

Variables & Constants

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

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

```
const totalUsers = 15;
```

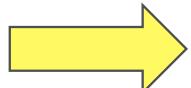
```
totalUsers = 20;
```

A “data container” / “data storage”

...where the value can change!

A “data container” / “data storage”

...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

`let userName`

Best Practice:
camelCase

`let ageGroup5`

Only letters
and digits

`let $kindOfSpecial`

Starting with \$
is allowed

`let _internalValue`

Starting with _
is allowed

Not Allowed / Not Recommended

`let user_name`

Allowed but
bad practice!

`let 21Players`

Starting digits
not allowed

`let user-b`

No special
characters!

`let let`

Keywords not
allowed



Operators

+	Add two numbers	=	Assign value to variable
-	Subtract two numbers		
*	Multiply two numbers		
/	Divide two numbers		
%	Divide two numbers, yield remainder		
**	Exponentiation (e.g. $2^{**} 3 = 8$)		

Functions

“Code on Demand”

(1) Define Function



(2) Call 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**).

`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	=	Assign value to variable
-	Subtract two numbers	+ =, - =, ...	Perform calculation and re-assign result to variable
*	Multiply two numbers	++, --	Increment / Decrement variable value + re-assign
/	Divide two numbers		
%	Divide two numbers, yield remainder		
**	Exponentiation (e.g. $2^{**} 3 = 8$)		



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

Default value of
uninitialized variables

Never assumed by
default

You shouldn't assign
undefined as a value
manually

You can assign this is
a value if you want to
“reset” / “clear” a
variable

NaN

Not a type!

Technically, it's of type
number and can
therefore be used in
calculations

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

Inline JavaScript Code

```
<script>  
alert('Hi there!');  
</script>
```

Executed directly when the browser (HTML parser) reaches this tag

Typically only used for very trivial sites/ scripts (you'd create a huge HTML file otherwise)

External / Imported JavaScript Files

```
<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!

Requested & loaded when browser (HTML parser) reaches this tag

Execution behavior depends on configuration: **async** and **defer** allows HTML parser to continue

Use this approach!



How To Import Your Scripts

Inline

```
<script>  
alert('Hi!');  
</script>
```

Immediately executed,
blocks HTML parsing &
rendering

External File

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

Immediately loaded &
executed, blocks HTML
parsing & rendering

External File (async)

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

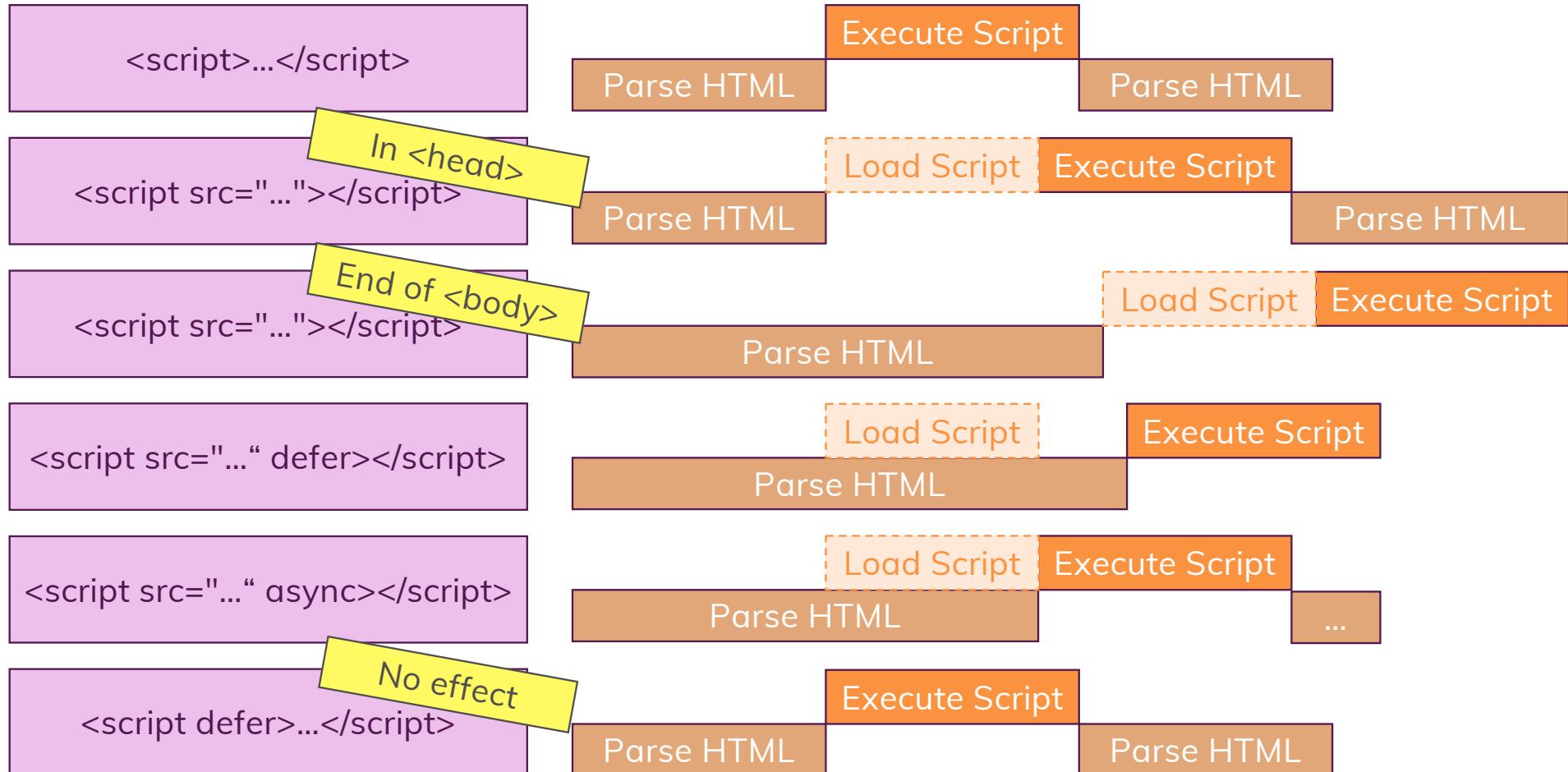
Immediately loaded &
executed thereafter, blocks
HTML parsing & rendering

External File (defer)

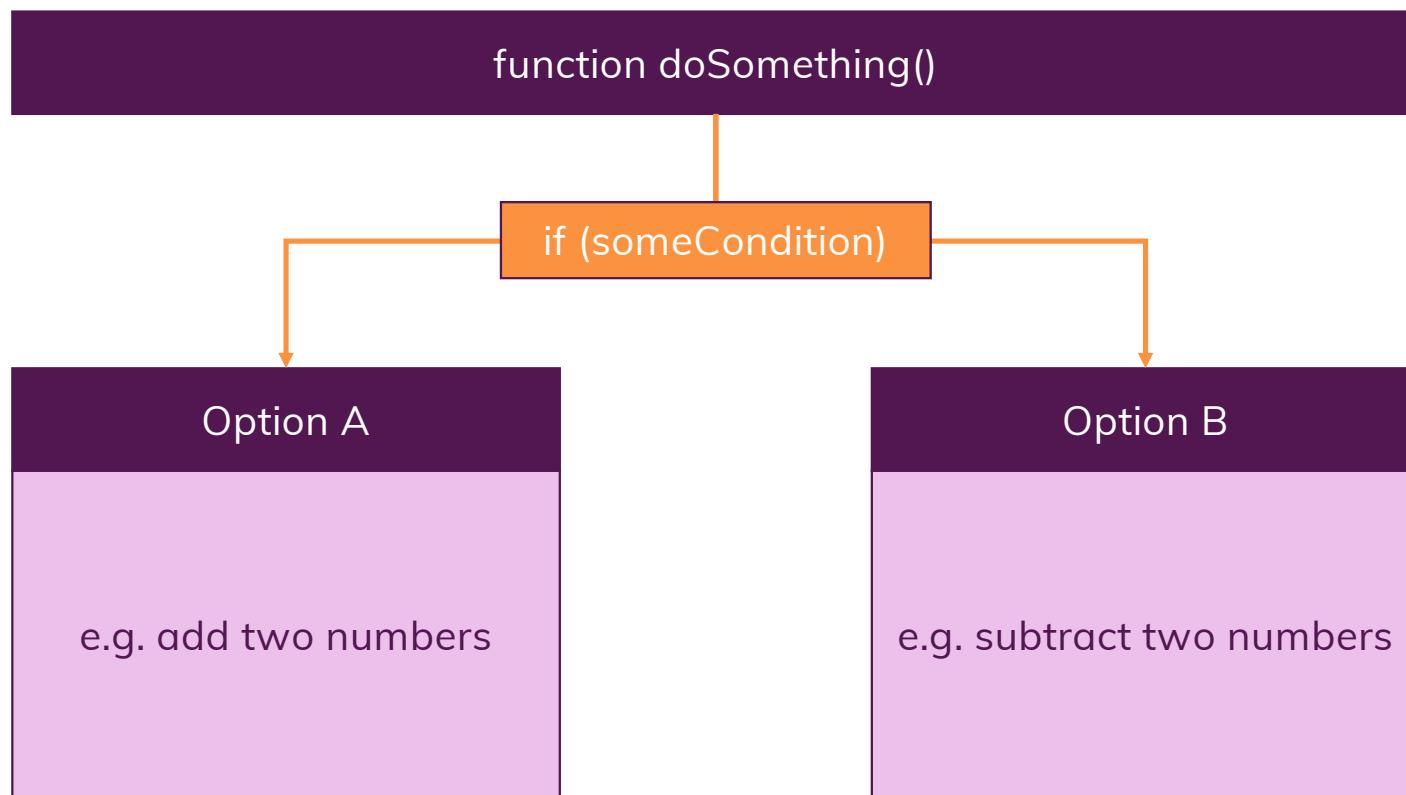
```
<script src="file.js" defer>  
</script>
```

Immediately loaded &
executed after HTML
parsing & rendering

Timeline Summary



Conditional Code Execution



Boolean Operators

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

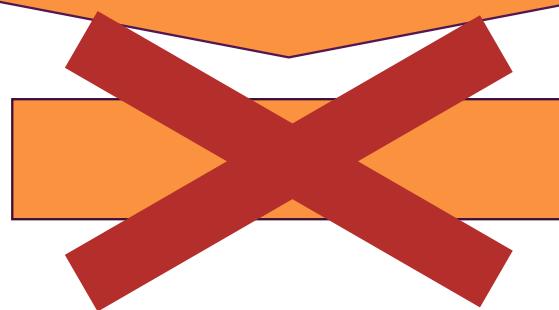
<code>==</code>	Check for value equality	e.g. <code>a == b</code>
<code>!=</code>	Check for value inequality	e.g. <code>a != b</code>
<code>==</code> and <code>!=</code> <i>Prefer over <code>==</code> and <code>!=</code></i>	Check for value AND type (in)equality	e.g. <code>a === b / a !== b</code>
<code>>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

Beware of Objects & Arrays in Comparisons!

{ name: 'Max' }

==== or ==

{ name: 'Max' }

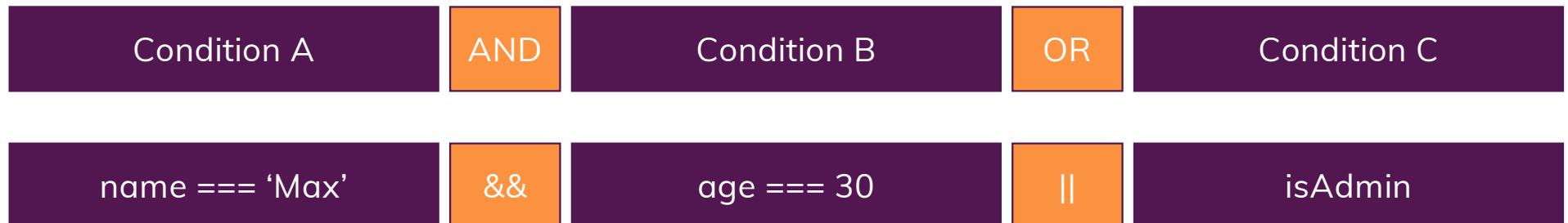


false!

Objects and arrays are kind of special in JavaScript!



Combining Conditions



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

JavaScript Conditions work with Booleans (true / false) OR with “falsy” / “truthy” values

Yields true
(i.e. a Boolean value)

```
const nameInput = 'Max';  
if (nameInput === 'Max') { ... }
```

```
const nameInput = 'Max';  
if (nameInput) { ... }
```

Yields a string,
NOT a boolean

Works!

Also works!

JavaScript tries to coerce values to a Boolean value if a Boolean is required!

Falsy andTruthy Values

0	→	false
ANY other number (incl. negative numbers)	→	true
"" (empty string)	→	false
ANY other non-empty string (incl. “false”)	→	true
{}, [] & all other objects or arrays	→	true
null, undefined, NaN	→	false

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

“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

Some errors can't be avoided (beyond your control as a developer)

User Input Errors

e.g. user enters text like 'hi' instead of a number

Network Errors

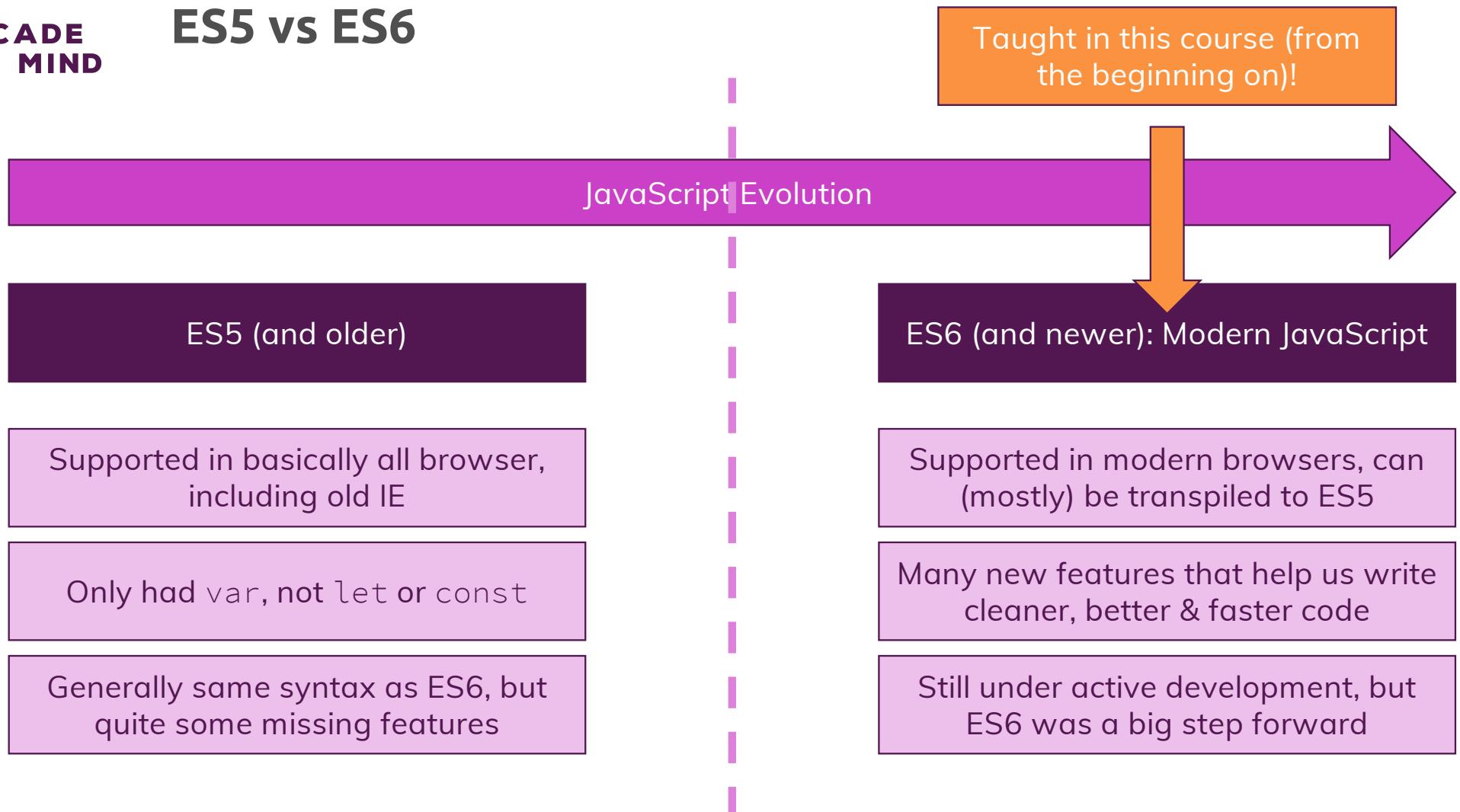
e.g. server is offline

...

Throw and catch errors to fail gracefully or recover if possible

