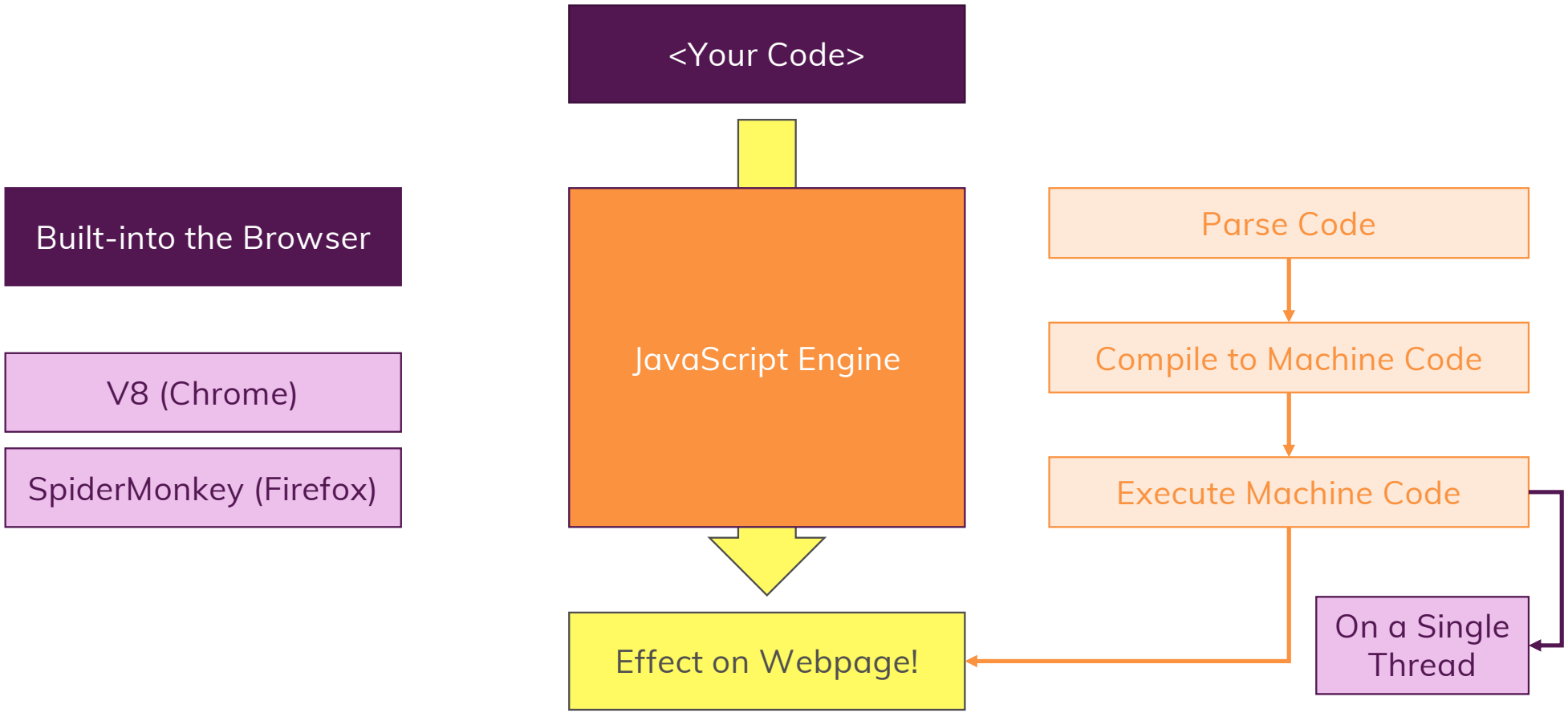
How Is JavaScript Executed?



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

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

Code Styles, Conventions & Syntax

There is a **difference** between writing **correct code** (i.e. avoiding syntax errors) and writing **readable**, properly formatted code.
Here's a **syntax error** example:

```
function greetUser {
alert('Hi there!');
}
```

What's the problem?

This function has no parameter list (the `()`), it should be:

```
function greetUser() {
alert('Hi there!');
}
```

This course teaches you the correct syntax - it teaches you how to define functions, how to declare variables etc. But of course, I also try to write readable code.

Take this as an example:

```
function greetUser() {alert('Hi there!');}
```

This would be a valid code snippet!

Whitespace, line breaks and indentation are all totally optional! Not required!
But of course this version of the code is more readable:

```
function greetUser() {
alert('Hi there!');
}
```

The same goes for variable definitions:

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

would be valid. And of course you can also read it.
But

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

is simply a bit more readable - it makes it easier to see where the variable name ends and the value starts.

So in general, **whitespace, adding a line break and indentation is optional** as long as you don't create syntax errors by omitting it - e.g. `functiongreet(){}` would be wrong since the function keyword can now no longer be identified.
You can also **structure longer expressions across multiple lines** to make them more readable, for example:

```
let someResult = 5 + 10 - 3 + 22 - 10000 + 5.344 * 1200;
```

could be written as:

```
let someResult = 5 + 10 - 3 +
22 - 10000 + 5.344 *
1200;
```

The same goes for long string concatenations:

```
let someLongString = 'Hi, this is going to be a bit longer, ' +
                     'so maybe split it across multiple lines by ' +
                     'concatenating multiple strings!';
```

This would not be valid, it would be a syntax error:

```
let someLongString = 'Hi, this is going to be a bit longer,
                      so maybe split it across multiple lines by
                      concatenating multiple strings!';
```

Why? Because **JavaScript can't find the end of the string in the first line** - and it doesn't look in the other lines. **Strings always have to be in one line** (or split into multiple strings, concatenated `via +`).

What about semi-colons?

Generally, you should place them after every expression you wrote. The exception are functions and similar code snippets where you use `{}` (exception from that rule: objects!).

Adding a semi-colon after a function wouldn't be a problem, you wouldn't get an error. But it's a common practice to NOT add semi-colons there.

Your IDE should normally highlight syntax errors right away. There also are tools & plugins that you can use that suggest certain code styles (e.g. how many whitespaces should be used for indentation) but the exact configuration of these tools and the exact rules you want to follow are of course up to you. We actually already use one plugin that helps us with code formatting - Prettier (https://github.com/prettier/prettier-vscode).
Later in the course, we'll dive into such tools ("linters" like ESLint) and of course I also follow a consistent code style which you can use throughout the course.

"Indirect" vs "Direct" Function Execution - Summary

It can be confusing to see that there seem to be two ways of executing a function:

```
function add() {
something = someNum + someOtherNum;
}
```

`add()` VS `add`

It's important to understand why we have these "two ways"!

In general, you call a function that you defined by **using its name** (e.g. `add`) and **adding parentheses** (with any parameters the function might need - or empty parentheses if no parameters are required like in the above example).

`add()`

This is how you execute a function from your code. Whenever JavaScript encounters this statement, it goes ahead and runs the code in the function. Period!

Sometimes however, you **don't want to execute the function immediately**. You rather want to "tell JavaScript" that it should execute a certain function **at some point in the future** (e.g. when some event occurs).
That's when you don't directly call the function but when you instead just provide JavaScript with the name of the function.

`someButton.addEventListener('click', add);`

This snippet would tell JavaScript: "*Hey, when the button is clicked, go ahead and execute add.*".
`someButton.addEventListener('click', add());` would be wrong.
Why? Because JavaScript would encounter that line when it parses/ executes your script and register the event listener AND immediately execute add - because you added parentheses => That means (see above): "*Please execute that function!*".
Just writing add somewhere in your code would do nothing by the way:

```
let someVar = 5;
add
alert('Do something else...');
```

Why?

Because you just throw the name of the function in there but **you don't give any other information to JavaScript**. It basically doesn't know what to do with that name ("*Should I run that when a click occurs? After a certain amount of time? I don't know...*") and hence JavaScript kind of ignores this statement.

Mixing Numbers & Strings

You saw the example with a number and a "text number" being added

`3 + '3'` => `'33'`
in JavaScript.

That happens because the `+` operator also supports strings (for string concatenation).
It's the only arithmetic operator that supports strings though. For example, this will not work:

`'hi' - 'i'` => `NaN`
`NaN` is covered a little later, the core takeaway is that you can't generate a string of 'h' with the above code.
Only `+` supports both strings and numbers.
Thankfully, JavaScript is pretty smart and therefore is actually able to handle this code:

`3 * '3'` => `9`
Please note: It yields the number (!) `9`, NOT a string `'9'`!
Similarly, these operations also all work:

`3 - '3'` => `0`
`3 / '3'` => `1`
Just `3 + '3'` yields `'33'` because here JavaScript uses the "*I can combine text*" mode of the `+` operator and generates a string instead of a number.

Objects - Common Syntax Gotchas

You learned the correct syntax for creating an object in the last lecture:

```
1.    const user = {
2.    name: 'Max',
3.    age: 30
4.    };
```

A couple of important things:

You use `{}` to "group the data" - a semicolon (`;`) is used after the closing `}`. On functions, we didn't do that. As a rule of thumb, you can keep in mind that **a semicolon is used after `{}` if the `{}` are on the right side of the equal sign**!
key-value pairs are **separated via a comma** (`,`), **NOT** via a semicolon. Using a semicolon inside of an object (i.e. between `{}`), would be a **syntax error**!
Values are assigned to keys/ properties **via a colon** (`:`), NOT via an equal sign (`=`). Using an equal sign inside of an object (i.e. between `{}`), would be a **syntax error**!
I.e. this would be WRONG and would throw an error:

```
1.    const worstPossibleUser = {
2.    name = 'Max';
3.    age = 30;
4.    };
```

"Shadowed Variables"

You learned about local ("function-internal") variables and global variables.

What happens if you have this code?

```
1.  let userName = 'Max';
2.  function greetUser(name) {
3.  let userName = name;
4.  alert(userName);
5.  }
6.  userName = 'Manu';
7.  greetUser('Max');
```

This **will actually show** an alert that says `'Max'` (NOT `'Manu'`).
You might've expected that an error gets thrown because we use and declare `userName` more than once - and as you learned, that is not allowed.
It indeed is **not allowed on the same level/ in the same scope.**
So this would fail:

```
1.  let userName = 'Max';
2.  let userName = 'Manu';
```

Why does it work in the first code snippet though?

Because we first create a global variable `userName` via
`let userName = 'Max';`

But then we never re-declare that on the global level (that would not be allowed).

We only declare another variable inside of the function. But since variables in functions get their **own scope**, JavaScript does something which is called **"shadowing"**.
It **creates a new variable on a different scop**e - this variables does not overwrite or remove the global variable by the way - **both co-exist**.
When referring to `userName` inside of the `greetUser` function we now **always refer to the local, shadowed variable**.
Only **if no such local variable existed**, JavaScript would **fall back to the global variable**.

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

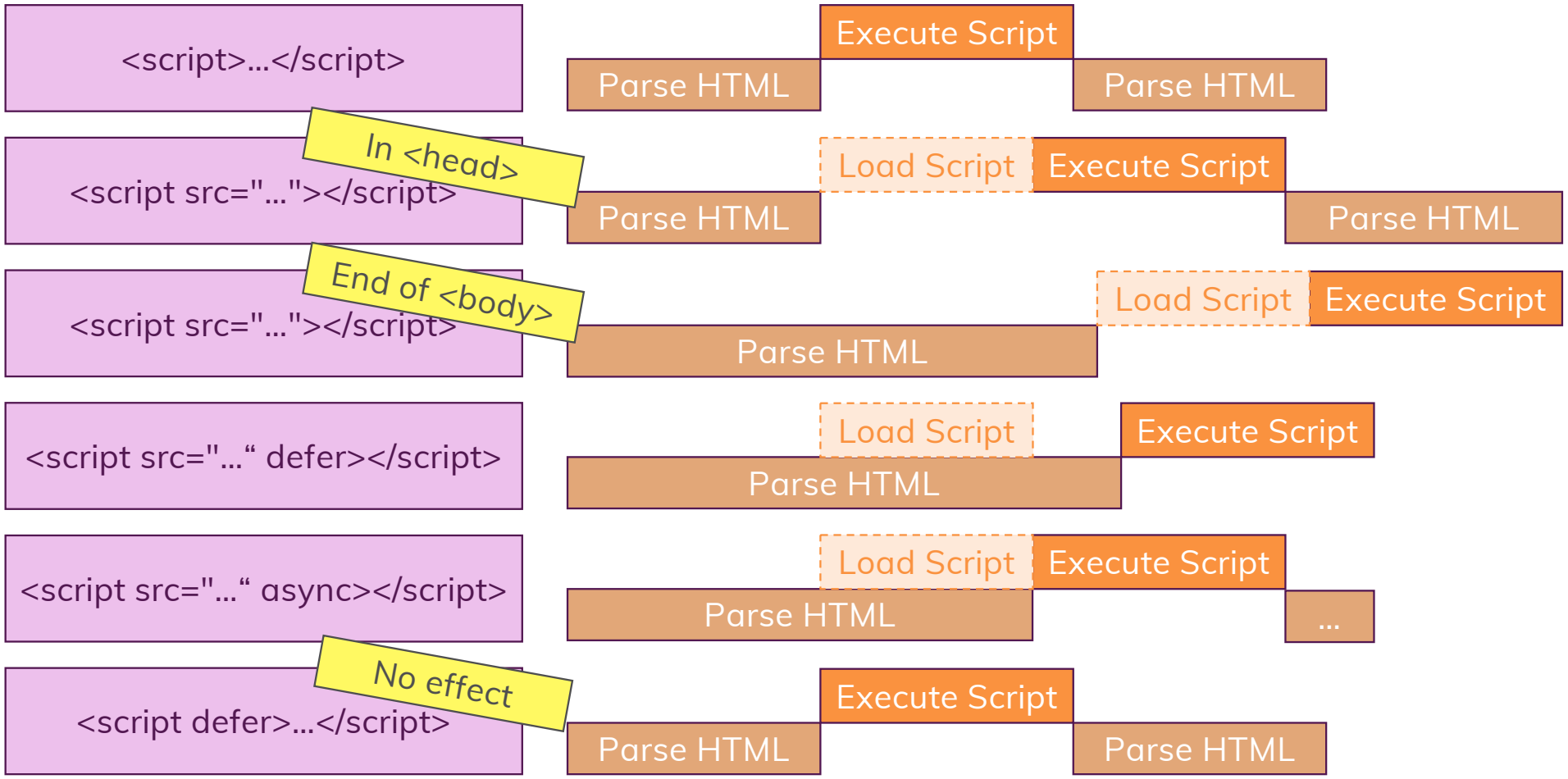
`<script>`
`alert('Hi there!');`
`</script>`
	`<script src="someFile.js"></script>`
	`<script src="otherFile.js"></script>`
	`<script src="otherFile.js"></script>`
	Multiple imports (even of the same file) are possible and normal!

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

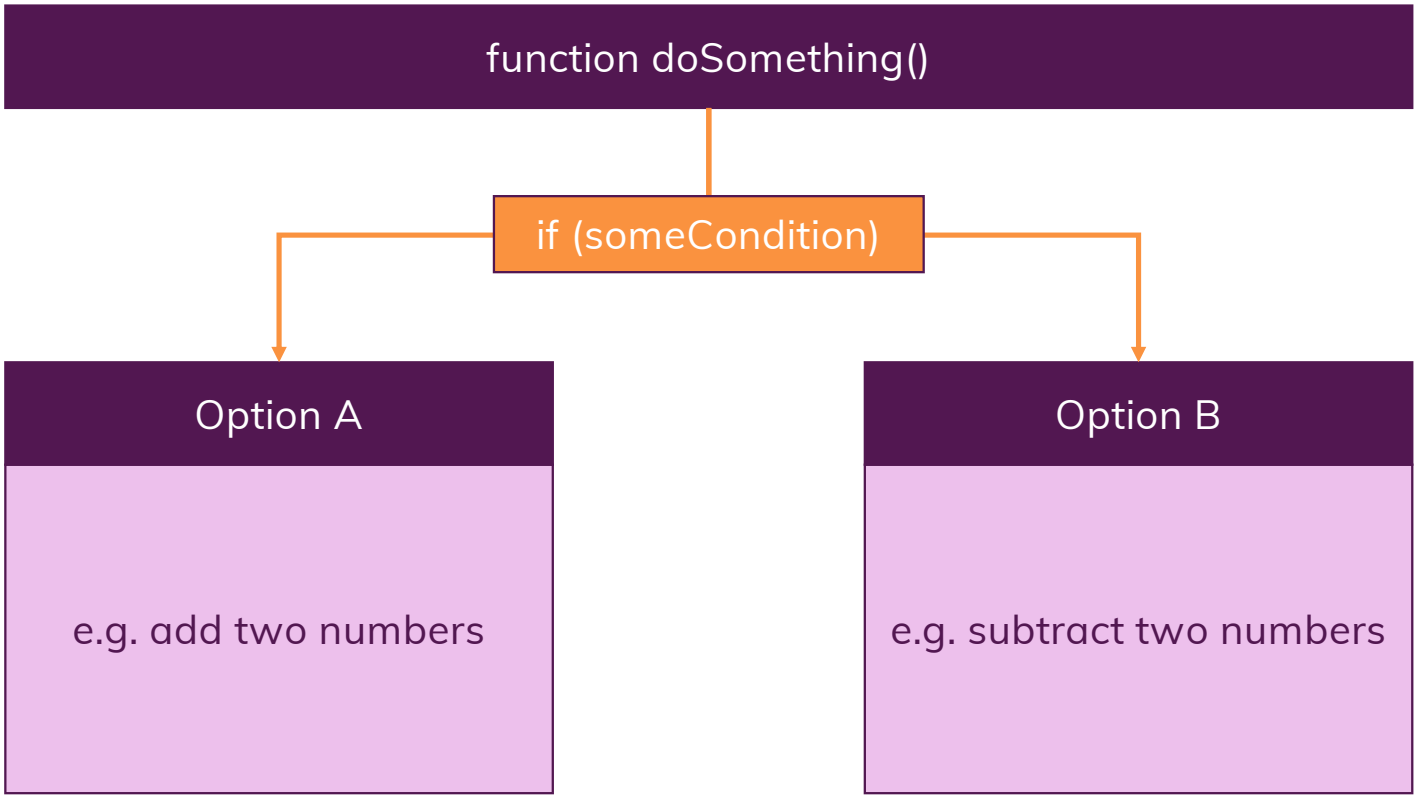
How To Import Your Scripts

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



Using Booleans in Conditions & More on Text Comparisons

Understanding the "Condition"

Always keep in mind that `condition` in

```
if (condition) { ... }
```

simply has to be a **boolean value**.
Often, you'll **generate** such a boolean value with the help of `===`, `>`, `<` etc. **All these operators yield boolean values** (without changing the variables/ values you're using them on).
Since `if` only wants a boolean, you of course **don't have to use such an operator**. If you already got a variable that holds a boolean, you can use it without any extra operator.
Example:

```
const isLoggedIn = true;
if (isLoggedIn) {
// This code will execute because isLoggedIn is true => A valid condition
}
```

You could write

```
const isLoggedIn = true;
if (isLoggedIn === true) {
...
}
```

but that would be **redundant**. You'd generate another new boolean where you already got one.
You can use the `!` operator to negate ("invert") the value:

```
const isLoggedIn = true;
if (!isLoggedIn) {
// This code will NOT execute because isLoggedIn is true but ! inverts it (in this check)
} else {
// This would execute because !isLoggedIn yields false => else block executes
}
```

Again, that would be similar to:

```
const isLoggedIn = true;
if (isLoggedIn !== true) {
// This would NOT execute
} else {
// This would execute because isLoggedIn is true and hence !== true yields false
}
```

But again, that would be redundant.

More on Text (String) Comparisons

Strings can also be compared with greater than (`>`) or lower/ smaller than (`<`) operators.
JavaScript compares strings based on standard lexicographical ordering, using Unicode values.

That means that `b` is greater than `a` for example.
JavaScript always looks at the first character and only considers other characters if the first character is similar. In addition, capital characters are considered to be smaller than lowercase characters.

See these examples:

```
'ab' > 'aa' // true

'a' > 'B' // true

'a' > 'b' // false
```

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
Prefer over == === and !==	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

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)

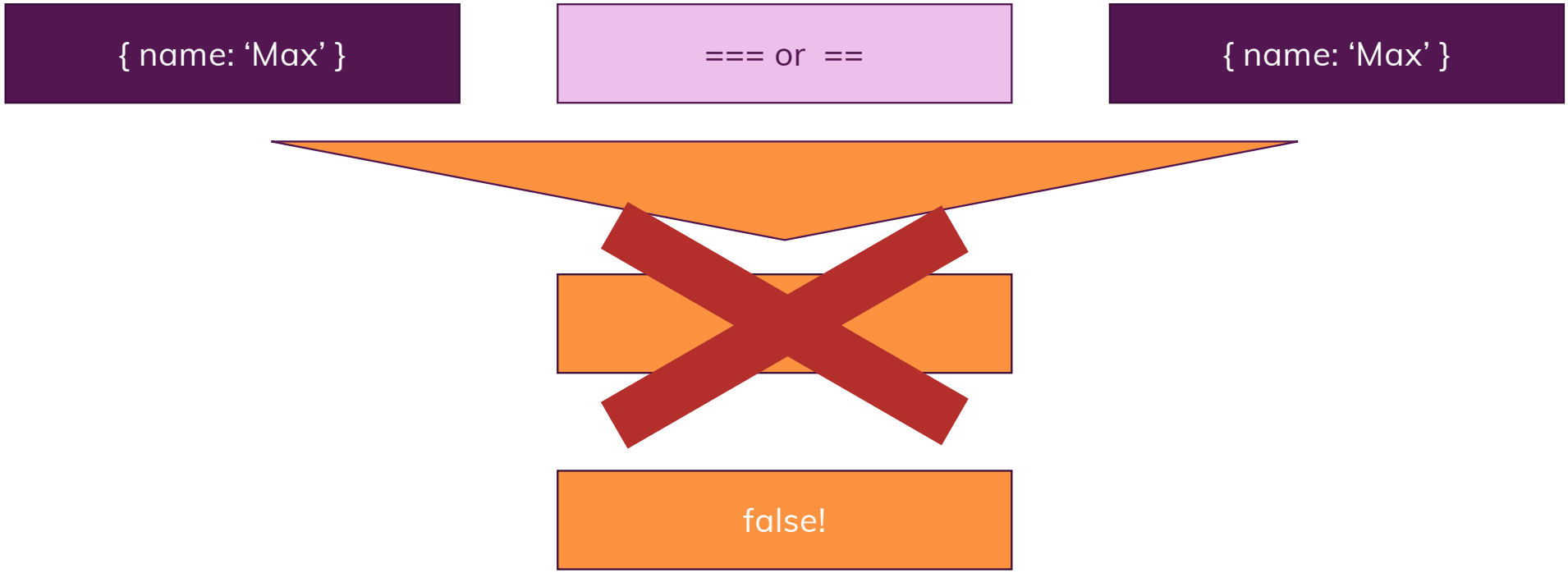
Part 2

Evaluated as an alternative

Yields true if Part 1 OR Part 2 yields true

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

Beware of Objects & Arrays in Comparisons!



Objects and arrays are kind of special in JavaScript!

Coercion vs Conversion

It's important to understand that JavaScript is able to use variables in conditions - even without comparison operators.

This is kind of obvious, if we consider a boolean variable, for example:

```
1.  let isLoggedIn = true;
2.  if (isLoggedIn) {
3.  ...
4.  }
```

Since if just wants a condition that returns true or false, it makes sense that you can just provide a boolean variable or value and it works - without the extra comparison (`if (isLoggedIn === true)` - that would also work but is redundant).
Whilst the above example makes sense, it can be confusing when you encounter code like this for the first time:

```
1.  let userInput = 'Max';
2.  if (userInput) {
3.  ... // this code here will execute because 'Max' is "truthy" (all strings but empty strings are)
4.  }
```

JavaScript tries to coerce ("convert without really converting") the values you pass to if (or other places where conditions are required) to boolean values. That means that it tries to interpret `'Max'` as a boolean - and there it follows the rules outlined in the previous lecture (e.g. `0` is treated as `false`, all other numbers are treated as `true` etc.)
It's important to understand that JavaScript doesn't really convert the value though.

