logic

ACADEMIND

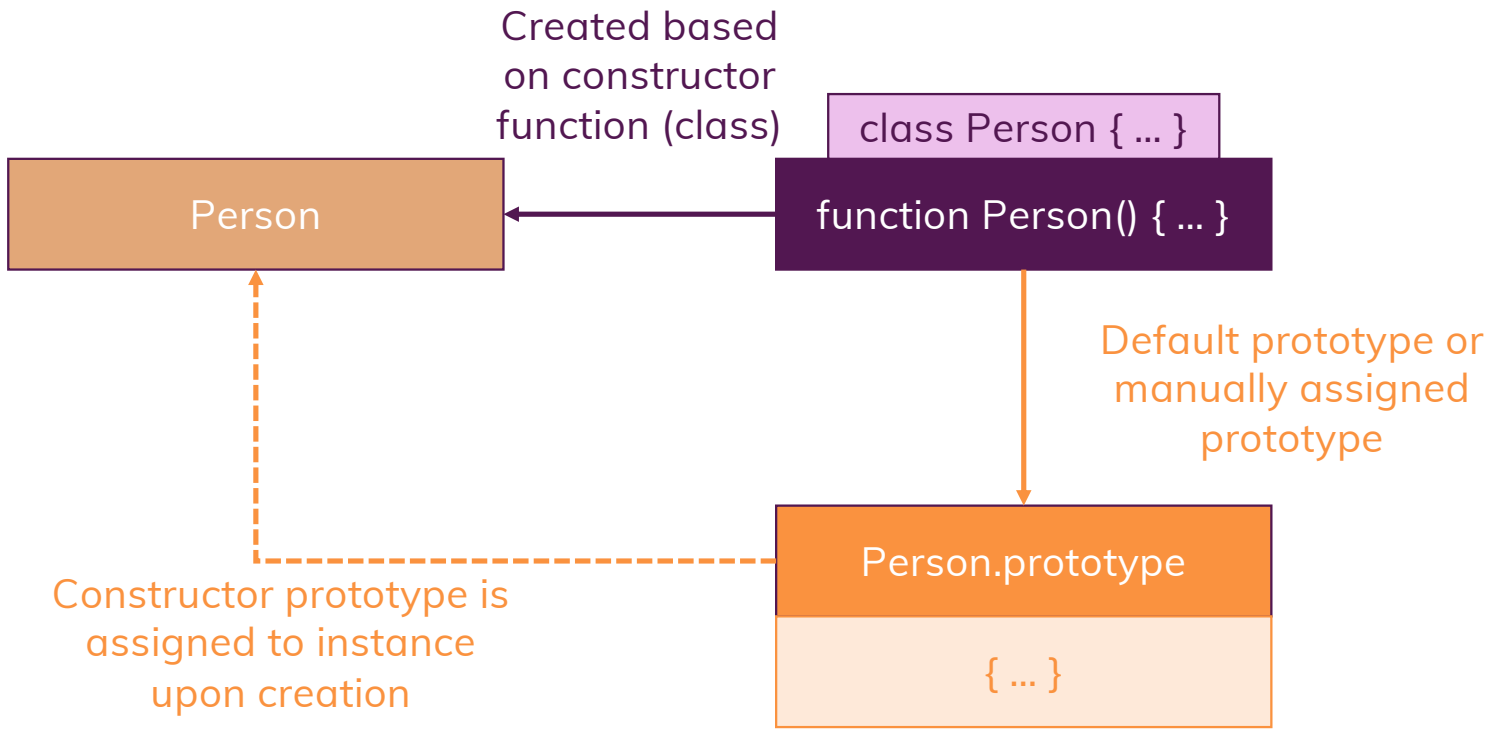
Constructor Functions vs Classes

Constructor Functions	Classes
Blueprint for Objects	Blueprint for Objects
Properties & Methods	Properties & Methods
Can be called with new	Must be called with new
All properties and methods are enumerable	Methods are non-enumerable by default
Not in strict mode by default	Always in strict mode

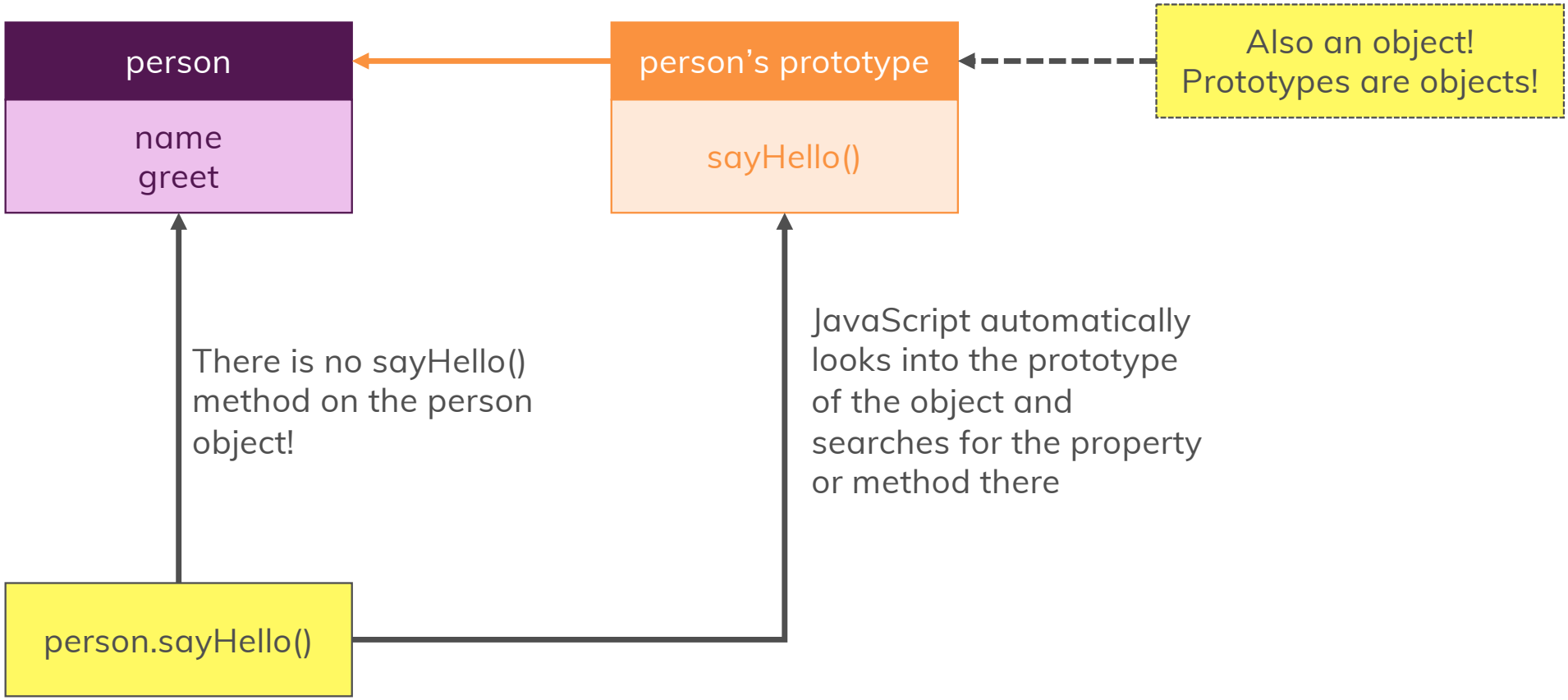
What are “Prototypes”?