```
try { ... } catch (error) { ... }
```

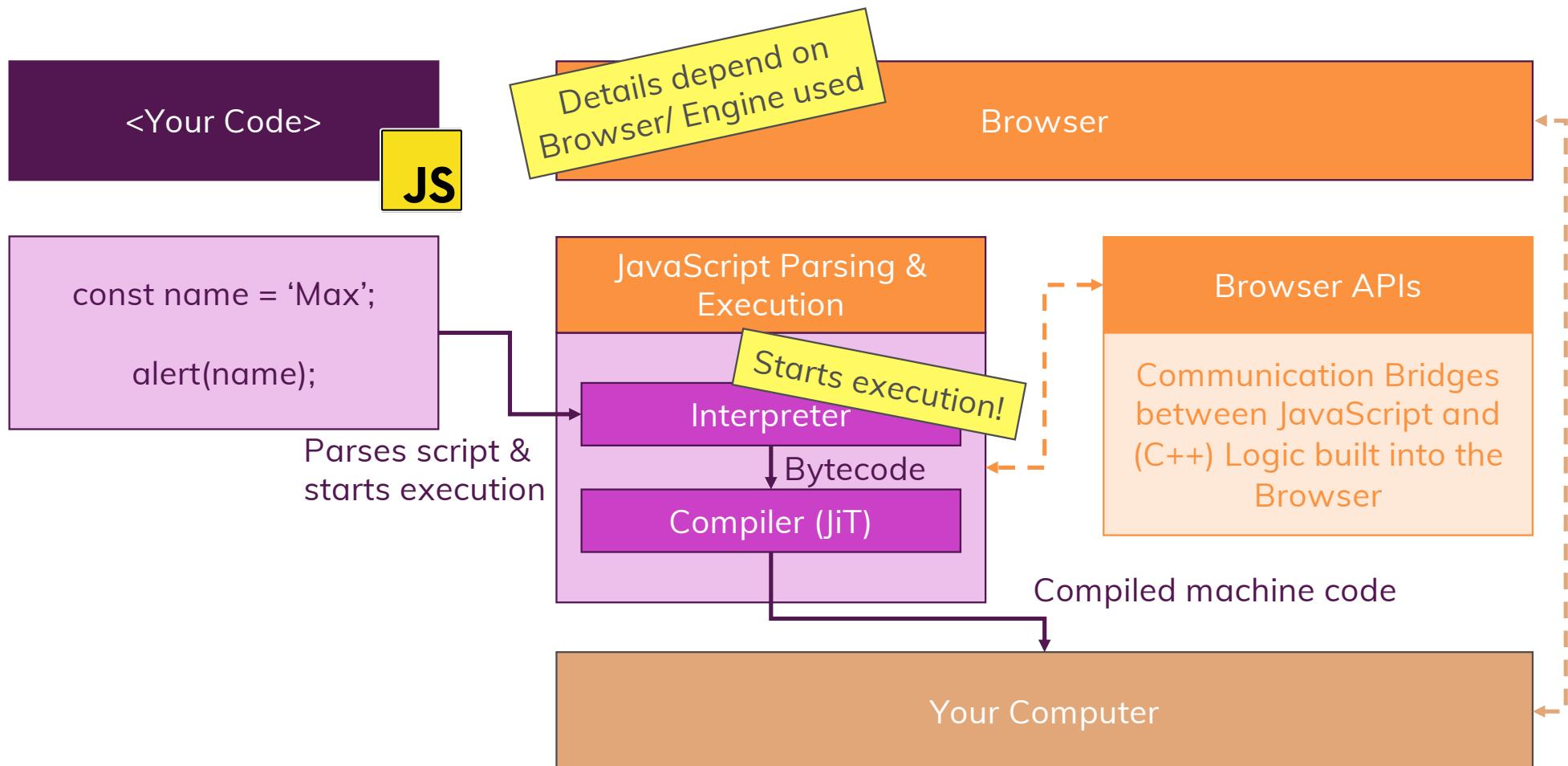
ES5 vs ES6



var vs let vs const



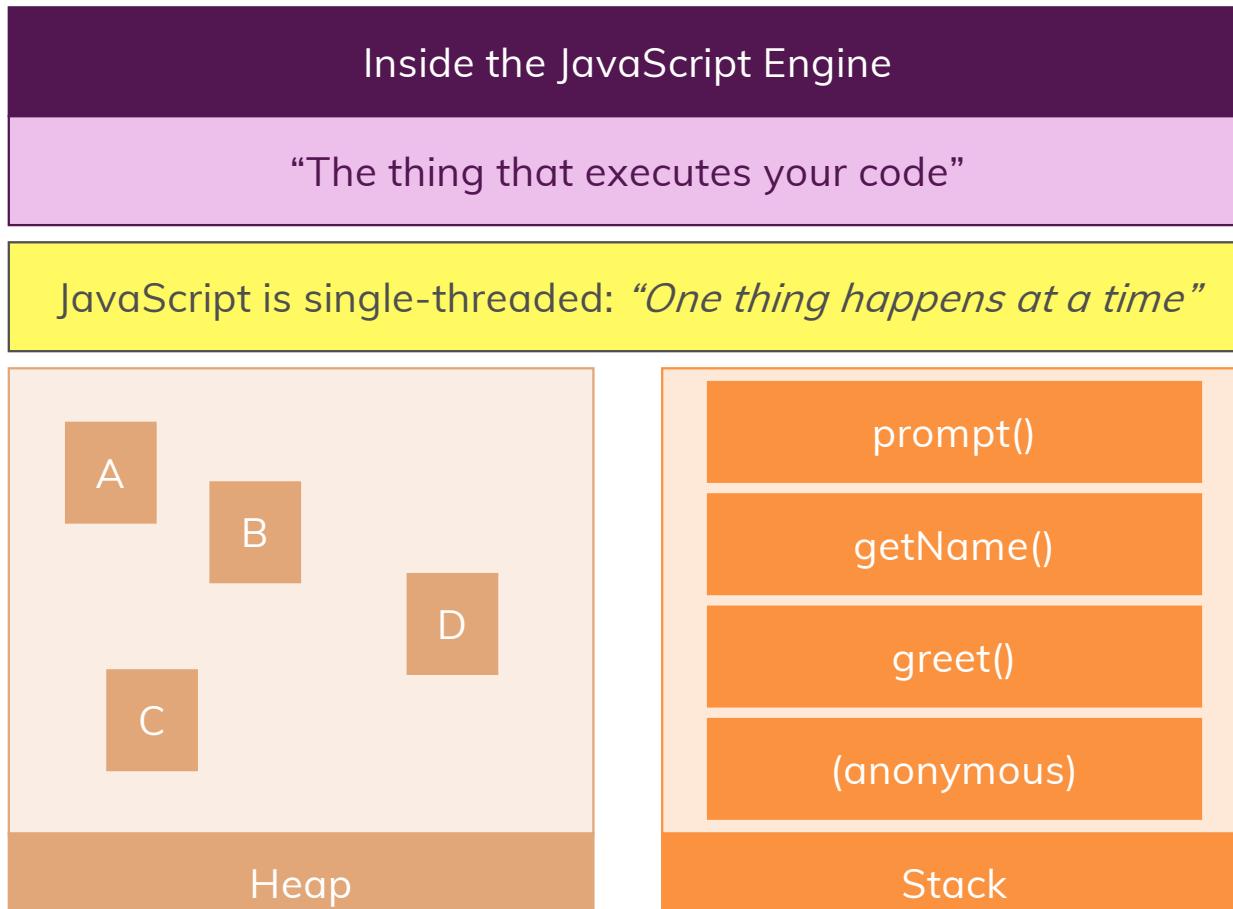
JavaScript Engines & What They Do



How Code Gets Executed

Memory allocation

Stores data in your system memory and manages access to it



Inside the JavaScript Engine

"The thing that executes your code"

JavaScript is single-threaded: "*One thing happens at a time*"

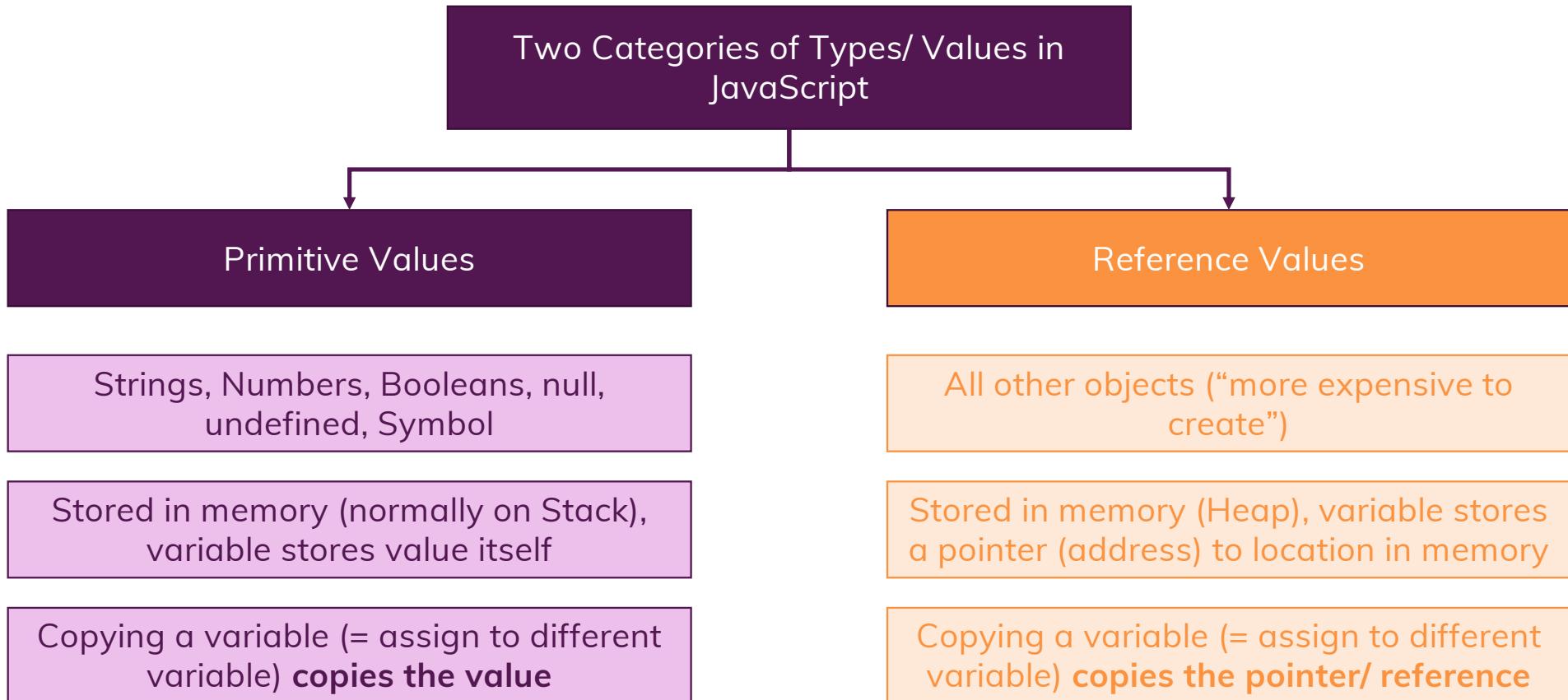
The Stack controls "which thing is happening"

Will become important later
(Event Loop)

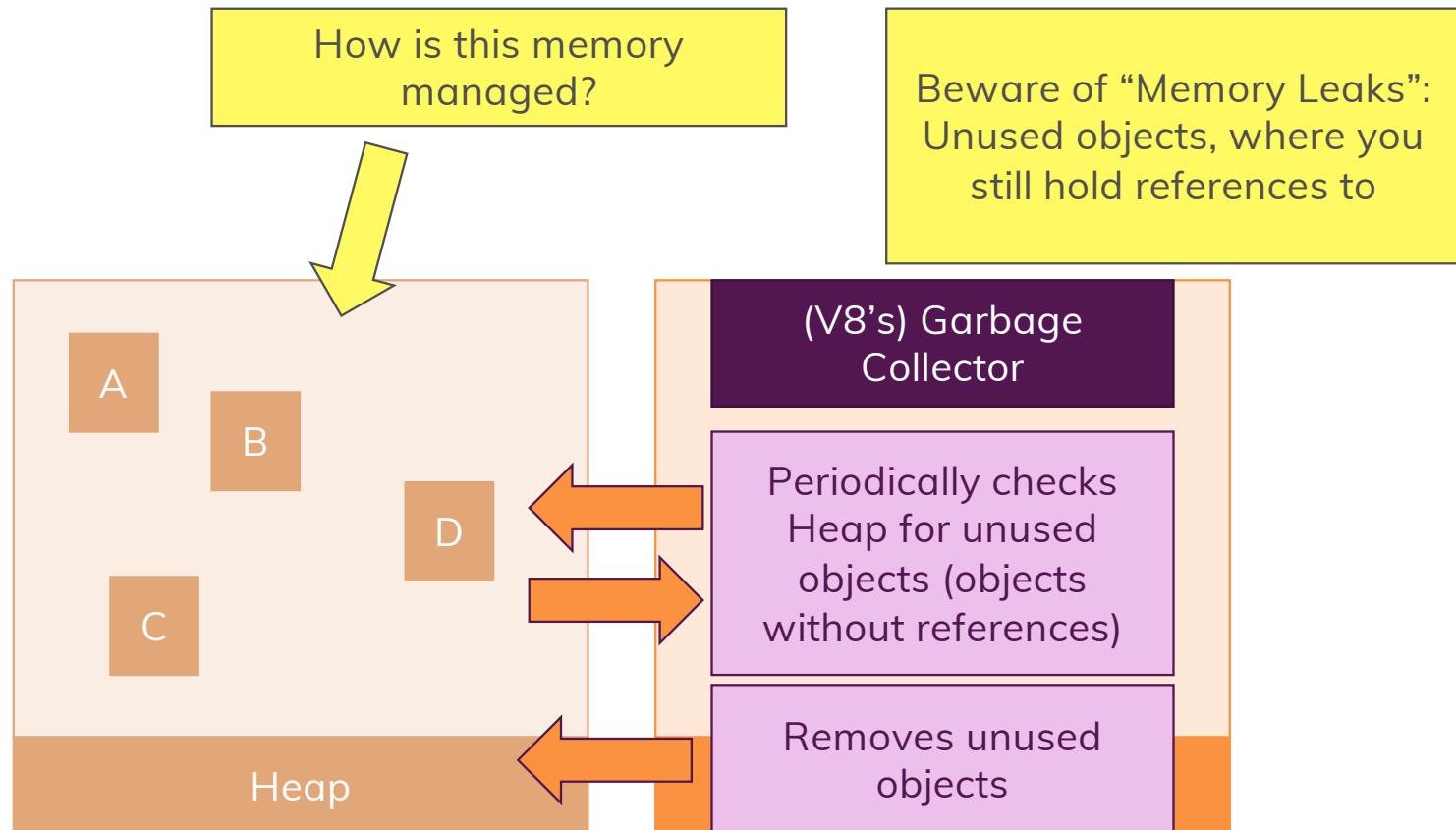
Execution Context

Manages your program flow (function calls and communication)

Primitive vs Reference Values



Garbage Collection





A Different Way of Defining Functions

Function Declaration / Function Statement

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

Hoisted to top, can be declared anywhere in the file (i.e. also after it's used)

Function Expression

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

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

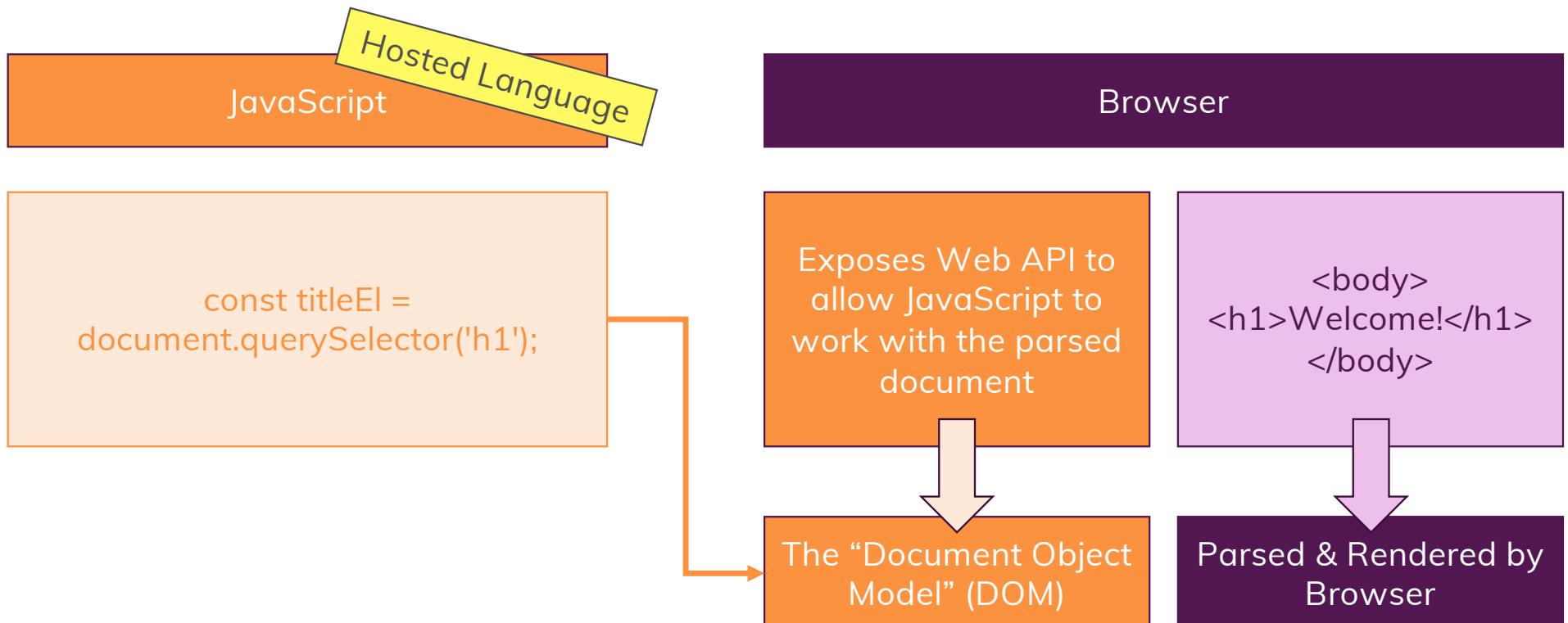
Curly braces can be
omitted, result is returned

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)



DOM is not strictly tied to Browsers! There are other tools that can parse HTML!



document & window

document

Root DOM Node

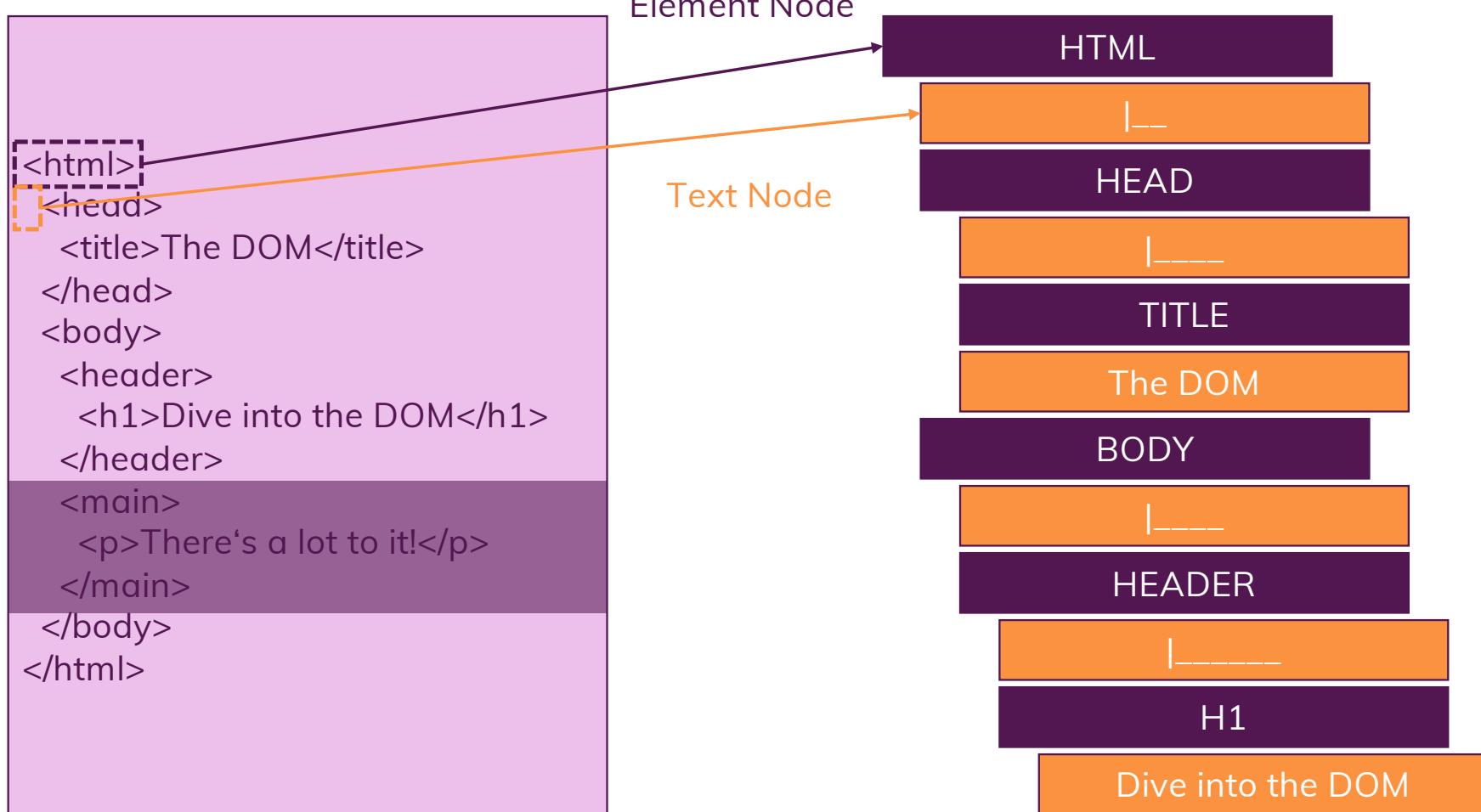
Provides access to element querying, DOM content etc

window

The active Browser Window / Tab

Acts as global storage for script, also provides access to window-specific properties and methods

The Document Object Model (DOM)



Nodes & Elements

Nodes



Elements

The objects that make up the DOM

Elements are
one type of
nodes

HTML tags are “element nodes” (or
just “elements”)

Special properties and methods to
interact with the elements

Text creates “text nodes”

Available methods and properties
depend on the kind of element

Attributes create “attribute nodes”

Can be selected in various different
ways (via JavaScript)

Can be created and removed via
JavaScript



Querying Elements

`querySelector()`, `getElementById()`

Return single elements

Different ways of querying elements (by CSS selector, by ID)

Direct reference to DOM element is returned

`querySelectorAll()`,
`getElementsByTagName()`, ...

Return collections of elements (array-like objects): `NodeList`

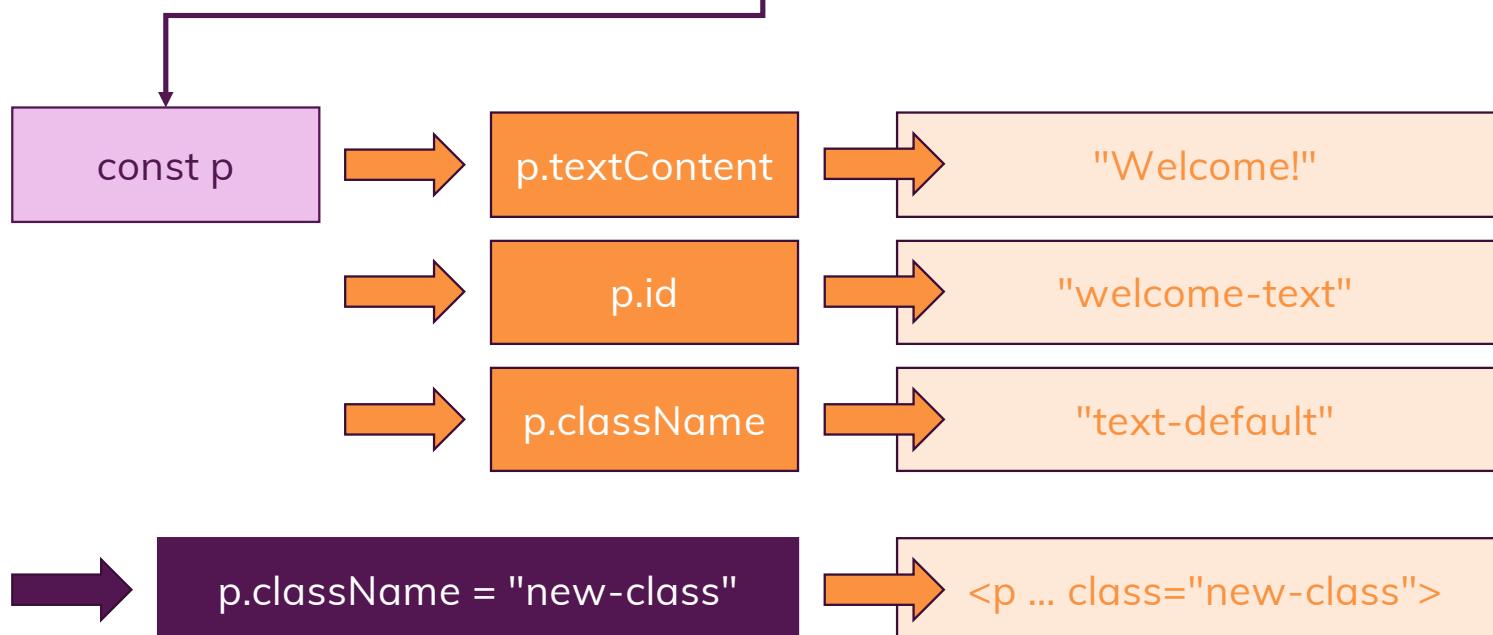
Different ways of querying elements (by CSS selector, by tag name, by CSS class)

`querySelectorAll()` returns a non-live `NodeList`, `getXY` return live `NodeLists`

Evaluating & Manipulating Elements