`userInput` still holds `'Max'` after being used in a condition like shown above - it's not converted to a boolean. That would be horrible because you'd invisibly lose the values stored in your variables.
Instead,

```
if (userInput) { ... }
```

is basically transformed (behind the scenes) to

```
if (userInput === true) {
```

And here, the `=== operator` generates and returns a boolean. It also doesn't touch the variable you're comparing - `userInput` stays a string. But it generates a new boolean which is temporarily used in the comparison.
And that's exactly what JavaScript automatically does when it finds something like this:

```
if (userInput) { ... }
```

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') { ... }
```

Works!

```
const nameInput = 'Max';

if (nameInput) { ... }
```

Yields a string, NOT a boolean

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

Also works!

Falsy and Truthy 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

Logical Operators - A Quick Summary

As a **reference** which you can come back to (or print out), here's a **quick summary** of how **logical operators** and **comparison operators** behave in JavaScript:

```
1. const userName = 'Max';
2. const altName = '';
3. console.log(userName === 'Max'); // generates and prints a boolean => true
4. console.log(userName); // wasn't touched, still is a string => 'Max'
```

```
5. console.log(userName || null); // userName is truthy and therefore returned by || => 'Max'
6. console.log(altName || 'Max'); // altName is falsy (empty string), hence 'Max' is returned => 'Max'
7. console.log(altName || ''); // both altName and '' are falsy but if the first operand is falsy, the second one is always returned => ''
8. console.log(altName || null || 'Anna'); // altName and null are falsy, 'Anna' is returned => 'Anna'
```

```
9. console.log(userName && 'Anna'); // userName is truthy, hence second (!) value is returned => 'Anna'
10.   console.log(altName && 'Anna'); // altName is falsy, hence first value is returned => ''
11.   console.log(userName && ''); // userName is truthy, hence second value is returned => ''
```

Always keep in mind: **NO** operator (neither `===`, `>` etc. nor `&&` or `||`) changes the variable you might be using in the comparison. In the above examples, the values stored in `userName` and `altName` **are NEVER changed**.
`===`, `>` etc. just **generate new boolean values** which are used in the comparison. `||` and `&&` **generate NO booleans**, they just treat the **values before and after them as conditions** (which therefore need to yield boolean values and are coerced to booleans if required).
Because of the above-described behaviors, you often use `||` in JavaScript to assign default/ fallback values to variables/ constants:

```
1. const enteredValue = ''; // let's assume this is set based on some input provided by the user, therefore it might be an empty string
```

```
2. const userName = enteredValue || 'PLACEHOLDER'; // will assign 'PLACEHOLDER' if ent
```

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

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) { ... }

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) {`
			`  ...`
			`}`

JavaScript Language vs Browser APIs

In the last lectures, we covered the JavaScript engine and what it does inside of the browser. You also learned that there is a difference between the JS code execution and Browser APIs you might tap into during that execution.

Essentially, you can split the code you write into these two pieces:

1) The JavaScript Language
Understands core syntax (`let`, `const` etc) but does NOT know anything about the DOM for example
2) Browser APIs
Not responsible for understanding your code (that's what 1) does) but instead responsible for exposing APIs like the DOM API which you can use from inside your script code.

The JavaScript language (1) is advanced by the **Ecma International Technical Committee 39 (TC39)**, which is a group that's part of the Ecma organization. It's responsible for adding new features to the JavaScript language itself. For example, in the past, it was responsible for adding `let` and `const`.
You can **learn more about TC39** here: https://tc39.es/
And you can explore the current proposals that are being discussed by that group - features that potentially make it into the core JavaScript language in the future: https://github.com/tc39/proposals
IMPORTANT: Just because a feature becomes part of the language does **NOT** mean that all JavaScript engines immediately support that feature. Of course the people developing the engines are doing their best to provide support as soon as possible but that process simply also takes time.
On the other hand, engine vendors also **sometimes start supporting certain features BEFORE TC39 made a feature an official part** of JavaScript. Because in the end, it's of course totally up to the people working on the engines to decide which syntax their JS engine understands.

Browser APIs also are standardized because the different browser vendors (Google for Chrome, Microsoft for Edge etc.) of course want to (roughly) provide feature parity and similar APIs. It wouldn't be a great developer experience if you had different functions which you need to call to make your scripts work in different browsers. Although, in the past, this was pretty normal.
Nowadays, thankfully, this is getting way better because **there also is a working group that works on browser APIs** - so that different features and APIs in different browsers are avoided as good as possible.
That working group has the name **WHATWG** and you can learn more about it here: https://whatwg.org/
If you're interested in learning more about the APIs that were/ are "managed" by this group, you can check this site: https://spec.whatwg.org/
This working group is not related to TC39!

ES5 vs ES6

Taught in this course (from the beginning on)!

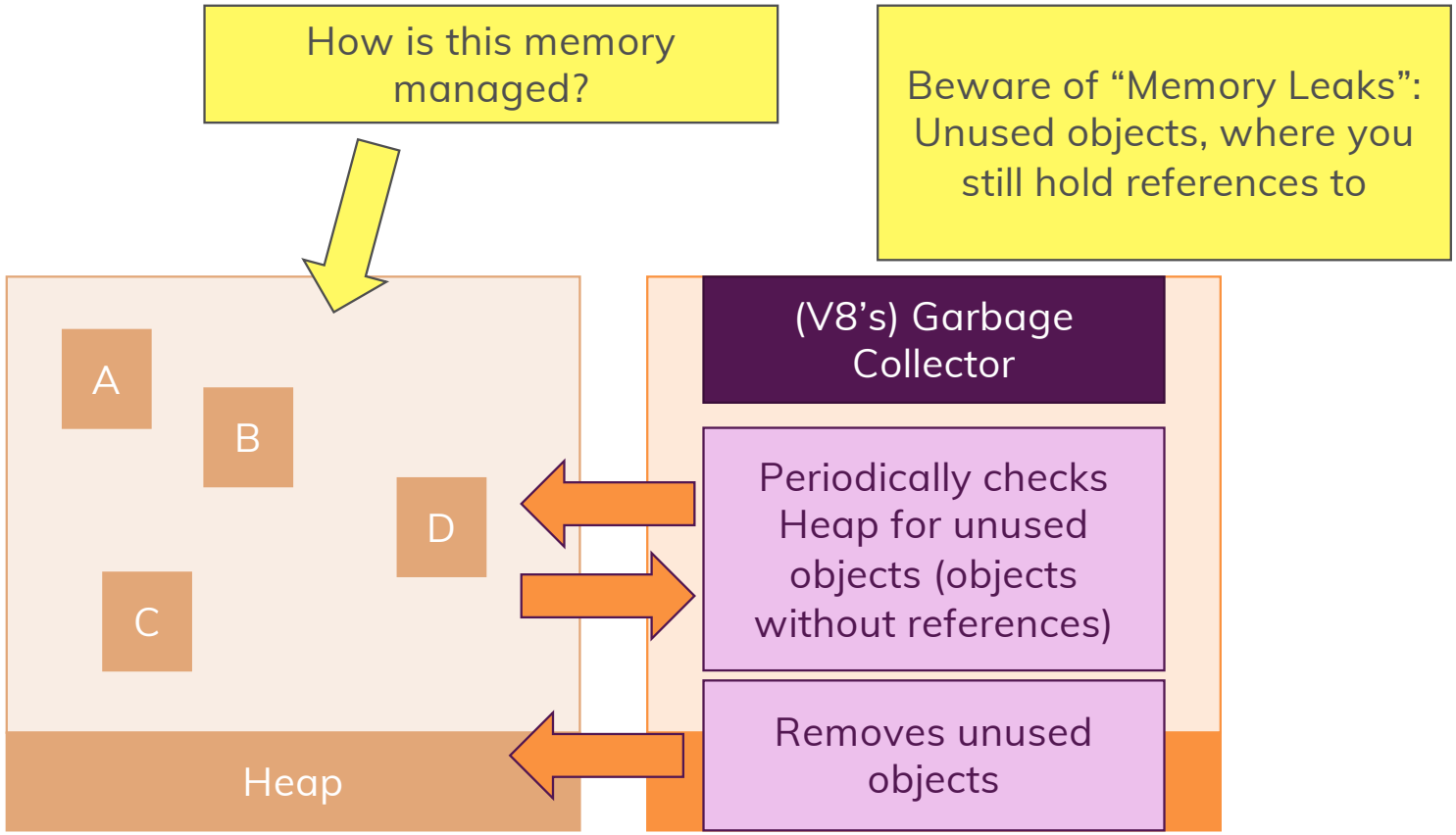
JavaScript Evolution

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

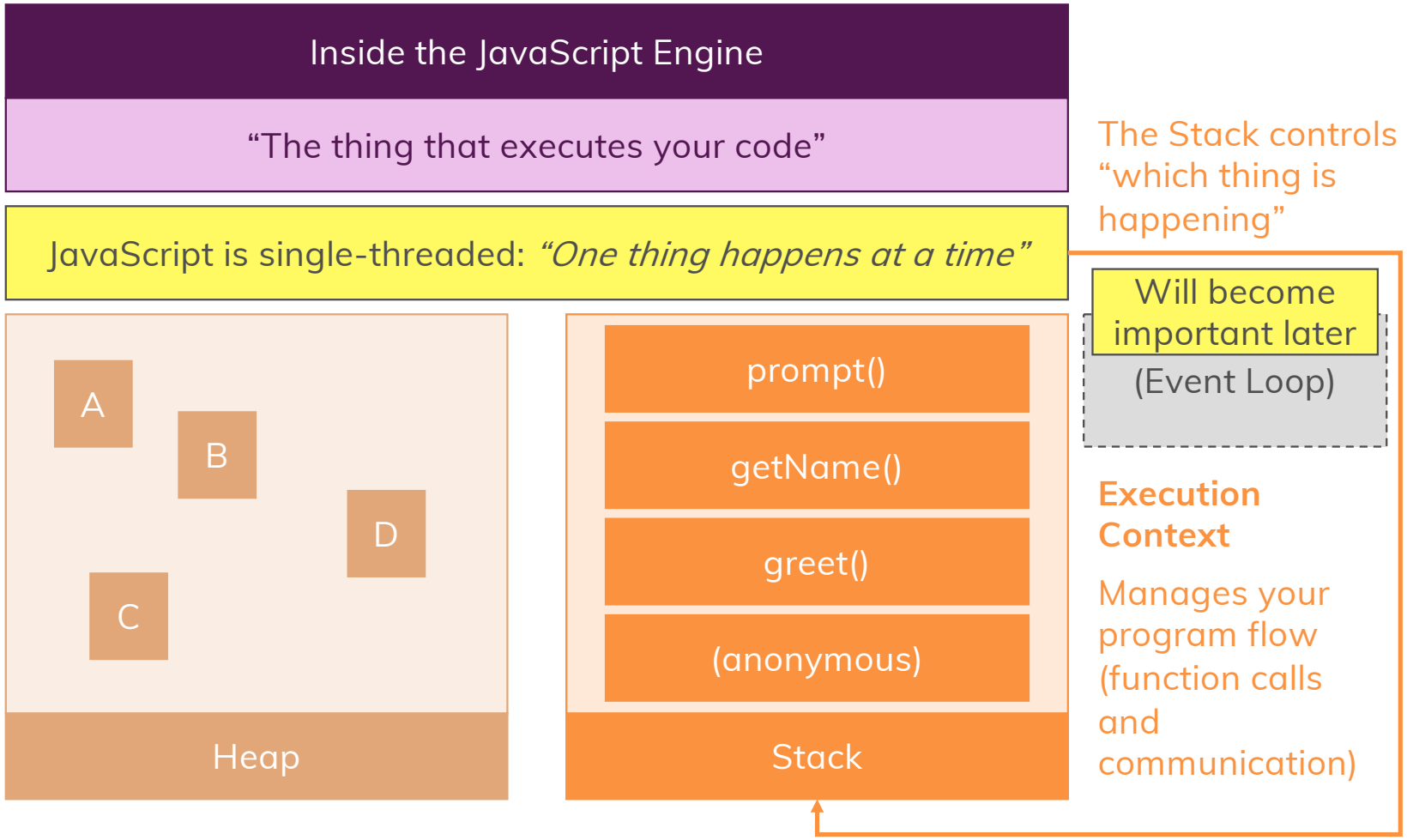
var vs let vs const

var	let	const
Creates a variable	Creates a variable	Creates a constant
Available since ... ever	Available since ES6	Available since ES6
Function & Global scope	Block scope	Block scope

Garbage Collection



How Code Gets Executed



The Stack controls "which thing is happening"

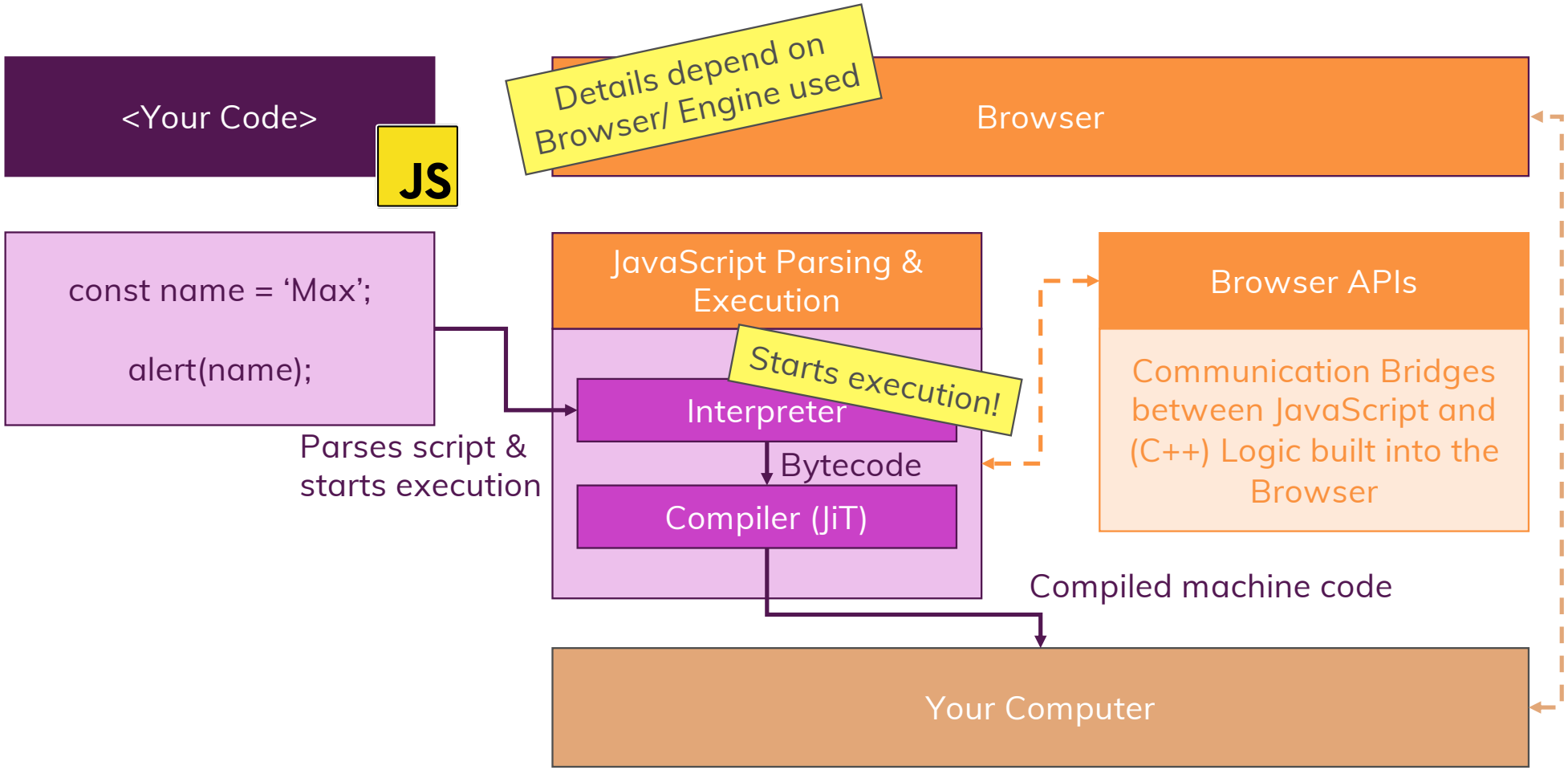
Memory allocation

Stores data in your system memory and manages access to it

Execution Context

Manages your program flow (function calls and communication)

JavaScript Engines & What They Do



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**

Reference vs Primitive Values

What are Primitives?

This article and videos is named “Reference vs Primitive Values”.

So let’s start - what are “Primitives”?

Here’s an example:

```
var age = 28
```

The **`age`** variable (you could also use **`let`** or **`const`** by the way) stores a number value. The number **`28`**.

Number values are called “primitive values” because they’re very simple building blocks of JavaScript apps.

Other simple core building blocks are:

```
var name = 'Max' // strings are primitives, too!
var isMale = true // so are booleans
```

So numbers, string, booleans - these are probably very well-known to you. **`undefined`** and **`null`** are additional primitive types.

JavaScript - The Complete Guide

Learn JavaScript in like no time - this course is optimized to teach you JavaScript as quick as possible!

Accelerated ES6 Training

Mastered the JavaScript basics? Take the next step with this course and dive into all new ES6 features!

What are Reference Types Then?

So we learned what “Primitives” (or “primitive types”) are.

What are “reference types” then?

`Object`s and **`Array`**s!

```
var person = {
  name: 'Max',
  age: 28,
}

var hobbies = ['Sports', 'Cooking']
```

Here, **`person`** is an object and therefore a so-called reference type. Please note that it holds properties that in turn have primitive values. This doesn’t affect the object being a reference type though. And you could of course also have nested objects or arrays inside the **`person`** object.

The **`hobbies`** array is also a reference type - in this case, it holds a list of strings. A **`string`** is a primitive value/ type as you learned but this doesn’t affect the **`array`**. Arrays are **always** reference types.

What’s the Difference?

Cool, we got two different types of values. What’s the idea behind all of that?

It’s related to memory management.

Behind the scenes, JavaScript of course has to store the values you assign to properties or variable in memory.

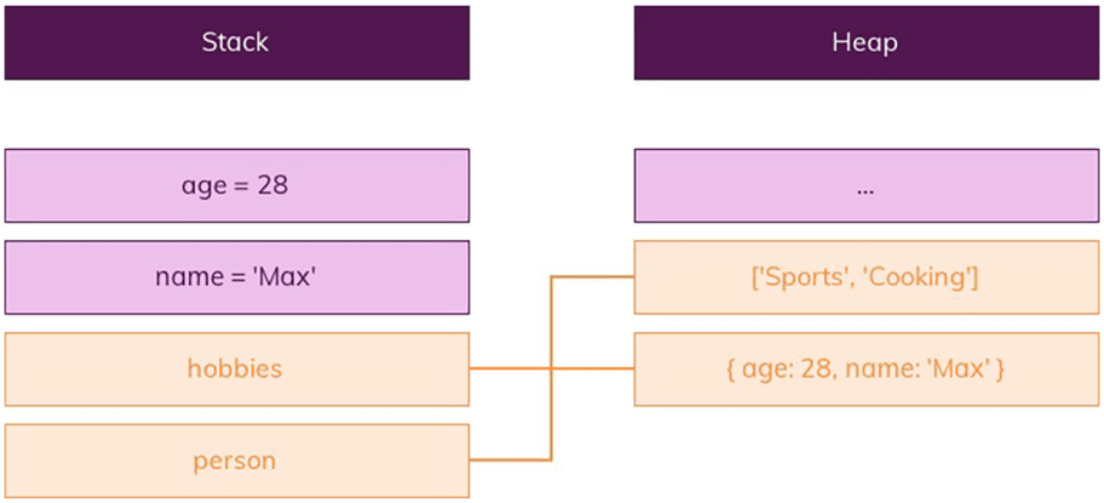
JavaScript knows two types of memory: The **Stack** and the **Heap**. You can dive much deeper if you want to.

Here’s a super-short summary: The stack is essentially an easy-to-access memory that simply manages its items as a - well - stack. Only items for which the size is known in advance can go onto the stack. This is the case for numbers, strings, booleans.

The heap is a memory for items of which you can’t pre-determine the exact size and structure. Since objects and arrays can be mutated and change at runtime, they have to go into the heap therefore.

Obviously, there’s more to it but this rough differentiation will do for now.

For each heap item, the exact address is stored in a pointer which points at the item in the heap. This pointer in turn is stored on the stack. That will become important in a second.



Okay, so we got different memories. But how does that make a difference to us, the developer?

Strange Behavior of "Reference Types"

The fact that only pointers are stored on the stack for reference types matters a lot!

What's actually stored in the **`person`** variable in the following snippet?

```
var person = { name: 'Max' }
```

Is it: a) The object (**`{ name: 'Max' }`**)

b) The pointer to the object

c) A pointer to the **`name`** property?

It's **b)**. A pointer to the **`person`** object is stored in the variable. The same would be the case for the **`hobbies`** array.

What does the following code spit out then?

```
var person = { name: 'Max' }
var newPerson = person
newPerson.name = 'Anna'
console.log(person.name) // What does this line print?
```

You'll see **`'Anna'`** in the console!

Why?

Because you never copied the person object itself to **`newPerson`**. You only copied the pointer! It still points at the same address in memory though. Hence changing **`newPerson.name`** also changes **`person.name`** because newPerson points at the exactly same object!

This is really important to understand! You're pointing at the same object, you didn't copy the object.

It's the same for arrays.

```
var hobbies = ['Sports', 'Cooking']
```

```
var copiedHobbies = hobbies
copiedHobbies.push('Music')
console.log(hobbies[2]) // What does this line print?
```

This prints `'Music'` - for the exact same reason as stated above.

How can you copy the actual Value?

Now that we know that we only copy the pointer - how can we actually copy the value behind the pointer? The actual object or array?

You basically need to construct a new object or array and immediately fill it with the properties or elements of the old object or array.

You got multiple ways of doing this - also depending on which kind of JavaScript version you're using (during development).

Here are the two most popular approaches for arrays:

1) Use the `slice()` method

`slice()` is a standard array method provided by JavaScript. You can check out its full documentation here.

```
var hobbies = ['Sports', 'Cooking']
var copiedHobbies = hobbies.slice()
```

It basically returns a new array which contains all elements of the old element, starting at the starting index you passed (and then up to the max number of elements you defined). If you just call **`slice()`**, without arguments, you get a new array with all elements of the old array.

2) Use the spread operator

If you're using ES6+, you can use the spread operator.

```
var hobbies = ['Sports', 'Cooking']
var copiedHobbies = [...hobbies]
```

Here, you also create a new array (manually, by using **`[]`**) and you then use the spread operator (**`...`**) to "pull all elements of the old array out" and add them to the new array.

For objects

1) `Object.assign()`

You can use the **`Object.assign()`** syntax which is explained in greater detail here.

```
var person = { name: 'Max' }
var copiedPerson = Object.assign({}, person)
```

This syntax creates a new object (the **`{}`** part) and assigns all properties of the old object (the second argument) to that newly created one. This creates a copy.

2)

Just as with arrays, you can also use the spread operator on objects.

```
var person = { name: 'Max' }
var copiedPerson = { ...person }
```

This will also create a new object (because you used **`{ }`**) and will then pull all properties of **`person`** out of it, into the brand-new objects.

Deep Clones?

Now you know how to clone arrays and objects.

Here's one super-important thing to note though: You're not creating deep clones with either approach!

If you cloned array contains nested arrays or objects as elements or if your object contains properties that hold arrays or other objects, then these nested arrays and objects will **not** have been cloned!

You still have the old pointers, pointing to the old nested arrays/ objects!

You'd have to manually clone every layer that you plan on working with. If you don't plan on changing these nested arrays or objects though, you don't need to clone them.

More about cloning strategies can be read here

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

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)

Different Arrow Function Syntaxes

For arrow functions, you got a couple of different syntaxes which you can use - here's a summary.

Important: Don't miss the "function only returns an object" special case at the end of this article!

Default syntax:

```
1.   const add = (a, b) => {
          a.   const result = a + b;
          b.   return result; // like in "normal" functions, parameters and return statement are OPTIONAL!
2.   };
```

Noteworthy: Semi-colon at end, no function keyword, parentheses around parameters/ arguments.

2) Shorter parameter syntax, if exactly one parameter is received:

```
1.   const log = message => {
          a.   console.log(message); // could also return something of course - this example just doesn't
2.   };
```

Noteworthy: Parentheses around parameter list can be omitted (for exactly one argument).

3) Empty parameter parentheses if NO arguments are received:

```
1.   const greet = () => {
          a.   console.log('Hi there!');
2.   };
```

Noteworthy: Parentheses have to be added (can't be omitted)

4) Shorter function body, if exactly one expression is used:

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

Noteworthy: Curly braces and return statement can be omitted, expression result is always returned automatically

5) Function returns an object (with shortened syntax as shown in 4)):

```
const loadPerson = pName => ({name: pName });
```

Noteworthy: Extra parentheses are required around the object, since the curly braces would otherwise be interpreted as the function body delimiters (and hence a syntax error would be thrown here).

That last case can be confusing: Normally, in JavaScript, curly braces always can have exactly one meaning.