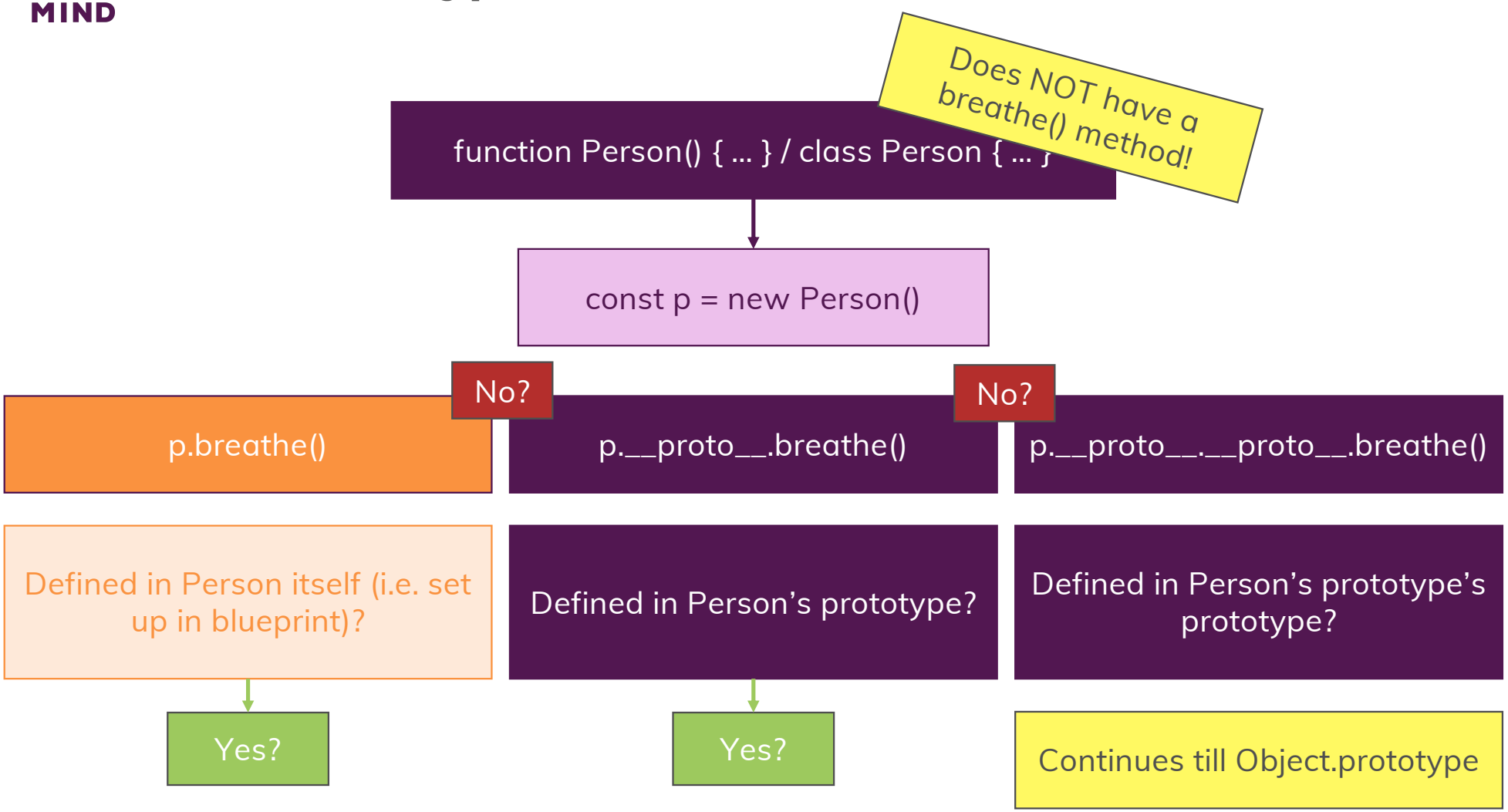
Prototypes


Created based on constructor function (class)
class Person { ... }
Person
function Person() { ... }
Default prototype or manually assigned prototype
Person.prototype
{ ... }
Constructor prototype is assigned to instance upon creation

Prototype Objects == “Fallback Objects”



The Prototype Chain



Prototypes & “Method Types”

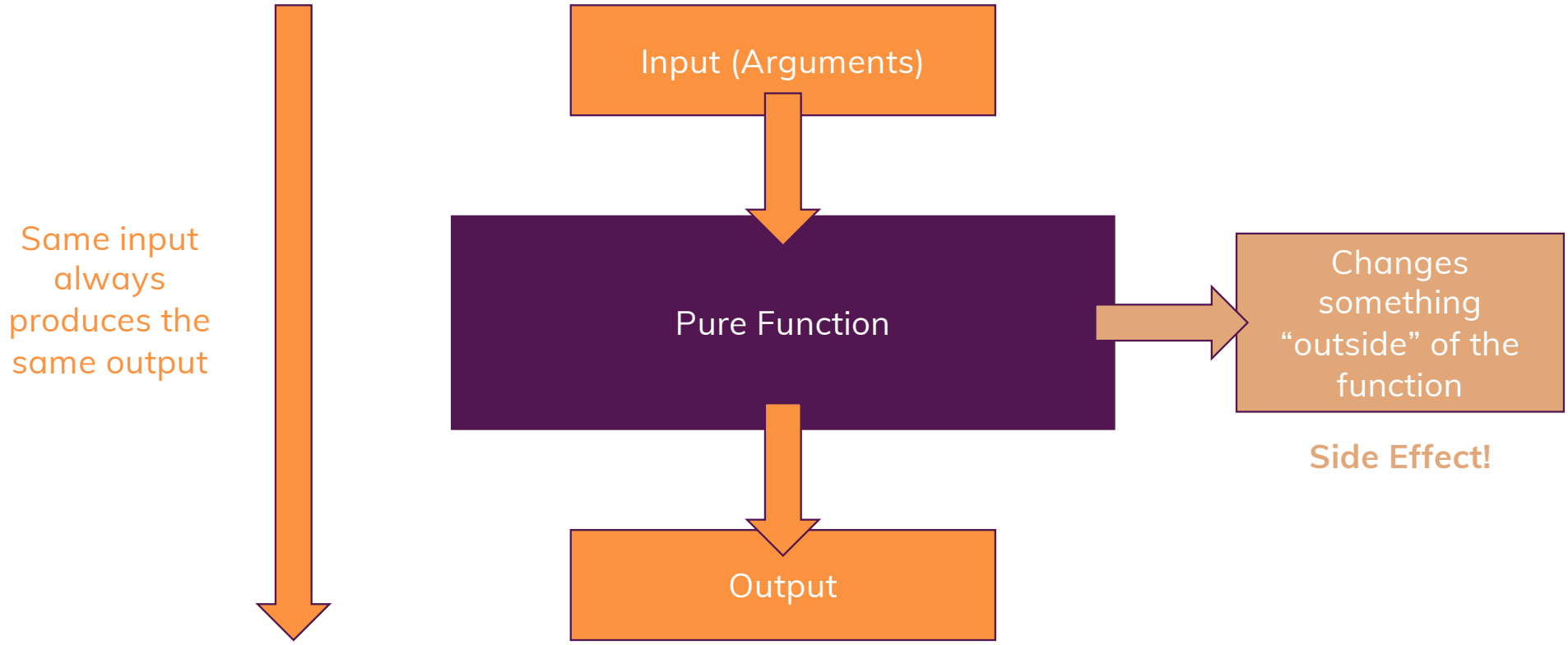
Method Shorthand	Property Function	Property Arrow Function

class Person {
  greet() {
  console.log(‘Hello’);
  }
}
	class Person {
	  greet = function() { … }
	  constructor() {
	  this.greet2 = function() { … }
	  }
	}
		class Person {
		  greet = () => { … }
		  constructor() {
		  this.greet2 = () => { … }
		  }
		}

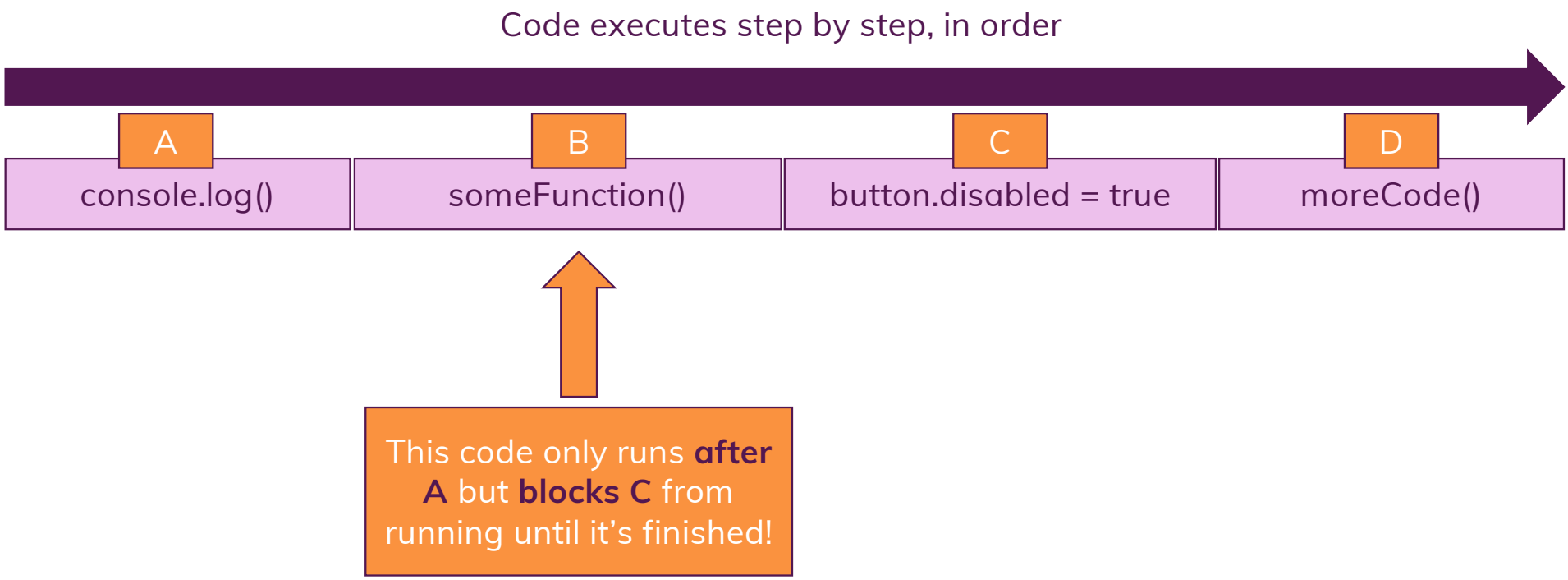
Assigned to Person’s prototype and hence shared across all instances (NOT re-created per instance).	Assigned to individual instances and hence re-created per object. `this` refers to “what called the method”.	Assigned to individual instances and hence re-created per object. `this` always refers to instance.