```
<p id="welcome-text" class="text-default">Welcome!</p>
```

```
document.getElementById('welcome-text');
```



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

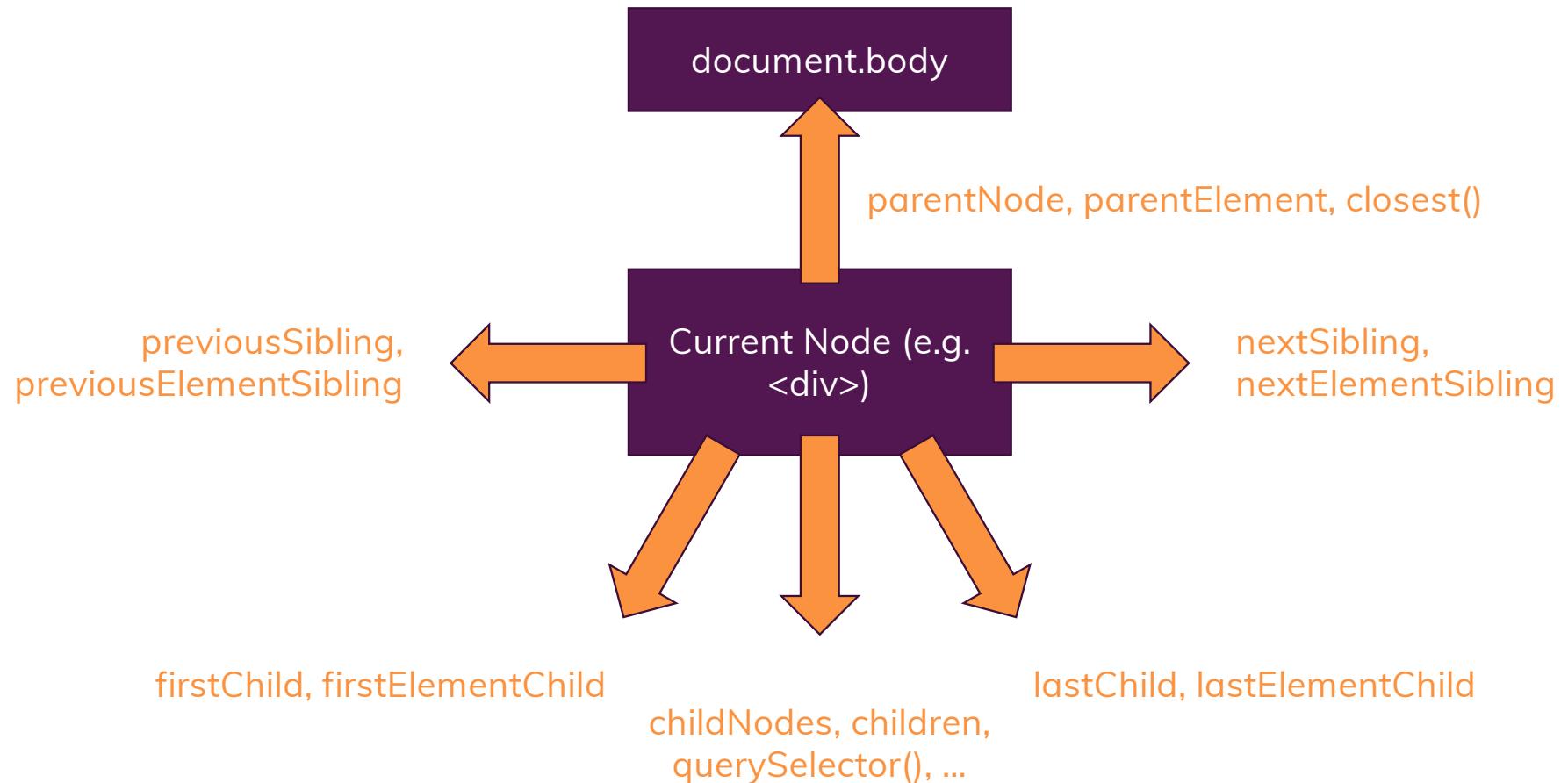
MDN for full reference of available properties + their behavior!

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
<pre><div> <p> A test! </p> </div></pre>			
<p>↓</p> <p><code><p> is a child of <div>. isn't!</code></p>	<p>↓</p> <p><code><p> is a descendant of <div>. So is !</code></p>	<p>↓</p> <p><code><div> is a parent of <p> but not of !</code></p>	<p>↓</p> <p><code><div> is an ancestor of <p> and of !</code></p>

Traversing the DOM





Styling DOM Elements

Via style Property

Directly target individual CSS styles (on the element)

Controls styles as inline styles on the element

Style property names are based on CSS properties but have adjusted names (e.g. backgroundColor)

Via className

Directly set the CSS classes assigned to the element

Set/ Control all classes at once

You can also control the id or other properties

Via classList

Conveniently add, remove or toggle CSS classes

Fine-grained control over classes that are added

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”?

An Iterable

Technically

Objects that implement the “iterable” protocol and have an @@iterator method (i.e. Symbol.iterator)

To us humans:

Objects where you can use the for-of loop.

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



What's an “Array-like Object”?

An Array-like Object

Technically

Objects that have a length property and use indexes to access items

To us humans:

See technical explanation

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

Maps & Sets

Arrays

Store (nested) data of any kind and length

Iterable, also many special array methods available

Order is guaranteed, duplicates are allowed, zero-based index to access elements

Sets

Store (nested) data of any kind and length

Iterable, also some special set methods available

Order is NOT guaranteed, duplicates are NOT allowed, no index-based access

Maps

Store key-value data of any kind and length, any key values are allowed

Iterable, also some special map methods available

Order is guaranteed, duplicate keys are NOT allowed, key-based access



Maps vs Objects

Maps

Can use ANY values (and types) as keys