```
1. const person = { name: 'Max' }; // Clearly creates an object
2. if (something) { ... } // Clearly used to mark the if statement block
```

But when using arrow functions, curly braces can have two meanings:

Mark the function body (in default form)

Create an object which you want to return (in shorter function body form)

To "tell" JavaScript what you want to do, wrap the expression (e.g. object creation) in parentheses like shown above.

Parameters vs Arguments

Throughout this course, you'll hear me use the words "**parameters**" and "**arguments**" interchangeably.

Technically, there is a difference though:

Parameters are these variables which you **specify between parentheses** when defining a function.

```
function sayHi(name) { ... }
```

In this example, `name` is a parameter.

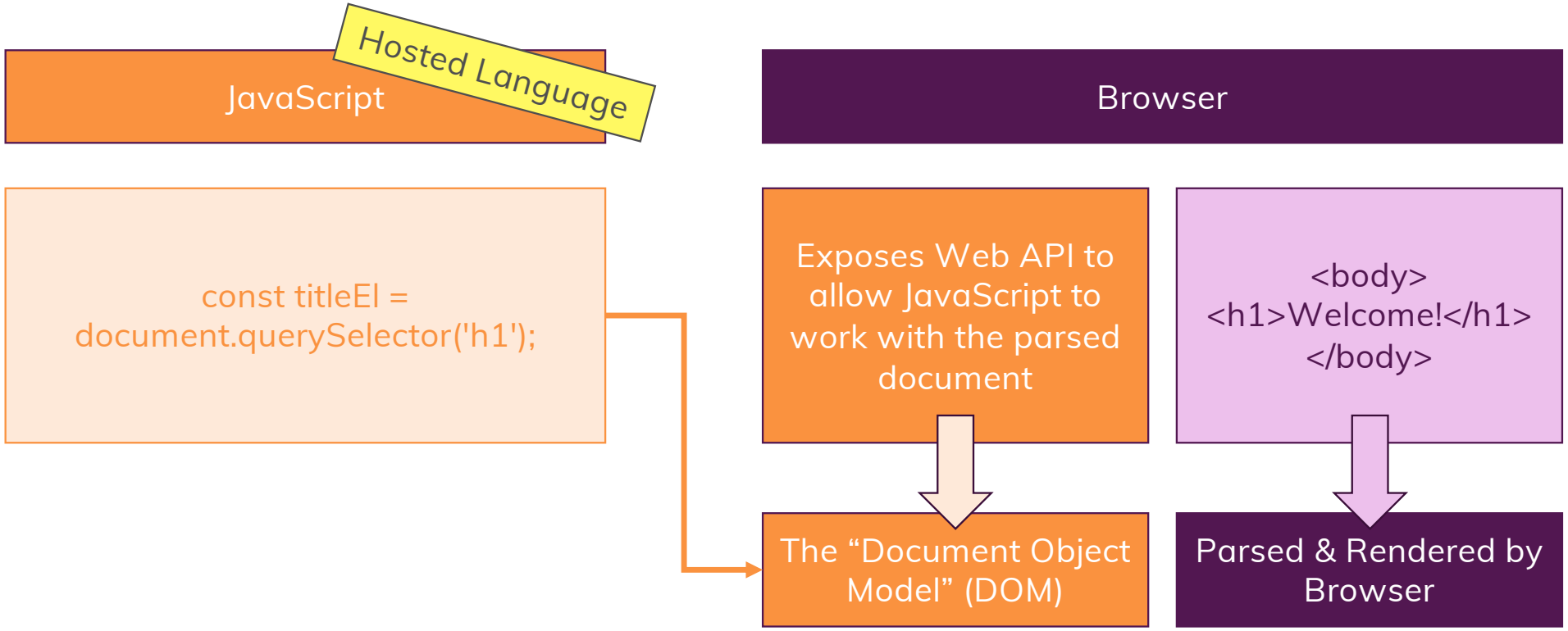
Arguments then are the **concrete values** you pass to a function when calling that function:

```
sayHi('Max');
```

`'Max'` is an **argument** of the function therefore - for the `name` parameter to be precise.

Since both concepts obviously are extremely close connected, I will often say "*let's define which arguments a function receives*" or something comparable, since defining the arguments of a function in the end means that you set up its parameters (and vice-versa).

The Document Object Model (DOM)



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

document & window

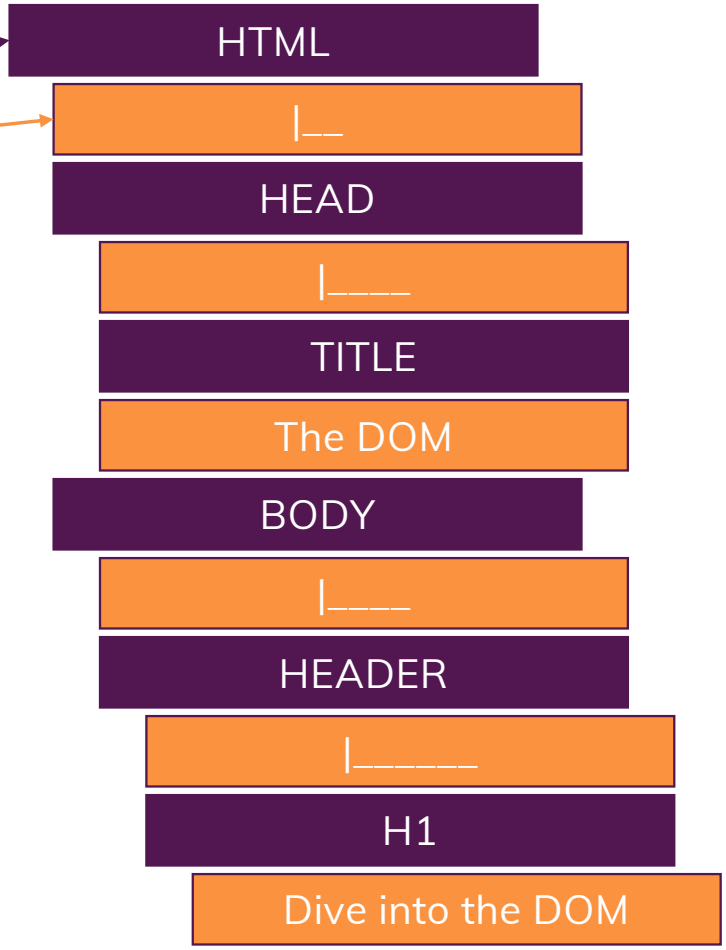
document	window
Root DOM Node	The active Browser Window / Tab
Provides access to element querying, DOM content etc	Acts as global storage for script, also provides access to window-specific properties and methods

The Document Object Model (DOM)

```
<html>
  <head>
    <title>The DOM</title>
  </head>
  <body>
    <header>
      <h1>Dive into the DOM</h1>
    </header>
    <main>
      <p>There's a lot to it!</p>
    </main>
  </body>
</html>
```

Element Node

Text Node



Nodes & Elements

Nodes	Elements
The objects that make up the DOM	Special properties and methods to interact with the elements
HTML tags are “element nodes” (or just “elements”)	Available methods and properties depend on the kind of element
Text creates “text nodes”	Can be selected in various different ways (via JavaScript)
Attributes create “attribute nodes”	Can be created and removed via JavaScript

Elements are one type of nodes

Summary: Node Query Methods

Here's a summary of the various methods you got to reach out to DOM elements (note: you can only query for element nodes).

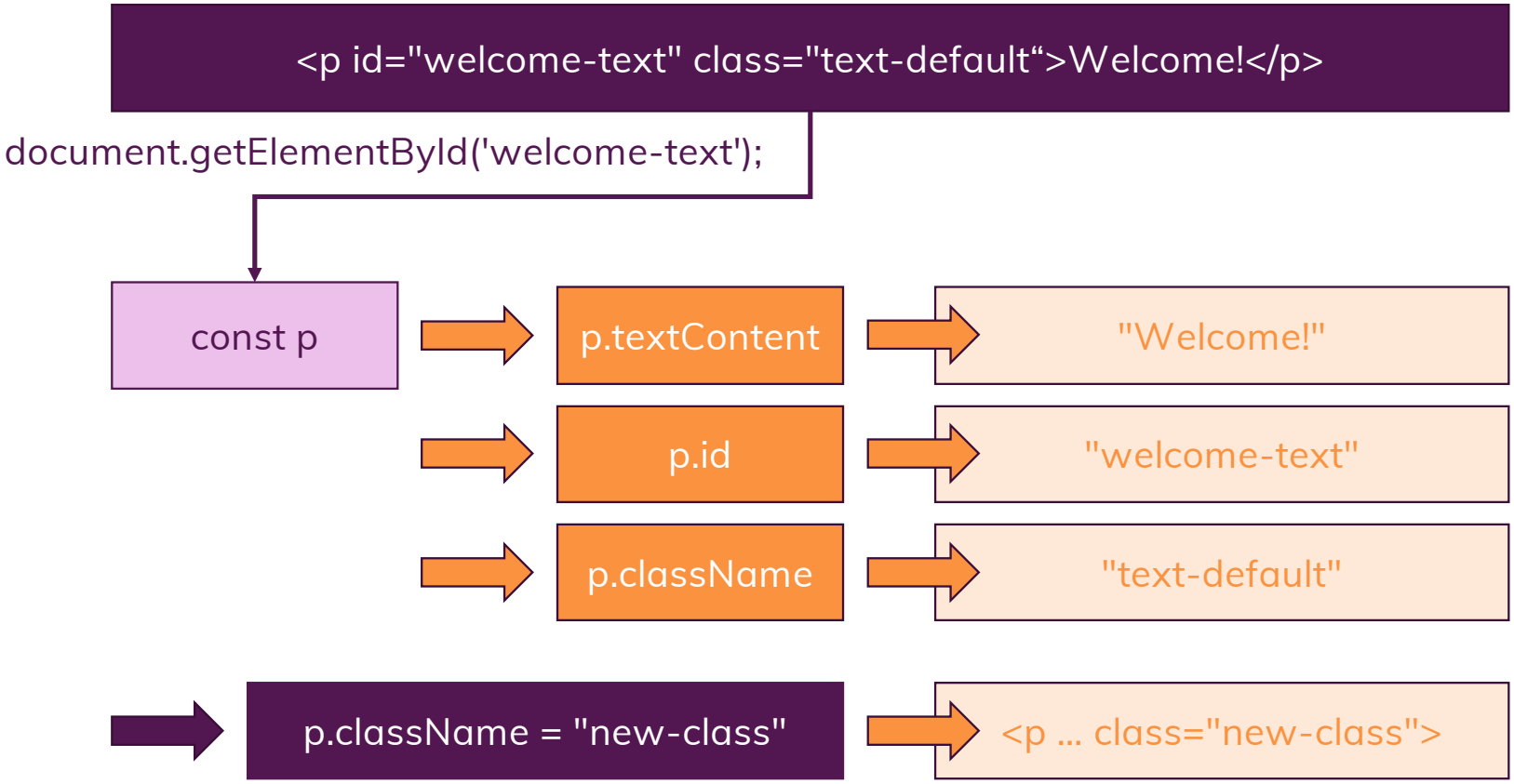
Besides the below query methods, you also got these special properties on the document object to select parts of the document:

`document.body` => Selects the `<body>` element node.
`document.head` => Selects the `<head>` element node.
`document.documentElement` => Selects the `<html>` element node

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

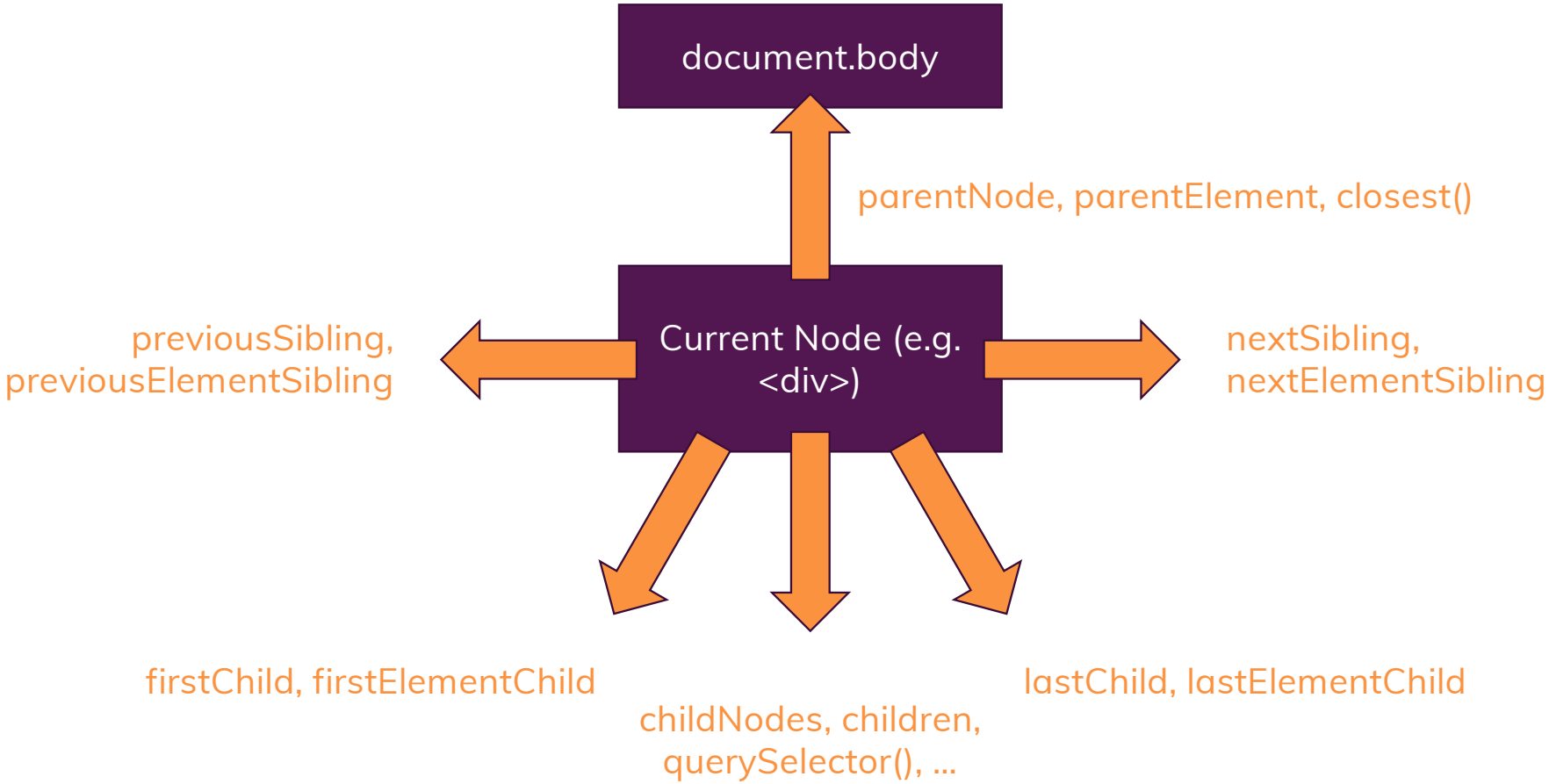
MDN for full reference of available properties + their behavior!

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
   A test! 	 A test! 	 A test! 	 A test!
is a child of . isn't!	is a descendant of . So is !	is a parent of but ot of !	is an ancestor of 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()

Summary: Insert, Replace, Remove

There are many ways of creating, inserting, replacing and removing DOM elements - here's a summary of the options you have.

For browser support, check the provided links and also the "Browser Support" module you find later in the course.

Create & Insert

You got two main options: Provide an HTML snippet (e.g. via `innerHTML`) to a valid HTML snippet and let the browser render it OR create a DOM object in JS code and append/ insert it manually. The latter approach has the advantage of giving you direct access to the DOM object (useful for setting its properties or adding event listeners). The downside is that you have to write more code.

Adding HTML Code:

```
const root = document.getElementById('root-el'); // selects something like <div id="root-el">
root.innerHTML = `
<div>
<h2>Welcome!</h2>
<p>This is all create & rendered automatically!</p>
</div>
`;
```

Important: Any existing content in `root` is completely replaced when using `innerHTML`. If you want to append/ insert HTML code, you can use `insertAdjacentHTML` instead: https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentHTML

```
const root = document.getElementById('root-el'); // selects something like <div id="root-el">
root.insertAdjacentHTML('afterbegin', `
<div>
<h2>Welcome!</h2>
<p>This is all create & rendered automatically!</p>
</div>
`);
```

Creating & Inserting DOM Objects Manually:

```
const someParagraph = document.createElement('p'); // creates a "p" element (i.e. a <p> element)
const root = document.getElementById('root-el'); // selects something like <div id="root-el">
root.append(someParagraph);
```

In this example, we create a paragraph and append it to `root` - append means that it's inserted at the end of `root` (i.e. inside of it but AFTER all other child nodes it holds).

Insertion Methods:

`append()` => https://developer.mozilla.org/en-US/docs/Web/API/ParentNode/append

Browser support is decent but for IE, `appendChild()` could be preferred => https://developer.mozilla.org/en-US/docs/Web/API/Node/appendChild

`prepend()` => https://developer.mozilla.org/en-US/docs/Web/API/ParentNode/prepend

Browser support is decent but for IE, `insertBefore()` could be preferred => https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore

`before()`, `after()` => https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/before & https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/after

Browser support is okay but IE and Safari don't support it.

Consider `insertBefore()` (https://developer.mozilla.org/en-US/docs/Web/API/Node/insertBefore) or `insertAdjacentElement()` (https://developer.mozilla.org/en-US/docs/Web/API/Element/insertAdjacentElement) as substitutes.

Important (no matter how you insert elements): Whenever you insert elements, you **MOVE the element** to that new place if you already inserted it before. It's **NOT copied** (you can copy an element via `someElement.cloneNode(true)` though).

Replace

You can replace elements in the DOM with two main methods:

`replaceWith()` => https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/replaceWith

`replaceChild()` => https://developer.mozilla.org/en-US/docs/Web/API/Node/replaceChild

`replaceWith()` is a bit easier to use and has decent browser support - with IE being the exception. To support that as well, consider using `replaceChild()`.

Remove

You can remove elements with three main methods:

`someElement.innerHTML = ''` => Clears all HTML content of `someElement` and hence removes any objects rendered in there.

`someElement.remove()` => Removes a single element (`someElement`) from the DOM (https://developer.mozilla.org/en-US/docs/Web/API/ChildNode/remove). Browser support is good, IE again doesn't like it though. Use `removeChild` (see below) instead.

`someElement.parentNode.removeChild(someElement)` => Removes the provided child element (NOT the element on which you call it). Provides broad browser support but of course requires a bit more code (https://developer.mozilla.org/en-US/docs/Web/API/Node/removeChild).

What about Text Nodes?

You can easily create & insert text nodes in one go:

```
someElement.textContent = 'Hi there!';
```

This creates and inserts the text node with a content of `'Hi there!'`.

Want to append to existing text?

Just use:

```
someElement.textContent = someElement.textContent + 'More text!';
```

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

Chaining Methods in JavaScript

With all these useful array methods you learned about, it's important to understand how you can combine them. Let's take `map()` and `reduce()` as an example:

```
1.  const originalArray = [{price: 10.99}, {price: 5.99}, {price: 29.99}];
2.  const transformedArray = originalArray.map(obj => obj.price); // produces [10.99, 5.99, 29.99]
3.  const sum = transformedArray.reduce((sumVal, curVal) => sumVal + curVal, 0); // => 46.97
```

Of course, you could skip the map step and just add the extraction logic to `reduce()`:

```
1.  const originalArray = [{price: 10.99}, {price: 5.99}, {price: 29.99}];
2.  const sum = originalArray.reduce((sumVal, curVal) => sumVal + curVal.price, 0); // => 46.97
```

But let's say you have a more complex extraction logic and hence want to split this into multiple method calls. Or you have a re-usable map function which you want to be able to use in different places of your app. Then you can still write the initial example in a more concise way if you **leverage method chaining**:

```
1.  const originalArray = [{price: 10.99}, {price: 5.99}, {price: 29.99}];
2.  const sum = originalArray.map(obj => obj.price)
3.  .reduce((sumVal, curVal) => sumVal + curVal, 0); // => 46.97
```

We call `.reduce()` directly on the result of `map()` (which produces an array, that's why this is possible). Hence we can avoid storing the mapped array in a separate constant or variable that we might not need in any other place.

What are Objects?

Core Data Structure in JavaScript

Typically reflect “real-world” entities

- e.g. button, movie
- Allow us to apply real-world logic to coding

Made up of properties & methods

- Store data in properties and actions in methods
- Allow you to “group” related data together and split your code into logical pieces

Objects in JavaScript

Primitive Values

- Numbers
- Strings
- Booleans
- null
- undefined
- Symbol

Reference Values (= Objects)

- Everything else!
- { ... }
- Arrays
- DOM Nodes
- ... other built-in objects

Objects & Primitive Values

Objects are **reference values** - you learned that.
It might not have been obvious yet but it's also important to recognize that, in the end, **objects are of course made up of primitive values**.
Here's an example:

```
1.  const complexPerson = {
2.  name: 'Max',
3.  hobbies: ['Sports', 'Cooking'],
4.  address: {
5.  street: 'Some Street 5',
6.  stateId: 5,
7.  country: 'Germany',
8.  phone: {
                i.  number: 12 345 678 9,
               ii.  isMobile: true
9.  }
10. },
11. };
```

Event though `complexPerson` has multiple nested reference values (nested arrays and objects), you end up with primitive values if you drill into the object.
`name` holds a string (`'Max'`) => **Primitive value**
`hobbies` holds an array (i.e. a reference value) which is full of strings (`'Sports'`, `'Cooking'`) => **Primitive values**
`address` holds an object which in turn holds a mixture of primitive values like `'Some Street 5'` and nested objects (`phone`), but if you dive into `phone`, you find only numbers and booleans in there => **Primitive values**
So you could say: **Primitive** values are the **core building blocks** that hold your data, **objects (and arrays)** are helpful for **organizing** and **working** with that data.

"this" - Summary

The `this` keyword can lead to some headaches in JavaScript - this summary hopefully acts as a remedy.
`this` refers to different things, depending on where it's used and how (if used in a function) a function is called.
Generally, `this` refers to the "thing" which called a function (if used inside of a function). That can be the global context, an object or some bound data/ object (e.g. when the browser binds `this` to the button that triggered a click event).

1) `this` in Global Context (i.e. outside of any function)

```
function something() { ... }

console.log(this); // logs global object (window in browser) - ALWAYS (also in strict mode)!
```

2) `this` in a Function (non-Arrow) - Called in the global context

```
function something() {
    console.log(this);
}

something(); // logs global object (window in browser) in non-strict mode, undefined in strict mode
```

3) `this` in an Arrow-Function - Called in the global context

```
const something = () => {
    console.log(this);
}

something(); // logs global object (window in browser) - ALWAYS (also in strict mode)!
```

4) `this` in a Method (non-Arrow) - Called on an object

```
const person = {
    name: 'Max',
    greet: function() { // or use method shorthand: greet() { ... }
        console.log(this.name);
    }
};

person.greet(); // logs 'Max', "this" refers to the person object
```

5) `this` in a Method (Arrow Function) - Called on an object