Unnecessary re-creations of function can cost performance!

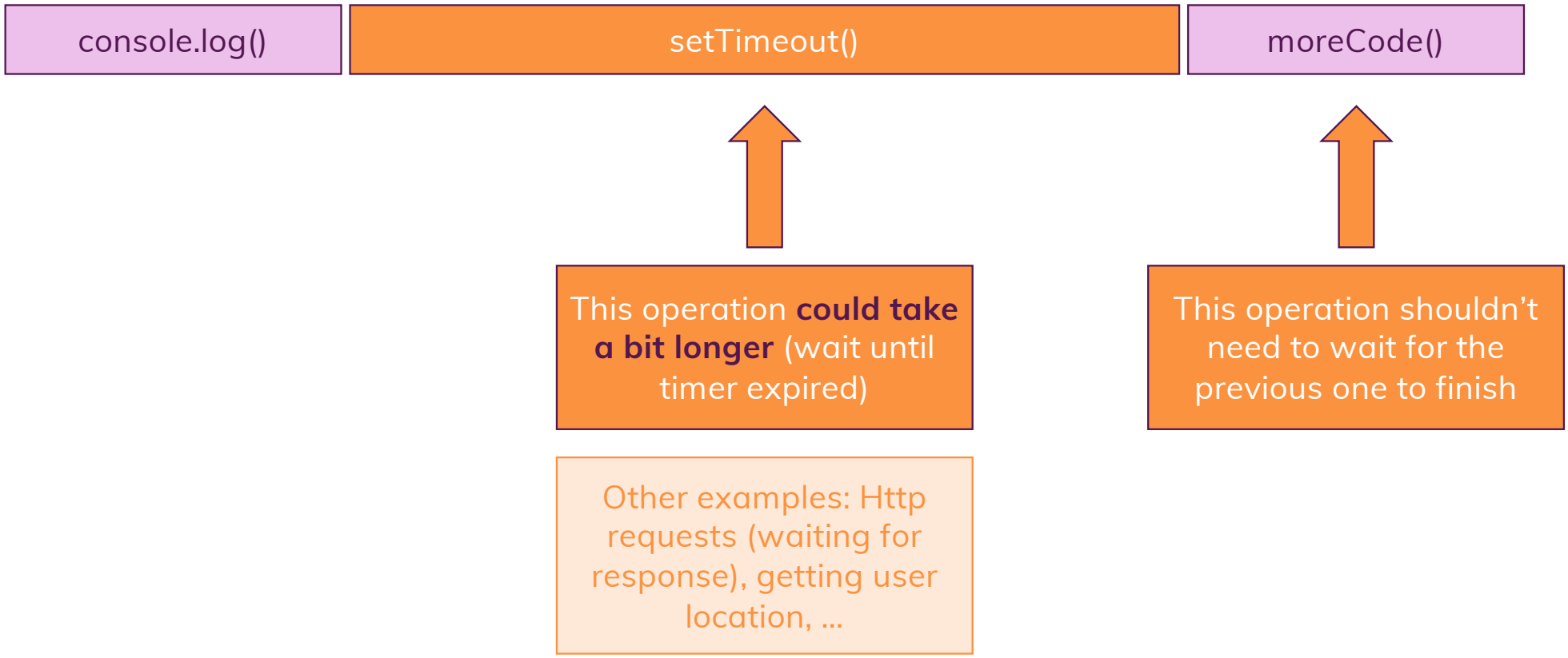
What's a "Pure" Function without "Side Effects"?