Better performance for large quantities of data

Better performance when adding + removing data frequently

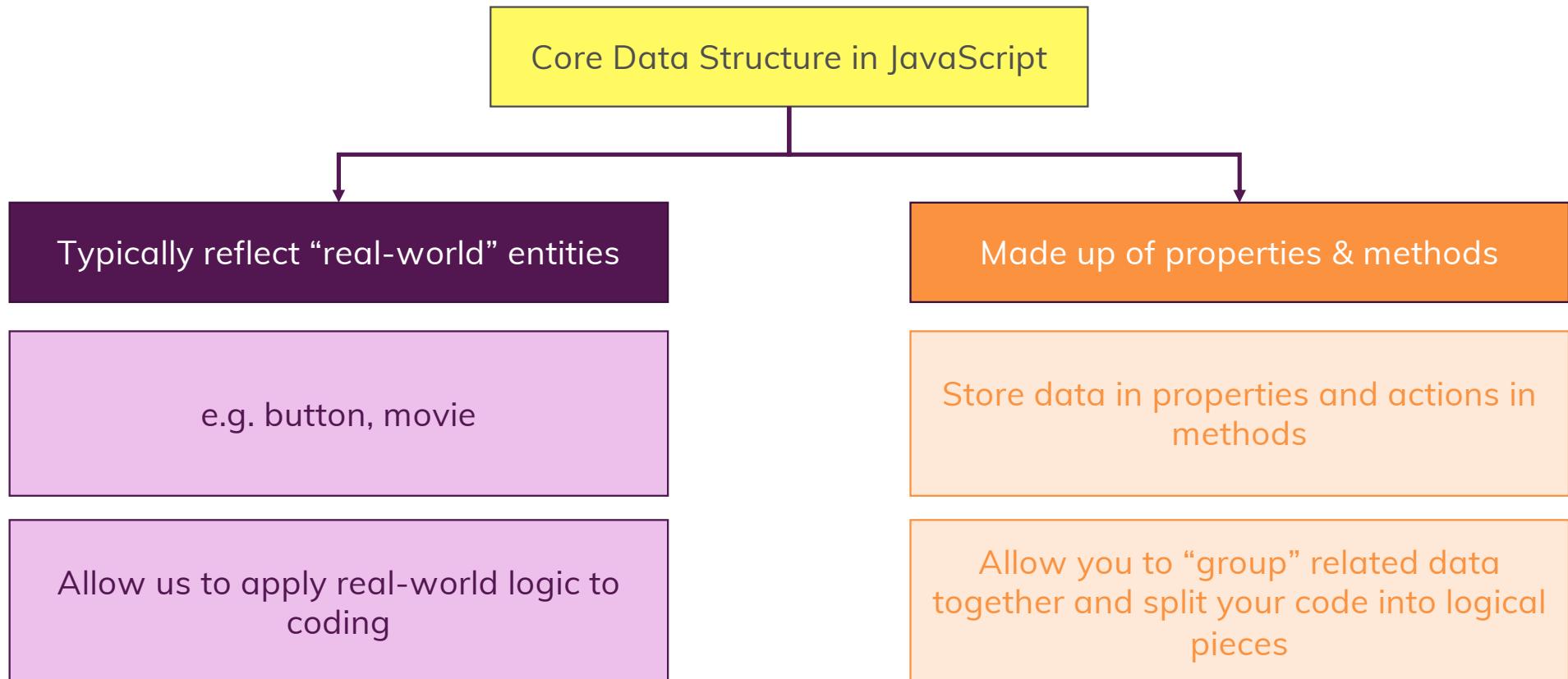
Objects

Only may use strings, numbers or symbols as keys

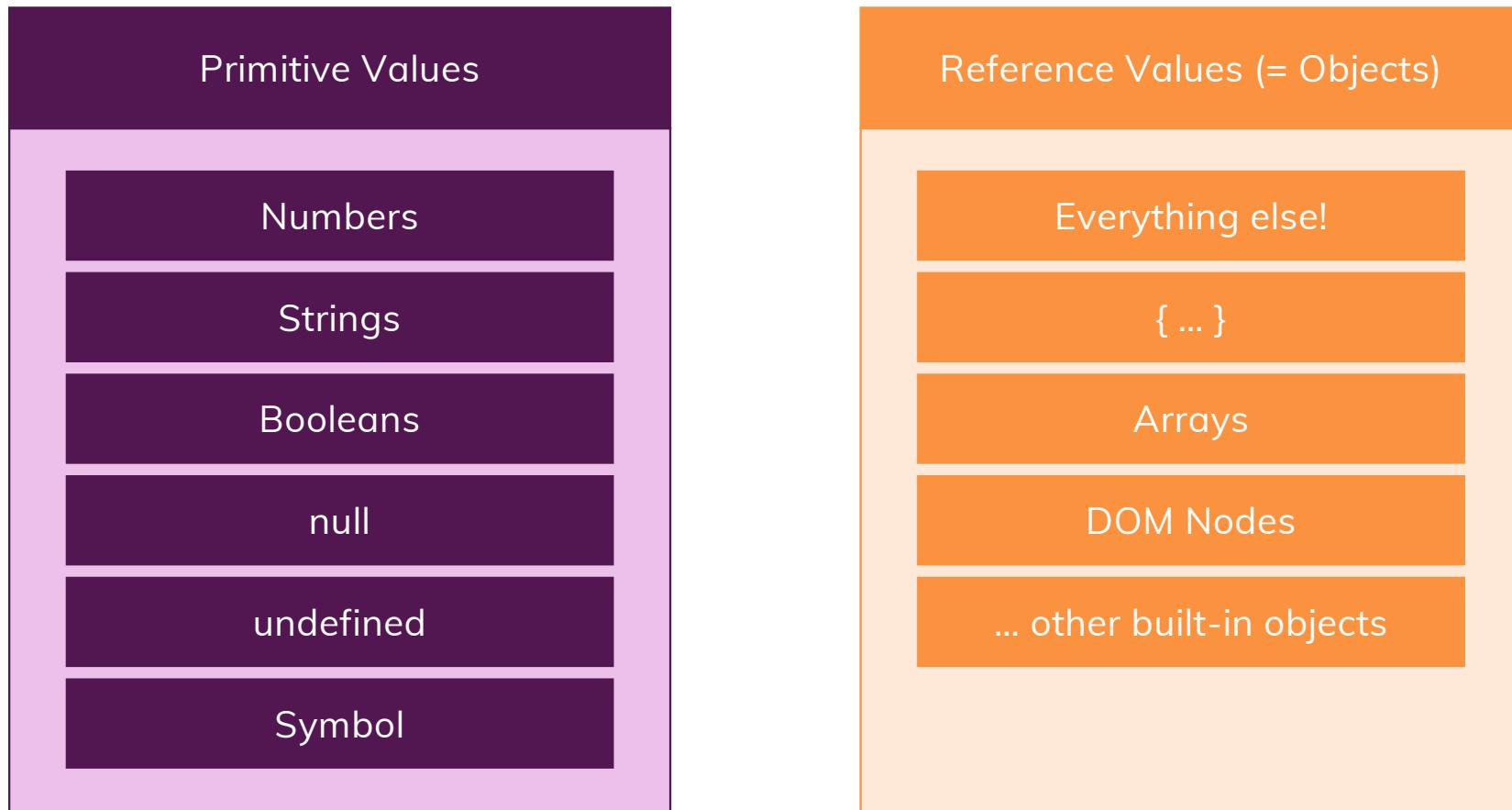
Perfect for small/ medium-sized sets of data

Easier/ quicker to create (typically also with better performance)

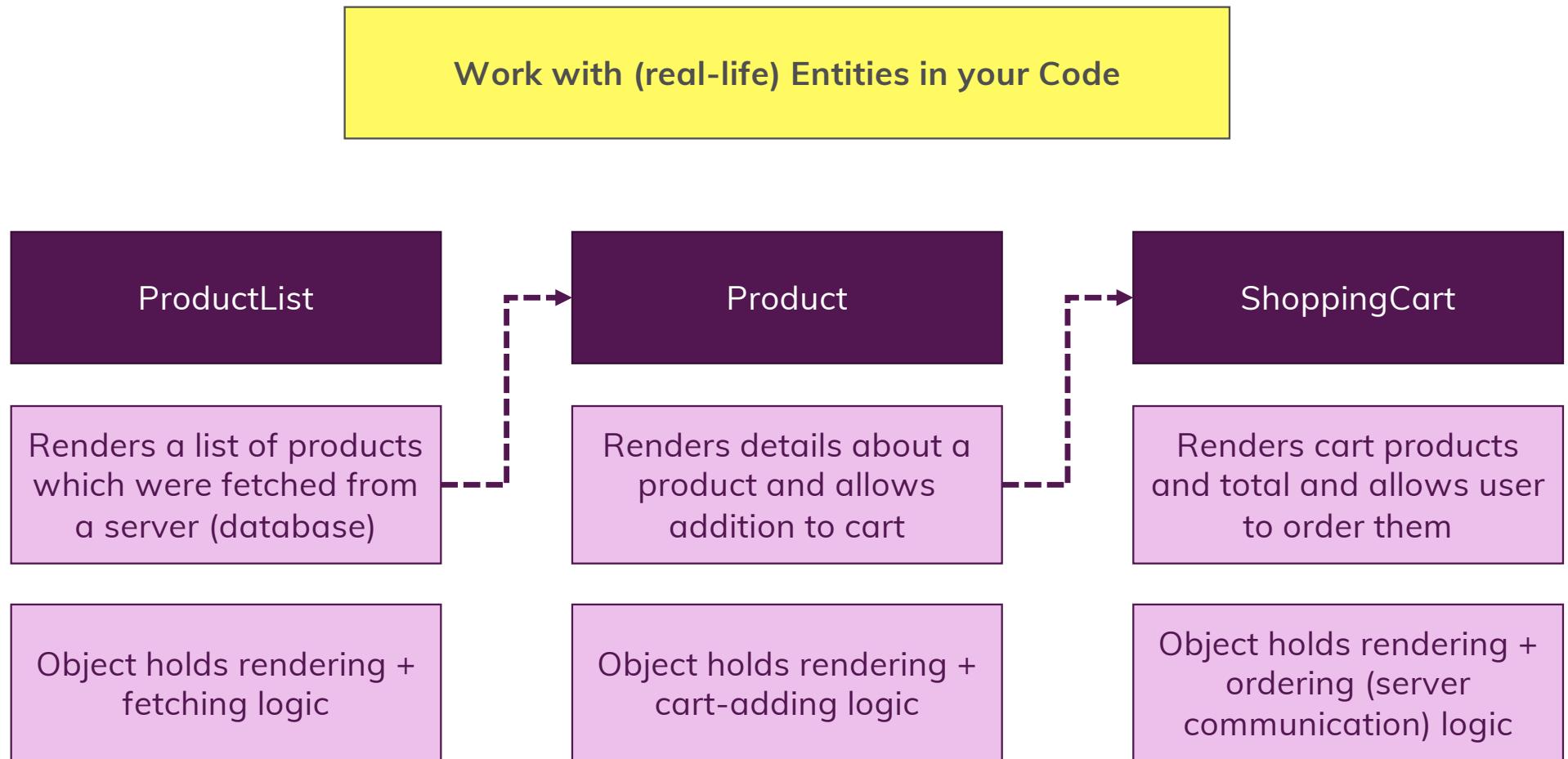
What are Objects?



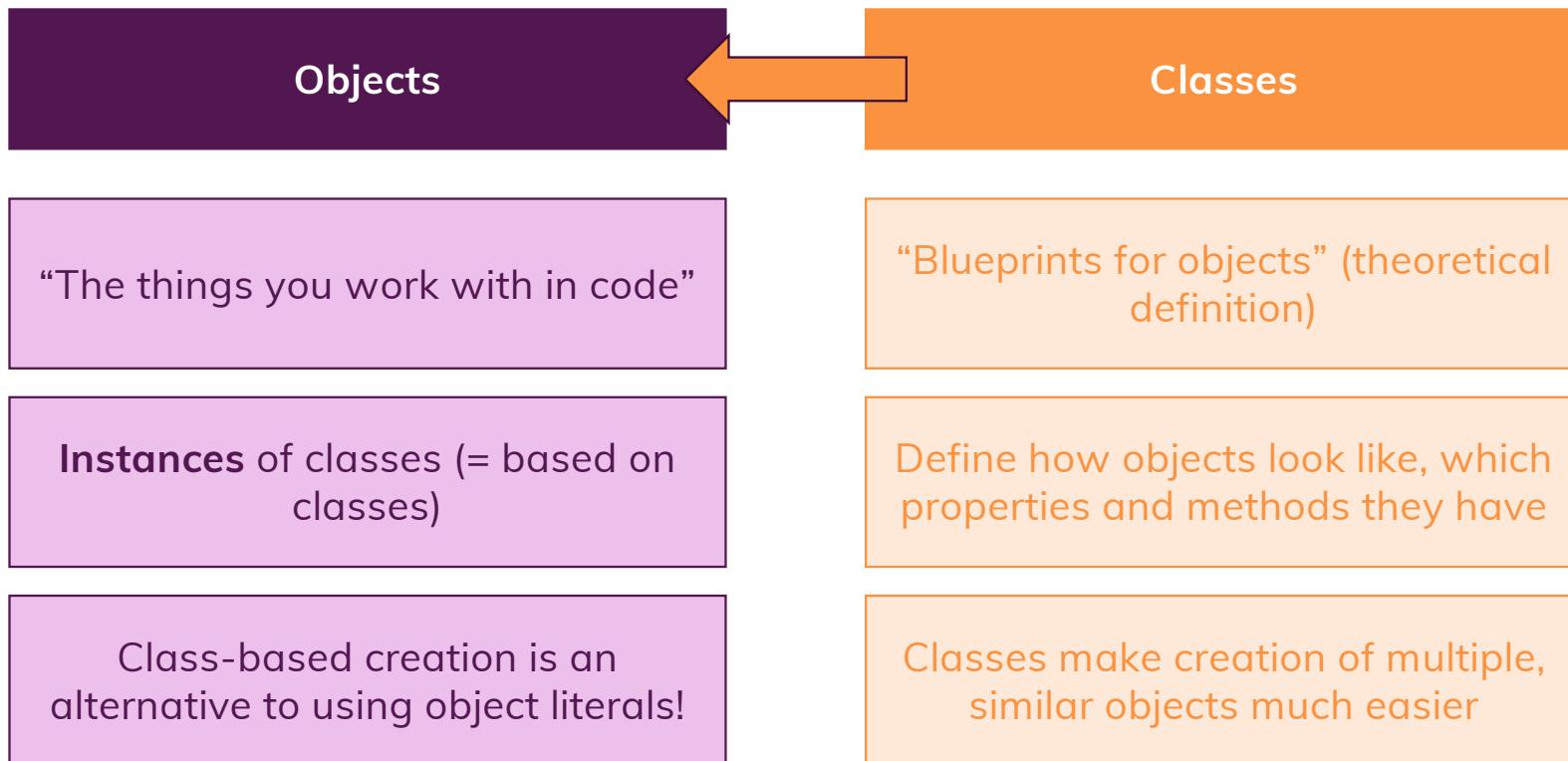
Objects in JavaScript



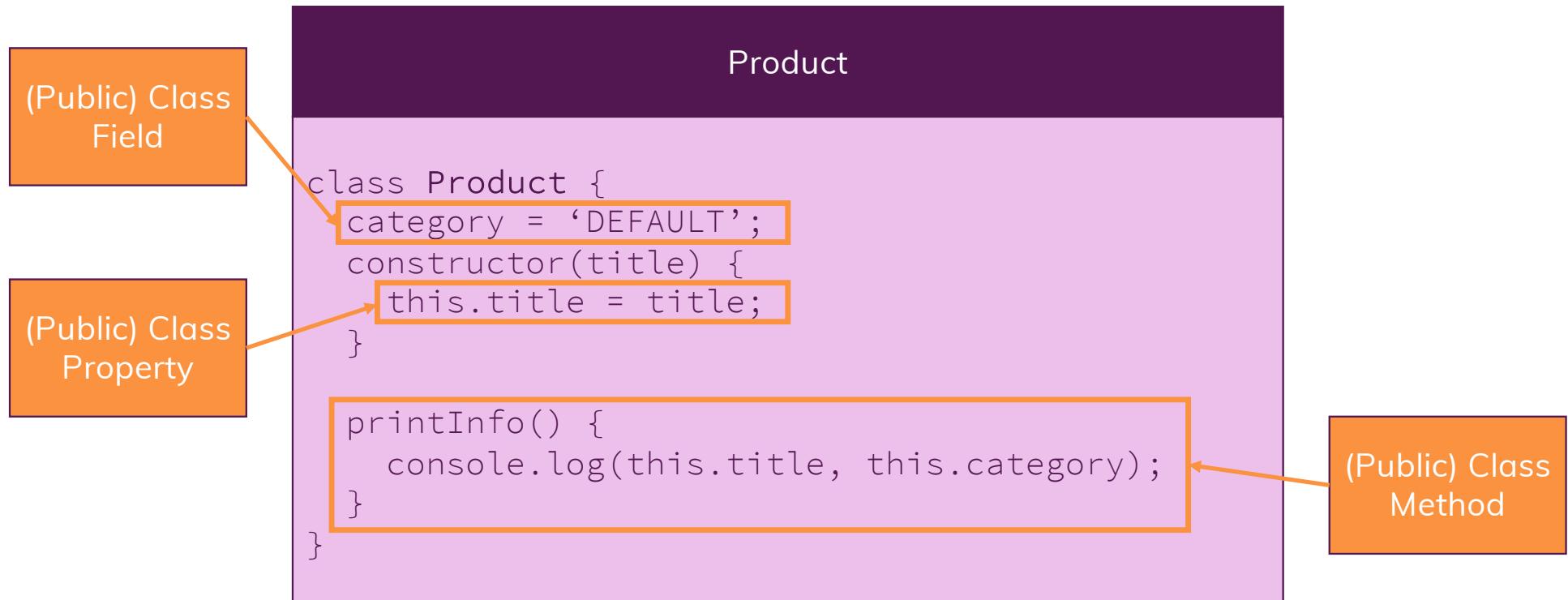
What's Object-oriented Programming (OOP)?



Classes & Instances



Class Fields (vs Properties)



“Nice to know”: Fields define properties for classes

Static Properties, Fields & Methods

Static Field / Property / Method

Defined with `static` keyword

Only accessible on class itself,
without instantiation (i.e. not on
instance)

Typically used in helper classes,
global configuration etc.

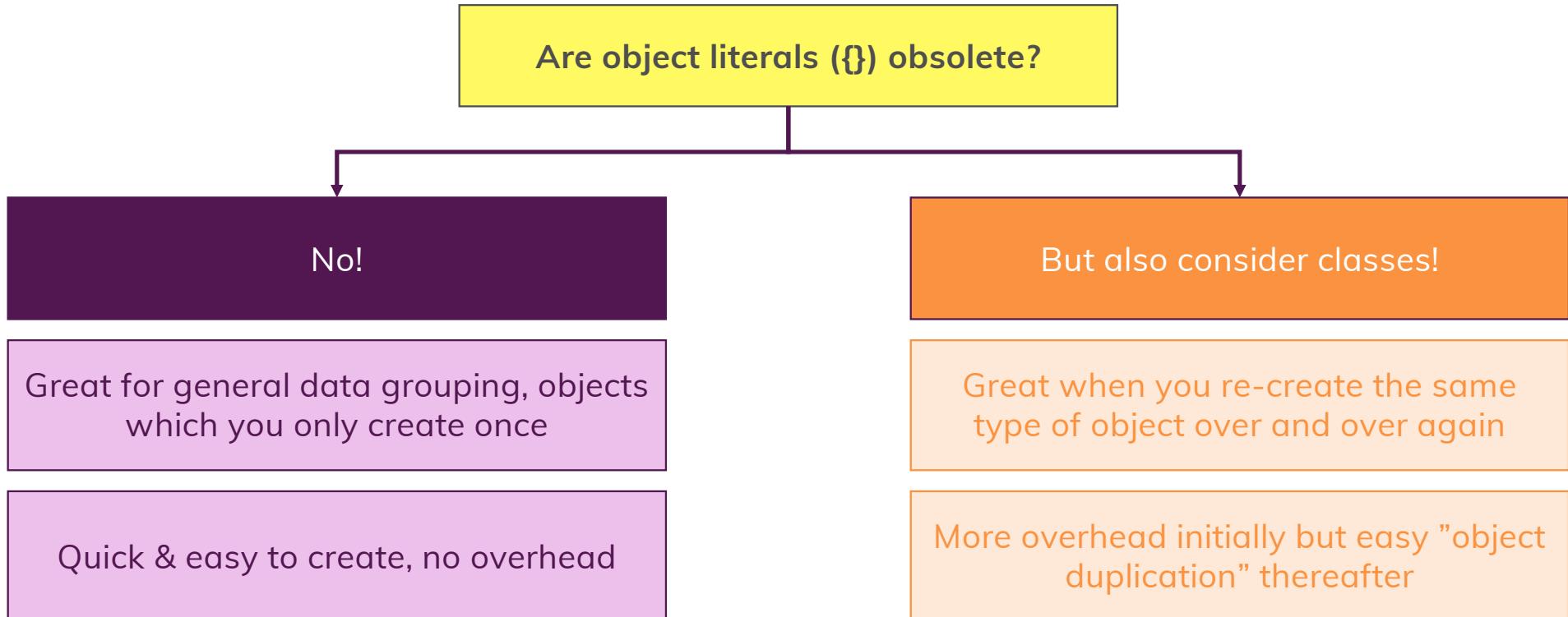
Instance Field / Property / Method

Defined without `static` keyword

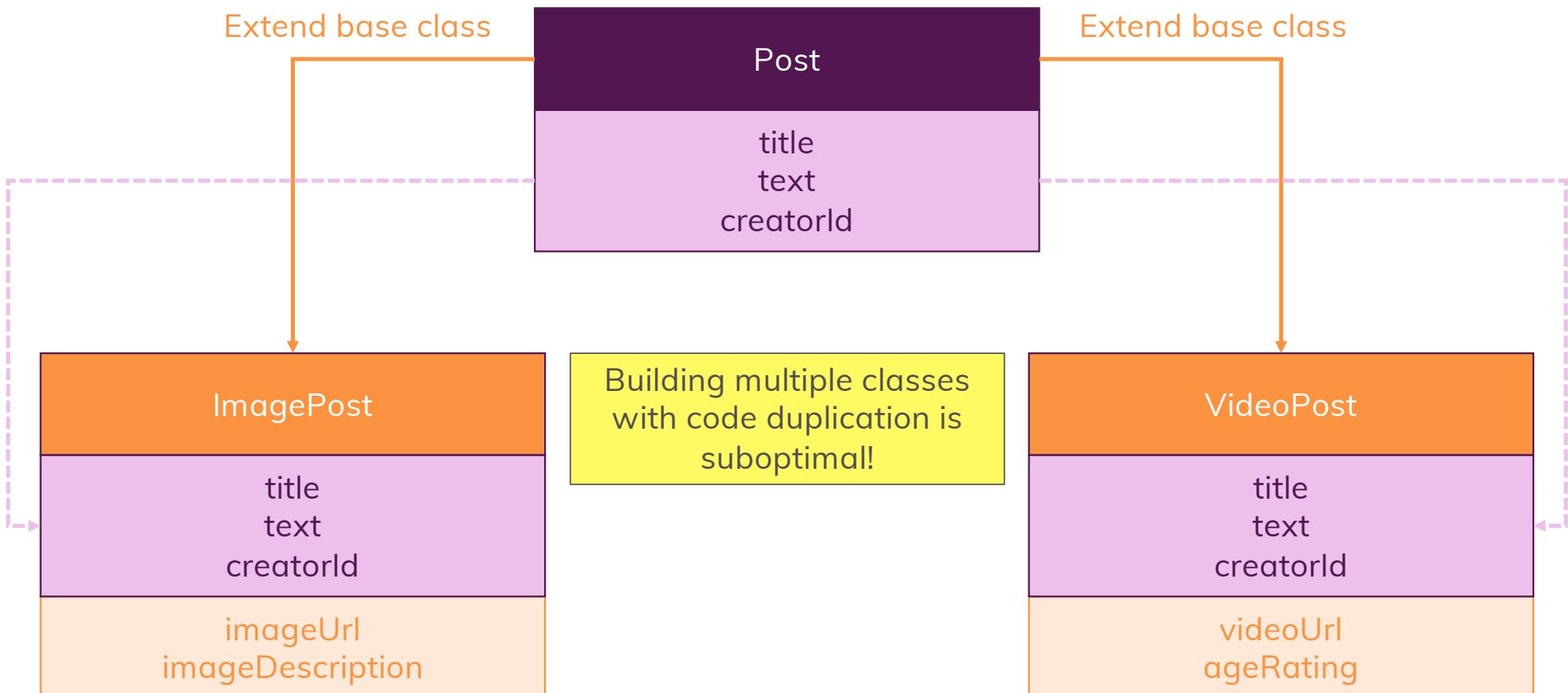
Only accessible on instances (=
objects) based on class

Used for core, re-usable logic

When to use Classes



Inheritance





Private Fields, Properties & Methods

Public

Accessible OUTSIDE of the class/
object

The “things” you work with in your
other code

Example: product.buy()

Private

Accessible ONLY INSIDE of the
class/ object

The “things” you work with in your
class only (to split & re-use code)

Example: Hard-coded (fallback)
values, re-used class-specific logic

Constructor Functions vs Classes

Constructor Functions

Blueprint for Objects

Properties & Methods

Can be called with new

All properties and methods are enumerable

Not in strict mode by default

Classes

Blueprint for Objects

Properties & Methods

Must be called with new

Methods are non-enumerable by default

Always in strict mode



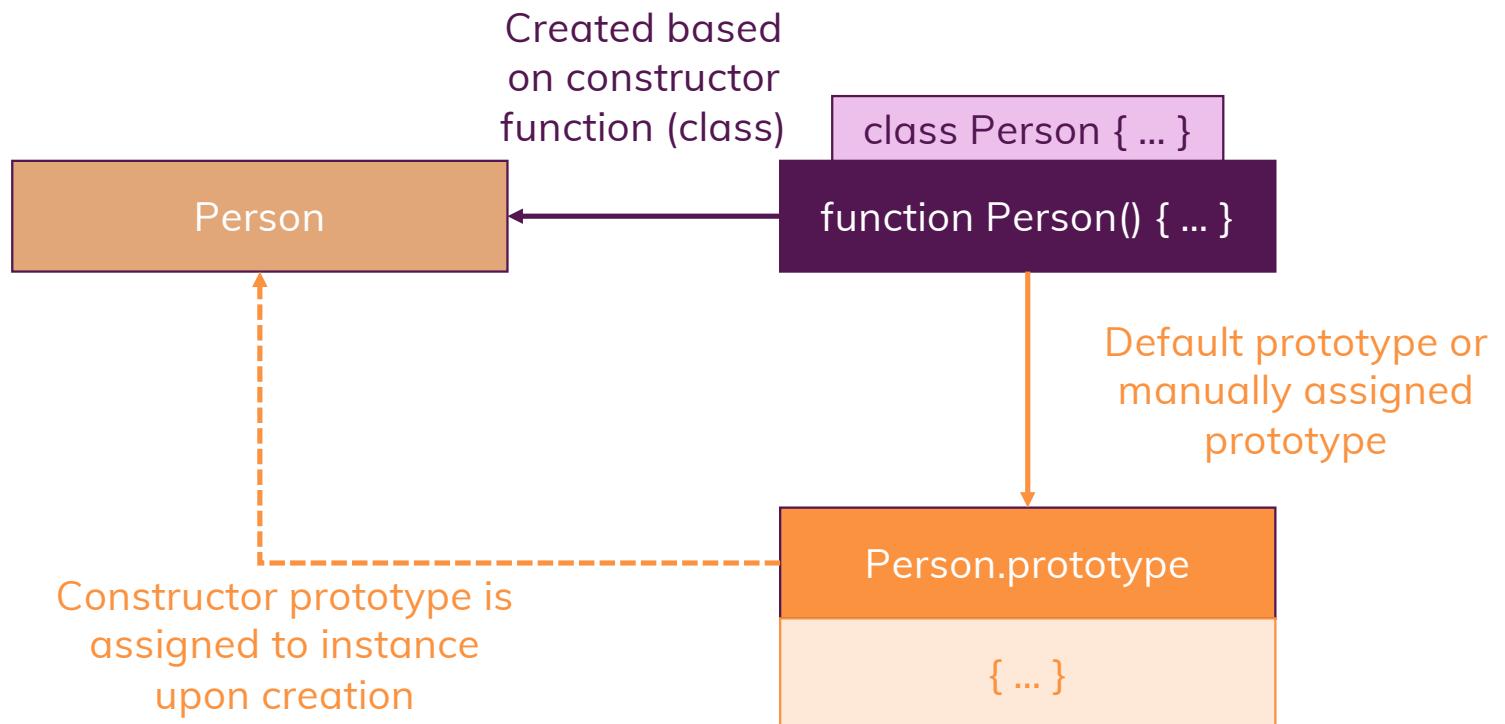
What are “Prototypes”?

JavaScript uses “Prototypical Inheritance”

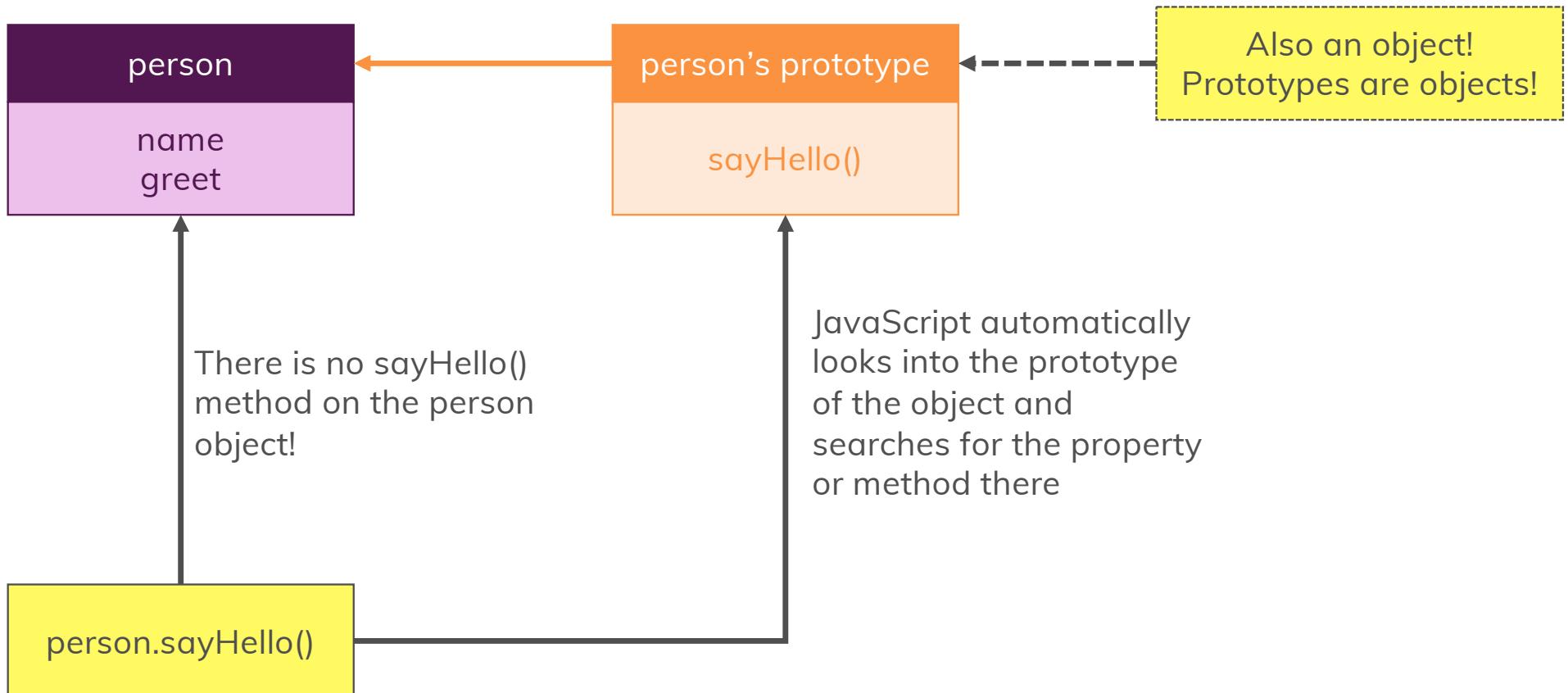
The class Syntax is basically just “syntactic sugar”

Constructor Functions & Prototypes power JavaScript Objects

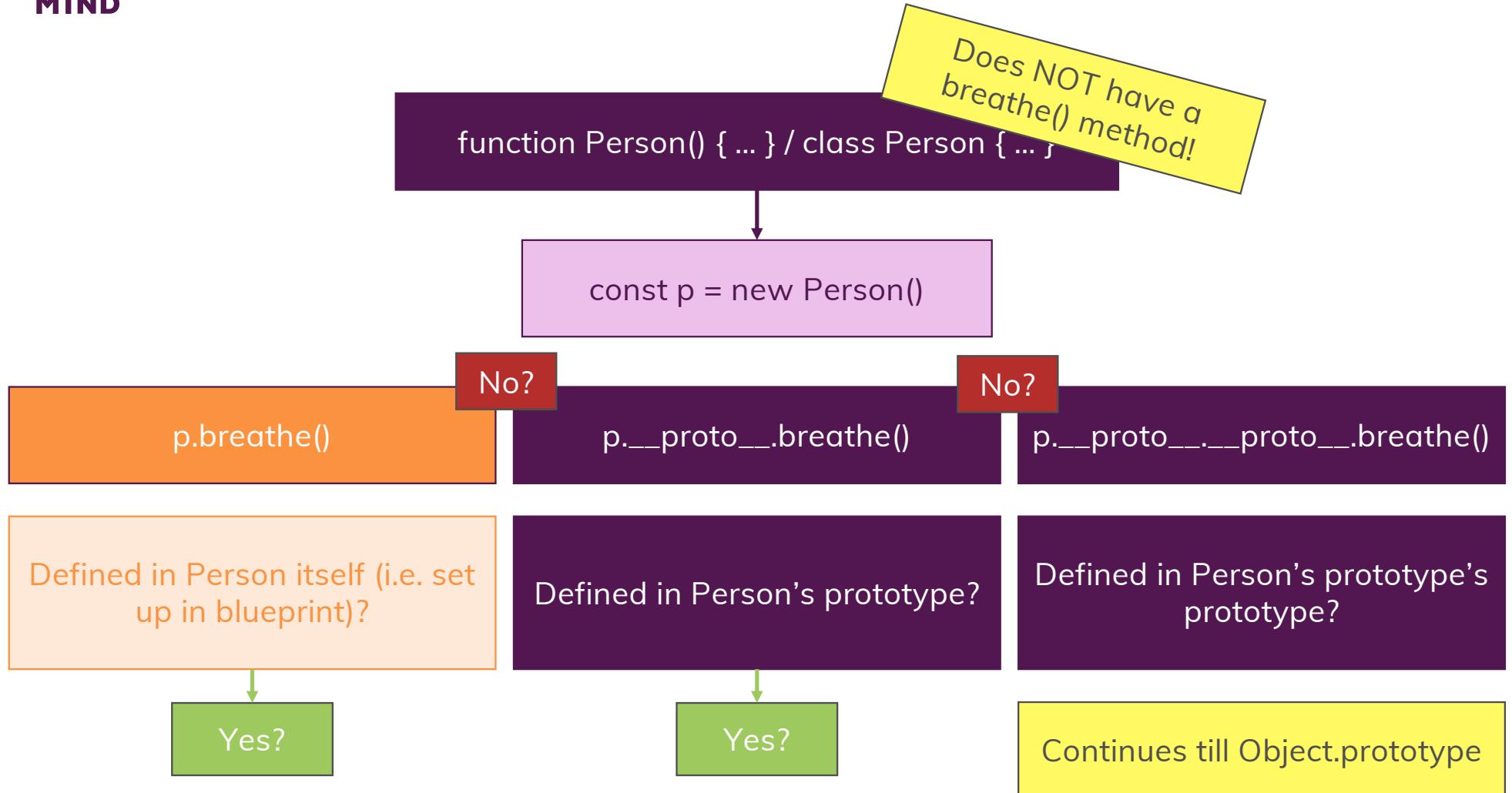
Prototypes



Prototype Objects == “Fallback Objects”



The Prototype Chain



Prototypes & “Method Types”

Method Shorthand