```
const person = {
    name: 'Max',
    greet: () => {
        console.log(this.name);
    }
};

person.greet(); // logs nothing (or some global name on window object), "this" refers to global (window) object, even in stric
t mode
```

`this` can refer to unexpected things if you call it on some other object, e.g.:

```
const person = {
    name: 'Max',
    greet() {
        console.log(this.name);
    }
};

const anotherPerson = { name: 'Manuel' }; // does NOT have a built-in greet method!

anotherPerson.sayHi = person.greet; // greet is NOT called here, it's just assigned to a new property/ method on the "anotherP
erson" object

anotherPerson.sayHi(); // logs 'Manuel' because method is called on "anotherPerson" object => "this" refers to the "thing" whi
ch called it
```

If in doubt, a `console.log(this);` can always help you find out what `this` is referring to at the moment!

What's Object-oriented Programming (OOP)?

Work with (real-life) Entities in your Code

ProductList	Product	ShoppingCart
Renders a list of products which were fetched from a server (database)	Renders details about a product and allows addition to cart	Renders cart products and total and allows user to order them
Object holds rendering + fetching logic	Object holds rendering + cart-adding logic	Object holds rendering + ordering (server communication) logic

Classes & Instances

Objects	Classes
"The things you work with in code"	"Blueprints for objects" (theoretical definition)
Instances of classes (= based on classes)	Define how objects look like, which properties and methods they have
Class-based creation is an alternative to using object literals!	Classes make creation of multiple, similar objects much easier

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

Are object literals ({}) obsolete?

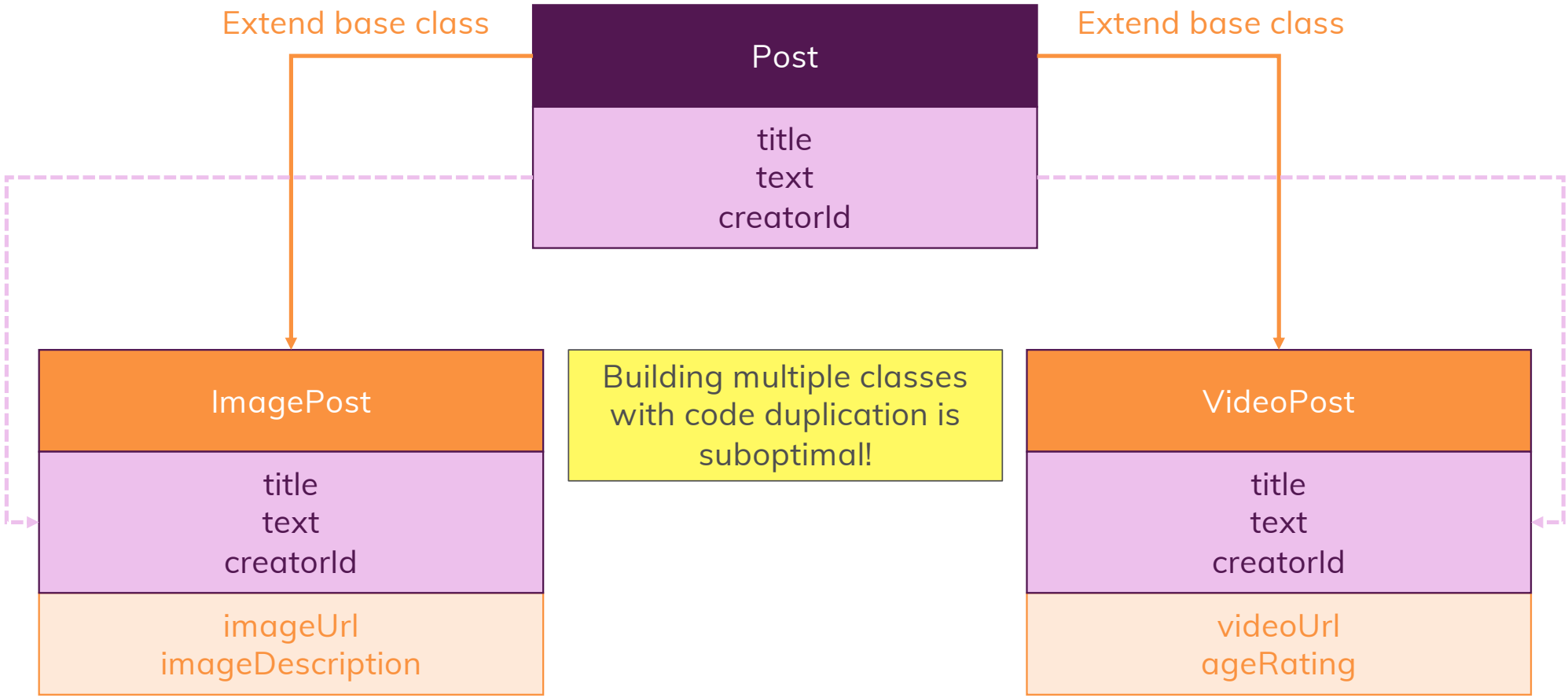
No!

- Great for general data grouping, objects which you only create once
- Quick & easy to create, no overhead

But also consider classes!

- Great when you re-create the same type of object over and over again
- More overhead initially but easy "object duplication" thereafter

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

"Pseudo-Private" Properties

The addition of private fields and properties is **relatively new** - in the past, such a feature was not part of JavaScript. Hence you might find many scripts that use a concept which you could describe as **"pseudo-private" properties**.

It would look like this:

```
class User {
constructor() {
      a. this._role = 'admin';
}
}
```

```
// or directly in an object
```

```
const product = {
_internalId: 'abc1'
};
```

What's that?

It's a quite common convention to prefix private properties with an underscore (`_`) to signal that they should not be accessed from outside of the object.

Important: It's **just a convention** that should signal something! It does **NOT technically prevent** access.

You **CAN** run this code without errors for example:

```
const product = {
_internalId: 'abc1'
};
console.log(product._internalId); // works!
```

It's really just a hint that developers should respect. It's not as strict as the "real" private properties introduced recently (`#propertyName`).

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

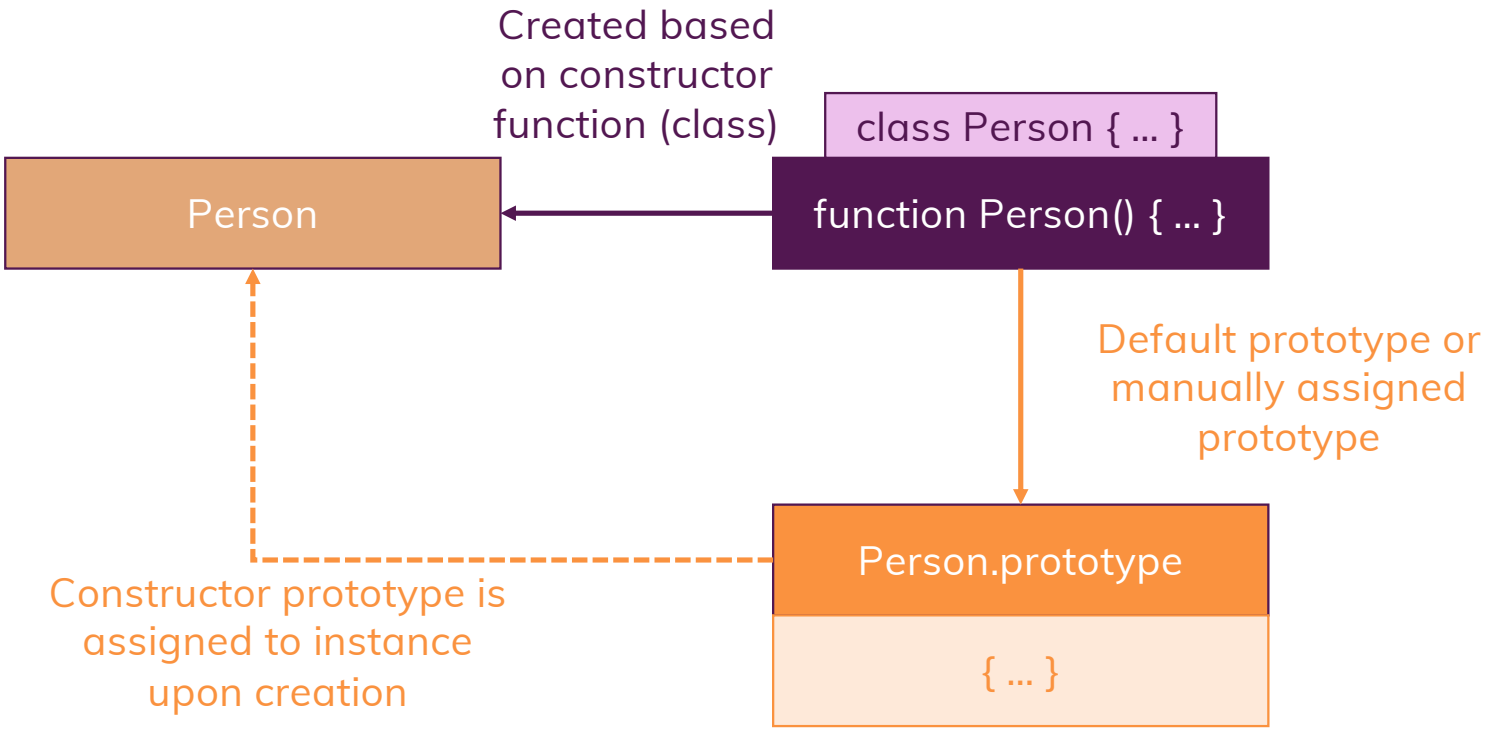
What are “Prototypes”?

JavaScript uses “Prototypical Inheritance”

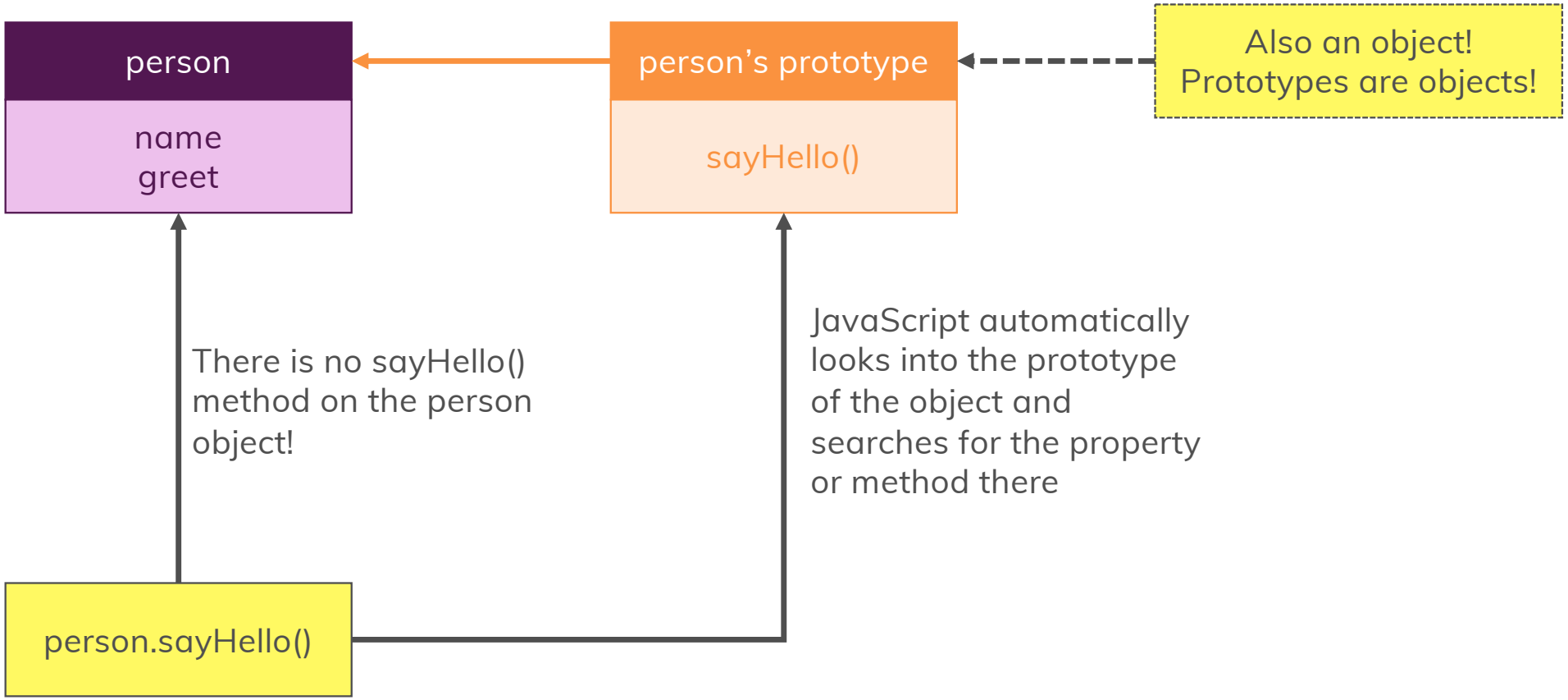
The `class` Syntax is basically just “syntactic sugar”

Constructor Functions & Prototypes power JavaScript Objects

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”



Prototypes - Summary

Prototypes can be a confusing and tricky topic - that's why it's important to really understand them.

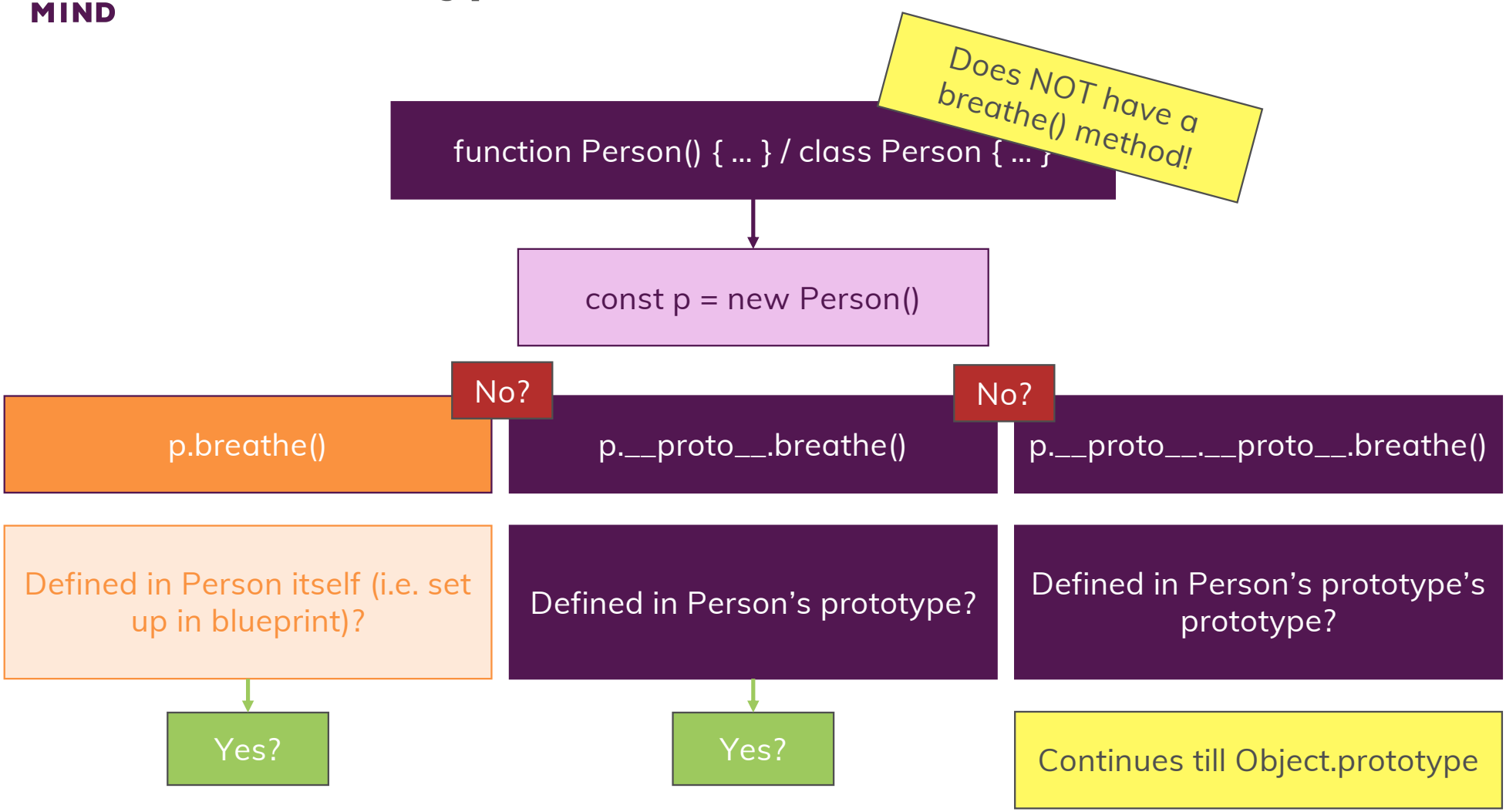
A prototype is an object (let's call it `"P"`) that is linked to another object (let's call it `"O"`) - it (the prototype object) kind of acts as a "**fallback object**" to which the other object (`"O"`) can reach out if you try to work with a property or method that's not defined on the object (`"O"`) itself.
EVERY object in JavaScript by default has such a fallback object (i.e. a prototype object) - more on that in the next lectures.
It can be especially confusing when we look at how you configure the prototype objects for "to be created" objects based on constructor functions (that is done via the `.prototype` property of the constructor function object).
Consider this example:

```
function User() {
... // some logic, doesn't matter => configures which properties etc. user objects will have
}
User.prototype = { age: 30 }; // sets prototype object for "to be created" user objects, NOT for User function object
```

The `User` function here also has a prototype object of course (i.e. a connected fallback object) - **but that is NOT the object the `prototype` property points at**. Instead, you access the connected fallback/ prototype object via the **special `__proto__` property** which EVERY object (remember, functions are objects) has.
The `prototype` property does something different: **It sets the prototype object, which new objects created with this User constructor function will have.**
That means:

```
const userA = new User();
userA.__proto__ === User.prototype; // true
userA.__proto__ === User.__proto__ // false
```

The Prototype Chain



Prototypes & “Method Types”

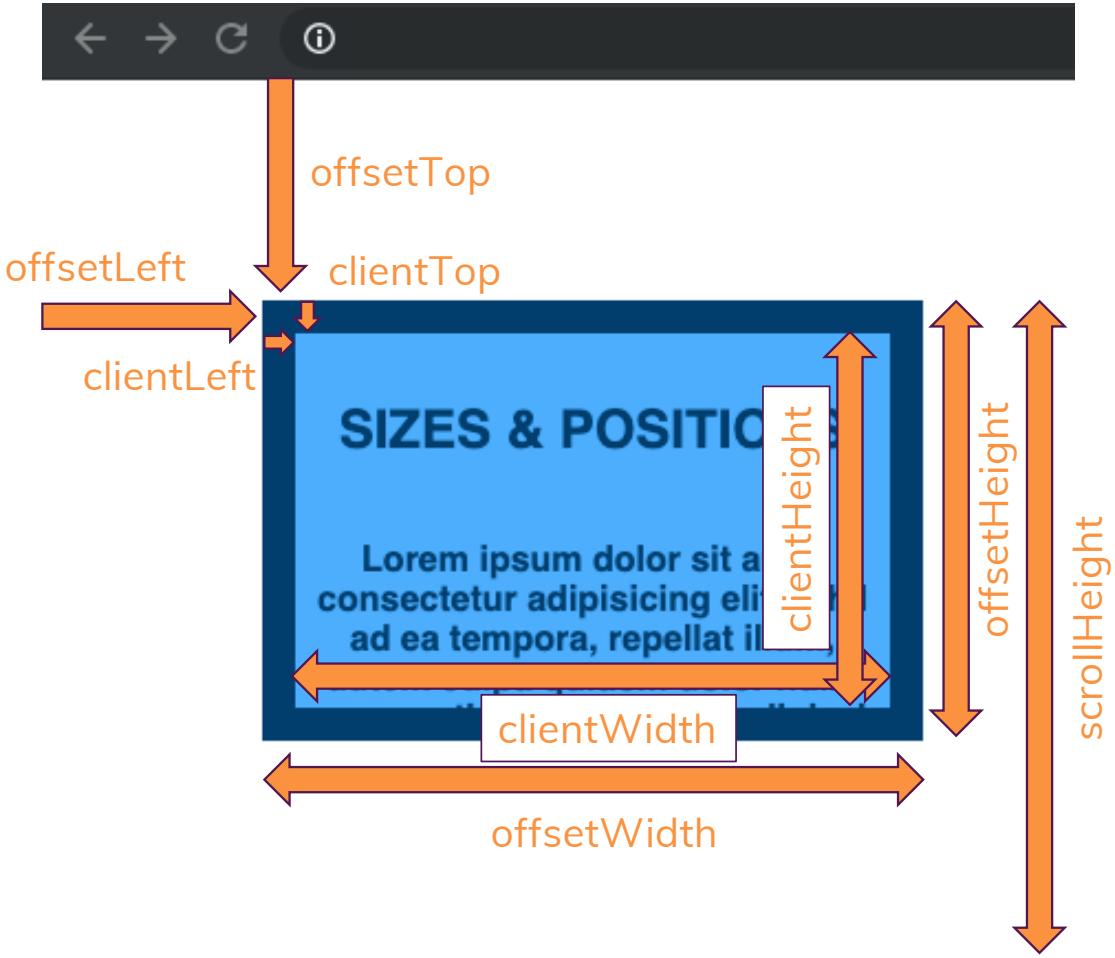
Method Shorthand	Property Function	Property Arrow Function

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!

Sizes & Positions



Events in JavaScript

Run Code Upon Certain Events

Exact Implementation Differs

e.g. Browser: addEventListener()

e.g. NodeJS: on()

Events Typically Transport Data

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

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

Example: Basic Infinite Scrolling

Let's have fun with the `scroll` event and create a list which you can scroll infinitely (explanations below)!

You can run this code snippet on any page - just make sure that **you can scroll vertically** (either by adding enough dummy content, by adding some styles that add a lot of `height` to some elements or by shrinking the browser window vertically).