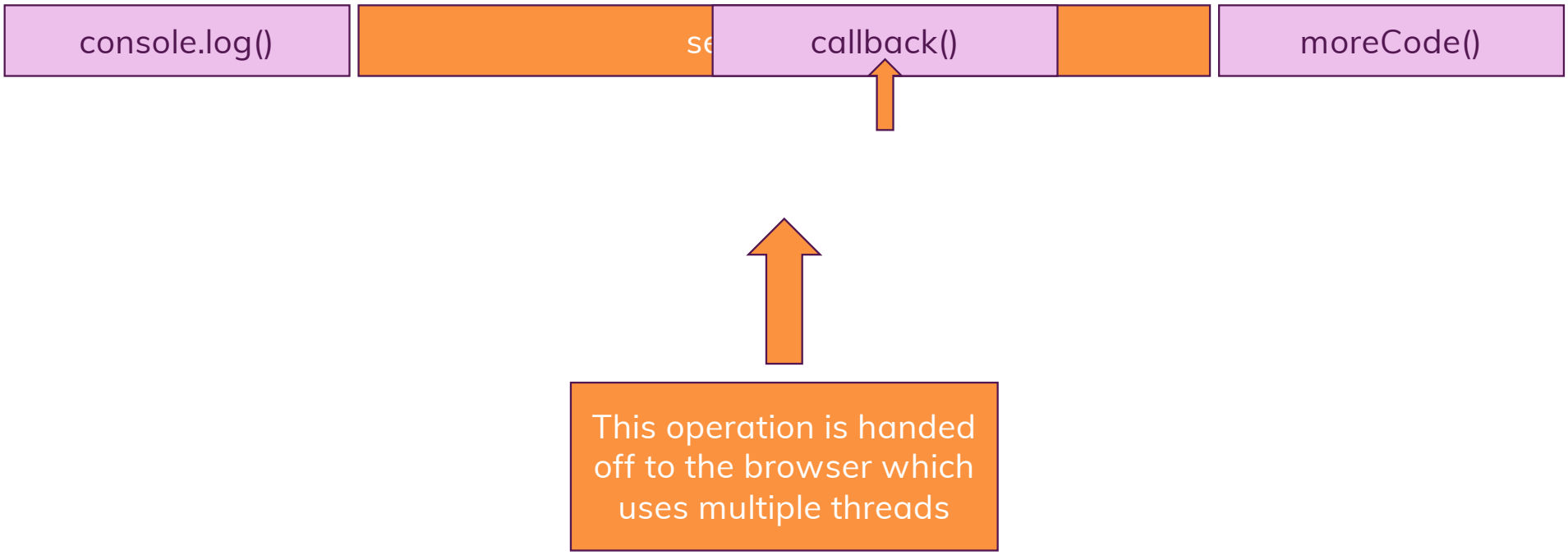
JavaScript is Single-Threaded



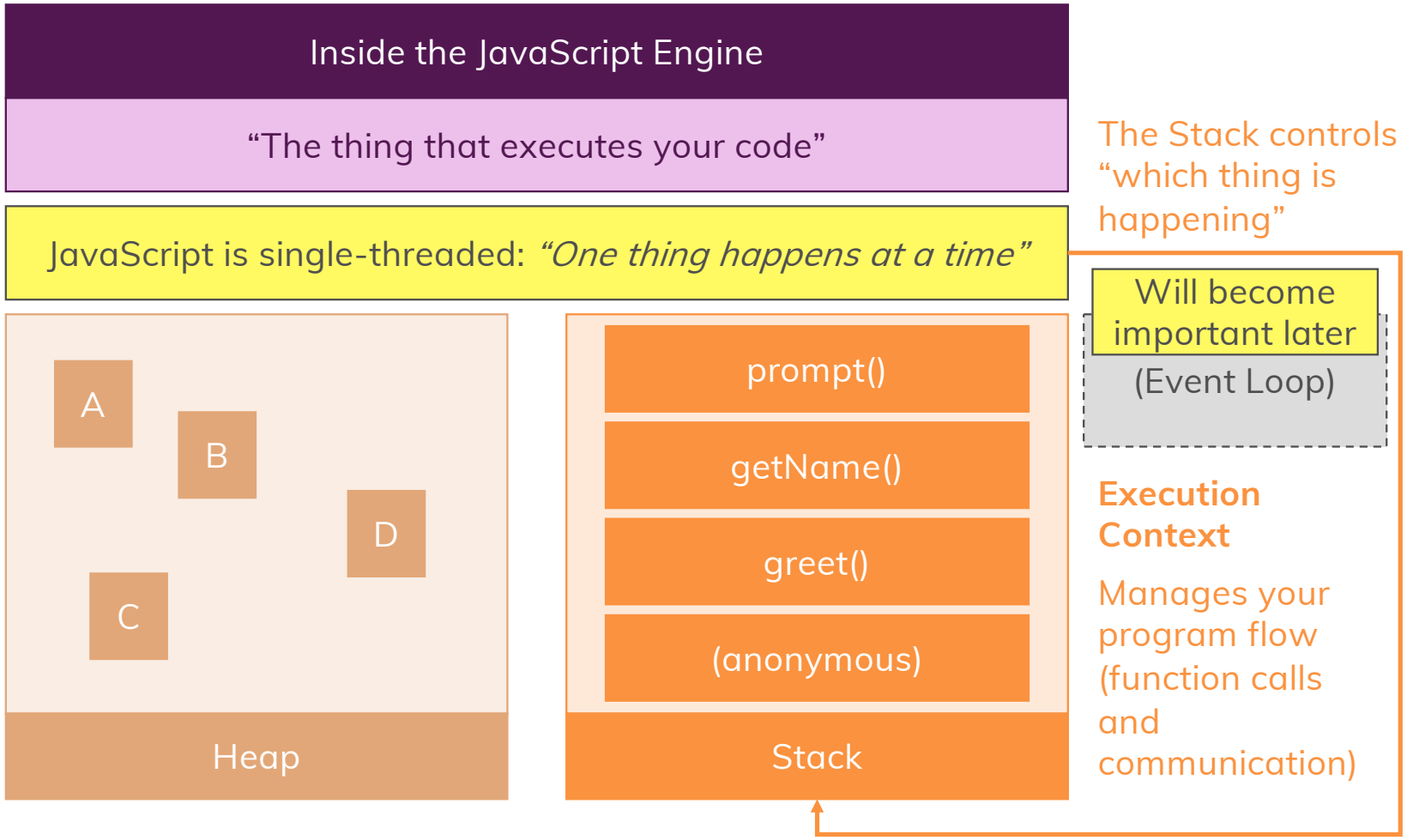
Certain Operations Take A Bit Longer



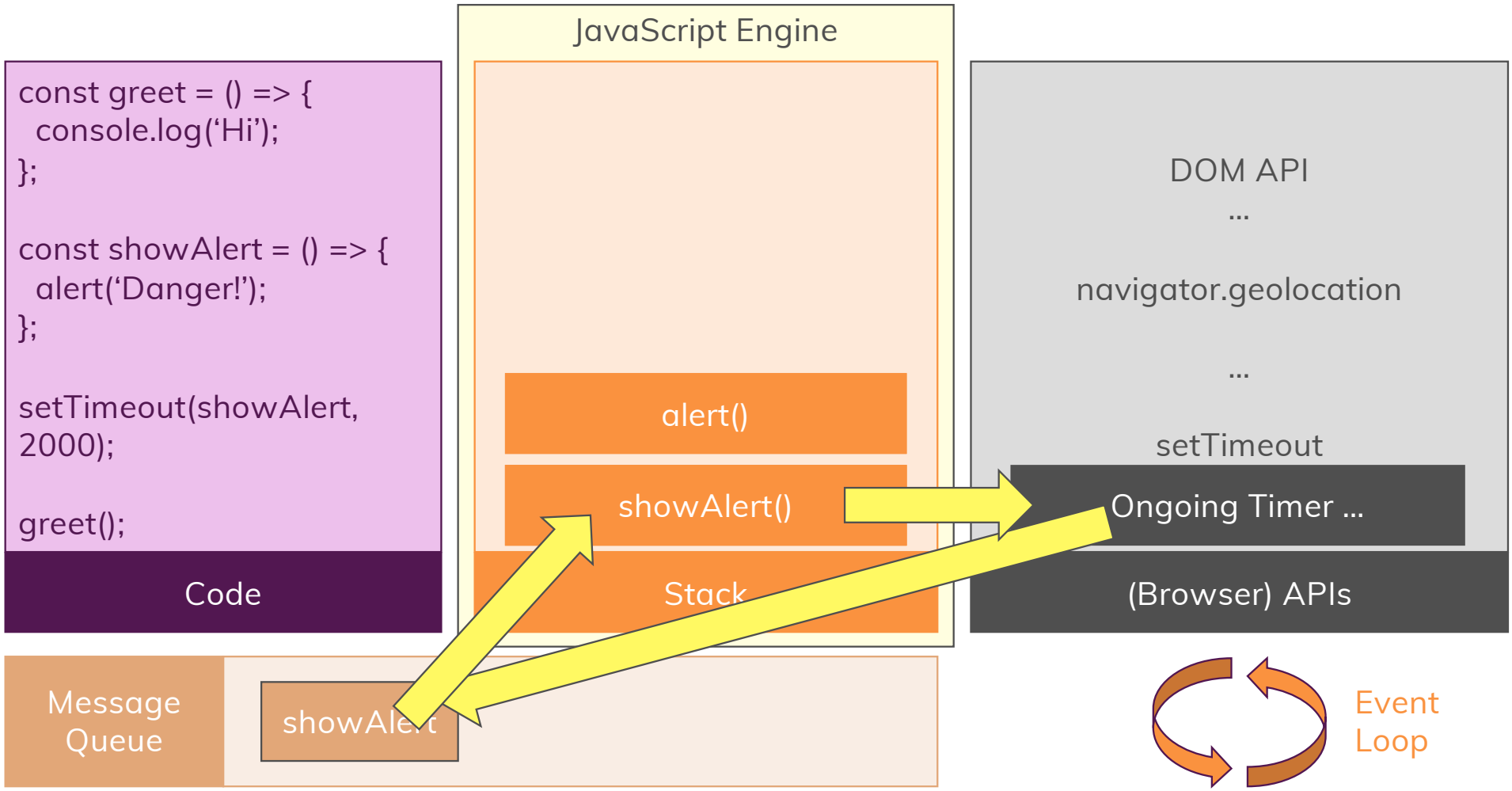
Asynchronous Code Execution!