```
class Person {  
  greet() {  
    console.log('Hello');  
  }  
}
```

Assigned to Person's prototype and hence shared across all instances (NOT re-created per instance).

Property Function

```
class Person {  
  greet = function() { ... }  
  constructor() {  
    this.greet2 = function() { ... }  
  }  
}
```

Assigned to individual instances and hence re-created per object. `this` refers to “what called the method”.

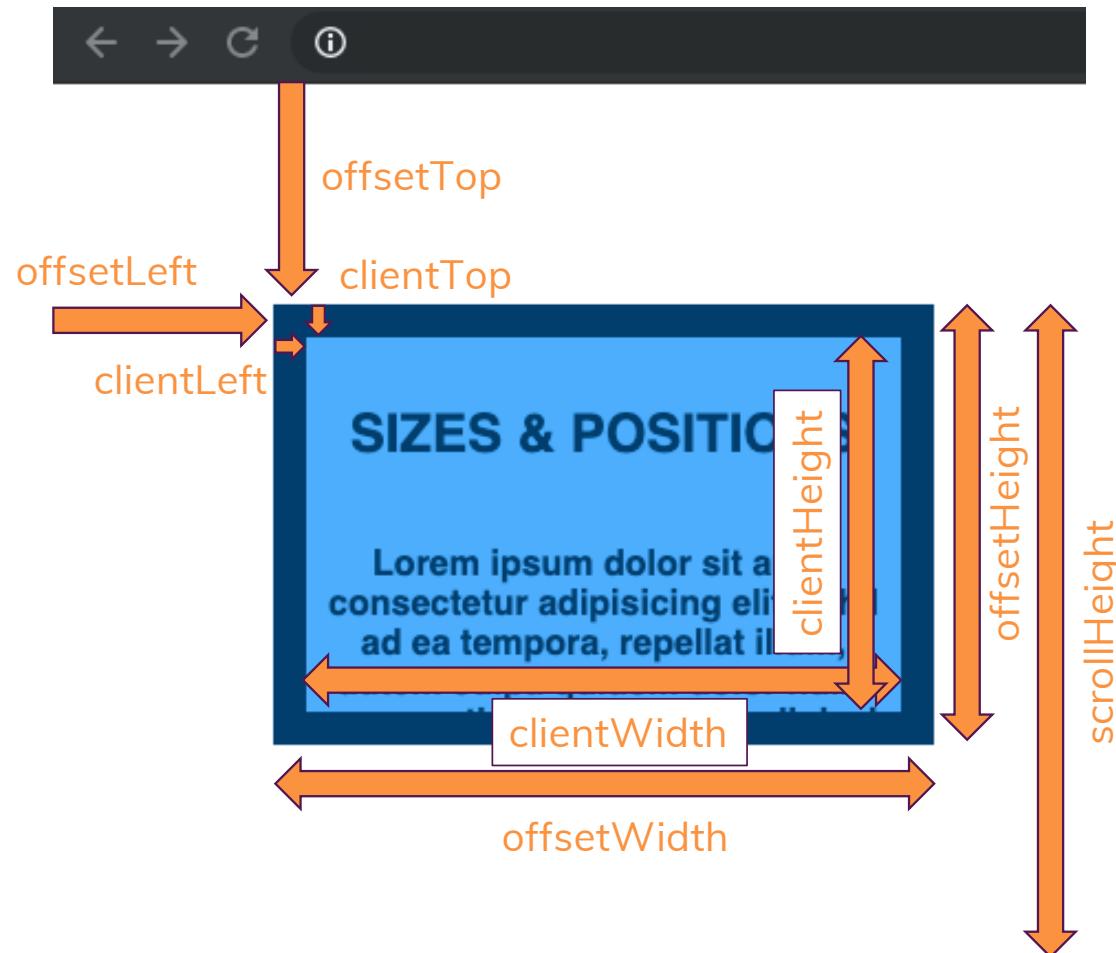
Property Arrow Function