```
1.   let curElementNumber = 0;

2.   function scrollHandler() {
3.   const distanceToBottom = document.body.getBoundingClientRect().bottom;

4.   if (distanceToBottom < document.documentElement.clientHeight + 150) {
          a. const newDataElement = document.createElement('div');
          b. curElementNumber++;
          c. newDataElement.innerHTML = `<p>Element ${curElementNumber}</p>`;
          d. document.body.append(newDataElement);
5.   }
6.   }

7.   window.addEventListener('scroll', scrollHandler);
```

So what's happening here?

At the very bottom, we register the `scrollHandler` function as a handler for the `'scroll'` event on our window object.

Inside that function, we first of all **measure the total distance** between our viewport (top left corner of what we currently see) and the end of the page (**not** just the end of our currently visible area) => Stored in `distanceToBottom`.

For example, if our browser window has a height of `500px`, then distanceToBottom could be `684px`, assuming that we got some content we can scroll to.

Next, we **compare the distance** to the bottom of our overall content (`distanceToBottom`) **to the window height + a certain threshold** (in this example `150px`). `document.documentElement.clientHeight` is preferable to `window.innerHeight` because it respects potential scroll bars.

If we have **less than `150px` to the end of our page content**, we make it into the if-block (where we append new data).

Inside of the if-statement, we then create a new `<div>` element and populate it with a `<p>` element which in turn outputs an incrementing counter value.

Firefox Adjustments

I do recommend to follow along in Chrome but in case you're using Firefox, you might be seeing some strange behaviors / errors.

Here are some code adjustments you can make to make it work in Firefox as well:

In

```
list.addEventListener('drop', event => {
```

add the following line at the beginning:

```
event.preventDefault
```

I.e. it should look like this:

```
list.addEventListener('drop', event => {
event.preventDefault();
// other code...
```

In

```
list.addEventListener('dragleave', event => {
```

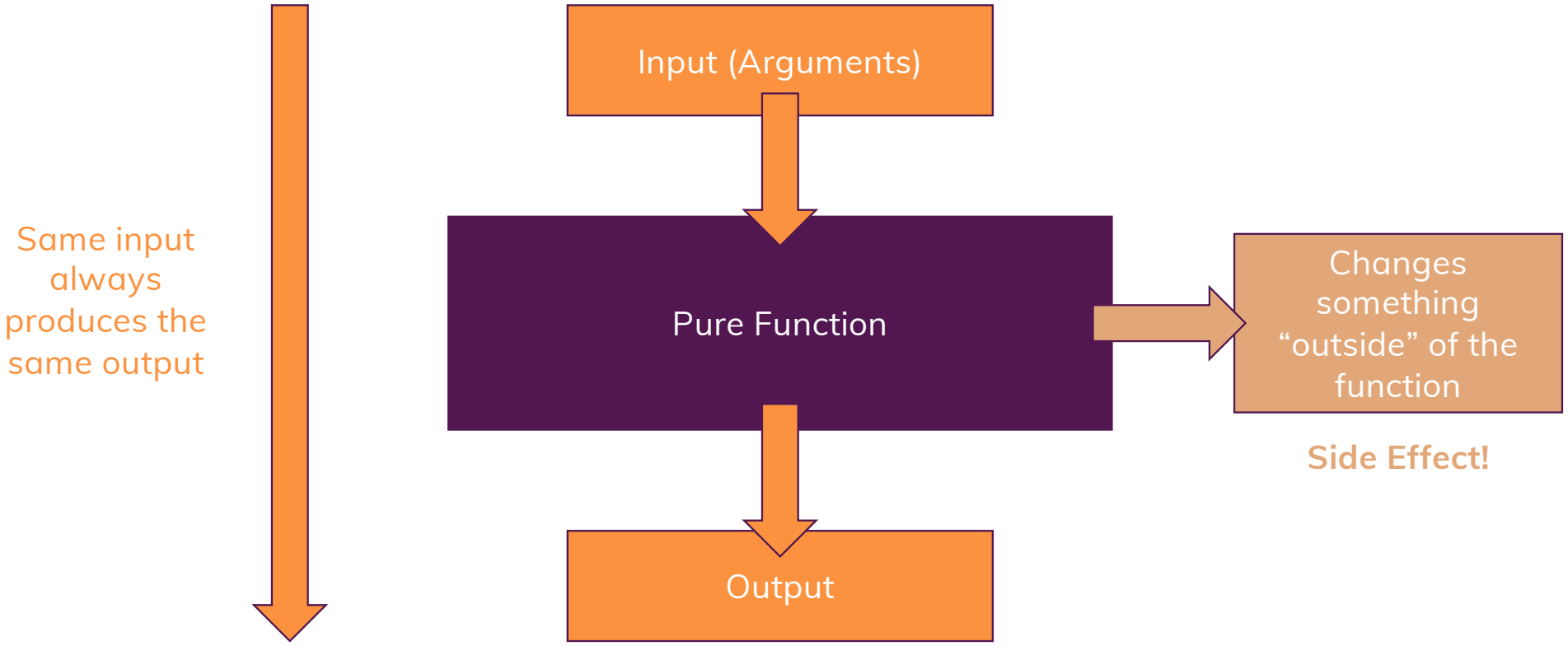
adjust the if statement to look like this:

```
if (event.relatedTarget.closest && event.relatedTarget.closest(...) {...}
```

I.e. it should look like this:

```
list.addEventListener('dragleave', event => {
if (event.relatedTarget.closest && event.relatedTarget.closest(...) {...}
```

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



Optional: IIFEs

In JavaScript - especially in older scripts - you sometimes find a pattern described as "**IIFEs**". IIFE stands for "**I**mmediately **I**nvoked **F**unction **E**xpression" and the pattern you might find looks like this (directly in a script file):

```
(function() {
var age = 30;
console.log(age); // 30
})()
```

```
console.log(age); // Error: "age is not defined"
```

What's that?

We see a function expression which calls itself (please note the `()` right after the function).
It's NOT a function declaration because it's wrapped in `()` - that happens on purpose since you can't immediately execute function declarations.

But why would you write some code?

Please note that the snippet uses `var`, NOT `let` or `const`. Remember that `var` does **NOT use block scope** but only differ between global and function scope.
As a consequence, it was hard to control where variables were available - variables outside of function always were available globally. Well, IIFEs solve that problem since the script (or parts of it) essentially are wrapped in a function => Function scope is used.

Nowadays, this is not really required anymore. With `let` and `const` we got block scope and if you want to restrict where variables are available (outside of functions, `if` statements, `for` loops etc - where you automatically have scoped variables since these structures create blocks), you can simply wrap the code that should have scoped variables with `{}`.

```
{
const age = 30;
console.log(age); // 30
}
```

```
console.log(age); // Error: "age is not defined"
```

Not something you see too often but something that is possible.

Tagged templates

A more advanced form of template literals are *tagged* templates.

Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.

The tag function can then perform whatever operations on these arguments you wish, and return the manipulated string. (Alternatively, it can return something completely different, as described in one of the following examples.)

The name of the function used for the tag can be whatever you want.

```
let person = 'Mike';

let age = 28;



function myTag(strings, personExp, ageExp) {

  let str0 = strings[0]; // "That "

  let str1 = strings[1]; // " is a "

  let str2 = strings[2]; // "."



  let ageStr;

  if (ageExp > 99){

    ageStr = 'centenarian';

  } else {

    ageStr = 'youngster';

  }



  // We can even return a string built using a template literal

  return `${str0}${personExp}${str1}${ageStr}${str2}`;

}



let output = myTag`That ${ person } is a ${ age }.`;



console.log(output);
```

```
// That Mike is a youngster.
```

Tag functions don't even need to return a string!

```
function template(strings, ...keys) {

  return (function(...values) {

    let dict = values[values.length - 1] || {};

    let result = [strings[0]];

    keys.forEach(function(key, i) {

      let value = Number.isInteger(key) ? values[key] : dict[key];

      result.push(value, strings[i + 1]);

    });

    return result.join('');

  });

}



let t1Closure = template`${0}${1}${0}!`;

//let t1Closure = template(["","","","!"],0,1,0);

t1Closure('Y', 'A');                      // "YAY!"



let t2Closure = template`${0} ${'foo'}!`;

//let t2Closure = template([""," ","!"],0,"foo");

t2Closure('Hello', {foo: 'World'}); // "Hello World!"



let t3Closure = template`I'm ${'name'}. I'm almost ${'age'} years old.`;

//let t3Closure = template(["I'm ", ". I'm almost ", " years old."], "name", "age");

t3Closure('foo', {name: 'MDN', age: 30}); //"I'm MDN. I'm almost 30 years old."

t3Closure({name: 'MDN', age: 30}); //"I'm MDN. I'm almost 30 years old."
```

Raw strings

The special `raw` property, available on the first argument to the tag function, allows you to access the raw strings as they were entered, without processing escape sequences.

```
function tag(strings) {

  console.log(strings.raw[0]);

}



tag`string text line 1 \n string text line 2`;

// logs "string text line 1 \n string text line 2" ,

// including the two characters '\' and 'n'
```

In addition, the `String.raw()` method exists to create raw strings—just like the default template function and string concatenation would create.

```
let str = String.raw`Hi\n${2+3}!`;

// "Hi\n5!"



str.length;

// 6



Array.from(str).join(',');

// "H,i,\,n,5,!"
```

Tagged templates and escape sequences

ES2016 behavior

As of ECMAScript 2016, tagged templates conform to the rules of the following escape sequences:

- Unicode escapes started by "`\u`", for example `\u00A9`
- Unicode code point escapes indicated by "`\u{}`", for example `\u{2F804}`
- Hexadecimal escapes started by "`\x`", for example `\xA9`
- Octal literal escapes started by "`\0o`" and followed by one or more digits, for example `\0o251`

This means that a tagged template like the following is problematic, because, per ECMAScript grammar, a parser looks for valid Unicode escape sequences, but finds malformed syntax:

```
latex`\unicode`

// Throws in older ECMAScript versions (ES2016 and earlier)

// SyntaxError: malformed Unicode character escape sequence
```

ES2018 revision of illegal escape sequences

Tagged templates should allow the embedding of languages (for example DSLs, or LaTeX), where other escapes sequences are common. The ECMAScript proposal Template Literal Revision (Stage 4, to be integrated in the ECMAScript 2018 standard) removes the syntax restriction of ECMAScript escape sequences from tagged templates.

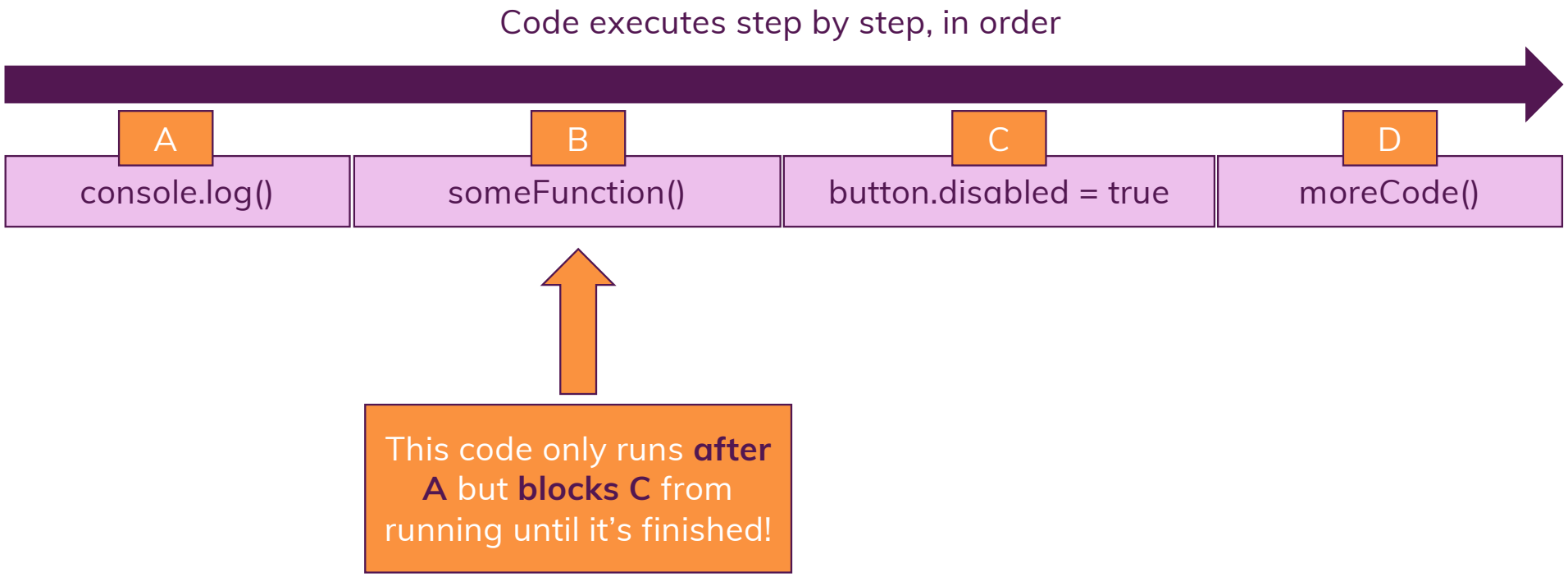
However, illegal escape sequences must still be represented in the “cooked” representation. They will show up as `undefined` element in the “cooked” array:

```
function latex(str) {

  return { "cooked": str[0], "raw": str.raw[0] }

}



latex`\unicode`



// { cooked: undefined, raw: "\\unicode" }
```

Note that the escape sequence restriction is only dropped from *tagged* templates—not from *untagged* template literals:

```
let bad = `bad escape sequence: \unicode`;
```

JavaScript is Single-Threaded



Certain Operations Take A Bit Longer

console.log()

setTimeout()

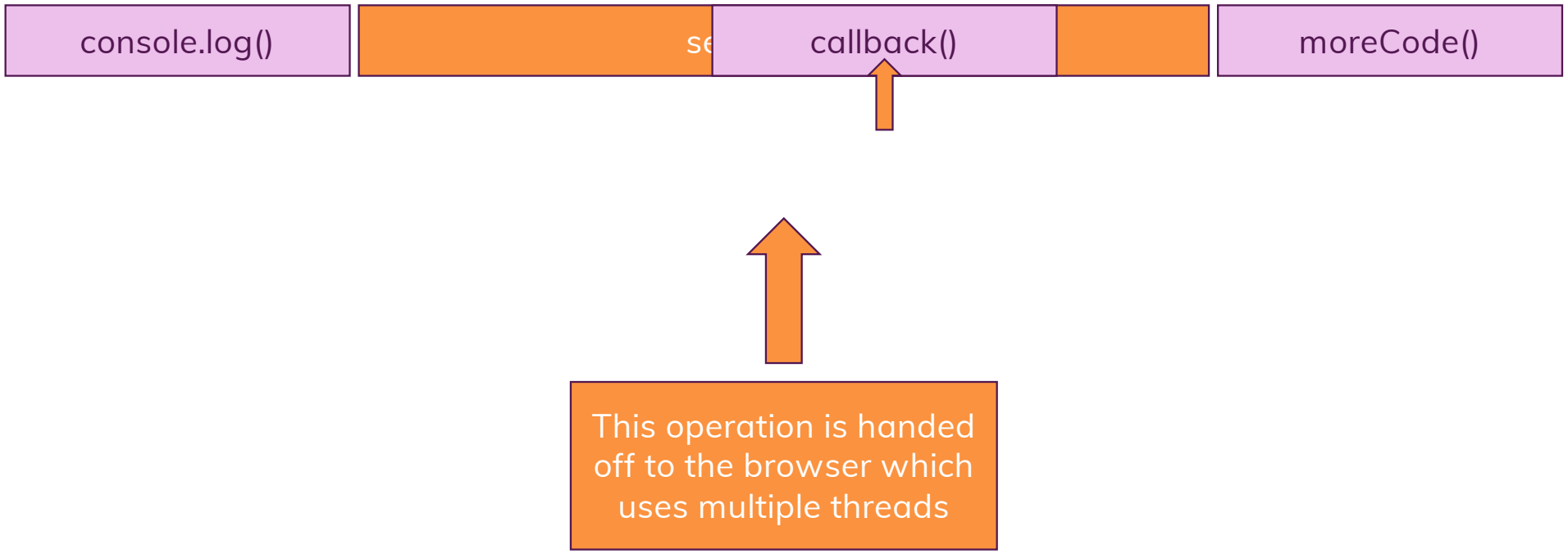
moreCode()

This operation **could take a bit longer** (wait until timer expired)

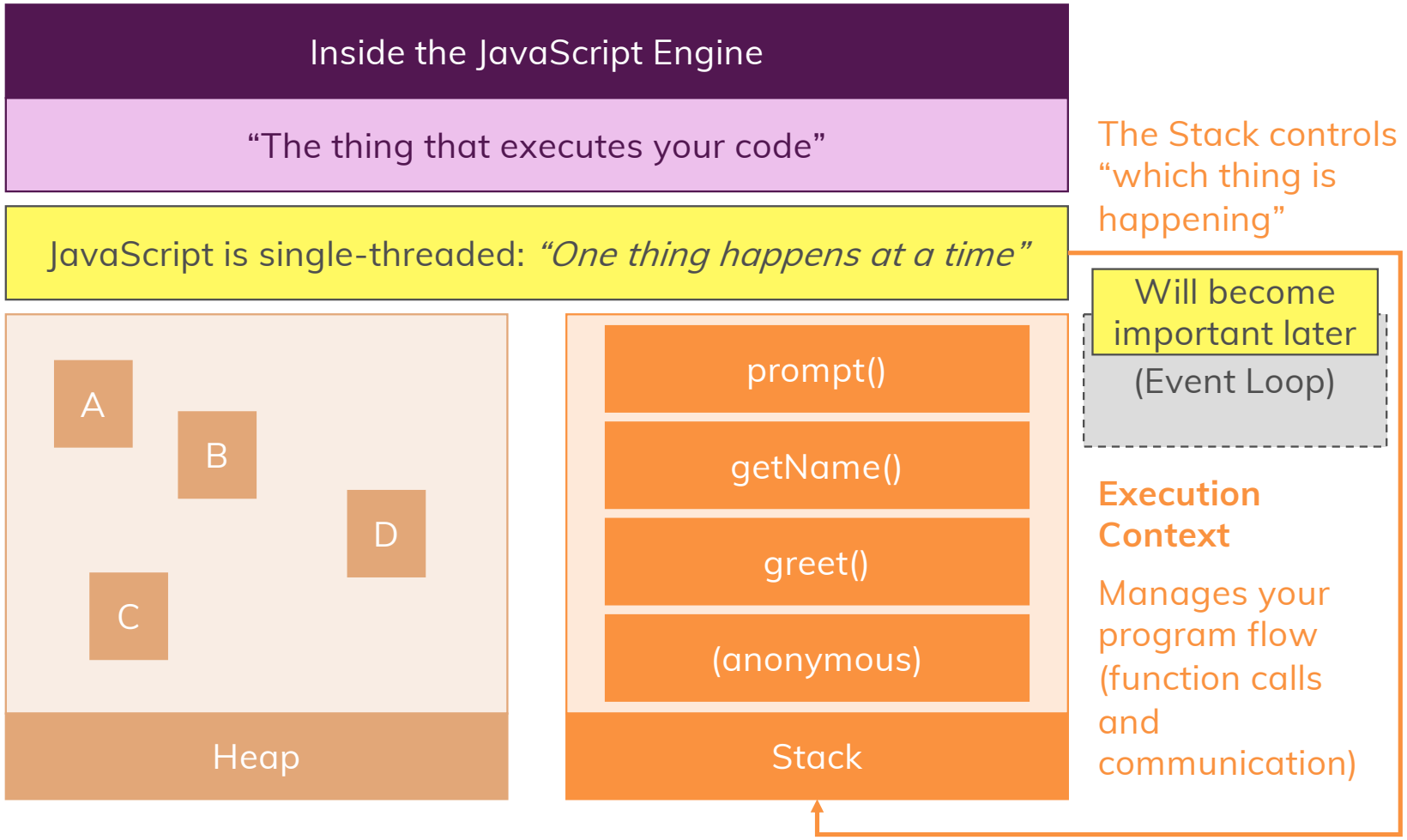
This operation shouldn't need to wait for the previous one to finish

Other examples: Http requests (waiting for response), getting user location, ...

Asynchronous Code Execution!



How Code Gets Executed



The Stack controls "which thing is happening"

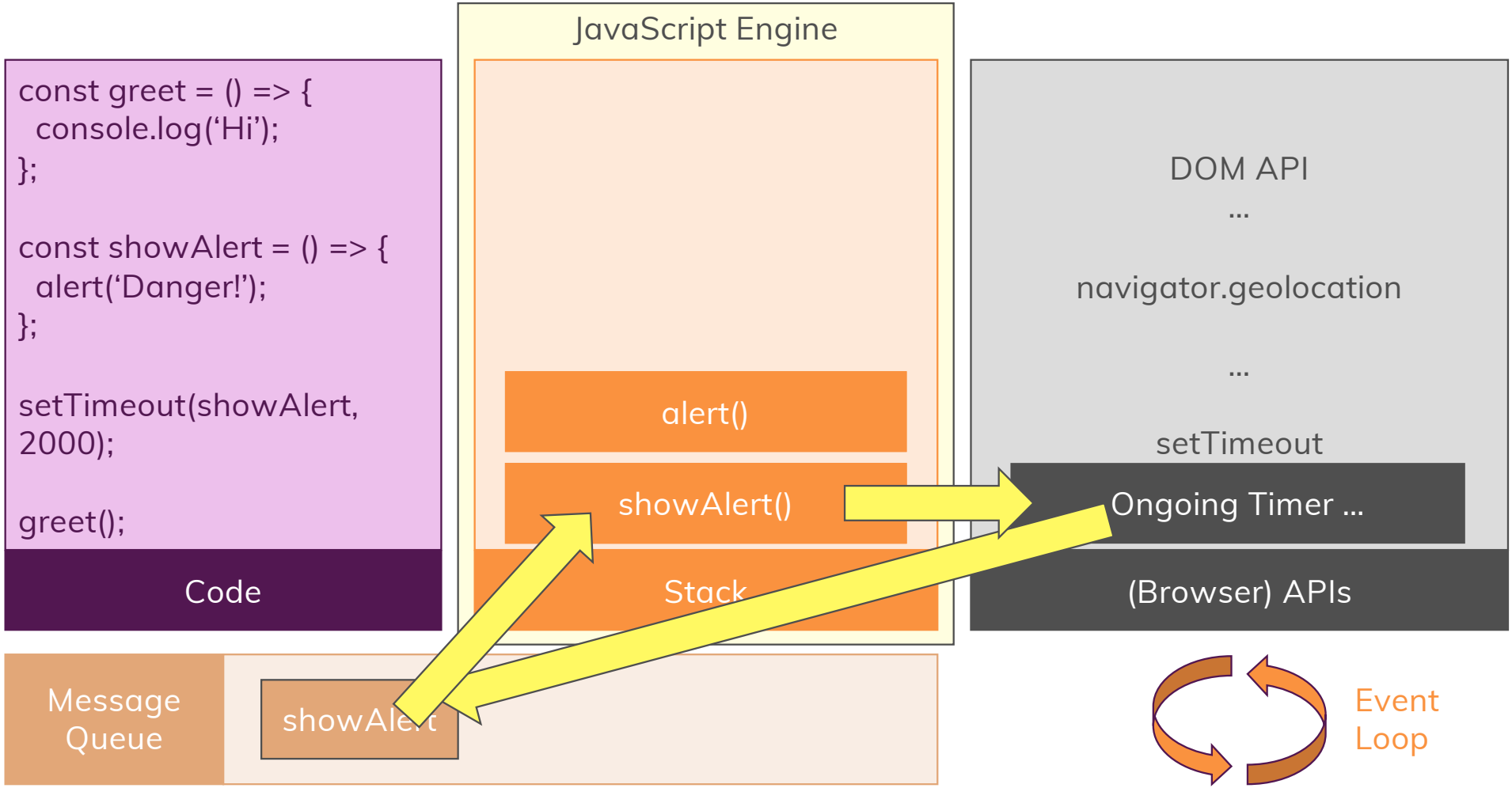
Memory allocation

Stores data in your system memory and manages access to it

Execution Context

Manages your program flow (function calls and communication)

Event Loop, Queue and Async Code



Promise States & "finally"

You learned about the different promise states:

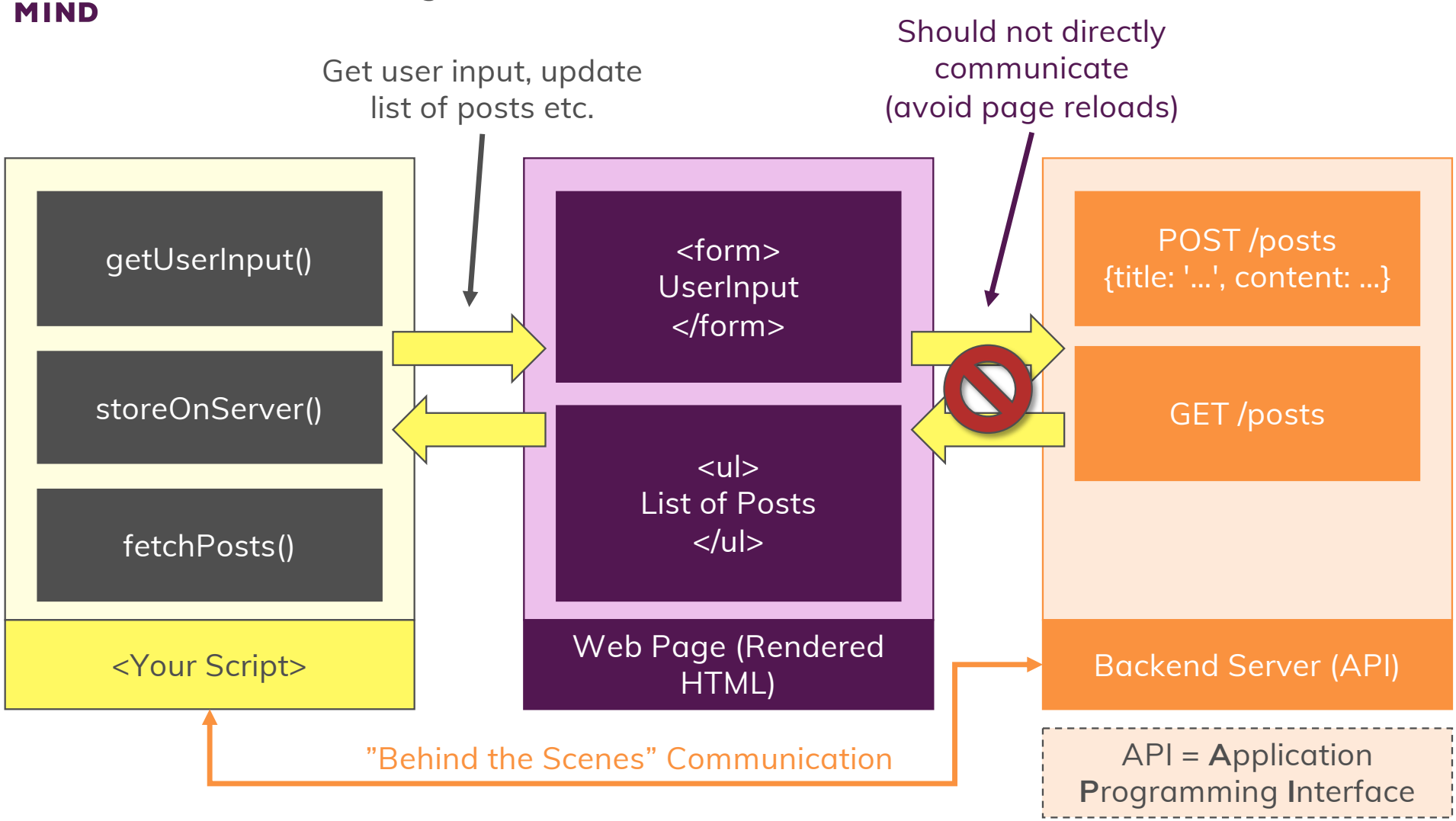
- **PENDING** => Promise is doing work, neither `then()` nor `catch()` executes at this moment
- **RESOLVED** => Promise is resolved => `then()` executes
- **REJECTED** => Promise was rejected => `catch()` executes

When you have another `then()` block after a `catch()` or `then()` block, the promise re-enters **PENDING** mode (keep in mind: `then()` and `catch()` always return a new promise - either not resolving to anything or resolving to what you `return` inside of `then()`). Only if there are no more `then()` blocks left, it enters a new, final mode: **SETTLED**.
Once **SETTLED**, you can use a special block - **`finally()`** - to do final cleanup work. `finally()` is reached no matter if you resolved or rejected before.
Here's an example:

```
somePromiseCreatingCode()
.then(firstResult => {
     a. return 'done with first promise';
})
.catch(err => {
     a. // would handle any errors thrown before
     b. // implicitly returns a new promise - just like then()
})
.finally(() => {
     a. // the promise is settled now - finally() will NOT return a new promise!
     b. // you can do final cleanup work here
});
```

You don't have to add a `finally()` block (indeed we haven't in the lectures).

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>
<ul> List of Posts </ul>
Web Page (Rendered HTML)
POST /posts {title: '...', content: ...}
GET /posts
Backend Server (API)
"Behind the Scenes" Communication
API = Application Programming Interface

How The Web Works

The web is everywhere!

We use it more than we ever did before - also in many places where you might not see it. Because “the web” is more than just websites you visit by entering a URL in your browser.

No matter if you check your e-mails on your mobile phone or if you’re sending a tweet - you are using the internet (i.e. “the web”).

How does all that work? Which technologies are involved and what do you need to learn (and to what extent) if you want to become a web developer?

In this article and video (see above), I will **not** dive into all technical details. This is meant to be a good overview of the web functionality.

CSS - The Complete Guide

Join this comprehensive 20h+ course to master CSS and learn how to create beautiful websites.

JavaScript - The Complete Guide

Learn JavaScript from scratch to build highly interactive and dynamic websites in this hands-on course!

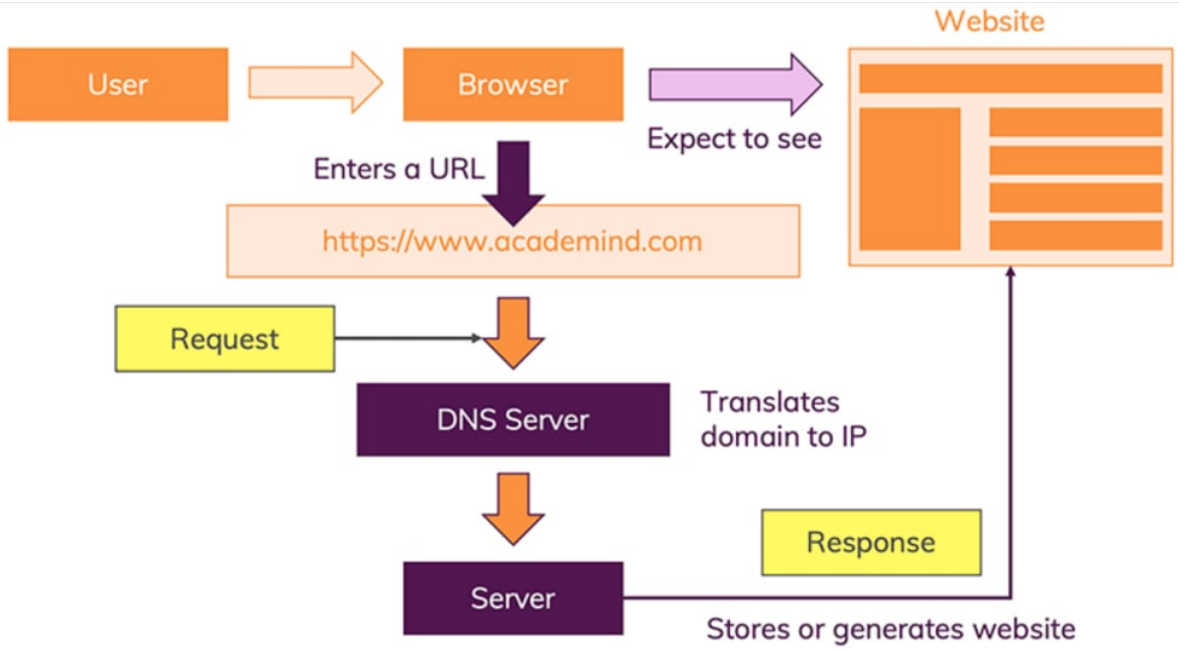
How Websites Work

Let’s start with the most obvious way of using the internet: You visit a website like academind.com.

The moment you enter this address in your browser and you hit ENTER, a lot of different things happen:

1. The URL gets resolved
2. A Request is sent to the server of the website

3. The response of the server is parsed
4. The page is rendered and displayed



Actually, every single step could be split up in multiple other steps, but for a good overview of how it all works, that's something we can ignore here. Let's have a look at all four steps.

Step 1 - URL Gets Resolved

The website code is obviously not stored on your machine and hence needs to be fetched from another computer where it is stored. This "other computer" is called a "server". Because it serves some purpose, in our case, it serves the website.

You enter "academind.com" (that is called "a domain") but actually, the server which hosts the source code of a website, is identified via IP (= Internet Protocol) addresses. The browser sends a "request" (see step 2) to the server with the IP address you entered (indirectly - you of course entered "academind.com").

In reality, you also often enter **`"academind.com/learn"`** or anything like that. **`"academind.com"`** is the **domain**, **`"/learn"`** is the so-called **path**. Together, they make up the **"URL"** ("Uniform Resource Locator").

In addition, you can visit most websites via **`"www.academind.com"`** or just **`"academind.com"`**. Technically, **`"www"`** is a **subdomain** but most websites simply redirect traffic to **`"www"`** to the main page.

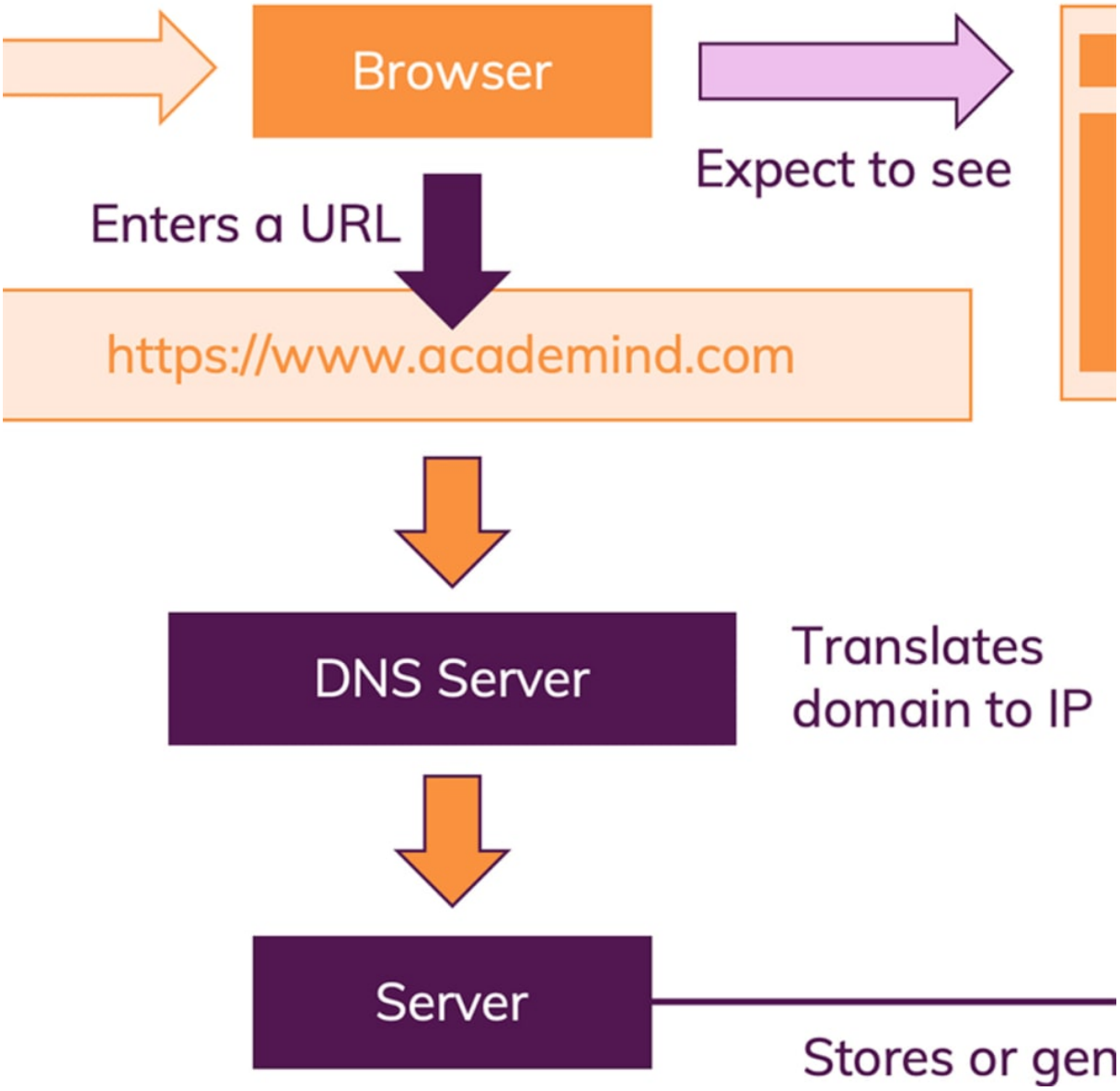
An IP address typically looks like this: **`172.56.180.5`** (though there also is a more "modern" form called IPv6 - but let's ignore that for now). You can learn more about IP addresses on Wikipedia.

How is the domain "academind.com" translated to its IP address?

There's a special type of server out there in the internet - not just one but many servers of that type. A so called "name server" or "DNS server" (where DNS = "Domain Name System").

The job of these DNS servers is to translate domains to IP addresses. You can imagine those servers as huge dictionaries that store translation tables: **Domain => IP address**.

When you enter "academind.com", the browser therefore first fetches the IP address from such a DNS server.



In case you're wondering: The browser knows the addresses of these domain servers by heart, they're programmed into the browser so to say.

Once the IP address is known, we advanced to step 2.

Step 2 - Request Is Sent

With the IP address resolved, the browser goes ahead and makes a request to the server with that IP address.

"A request" is not just a term. It really is a technical thing that happens behind the scenes.

The browser bundles up a bunch of information (What's the exact URL? Which kind of request should be made? Should metadata be attached) and sends that data package to the IP address.

▼ General

Request URL: `https://academind.com/`

Request Method: `GET`

Status Code: 🟢 `200`

Remote Address: `99.84.92.105:443`

Referrer Policy: `no-referrer-when-downgrade`

▶ Response Headers (11)

▼ Request Headers

:authority: `academind.com`

:method: `GET`

:path: `/`

:scheme: `https`

accept: `text/html,application/xhtml+xml,applicati`
`d-exchange;v=b3`

accept-encoding: `gzip, deflate, br`

The data is sent via the “HyperText Transfer Protocol” (known as “HTTP”) - a standardized protocol which defines what a request (and response) has to look like, which data may be included (and in which form) and how the request will be submitted. You can learn more about HTTP here.

Because HTTP is used, a full URL actually looks like this: http://academind.com. The browser auto-completes it for you.

And there also is **HTTPS** - it’s like HTTP but **encrypted**. Most modern pages (including academind.com) use that instead of HTTP. A full URL then becomes: https://academind.com.

Since the whole process and format is standardized, there is no guessing about how that request has to be read by the server.

The server then handles the request appropriately and returns a so called “response”. Again, a “response” is a technical thing and kind of similar to a “request”. You could say it’s basically a “request” in the opposite direction.

Like a request, a response can contain data, metadata etc. When requesting a page like academind.com, the response will contain the code that is required to render the page onto the screen.

What happens on the server?

That's defined by web developers. In the end, a response has to be sent. That response doesn't have to contain "a website". It can contain any data - including files or images.

Some servers are programmed to generate websites dynamically based on the request (e.g. a profile page that contains your personal data), other servers return pre-generated HTML pages (e.g. a news page). Or both is done - for different parts of a webpage. There also is a third alternative: Websites that are pre-generated but that change their appearance and data in the browser.

The different kinds of websites are not really the focus of this article. If you want to learn more about that, check out this article + video.

For our simple case we have a server that returns the code to display a website. So let's continue with step 3.

Step 3 - Response Is Parsed

The browser receives the response sent by the server. This alone, doesn't display anything on the screen though.

Instead, the next step is that the browser parses the response. Just as the server did it with the request. Again, the standardization enforced by HTTP helps of course.

The browser checks the data and metadata that's enclosed in the response. And based on that, it decides what to do.

You might've had cases where a PDF opened in your browser. That happened because the response informed the browser that the data is **not** a website but a PDF document instead. And the browser tries to pick the best handling mechanism for any data type it detects.

Back to our website scenario.

In that case, the response would contain a specific piece of metadata, that tells the browser that the response data is of type `text/html`.

This allows the browser to then parse the actual data that's attached to the response as HTML code.

HTML is the core "programming language" (technically, it's not a programming language - you can't write any logic with it) of the web. HTML stands for "Hyper Text Markup Language" and it describes the structure of a webpage.

The code looks like this:

```
<h1>Breaking News!</h1>
<p>Websites work because browser understand HTML!</p>
```

<h1> and **<p>** are so called "HTML tags" and if you want to learn more about HTML, this series is a great place to go.

Every HTML tag has some semantic meaning which the browser understands, because HTML is also standardized. Hence there is no guessing about what a **<h1>** tag means.

The browser knows how to parse HTML and now simply goes through the entire response data (also called "the response body") to render the website.

Step 4 - Page Is Displayed

As mentioned, the browser goes through the HTML data returned by the server and builds a website based on that.

Though it is important to know, that HTML does not include any instructions regarding **what** the site should look like (i.e. how it should be styled). It really only defines the structure and tells the browser which content is a heading, which content is an image, which content is a paragraph etc. This is especially important for accessibility - screen readers get all the useful information out of the HTML structure.

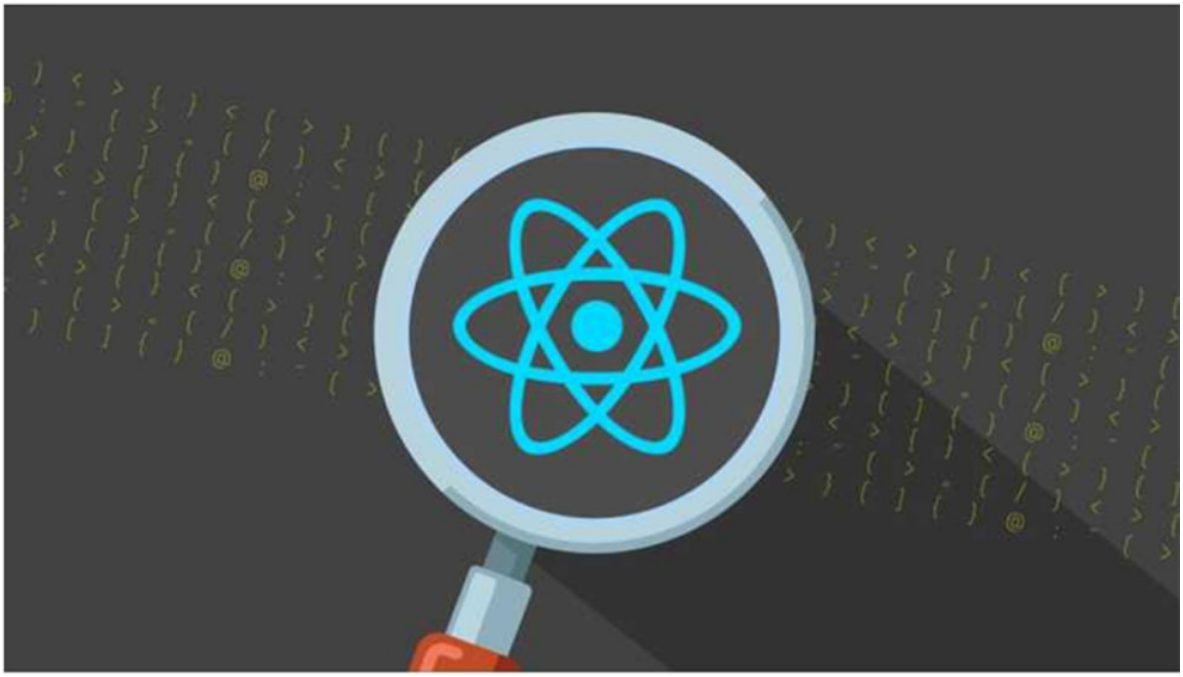
A page that only includes HTML would look like this though:

Angular - The Complete Guide

09 January, 2019

Join our comprehensive 28h+ course and find out why Angular is an awesome JavaScript framework for building modern and engaging web apps!

HOT



React - The Complete Guide

06 February, 2019

Learn Reactjs, Hooks, Redux, React Routing, Animations, Next.js basics and way more! What are all these things and how do we use them?

Not that beautiful, right?

That's why there's another important technology (another "programming language", which isn't really a programming language): CSS ("Cascading Style Sheets").

CSS is all about adding styling to the website. That is done via “CSS rules”:

```
h1 {
  color: blue;
}
```

This rule would color all `<h1>` tags blue.

Rules like that can be added inside of the HTML code but typically, they’re part of separate `.css` files which are requested separately.

Without diving into too many details here, that holds one important implication: A website can be made up of more than the data of the first response we get.

In practice, websites fetch a lot of additional data (via additional requests and responses) which are kicked off once the first response arrived.

How does that work?

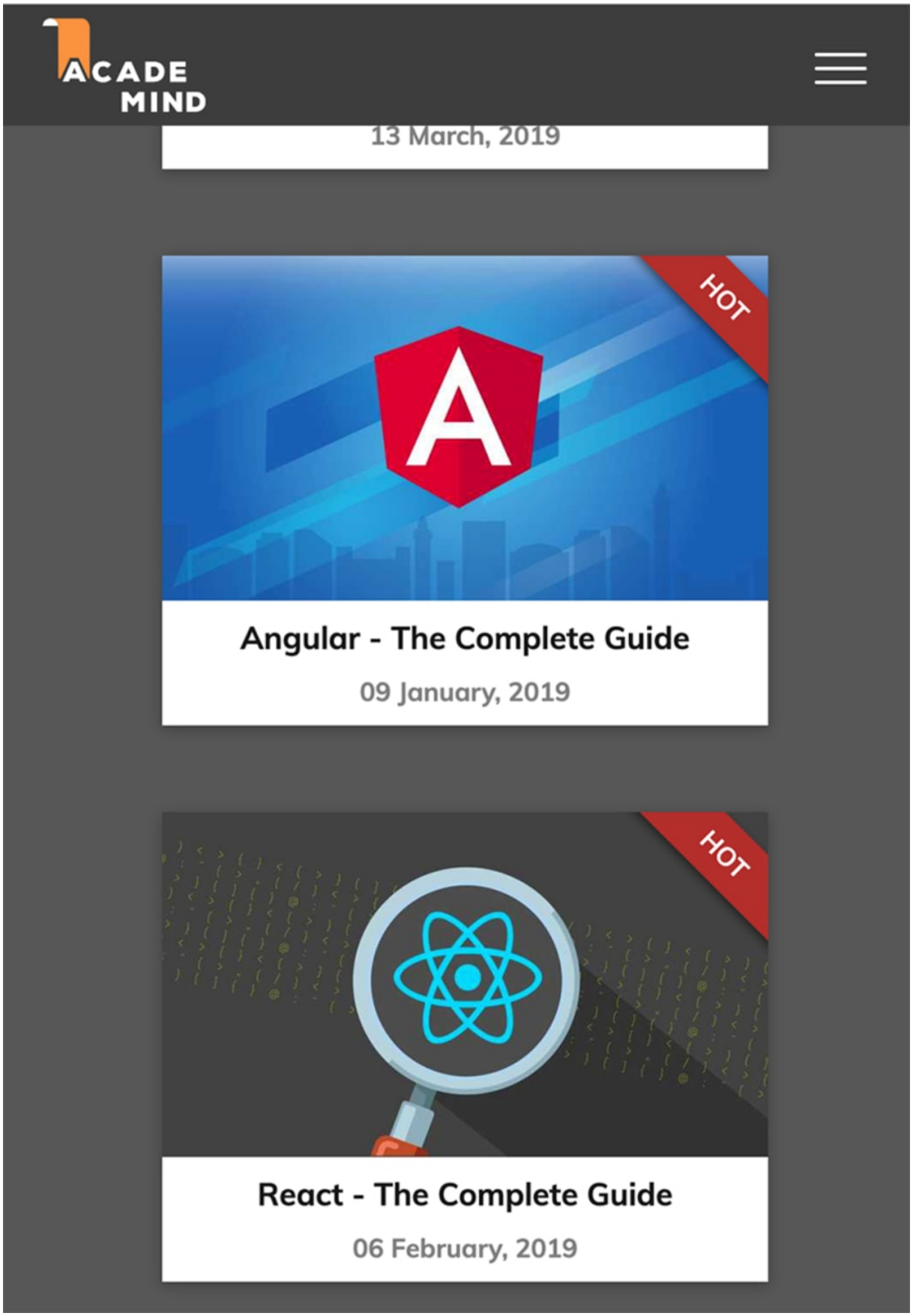
Well, the HTML code of the first response simply contains instructions to fetch more data via new requests - and the browser understands these instructions:

```
<link rel="stylesheet" href="/page-styles.css" />
```

Name	Status	Type
academind.com	200	document
component---node-mo…	200	script
styles-edc7da8ba2741…	200	script
app-10c9e1e6fd32855…	200	script
webpack-runtime-0d8…	200	script
path---offline-plugin-a…	200	json
css?family=Muli:300,7…	200	stylesheet
css?family=Muli:30…	200	fetch
analytics.js	200	script
analytics.js	200	fetch
pages-manifest-d586d…	200	script
1-8ff6e487b68be897ee…	200	script
2-db2daebb0d1fedf68…	200	script
3-325c965f8e273cc0c…	200	script
component---src-temp…	200	script

Again, I won’t dive into more details here. If you want to learn more about CSS, our Complete Guide will be very useful!

Together with CSS, the browser is able to display webpages like this:



There actually is another programming language involved (this time, it really is a programming language!): JavaScript.

It's not always visible but all dynamic content you find on a website (e.g. tabs, overlays etc.) is actually only possible because of JavaScript. It allows web developers to define code that runs in the browser (**not** on the server), hence JavaScript can be used to change the website whilst the user is viewing it.

As before, if you want to learn more, check out our JavaScript resources, for example our complete course.

These are the four steps that are **always** involved when you enter a page address like academind.com and you thereafter see the website content in your browser.

Server-side vs Browser-side

From the four steps above, you learned that we can differentiate two core "sides" when talking about the web: **Server-side** and **Browser-side** (or: **Client-side** since we can also access the internet without a browser - see below!).

If you're interested in becoming a web developer, it's important to know that you use different technologies and programming languages for these sides.

Server-side

You need server-side programming languages - i.e. languages that don't work in the browser but that can run on a normal computer (a server is in the end just a normal computer).

Examples would be:

- Node.js
- PHP
- Python

Important: With the exception of PHP, you can also use these programming languages for other purposes than web development.

Whilst Node.js is indeed primarily used for server-side programming (though it's technically not limited to that), Python is also very popular for data science and machine learning.

Browser-side

In the browser, there are exactly three languages/ technologies you need to learn. But whilst the server-side languages were alternatives, these three browser-side languages are all mandatory to know and understand:

- HTML (for the structure)
- CSS (for the styling)
- JavaScript (for dynamic content)

"Behind the Scenes" Internet

Thus far, we discussed websites. I.e. the case where you enter a URL (e.g. "academind.com/learn") into the browser and you get a website in return.

But the internet is more than that. Indeed, you use it for more than just that every day!

The core idea of requests and responses is always the same. But not every response is necessarily a website. And not every request wants a website.

The metadata which is attached to requests and responses controls which data is wanted and returned. Of course both parties that are involved (i.e. client and server) need to support the requestes and sent data.

You can't request a PDF from **`"academind.com"`** for example. You could send such a request but you wouldn't get back a PDF - simply because we don't support this kind of requested data for this specific URL.

But there are many servers that specialize in providing URLs that return certain pieces of data. Such services are also referred to as "APIs" ("Application Programming Interface").

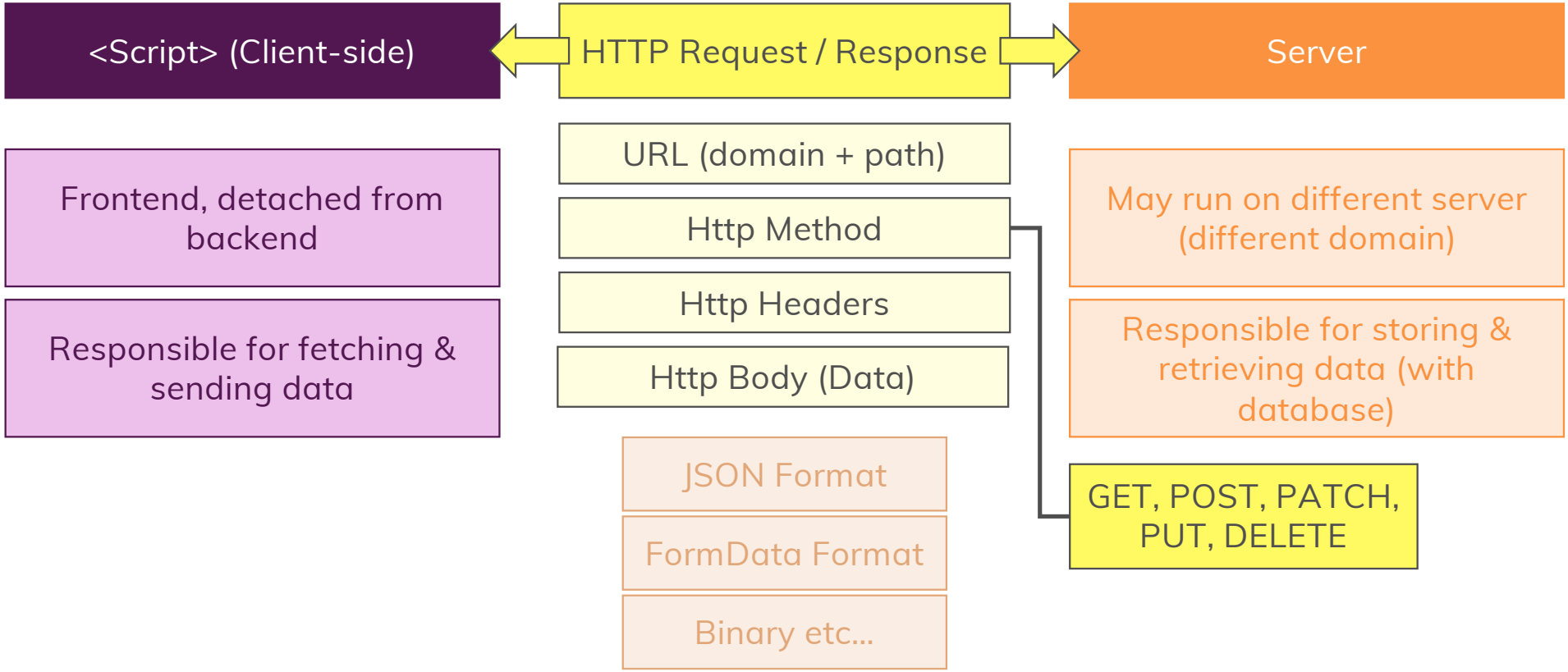
For example, mobile apps send "invisible" HTTP requests to such APIs (to specific URLs which are known to them) to get or store data. Twitter is fetching the tweet feed for example.

And even on webpages, such "invisible" requests are sent. If you sign up for our newsletter (which you of course should!), no new page will be loaded. Because data is exchanged behind the scenes. Even though the client is the browser in this case, the request which is sent wants no website in return. And the server URL that receives it offers no website - instead, the server knows how to handle your email address.

We could go into way more detail here but this is already a long article. You should now have a good understanding of how the web works and which core technologies are involved.

If you do want to dive deeper, explore the many resources we have on our page, sign up to our newsletter and of course subscribe to our YouTube channel for more in-depth tutorials and content! :)

(Background) Http Overview



JSON Data Deep Dive

Typically, data is transferred as "**JSON**" data between your client-side code and your backend ("the server").
JSON stands for **J**ava**S**cript **O**bject **N**otation and it looks like this:

```
{
"name": "Max",
"age": 30,
"hobbies": [
      a. { "id": "h1", "title": "Sports" },
      b. { "id": "h2", "title": "Cooking" }
],
"isInstructor": true
}
```

JSON data supports **objects** (`{}`), **arrays** (`[]`), **strings** (MUST use double-quotes!), **numbers** (NO quotes) and **booleans** (also NO quotes).
All object keys (e.g. "name") **HAVE to be wrapped by double quotes**. **No quotes** or **single quotes** are **NOT ALLOWED**!
Actually, the whole JSON "object" is wrapped in quotes itself because **JSON data in the end is just a string** that contains data in the format shown above.
You can test it yourself - take the following non-JSON JavaScript object and apply `JSON.stringify()` on it. This will convert it to JSON data. If you do that in the dev tools console, you'll see that you in the end get a string which contains data formatted as shown above.

```
const person = { // this is NOT JSON - it's a normal ("raw") JavaScript object!
name: 'Max',
age: 30,
hobbies: [
      a. { id: 'h1', title: 'Sports' },
      b. { id: 'h2', title: 'Cooking' }
],
isInstructor: true
};
```

```
const jsonData = JSON.stringify(person); // convert raw JS data to JSON data string
console.log(jsonData); // a string with machine-readable JSON data in it
console.log(typeof jsonData); // string
```

We use JSON data because it's easy to parse for machines - and as an extra benefit it's also quite readable to us humans.

If you receive some JSON data and you want to convert it back into normal JS data, you can use `JSON.parse()`:

```
const parsedData = JSON.parse(jsonData); // yields a "raw" JS object/ array etc.
```

Also nice to know: You're **NOT LIMITED to objects** when converting data to JSON. You can also convert numbers, arrays, booleans or just strings - all data types JSON supports:

```
const jsonNumber = JSON.stringify(2); // "2"
const jsonText = JSON.stringify('Hi there! I use single quotes in raw JS'); // ""Hi there! ...""
const jsonArray = JSON.stringify([1, 2, 3]); // "[1,2,3]"
const jsonBoolean = JSON.stringify(true); // "true"
```

Limitations of “Basic Projects”

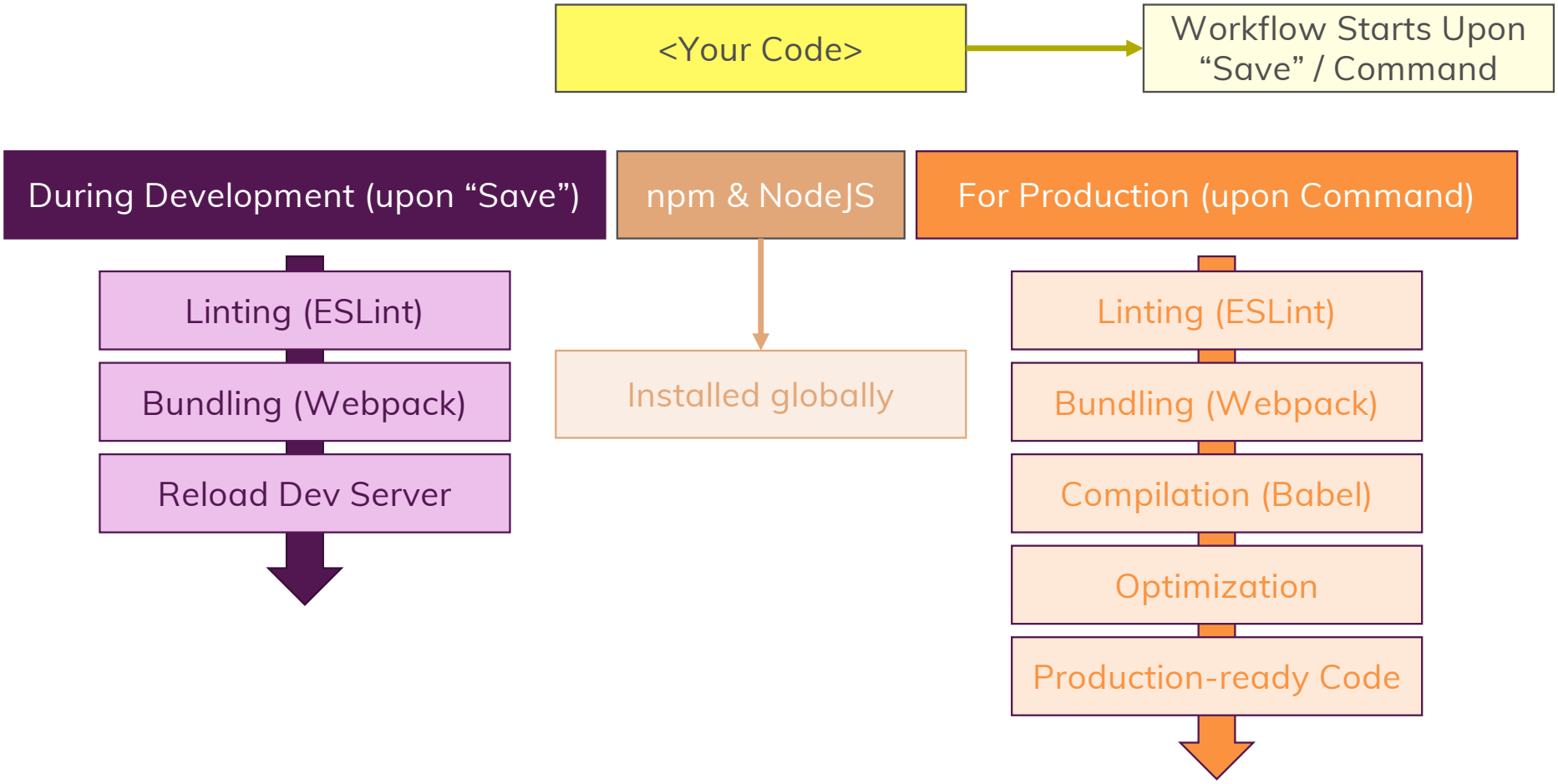
- Micromanagement of Imports OR Lots of Unnecessary Http Requests
- Unoptimized Code (not as small as possible)
- Potentially sub-optimal Browser Support
- Need to Reload Page Manually (after changes to code)
- Code Quality Is Not Checked

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

Configuring ESLint

ESLint **offers a lot of different options** so that you can really fine-tune it exactly to your requirements. You can set up your own rules from the ground up (basically what we started doing in the lectures) but you can also use presets and pre-configured rulesets.

To fully understand all options you can configure in `.eslintrc.json`, this part of the official docs should be helpful: https://eslint.org/docs/user-guide/configuring
To explore all available rules and what they mean, explore this part of the official docs: https://eslint.org/docs/rules/
Want to use a preset? Here you go: https://www.npmjs.com/search?q=eslint-config (just click on one of the results and follow the instructions provided there)
Also check out the docs in general: https://eslint.org/docs/user-guide/getting-started

Bonus: Multiple Entry Points

In the example project, we only have **one main entry point**: `app.js`.

In bigger projects - with multiple HTML pages - you **might have multiple scripts** for the different pages (HTML files) you might be building. Hence you **might need more than one entry point** because you want to build more than one bundle (i.e. not every HTML page uses the same script).

This can easily be configured with Webpack:

Instead of

```
1.  entry: './src/app.js'
```

2. use

```
3.  entry: {
4.      welcome: './src/welcome-page/welcome.js',
5.      about: './src/about-page/about.js',
6.      // etc.
7.  }
```

Now Webpack will **look up all these entry points** and create one bundle per entry point - you can then link to these bundles in your respective HTML files.

A simple rule that makes sense for most projects is:

One entry point per HTML file because you typically have one script per HTML file.

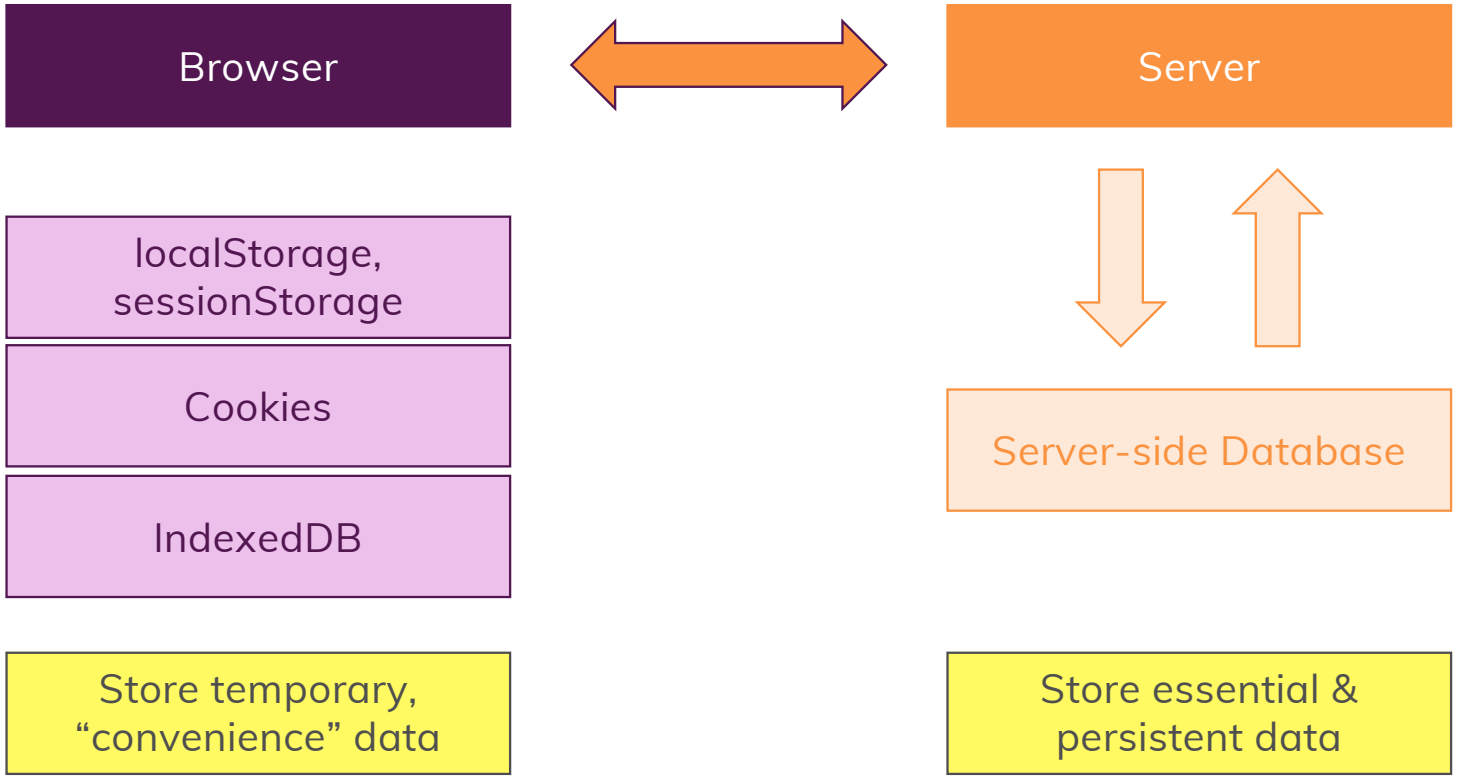
If you share a script across multiple HTML files or you have a file that does not need any script, you of course can deviate from that rule.

You can **learn more about multiple entry points** with these two resources:

- Code Splitting (i.e. generating more than one bundle): https://webpack.js.org/guides/code-splitting/
- Entry Point Configuration: https://webpack.js.org/concepts/#entry

And in general, check out the official Webpack docs to dive into it in detail: https://webpack.js.org/guides/

What is “Browser Storage”?



LocalStorage vs Cookies vs IndexedDB

localStorage, sessionStorage	Cookies	IndexedDB
Simple key-value store	Simple key-value store (some config options)	Relatively sophisticated, client-side database
Manage user preferences or basic user-data	Manage user preferences or basic user-data	Manage complex data your web app needs
Can be cleared by the user and via JS	Can be cleared by the user and via JS	Can be cleared by the user and via JS
Easy to use, quite versatile, bad for complex data	A bit clunky to use, quite versatile, sent to server, bad for complex data	A bit clunky to use, great for complex (non-critical) data, good performance

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)

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.

Checking Features & Browser Support

Is a feature available?

MDN	caniuse.com	Google	ES6/ JS Compat Table
Check feature article – browser support table at the bottom	Search for feature name to get a detailed overview table (incl. browser market shares!)	Google “fetch javascript browser support” and check results	For next-gen JavaScript syntax (!) features, check this overview table
Also often includes notes, fallbacks, workarounds (if support is missing)	Also often includes notes, fallbacks, workarounds (if support is missing)	For more complex issues, Stackoverflow discussions can be helpful	Provides detailed feature split and also includes support of transpilers

Which Browser Support Do You Need?

Naive Thought: Support **ALL Browsers** & **ALL Browser Versions**

DON'T do this!

Disadvantages: Ship too much code, pass up on some features & improvements

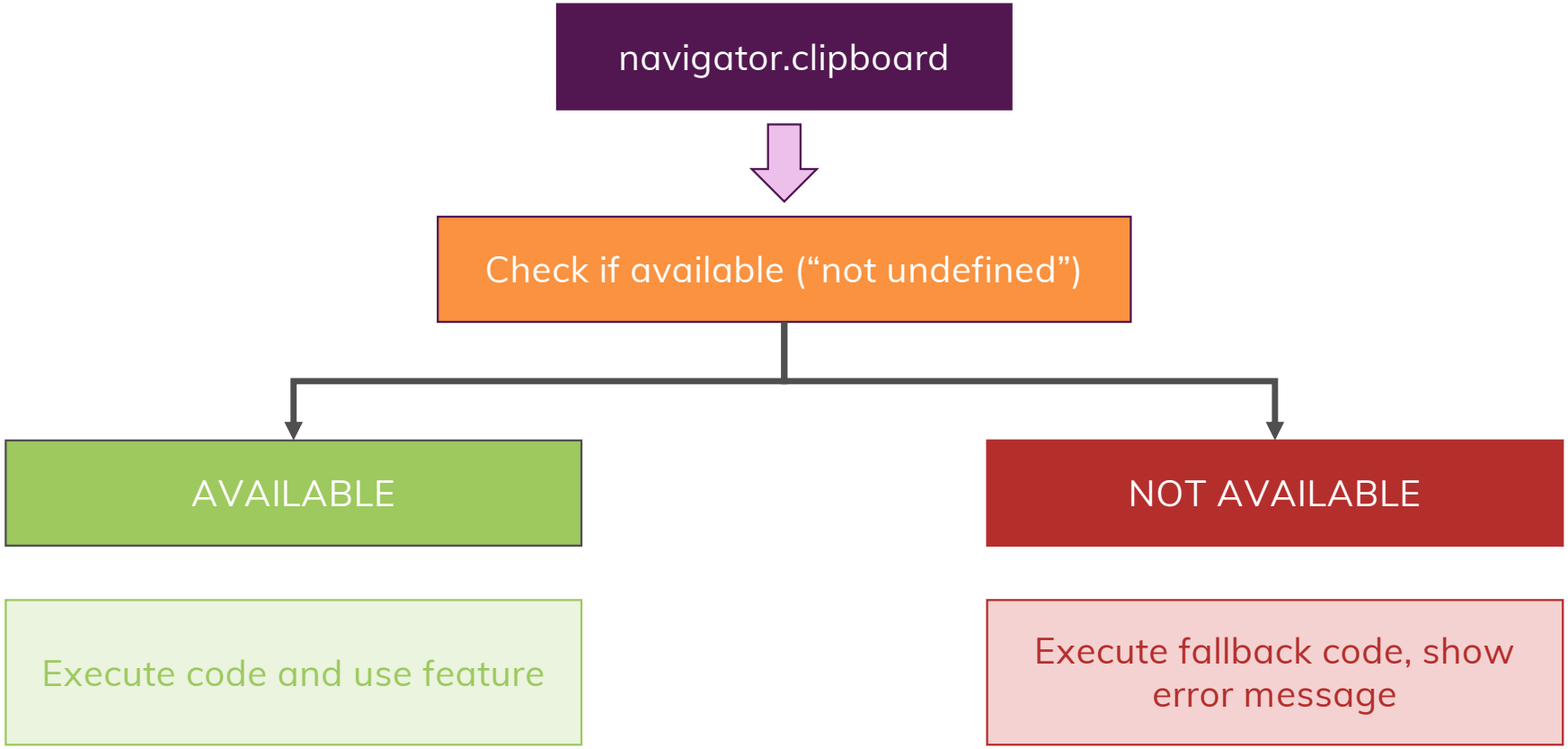
Analyse your market!
Who will be your users?

Internal tool/ app: Company browser might be clearly defined

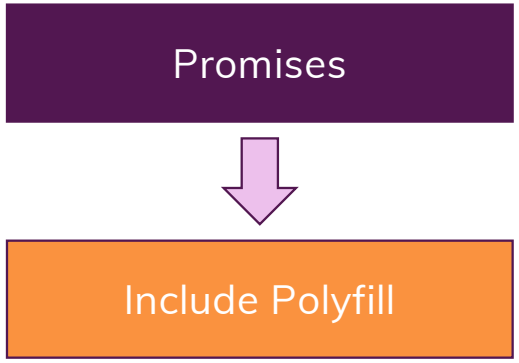
Tech-savvy users: Very likely to use cutting-edge browsers & versions

Older people, non-tech-savvy audience: Likely to use older browsers

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

Continuing without a Credit Card

Using Google Maps unfortunately requires a credit card, even though you got a generous free tier which you very likely wouldn't exceed.

If you got no credit card, you can look into OpenLayers as an alternative (here's how to render a map with it: https://openlayers.org/en/latest/doc/quickstart.html).
In our concrete example, this would render a map:

Include this in your HTML file:

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/gh/openlayers/openlayers.github.io@master/en/v6.1.1/css/ol.css" type="text/css">
<script src="https://cdn.jsdelivr.net/gh/openlayers/openlayers.github.io@master/en/v6.1.1/build/ol.js"></script>
```

In `Map.js`, use this JS code:

```
document.getElementById('map').innerHTML = ''; // clear the <p> in the <div id="map">
const map = new ol.Map({
target: 'map',
layers: [
new ol.layer.Tile({
source: new ol.source.OSM()
})
],
view: new ol.View({
center: ol.proj.fromLonLat([coordinates.lng, coordinates.lat]),
zoom: 16
})
});
```

You can explore the OpenLayers docs to learn how to render a broad variety of different things.