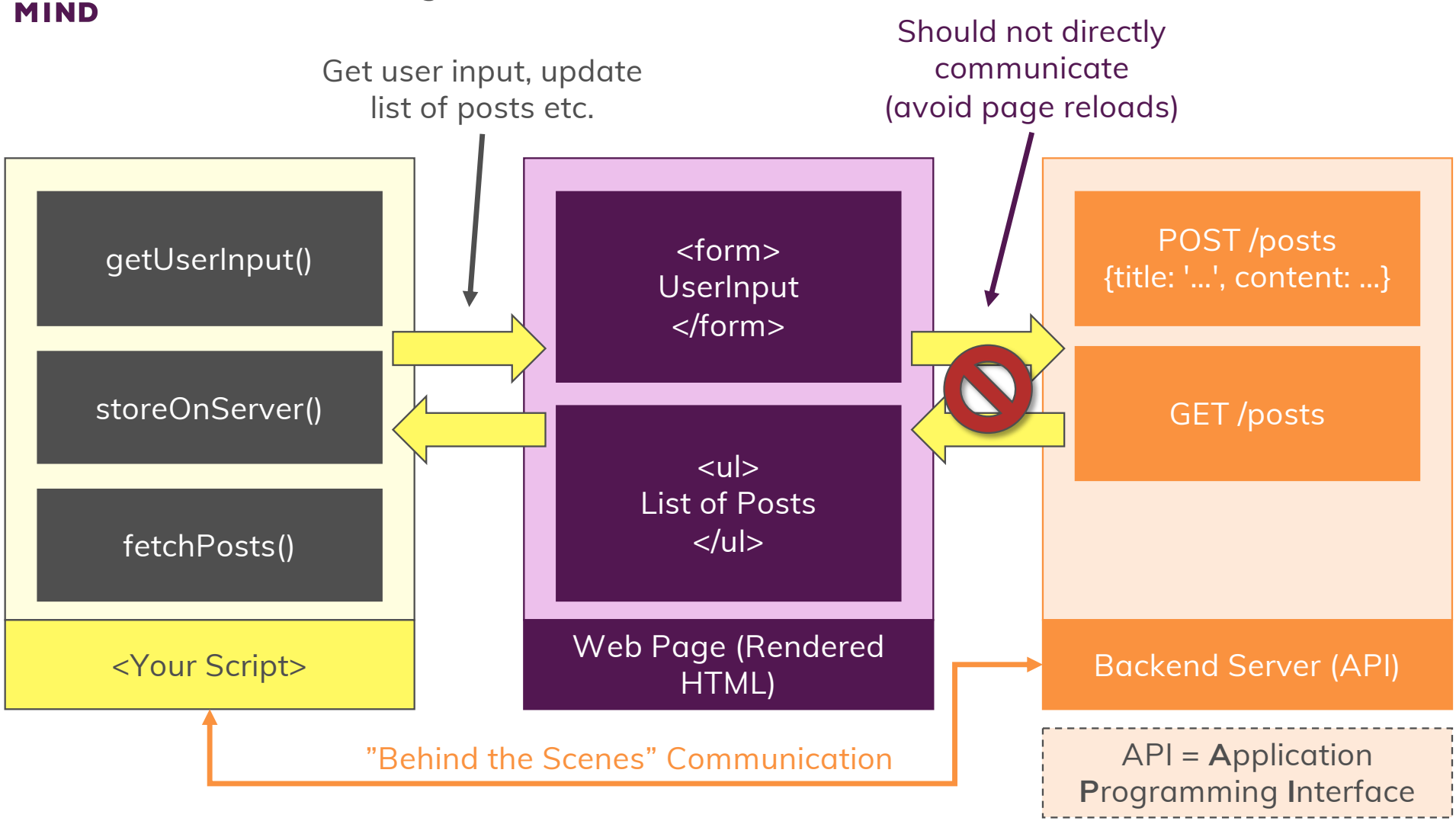
How Code Gets Executed



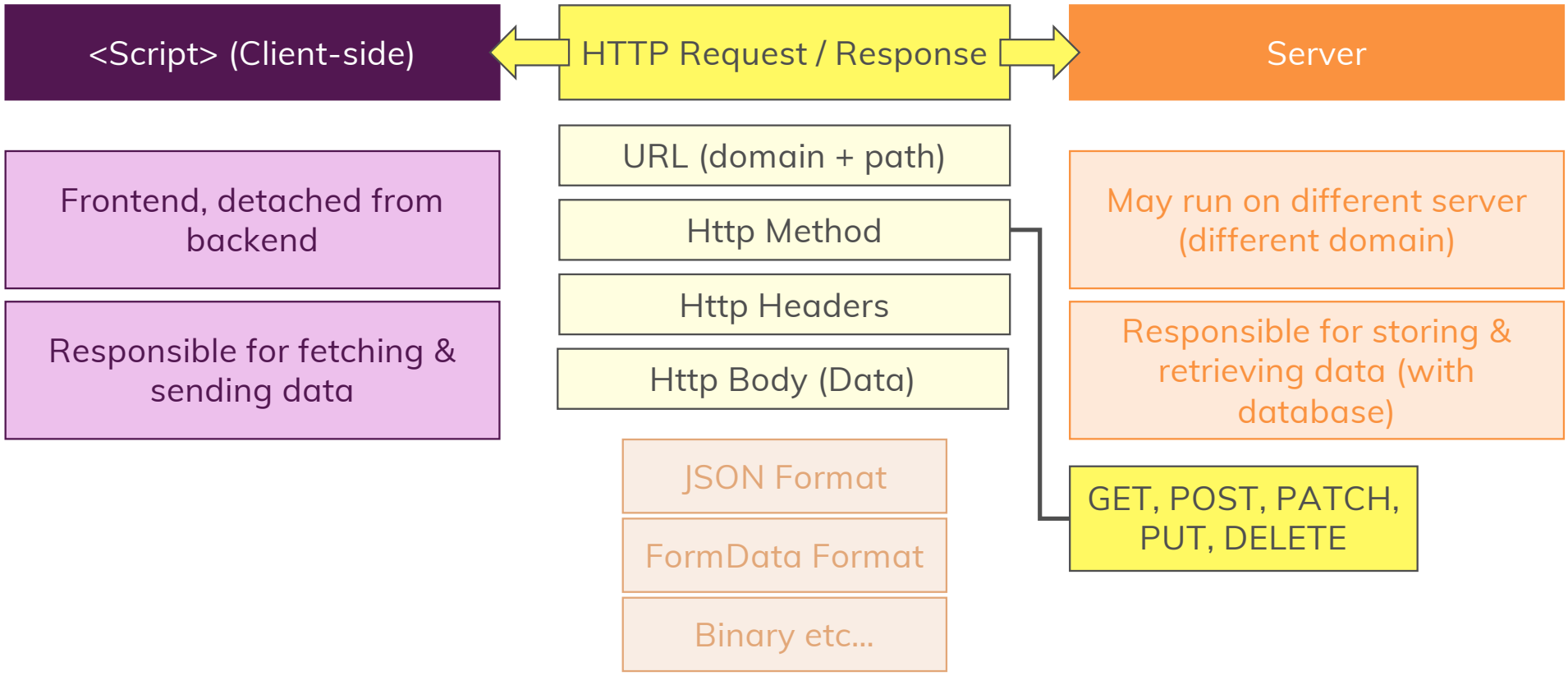
Event Loop, Queue and Async Code



What & Why?


Get user input, update list of posts etc.
Should not directly communicate (avoid page reloads)
getUserInput()
storeOnServer()
fetchPosts()
<Your Script>
<form> UserInput </form>
 List of Posts
Web Page (Rendered HTML)
POST /posts {title: '...', content: ...}
GET /posts
Backend Server (API)
"Behind the Scenes" Communication
API = Application Programming Interface

(Background) Http Overview



Limitations of “Basic Projects”

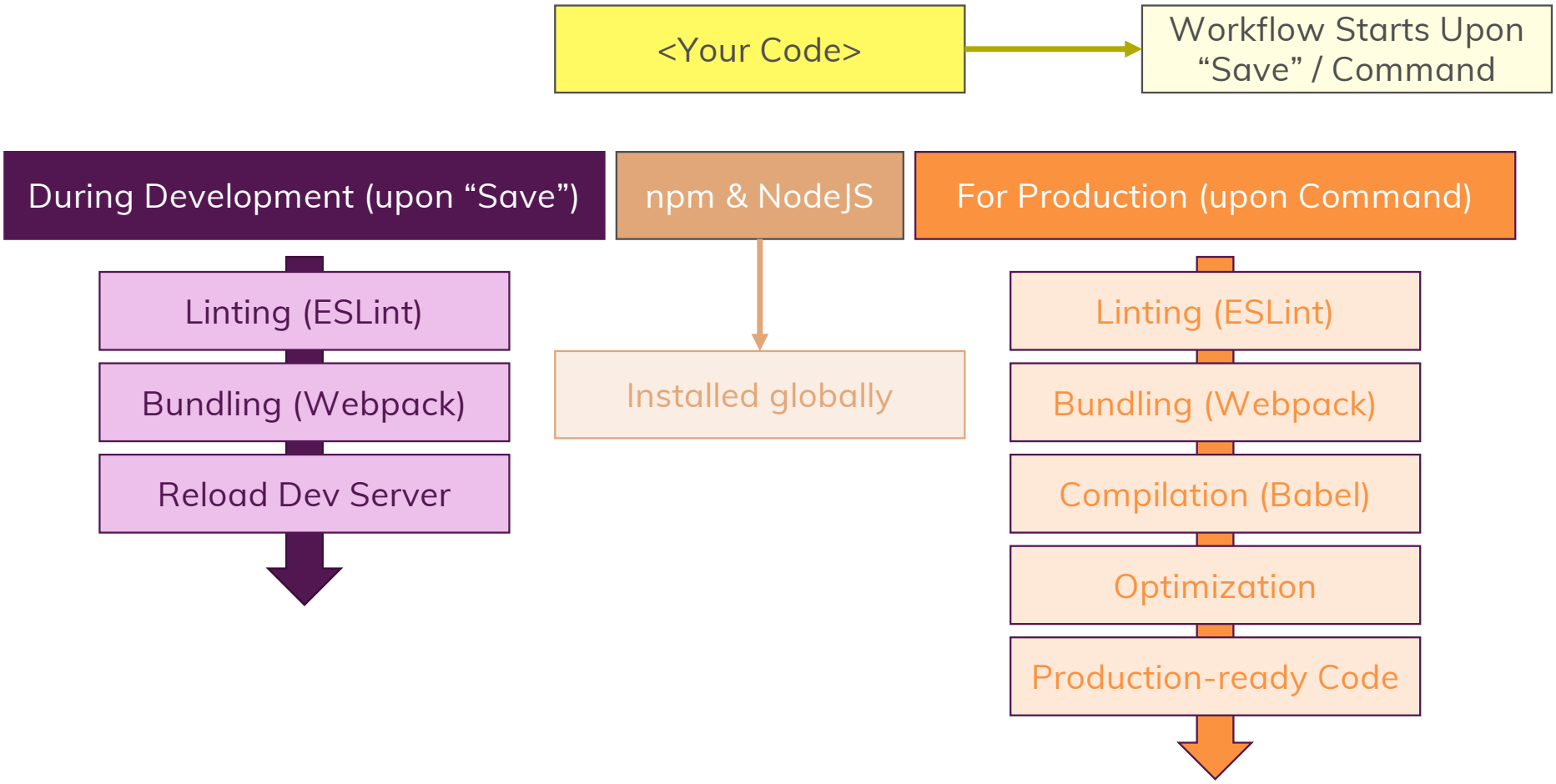
- Micromanagement of Imports OR Lots of Unnecessary Http Requests
- Unoptimized Code (not as small as possible)
- Potentially sub-optimal Browser Support
- Need to Reload Page Manually (after changes to code)
- Code Quality Is Not Checked