```
class Person {  
  greet = () => { ... }  
  constructor() {  
    this.greet2 = () => { ... }  
  }  
}
```

Assigned to individual instances and hence re-created per object. `this` always refers to instance.

Unnecessary re-creations of function can cost performance!

Sizes & Positions



Events in JavaScript

Run Code Upon Certain Events

Exact Implementation Differs

Events Typically Transport Data

e.g. Browser: addEventListener()



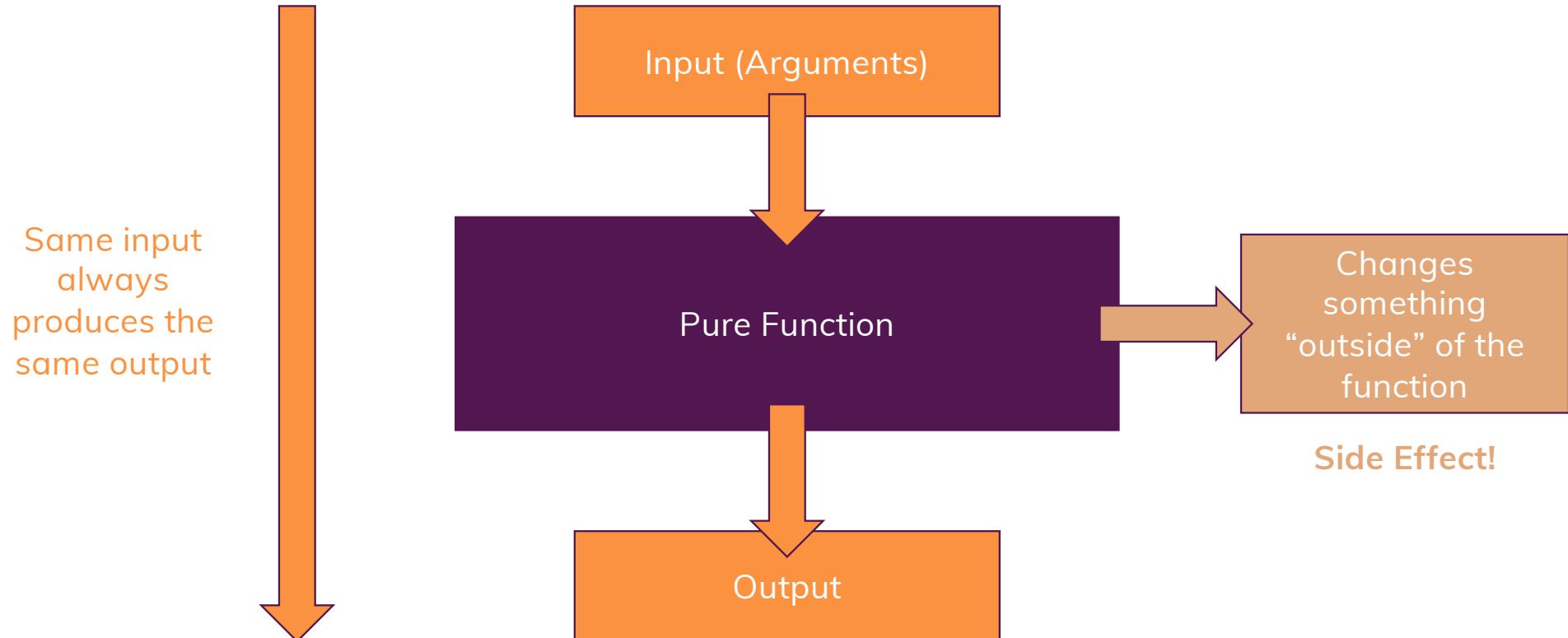
`addEventListener('...', event => ...)`

e.g. NodeJS: on()

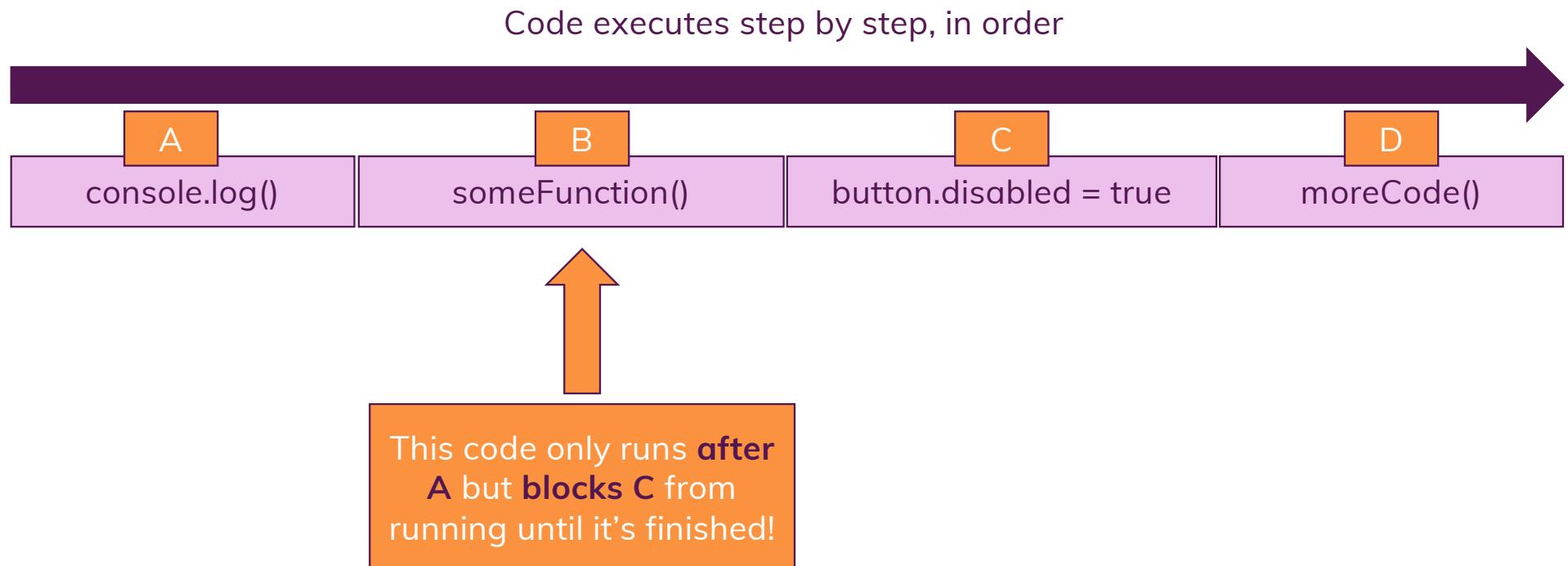


`on('...', event => ...)`

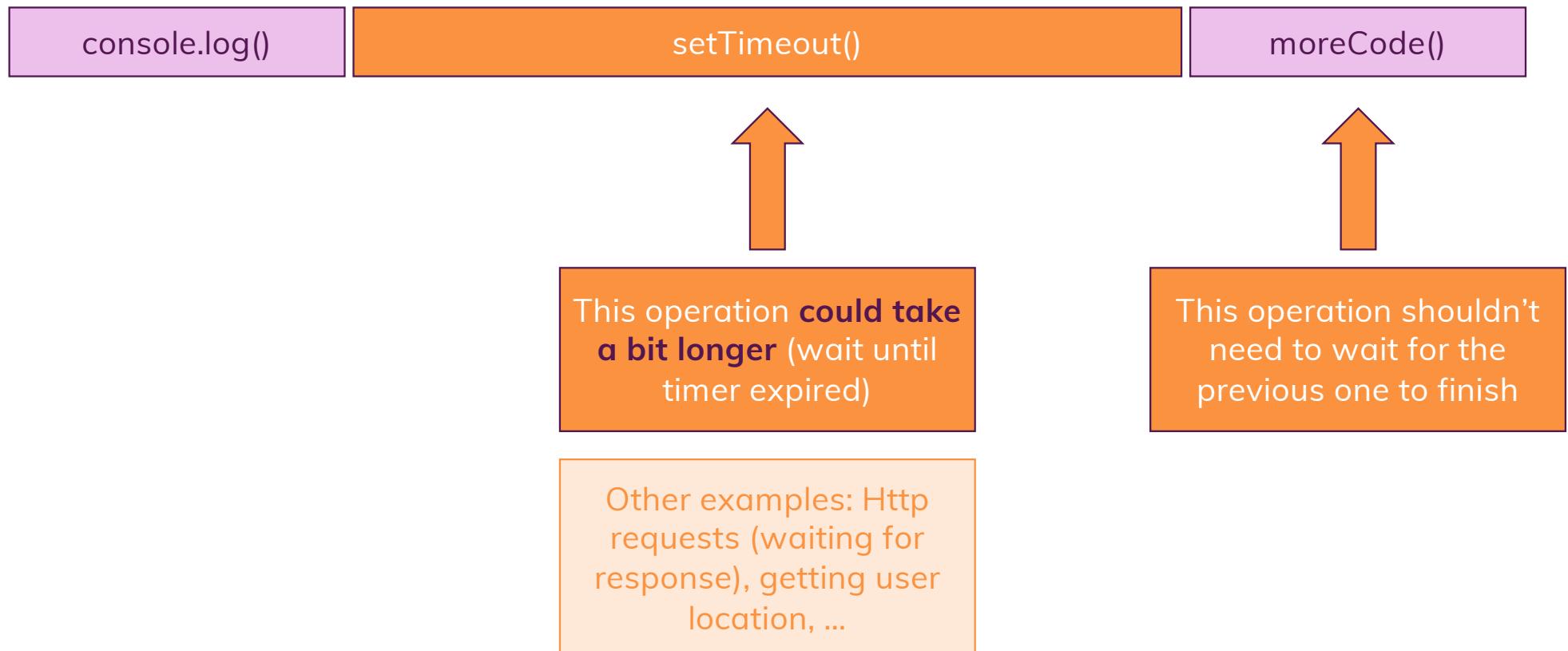
What's a “Pure” Function without “Side Effects”?



JavaScript is Single-Threaded

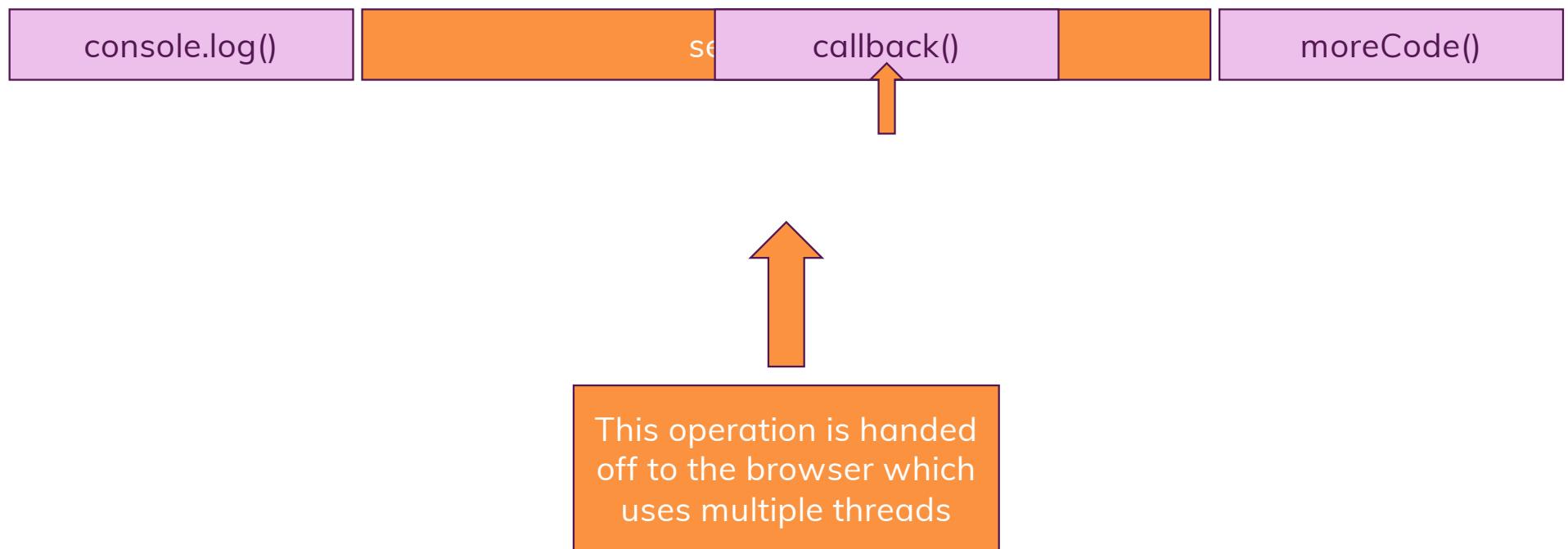


Certain Operations Take A Bit Longer





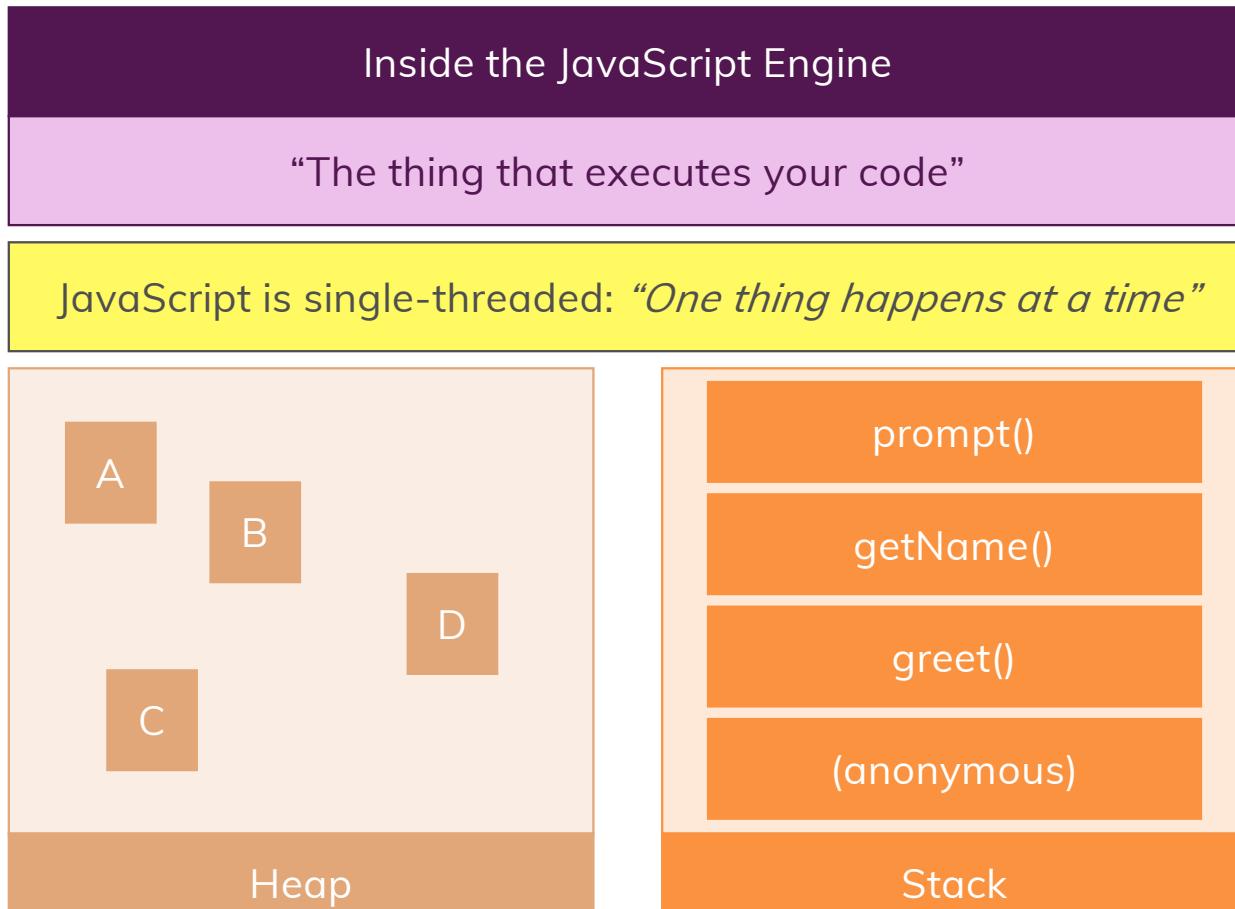
Asynchronous Code Execution!



How Code Gets Executed

Memory allocation

Stores data in your system memory and manages access to it



The Stack controls "which thing is happening"

Will become important later
(Event Loop)

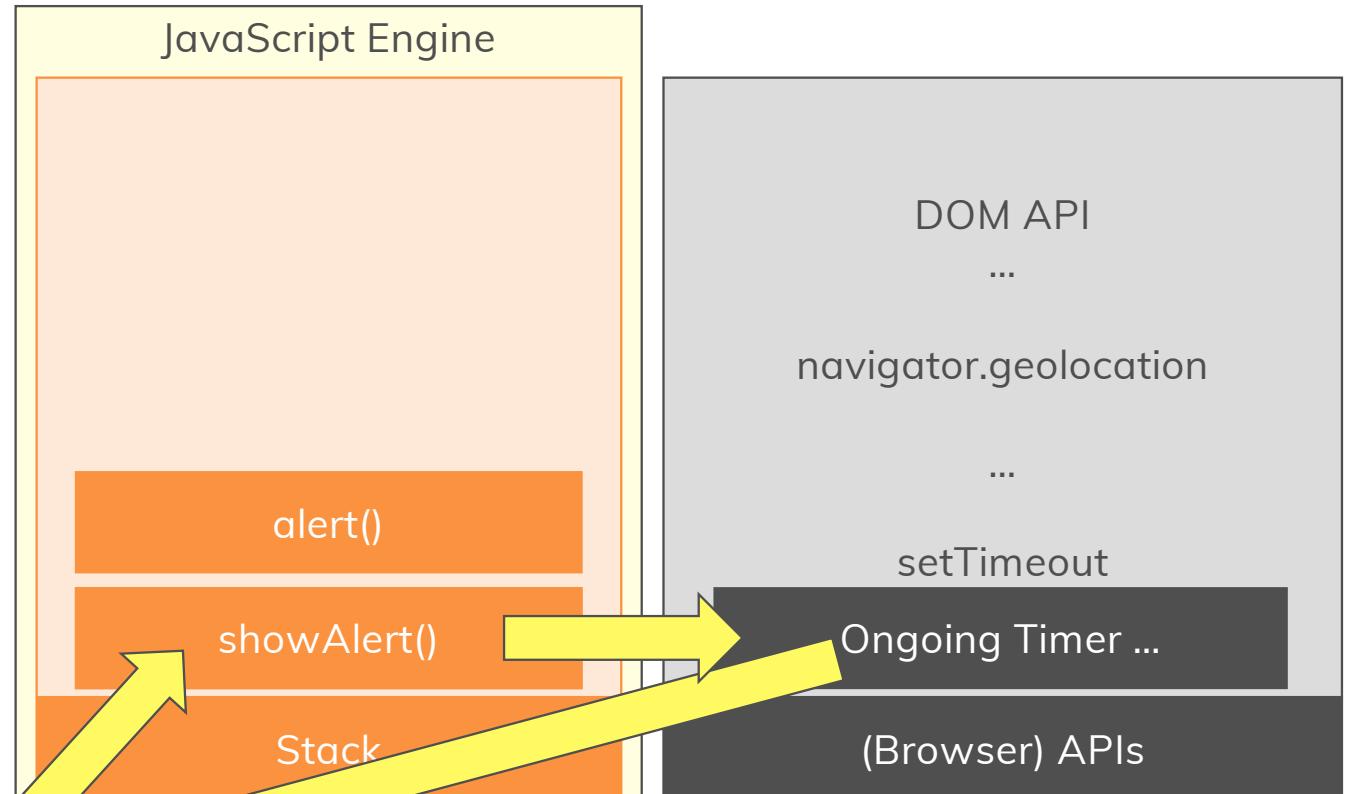
Execution Context

Manages your program flow (function calls and communication)

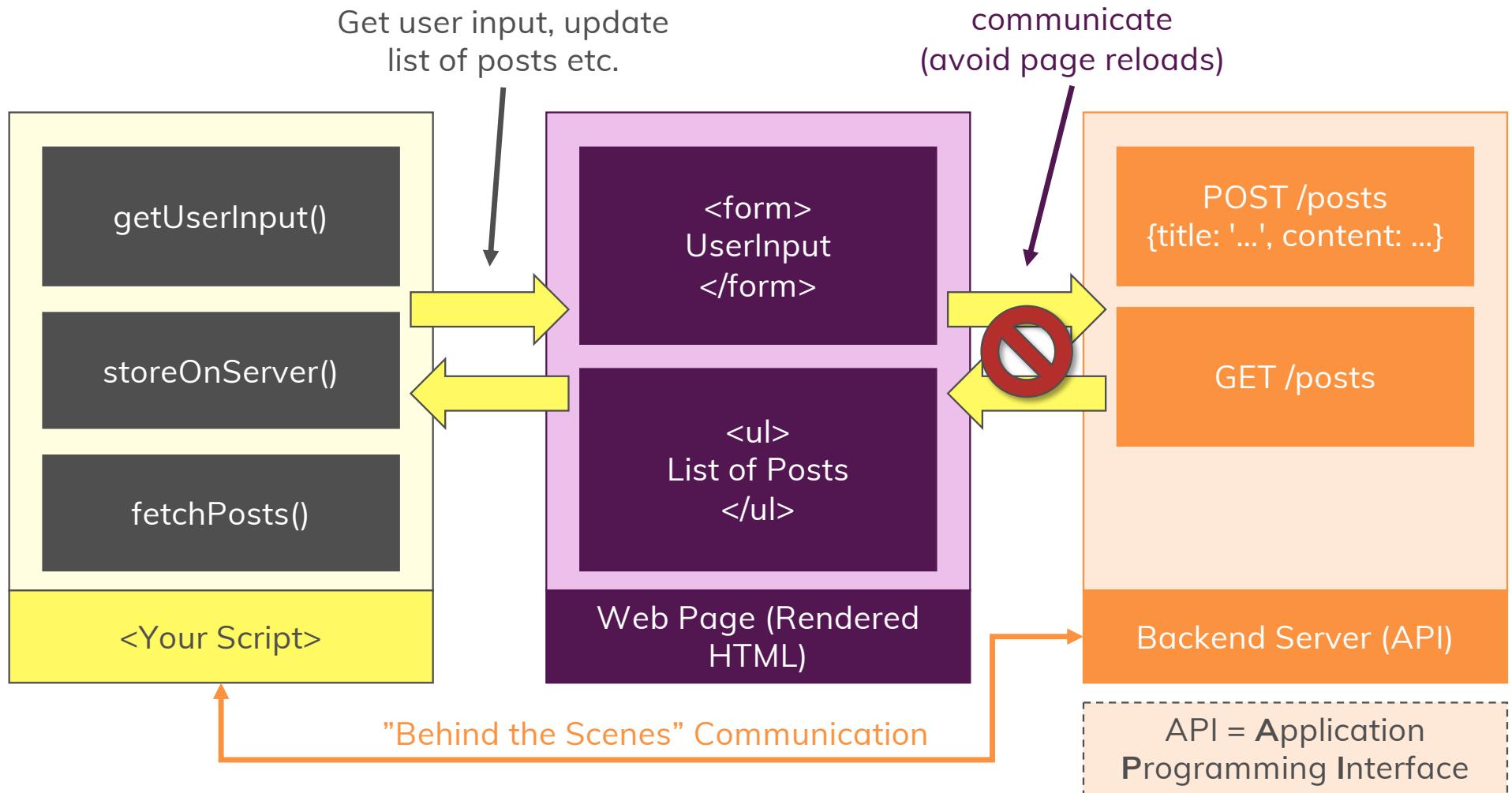
Event Loop, Queue and Async Code

```
const greet = () => {  
  console.log('Hi');  
};  
  
const showAlert = () => {  
  alert('Danger!');  
};  
  
setTimeout(showAlert, 2000);  
  
greet();
```

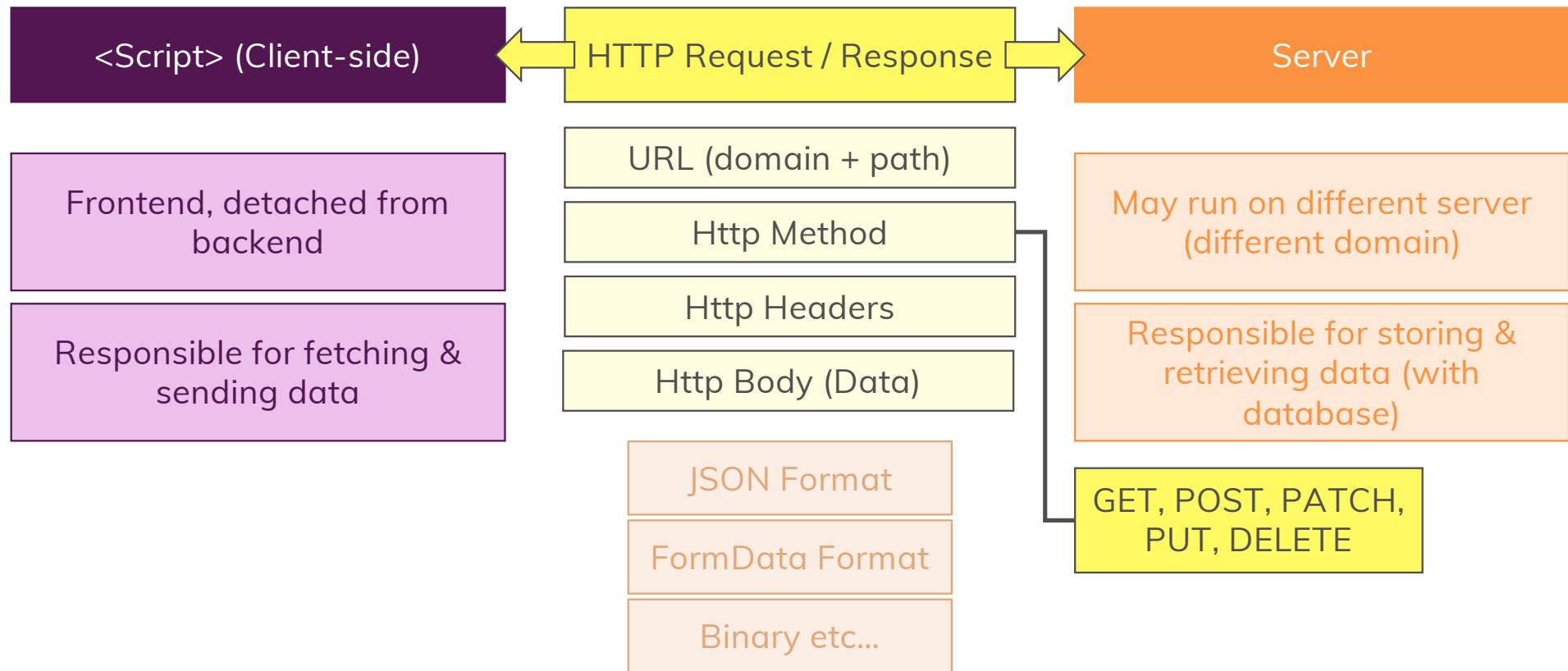
Code



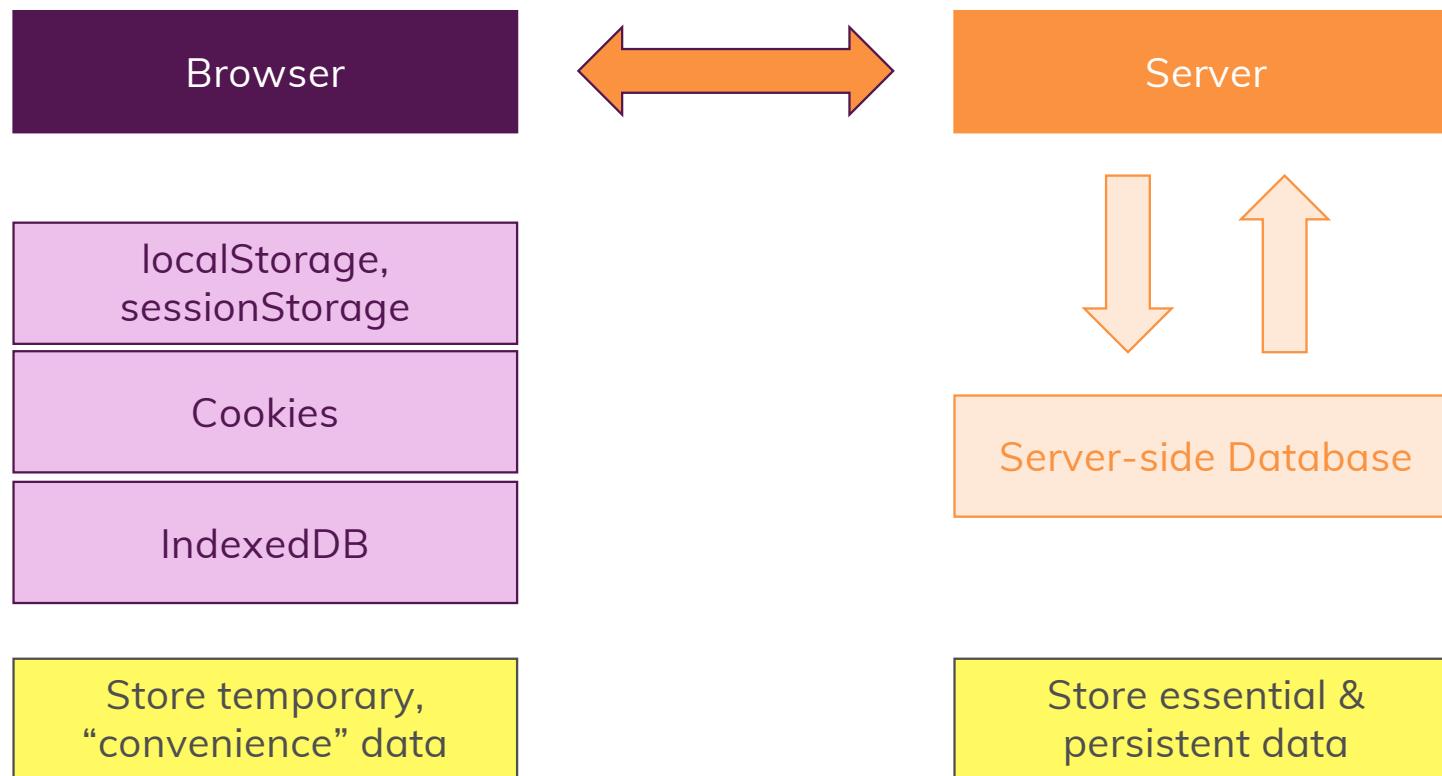
What & Why?



(Background) Http Overview



What is “Browser Storage”?



LocalStorage vs Cookies vs IndexedDB

localStorage,
sessionStorage

Simple key-value store

Manage user preferences
or basic user-data

Can be cleared by the
user and via JS

Easy to use, quite
versatile, bad for
complex data

Cookies

Simple key-value store
(some config options)

Manage user preferences
or basic user-data

Can be cleared by the
user and via JS

A bit clunky to use, quite
versatile, sent to server,
bad for complex data

IndexedDB

Relatively sophisticated,
client-side database

Manage complex data
your web app needs

Can be cleared by the
user and via JS

A bit clunky to use, great
for complex (non-critical)
data, good performance

Browser Features vs JavaScript Syntax

Kind of related but not the same!

Browser Features (“Browser APIs”)

JavaScript Syntax

The browser “decides” which features it offers

The browser’s JS engine defines which JS syntax/ version is supported

Examples: `fetch()`, geolocation, DOM API, ...

Examples: `let`, `const`, `async/ await`, Promises, ...

Implemented individually

Implemented individually or in chunks (e.g. ES6)

Not all browsers support all features – and support changes all the time in modern (“evergreen”) browsers. **Legacy** browsers (e.g. IE8) do not receive new features anymore.

What Exactly Is “Browser Support”?

fetch()

You (developer) want to use this

You can't control which browser
your users use



Chrome



Firefox



Internet Explorer



Edge



Safari

YES
(since 2015)