Helpful Tools & Why To Use Them

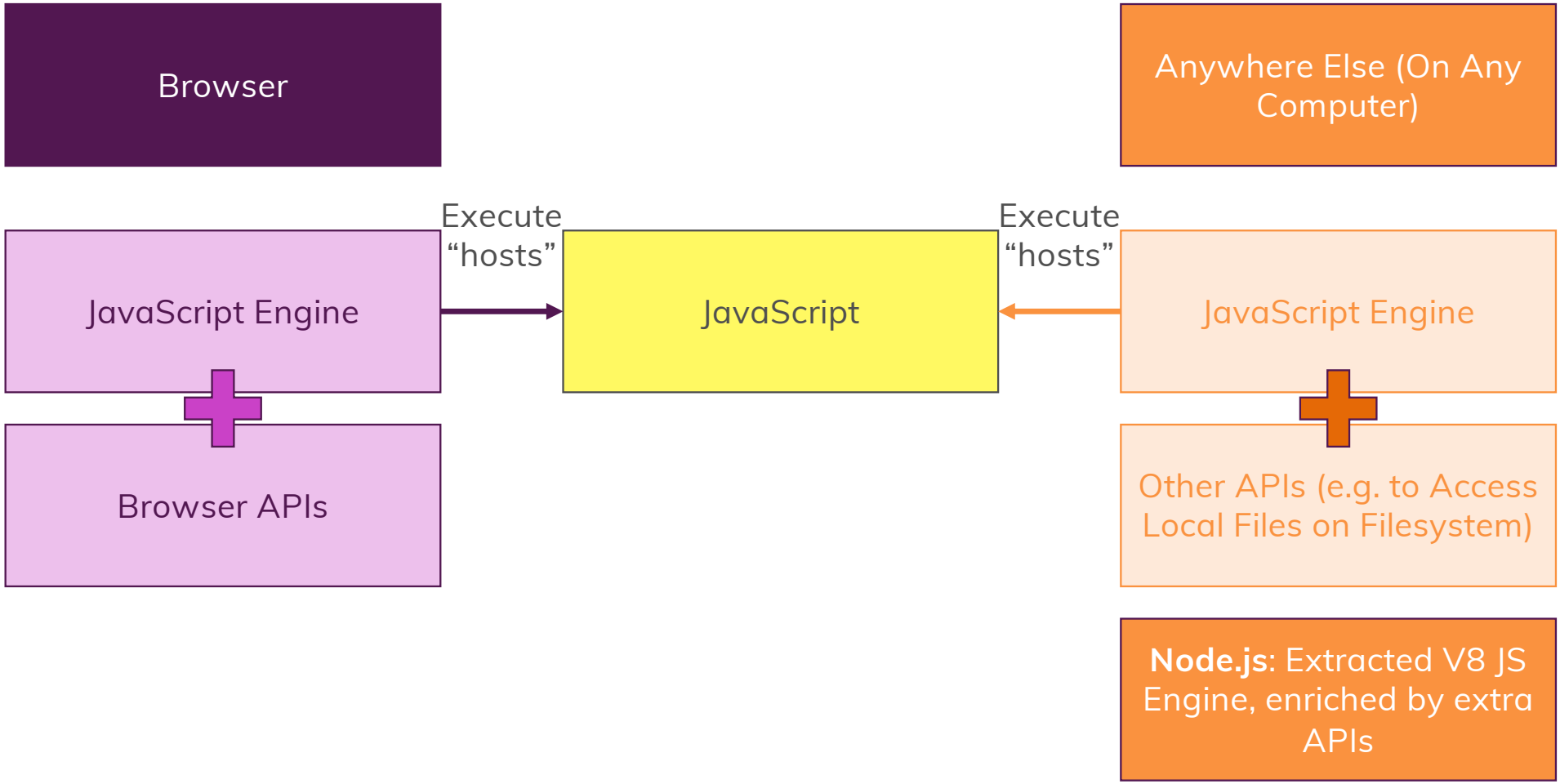
Tool Purpose	Tool Name	What it does & Why
A Development Server	**webpack-dev-server** OR **serve** (standalone tool)	Serve under (more) realistic circumstances, auto-reload
Bundling Tool	**Webpack**	Combine multiple files into bundled code (less files)
Code Optimization Tool	**Webpack Optimizer** Plugins	Optimize code (shorten function names, remove whitespace, ...)
Code Compilation Tool	**Babel**	Write modern code, get "older" code as output
Code Quality Checker	**ESLint**	Check code quality, check for conventions & patterns

Later!

Setup


<Your Code>
Workflow Starts Upon "Save" / Command
During Development (upon "Save")
npm & NodeJS
For Production (upon Command)
Linting (ESLint)
Bundling (Webpack)
Reload Dev Server
Installed globally
Linting (ESLint)
Bundling (Webpack)
Compilation (Babel)
Optimization
Production-ready Code