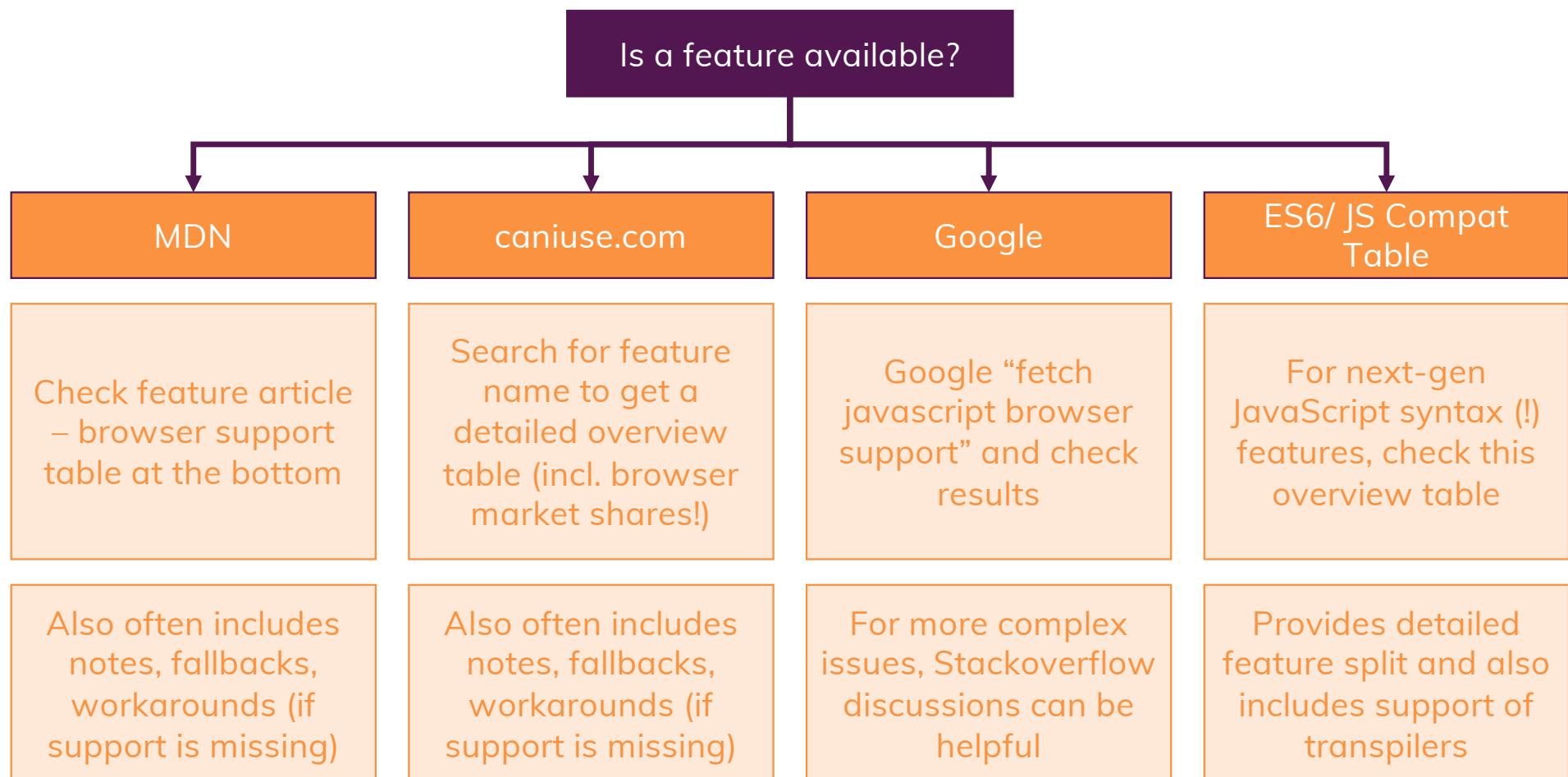
YES
(since 2015)

NO

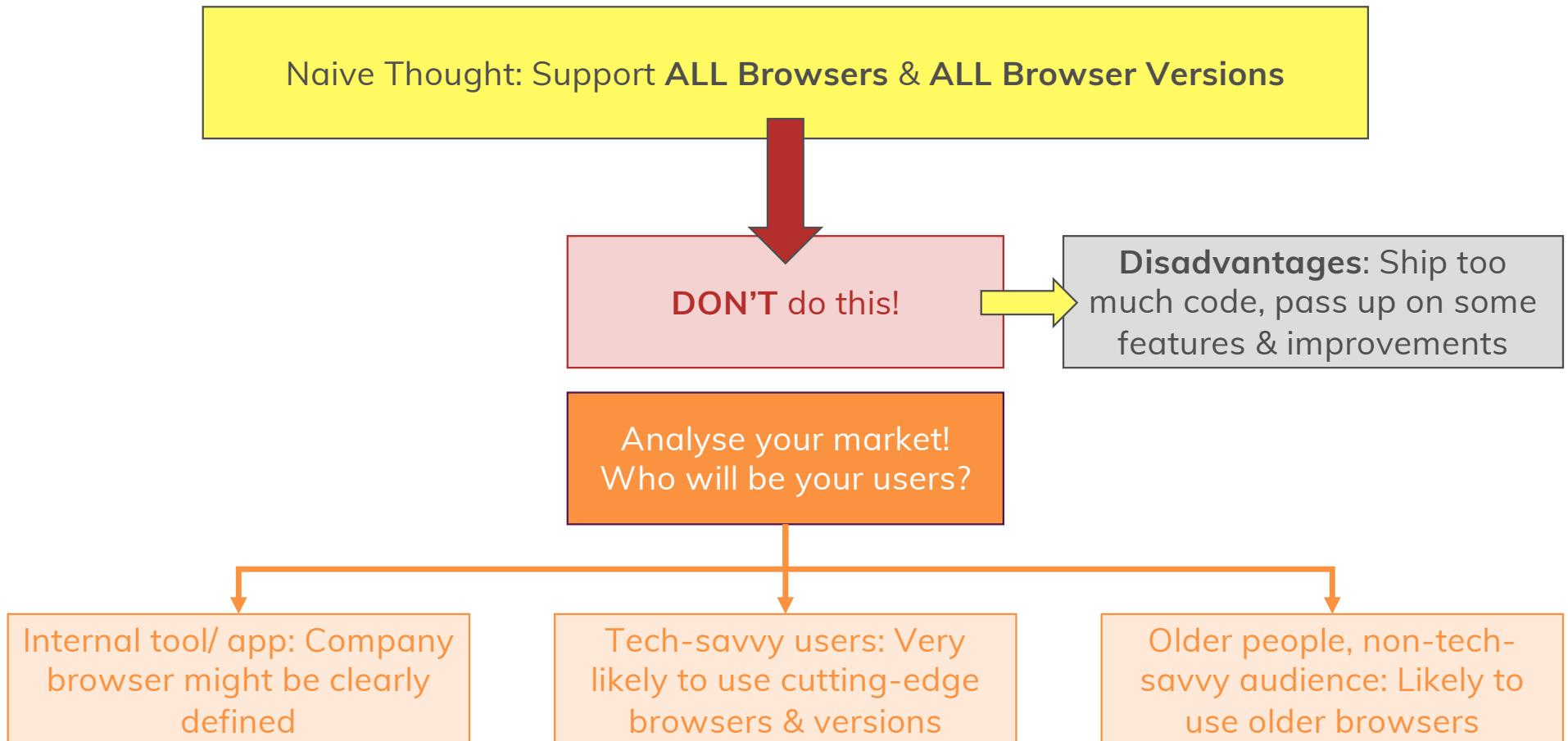
YES
(since 2016)

YES
(since 2017)

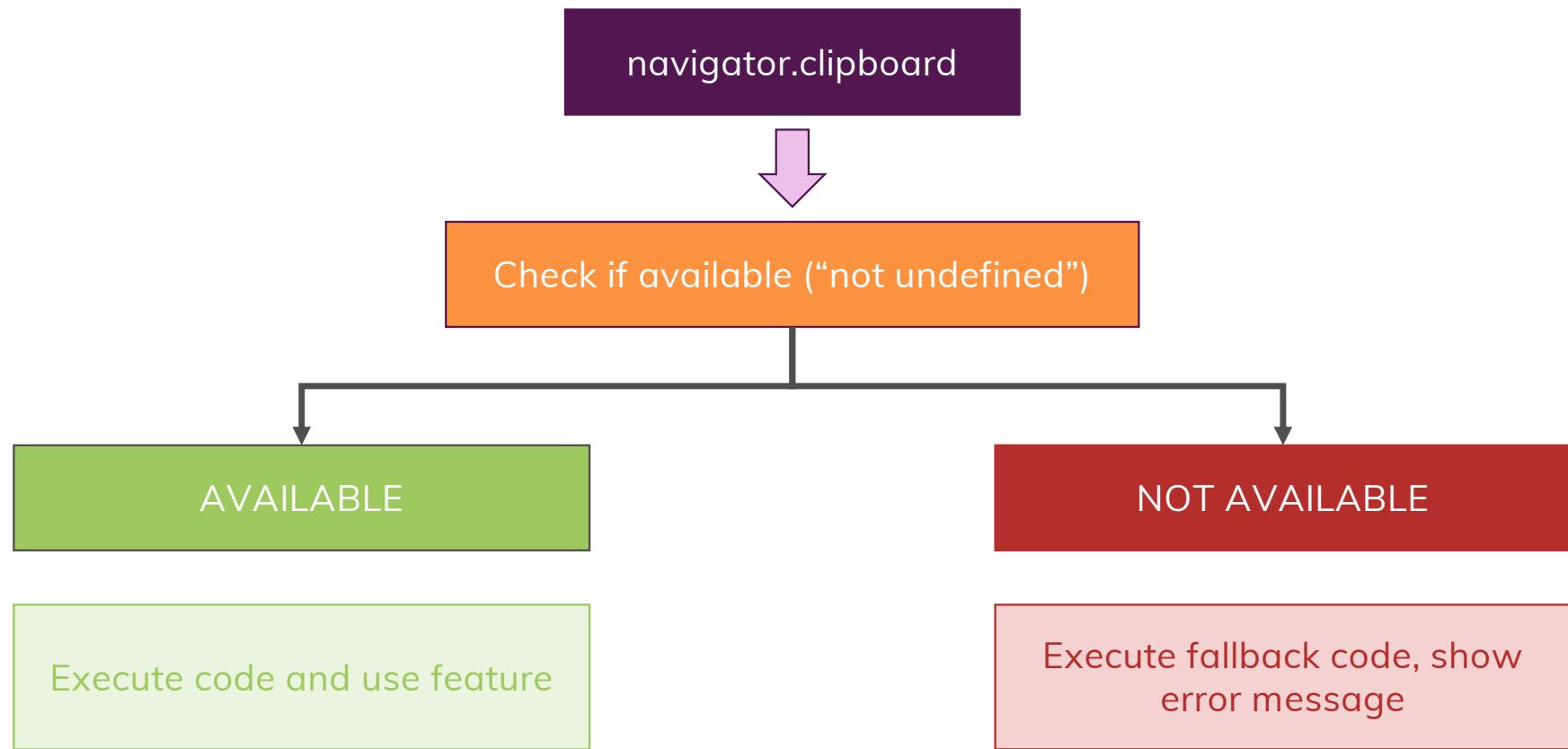
Checking Features & Browser Support



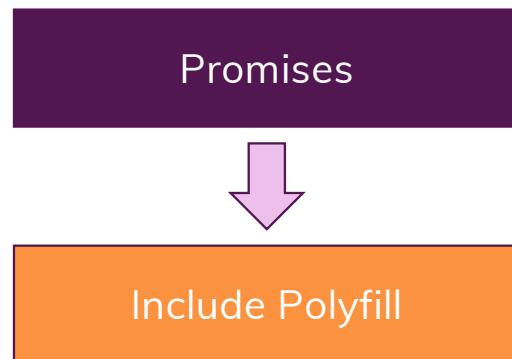
Which Browser Support Do You Need?



Using Feature Detection & Fallbacks



Using Polyfills



(Third-party) JavaScript code which adds an unsupported feature to the browser (for this script execution). Only possible for some features (i.e. where you can re-create the missing functionality)



Using Transpilation

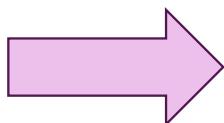
let, const, async/ await



Convert (“transpile”) modern code to “older” code

Third-party tools like Babel convert your code to code
that works in older browsers as well.

Write modern code, use
cutting-edge features



Ship (deploy) cross-browser
code

What?

Symbols

Iterators &
Generators

Reflect API

Proxy API

Primitive values

Create your own
“loopable” values

API to control objects

Create “traps” for
object operations

Used as object
property identifiers

What arrays, strings
etc. use internally

Standardized &
grouped methods

Step in and execute
code

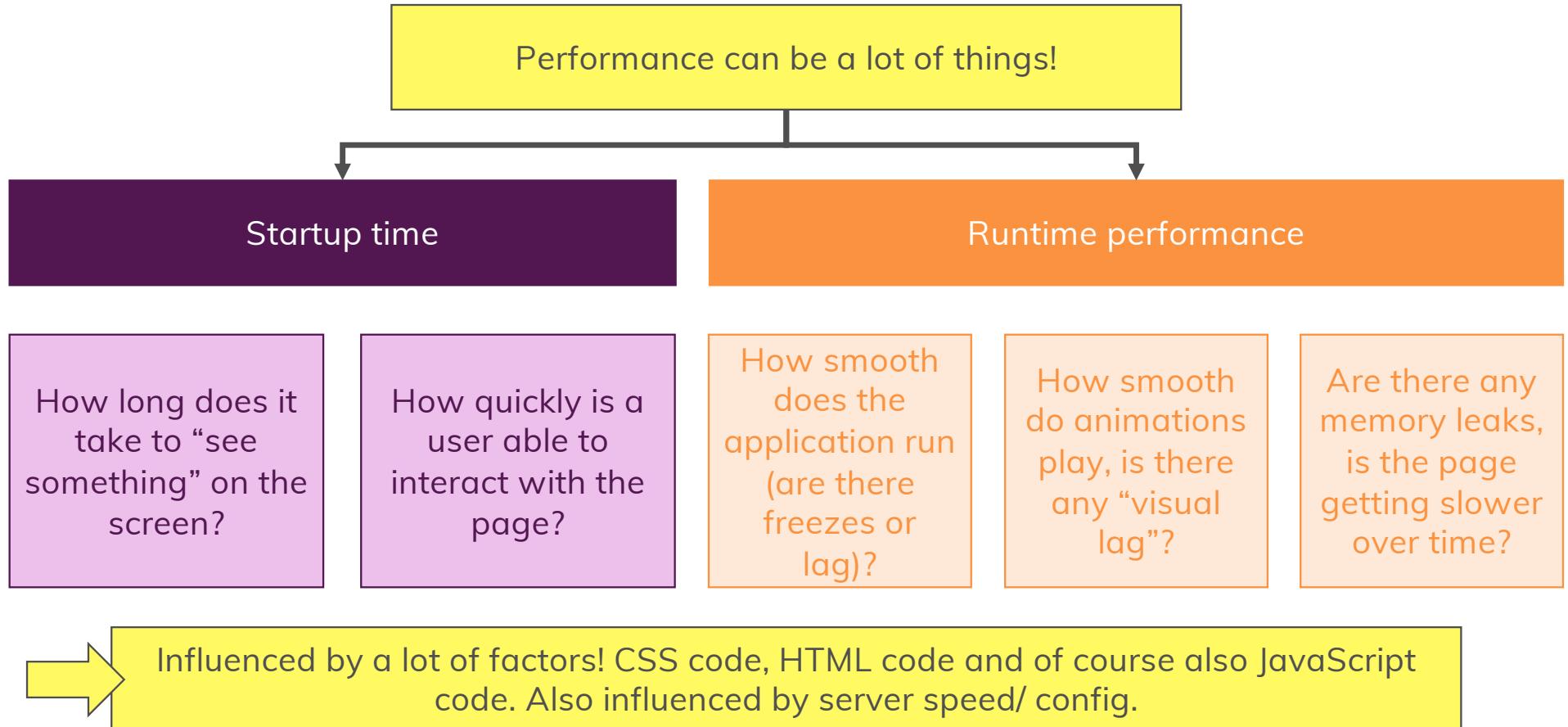
Built-in & creatable
by developers

Control code usage/
impact

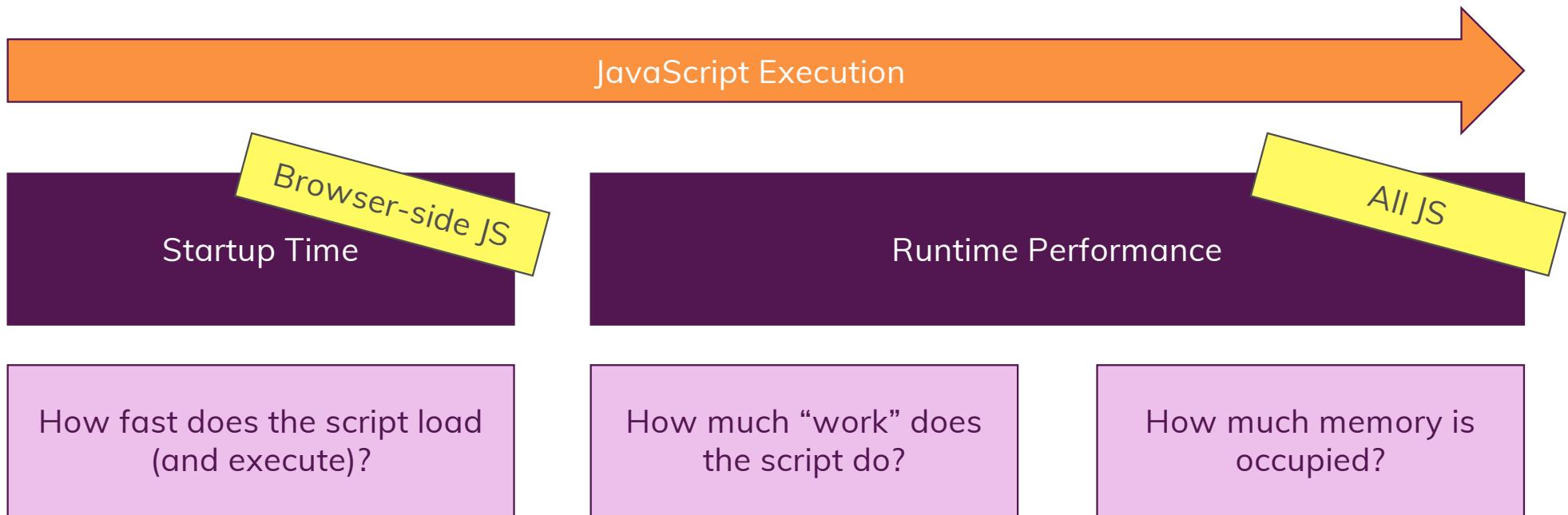
Control code usage/
impact

Uniqueness is
guaranteed

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

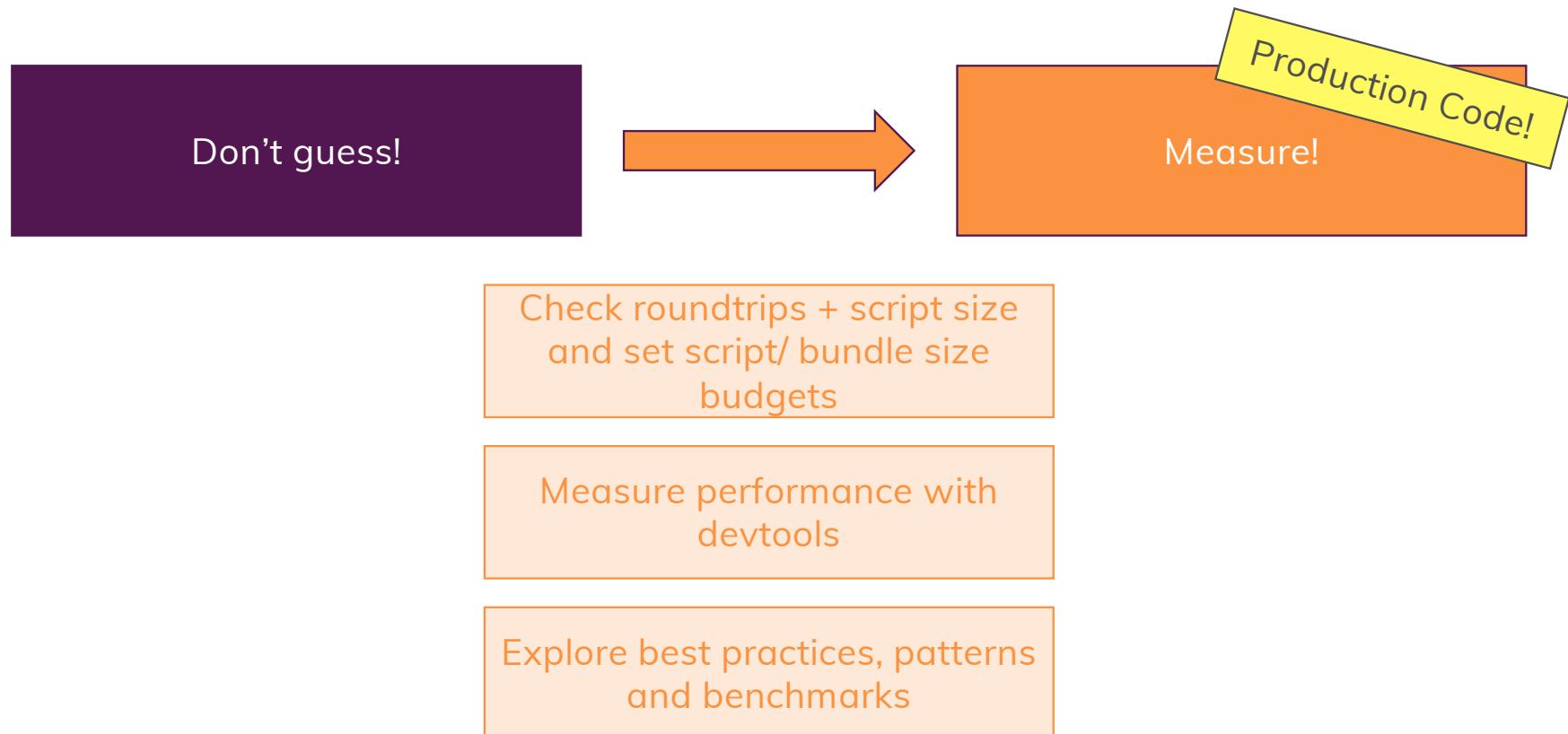
Optimize Code Execution, DOM Access: Avoid unnecessary code execution, especially unnecessary DOM operations/ repaints

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





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

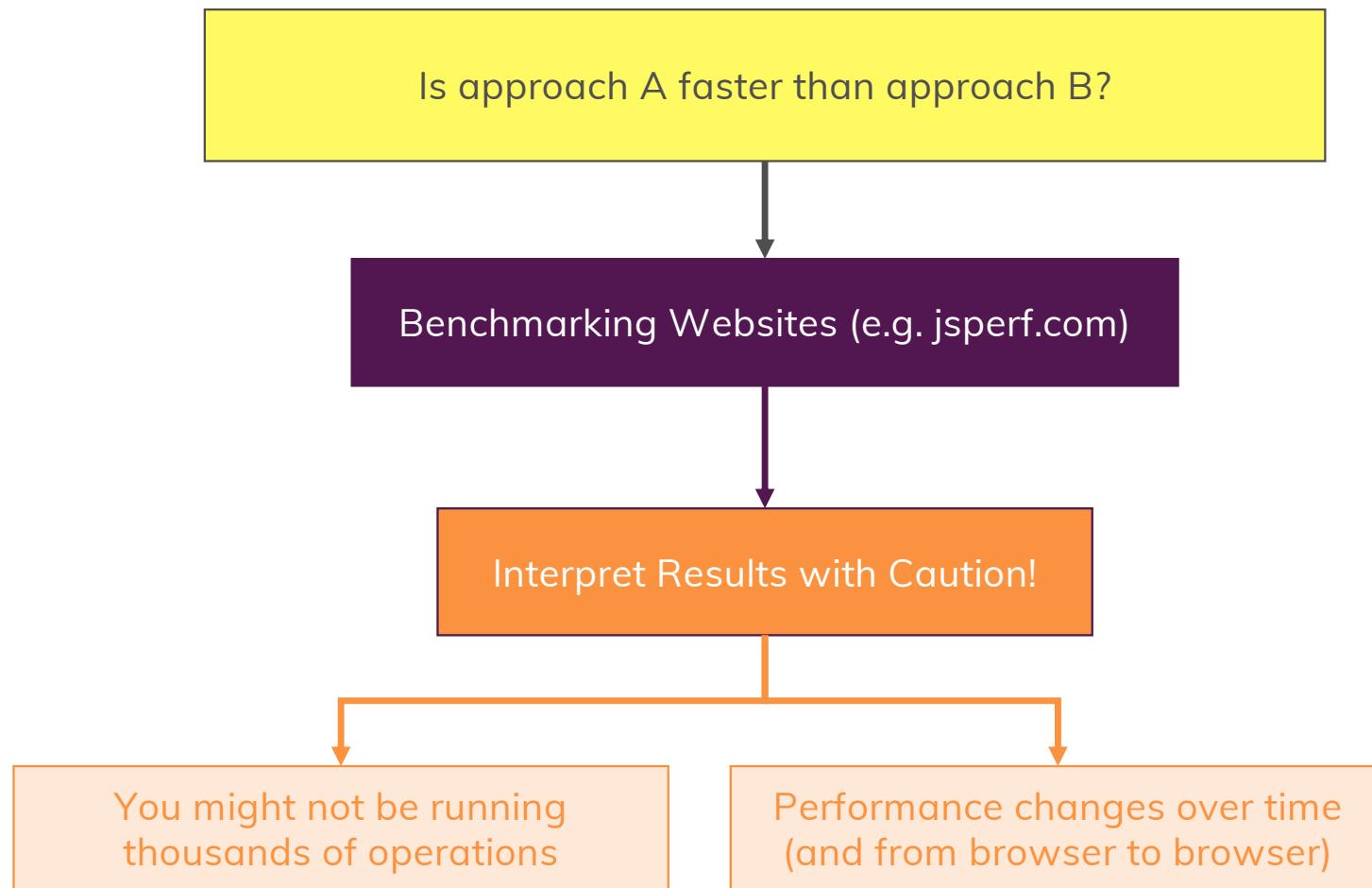
jsperf.com

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