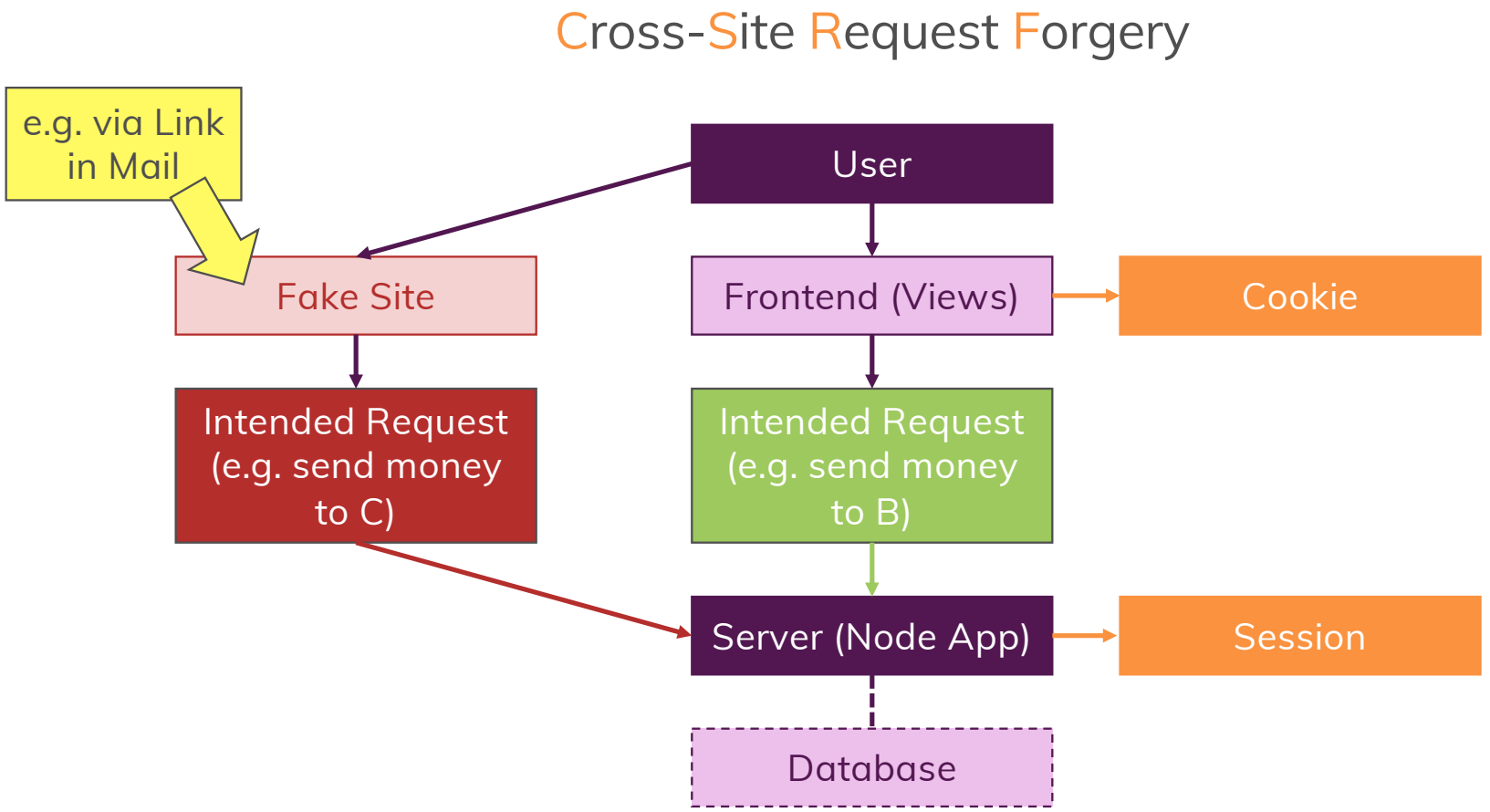
JavaScript Is A Hosted Language



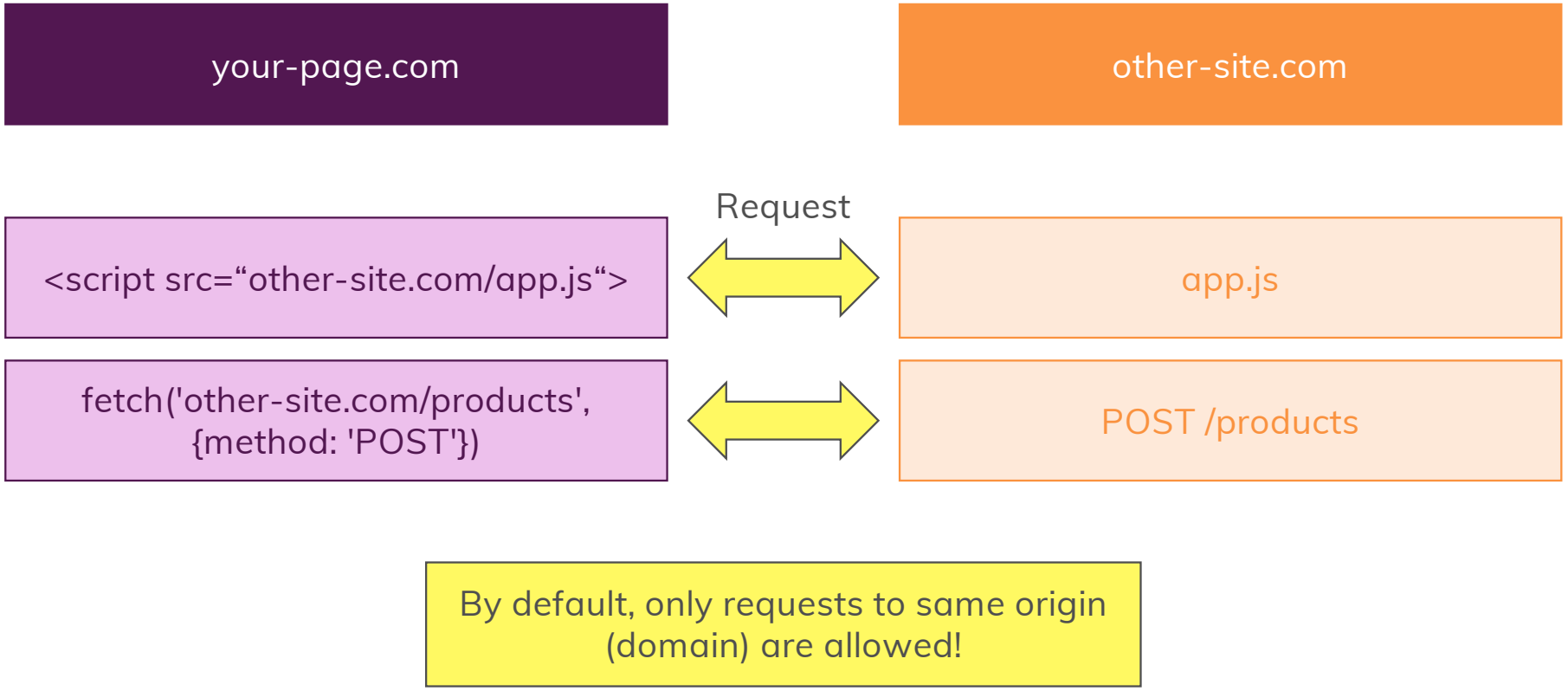
What Could Go Wrong?

Security Details In Your Code	Cross-Site Scripting Attacks (XSS)	Cross-Site Request Forgery (CSRF)	Cross-Origin Resource Sharing (CORS)
Your JavaScript code can be read by ANYONE	Attack pattern where malicious JS code gets injected + executed	Attack pattern that depends on injected content (e.g. image)	Not an attack pattern but a security concept
Security-relevant details can be read	Injected code can do ANYTHING your code could do as well	Requests to malicious servers are made with user's cookies	Requests are only allowed from same origin (domain)
Attackers may be able to abuse exposed data	Very dangerous: Full behind-the-scenes control for attacker	Actions can be executed without the user knowing	Controlled via server-side response headers and browser
Example: Database access credentials exposed in code	Example: Unchecked user-generated content	Example: Malicious image URL, XSS	Example: JavaScript Modules

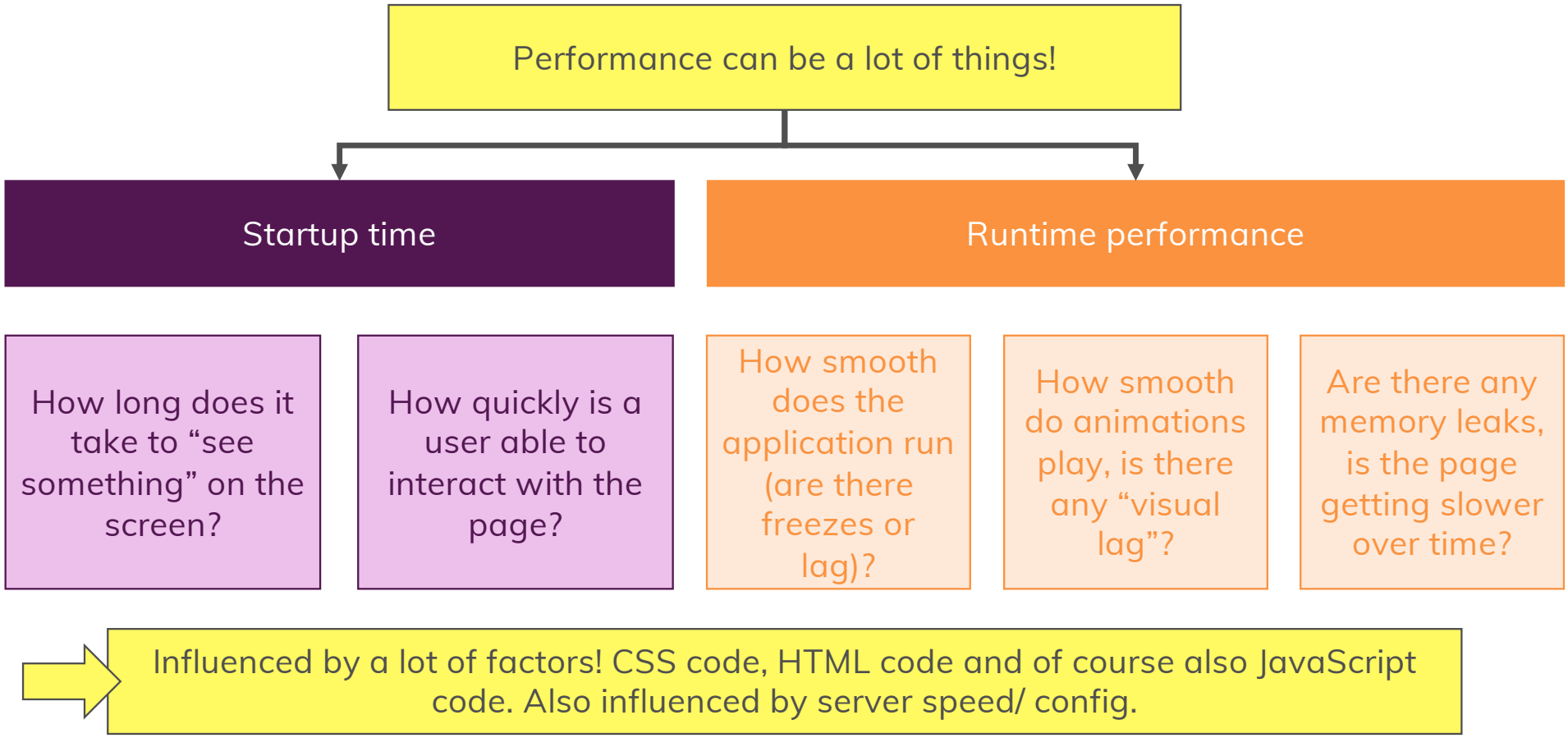
Cross-Site Request Forgery (CSRF)



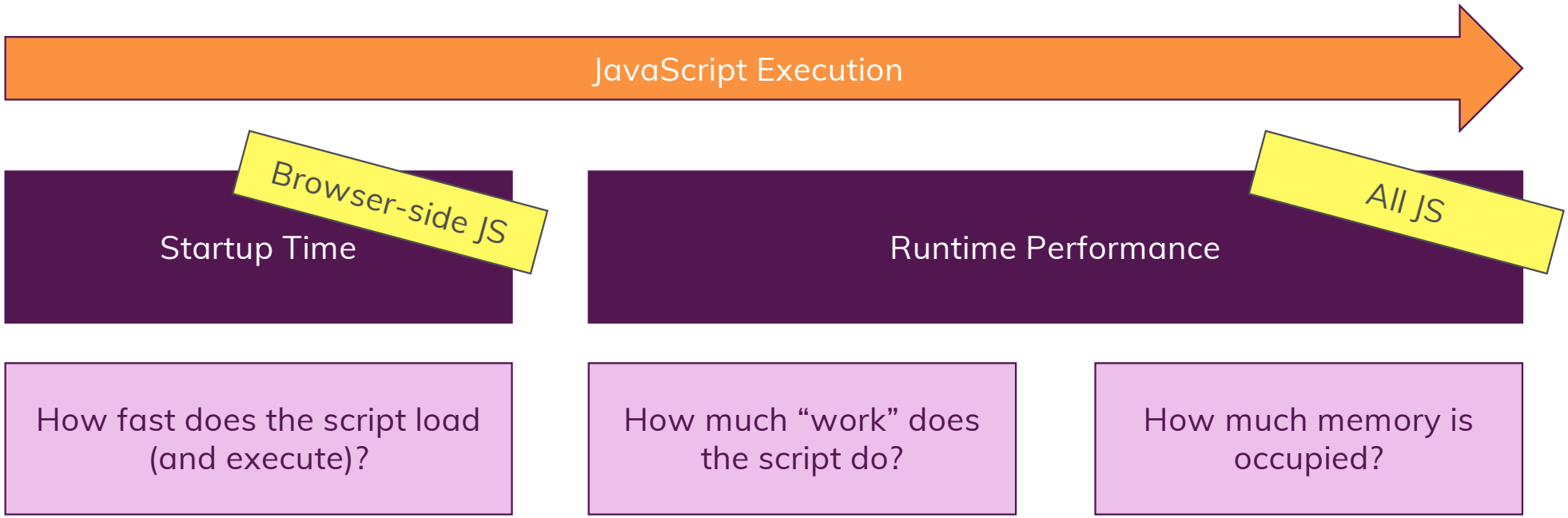
Cross-Origin Resource Sharing (CORS)



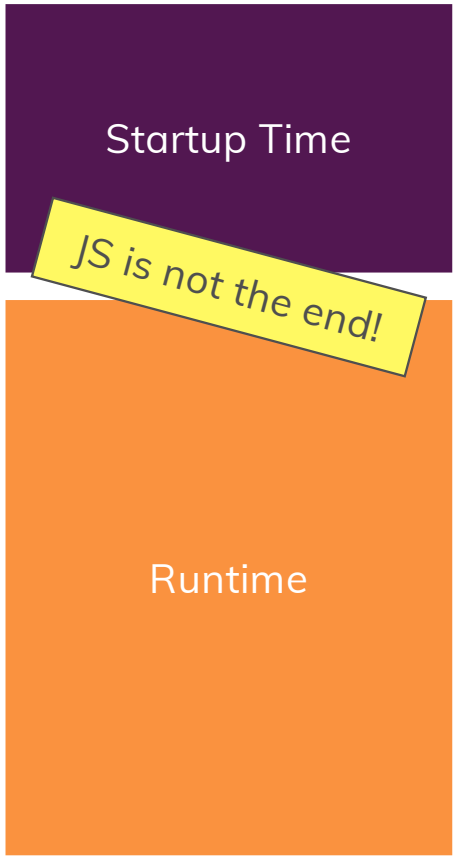
What is “Performance”?



What Influences Performance?



Different “Layers” of Performance Optimization



Bundle / Script **Size**: Delays initial parsing / execution

Number of **Http Roundtrips** (e.g. because of non-bundled code, third-party library CDNs): Delays initial parsing / execution

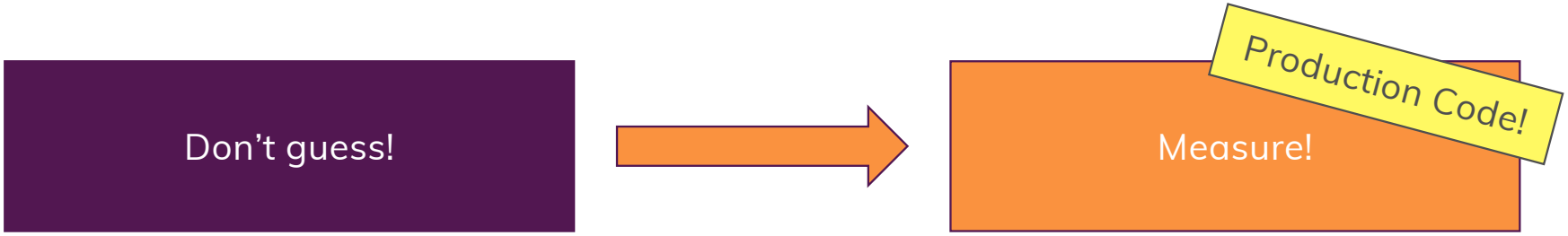
Optimize Code Execution, **DOM Access**: Avoid unnecessary code execution, especially unnecessary DOM operations/ repaints

Avoid **Memory Leaks**: Can crash your application in the worst case, but will slow it down in all cases

Find **Code Alternatives** with Better Performance: Especially important for “high-frequency” code parts

Micro-optimizations in your Code: Optimize for a very specific use-case (e.g. data structures for frequent access / changes)

Measuring & Auditing



Check roundtrips + script size and set script/ bundle size budgets

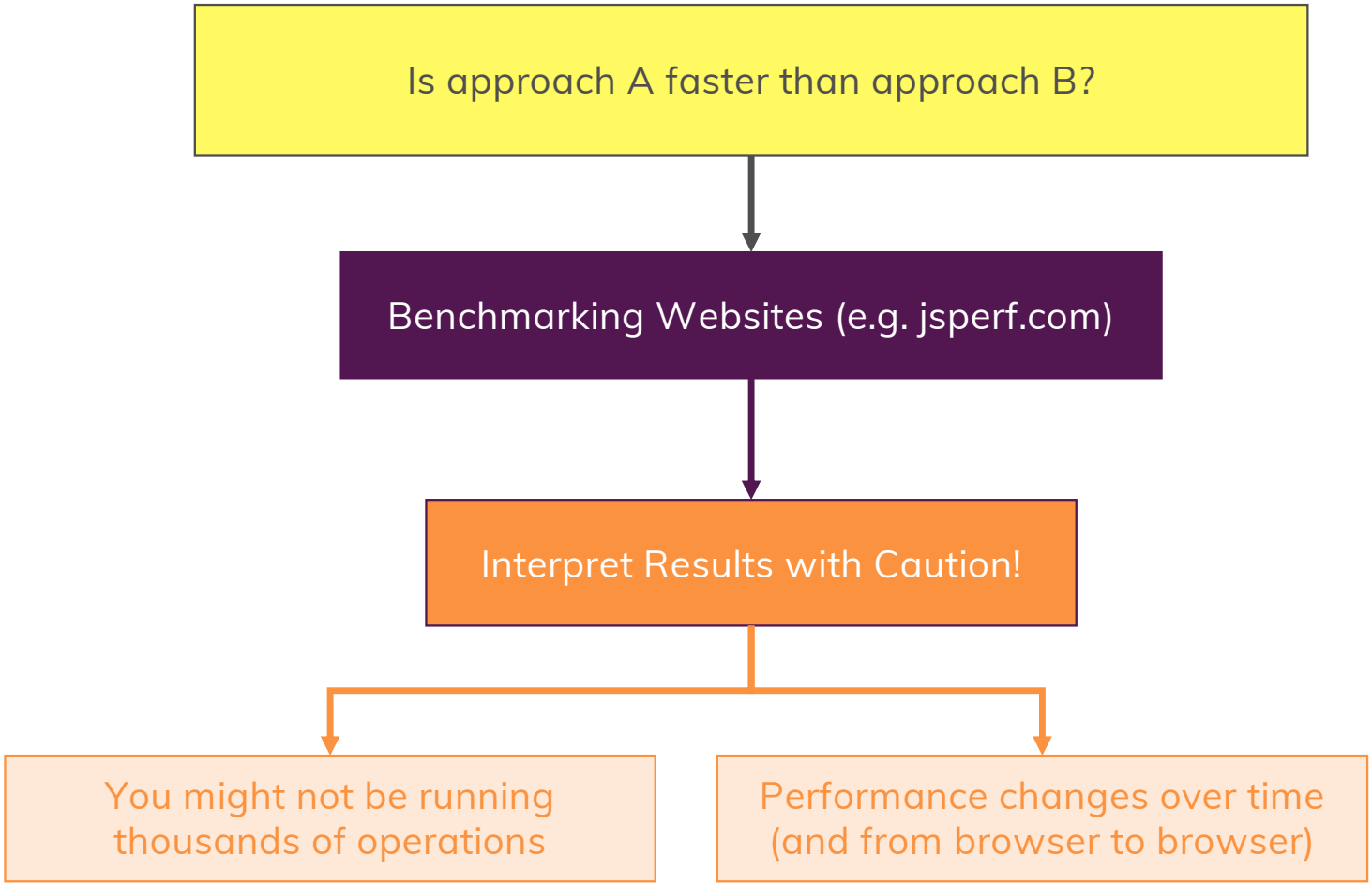
Measure performance with devtools

Explore best practices, patterns and benchmarks

Measuring Performance

performance.now()	Add this to your code (during development / testing) and check the execution time (difference) for certain operations
Browser DevTools	Use the many features of browser dev tools to detect unnecessary code executions, http requests and measure execution time + memory leaks
jsperf.com	Compare alternative code snippets and measure performance
webpagetest.com	Test your entire (live) web page to detect optimization potential

Benchmarking & Comparing Performance



Runtime Performance – Potential Improvements

Avoid unnecessary code executions, batch operations together

Find + fix memory leaks, even small ones

For high-frequency operations: Use optimal JavaScript approach (re-validate regularly)

Consider micro-optimizations

Startup Time – Potential Improvements

Eliminate unnecessary / unused code, avoid using too many libraries

Potentially add third-party libraries to bundle instead of via CDN

Bundle code & use “lazy loading”

Minify code