In the next lectures, we'll also use the Google API to convert a pair of coordinates into an address and vice versa. For that, we'll also use the Google APIs and hence need a credit card.

Unfortunately, I'm not aware of any credit-card-free alternatives, so for now the solution will be to simply return some dummy data in those utility functions we'll add:

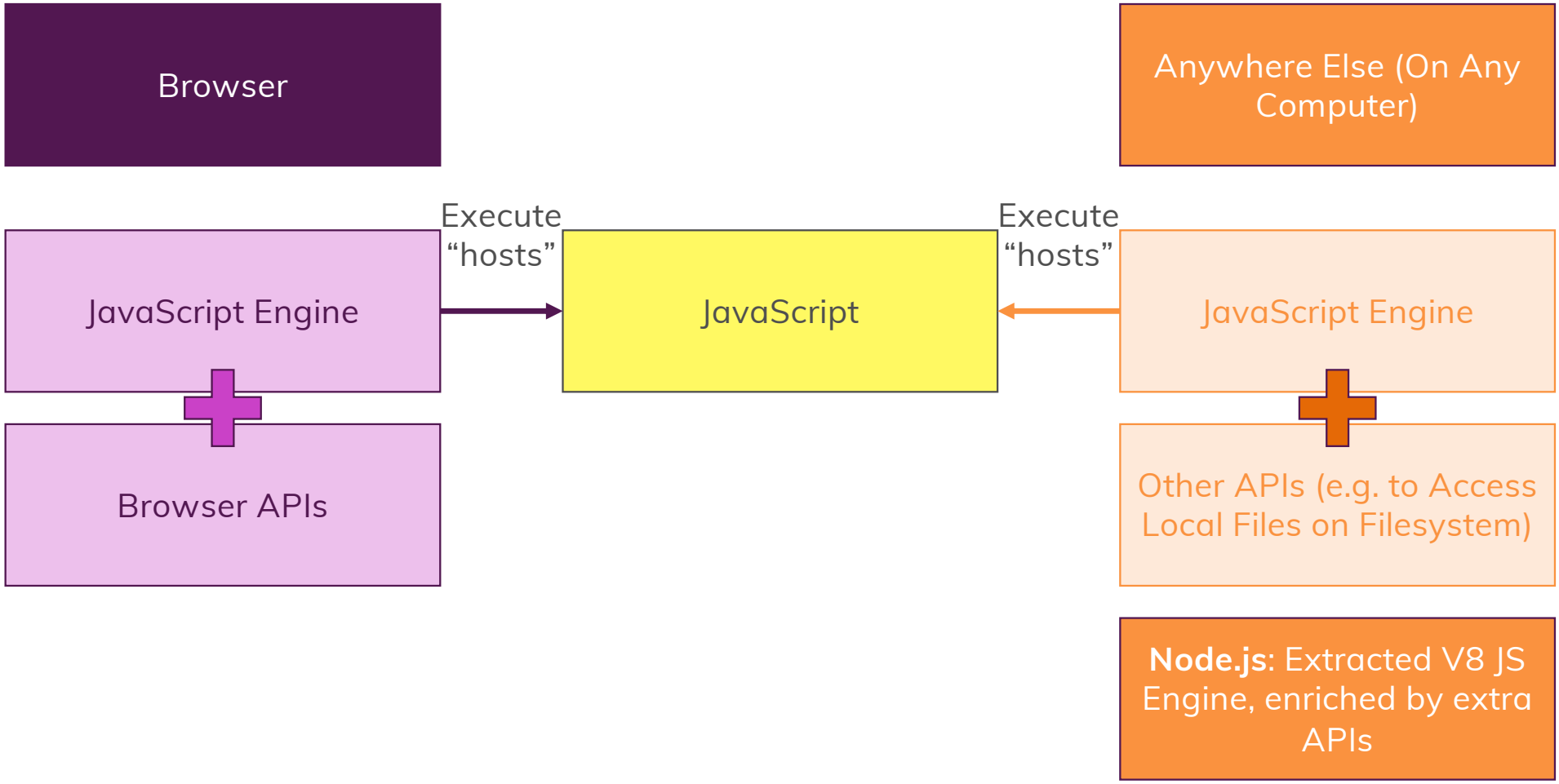
```
export async function getAddressFromCoords(coords) {
return '6th Avenue'; // return any dummy address you want
}
export async function getCoordsFromAddress(address) {
return {lat: 47.01, lng: 33.55}; // return any dummy coordinates you want
}
```

ACADEMIND

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			

JavaScript Is A Hosted Language



NodeJS Error Handling

The finished code of this section will crash if you try to visit **/my-place/:id** with an invalid ID (that can't be converted to the special MongoDB ObjectId type).
You can of course add error handling in a similar way as you learned it before in the course:

- `try-catch` for synchronous code (or `async` / `await`)
- `then()` / `catch()` for promise-based code (where you don't use `async` / `await`)

Example:

```
1.   router.get('/location/:lid', (req, res, next) => {
2.   const locationId = req.params.lid;
3.   client.connect(function(err, client) {
4.   const db = client.db('locations');
5.   // Insert a single document
6.   db.collection('user-locations').findOne(
7.   {
          a. _id: new mongodb.ObjectId(locationId)
8.   },
9.   function(err, doc) {
          a. // if (err) {}
          b. if (!doc) {
          c. return res.status(404).json({ message: 'Not found!' });
          d. }
          e. res.json({ address: doc.address, coordinates: doc.coords });
10.  }
11.  );
12.  });
13.  });
```

can be improved like this:

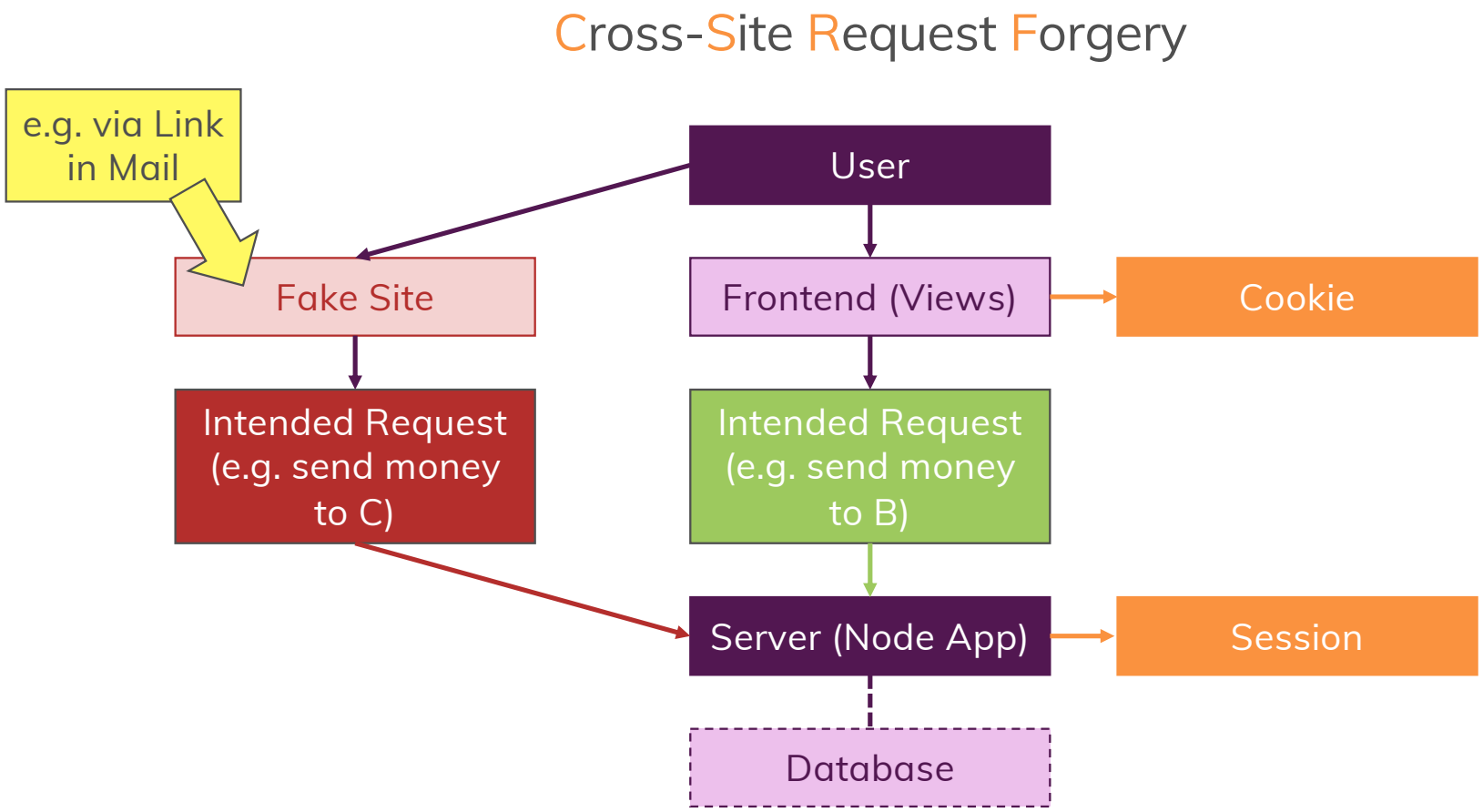
```
1.   router.get('/location/:lid', (req, res, next) => {
2.   const locationId = req.params.lid;
3.   client.connect(function(err, client) {
4.   const db = client.db('locations');
5.   // THIS WAS ADDED
6.   let locationId;
7.   try {
          a. locationId = new mongodb.ObjectId(locationId);
8.   } catch (error) {
          a. // return to make sure the other code does not execute
          b. return res.status(500).json({message: 'Invalid id!'});
9.   }
10.  // END OF ADDED CODE
11.  // Insert a single document
12.  db.collection('user-locations').findOne(
13.  {
          a. _id: locationId // will only be reached if the above code didn't throw an error
14.  },
15.  function(err, doc) {
          a. // if (err) {}
          b. if (!doc) {
          c. return res.status(404).json({ message: 'Not found!' });
          d. }
          e. res.json({ address: doc.address, coordinates: doc.coords });
16.  }
17.  );
18.  });
19.  });
```

I dive deeper into more generic NodeJS error handling in my "NodeJS - The Complete Guide" course but the important takeaway here is, that you have the same error handling tools as in browser-side JavaScript.

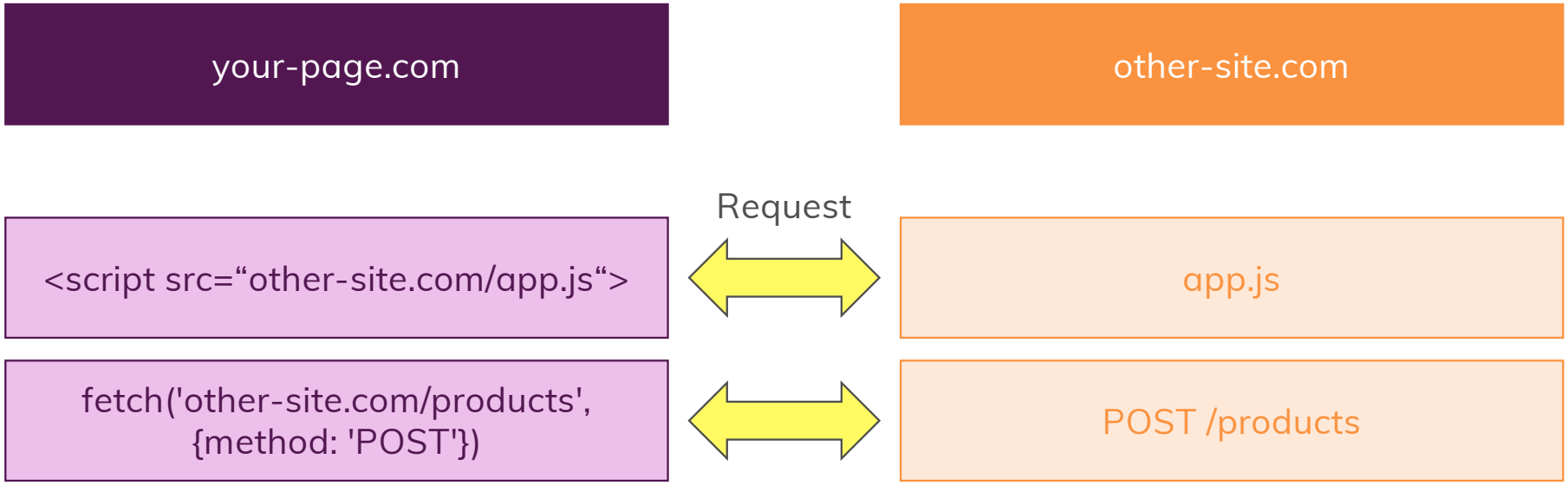
What Could Go Wrong?

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)



By default, only requests to same origin (domain) are allowed!

Different Kinds of Applications / Websites

Only HTML + JS + CSS

A **static host** (doesn't execute server-side code) suffices!

Server needs to host + provide HTML, JS and CSS assets

Example services: AWS S3, Firebase Hosting,...

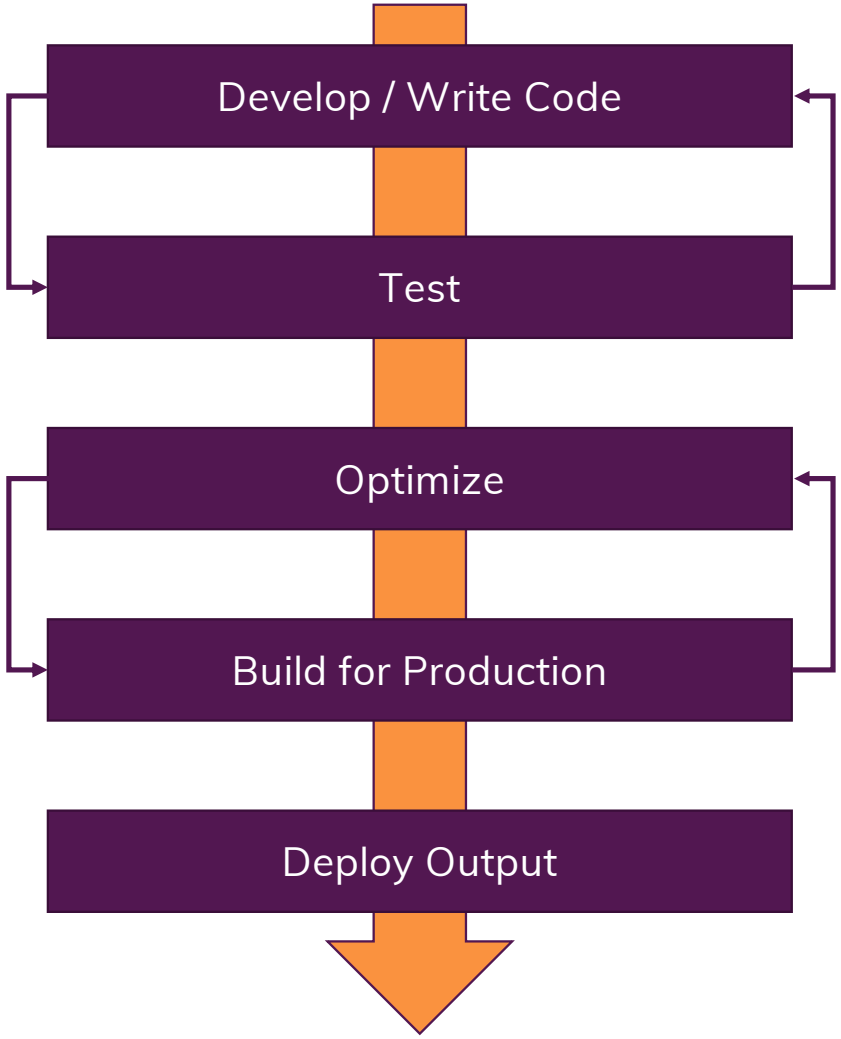
Server-side Code (Node.js)

Need **dynamic host** (able to execute server-side Node.js)

Server needs to host + execute files (server-side)

Example services: AWS Elastic Beanstalk, Heroku, ...

Deployment Steps



Different Types of Websites

You can build different kinds of websites/ web apps.

Specifically, there are three major types you can differentiate:

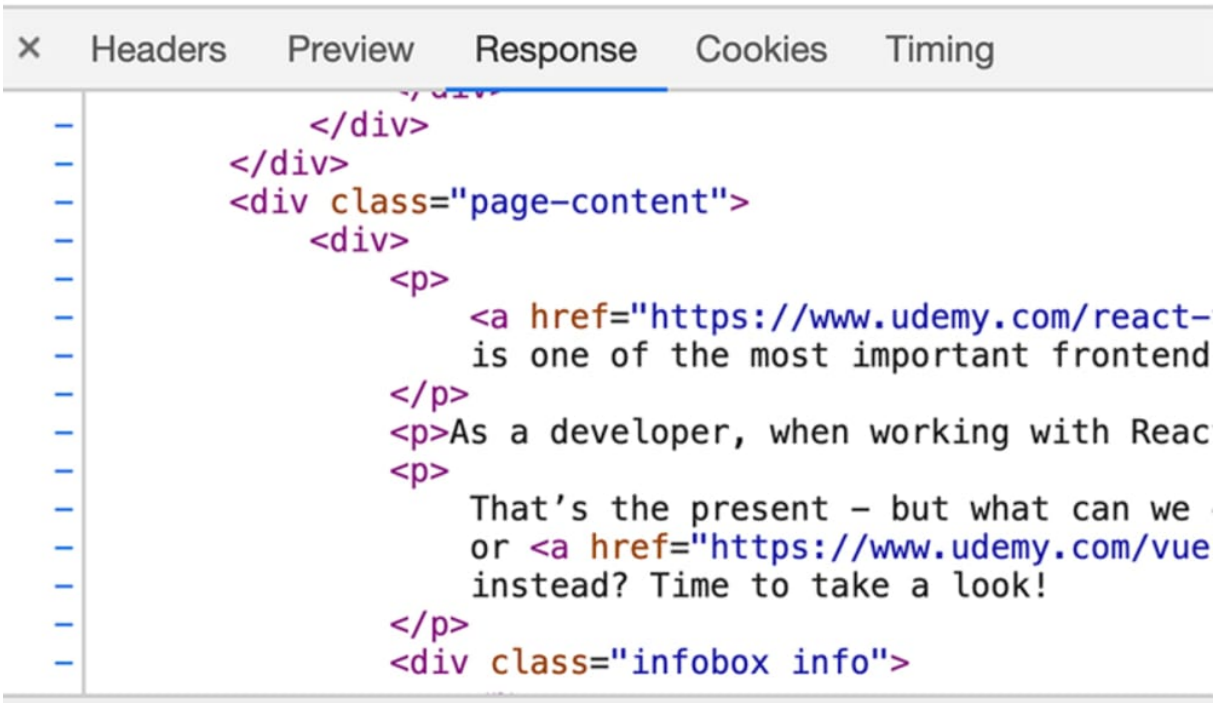
- Static Websites (just HTML + CSS + JS)
- Single-Page-Applications (SPAs, HTML + CSS + JS with only one HTML page being served, client-side JS is used to re-render the page dynamically)
- Dynamic/ Server-side rendered Web Applications: Websites where the HTML pages are created dynamically on the server (e.g. via templating engines like EJS).

When deploying such websites, it's important to understand that sites with NO server-side code (i.e. Static Websites and SPAs), required only a static host (e.g. AWS S3, Firebase Hosting).

Pages where HTML is generated dynamically on the server require a host that is capable of executing the server-side code (i.e. a server that supports NodeJS, PHP or whatever language is being used).

Dynamic vs SPA vs Static Websites

You know how the web works - the browser sends a request, the server returns a response. If a webpage was requested, the response contains HTML code which then is used by the browser to render the website.

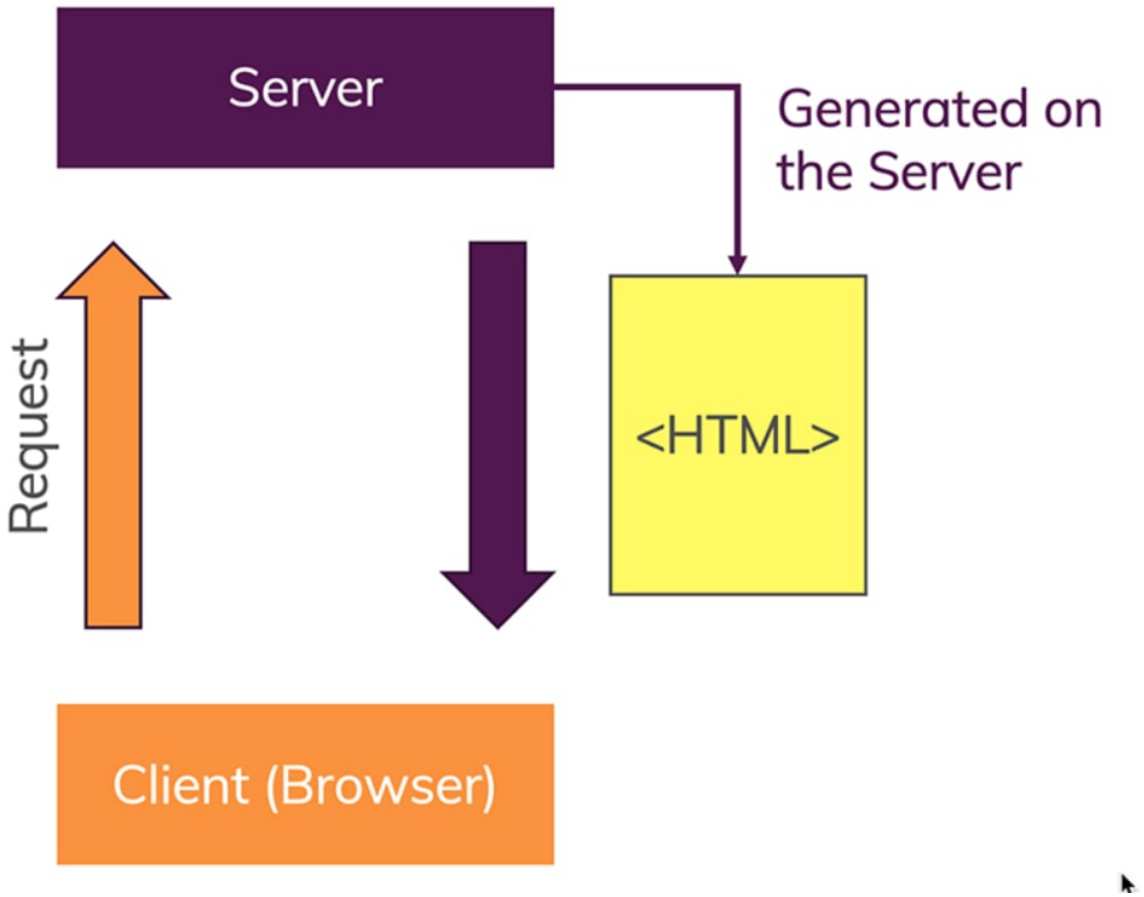


But how does the HTML code, which is returned, end up on the server? Is it generated "on the fly" (per request)? Is it "pre-generated" (i.e. are the finished HTML pages uploaded onto the server)?

It turns out, that there are three main alternatives when it comes to rendering websites (and the HTML code that is required):

1. Dynamic Pages

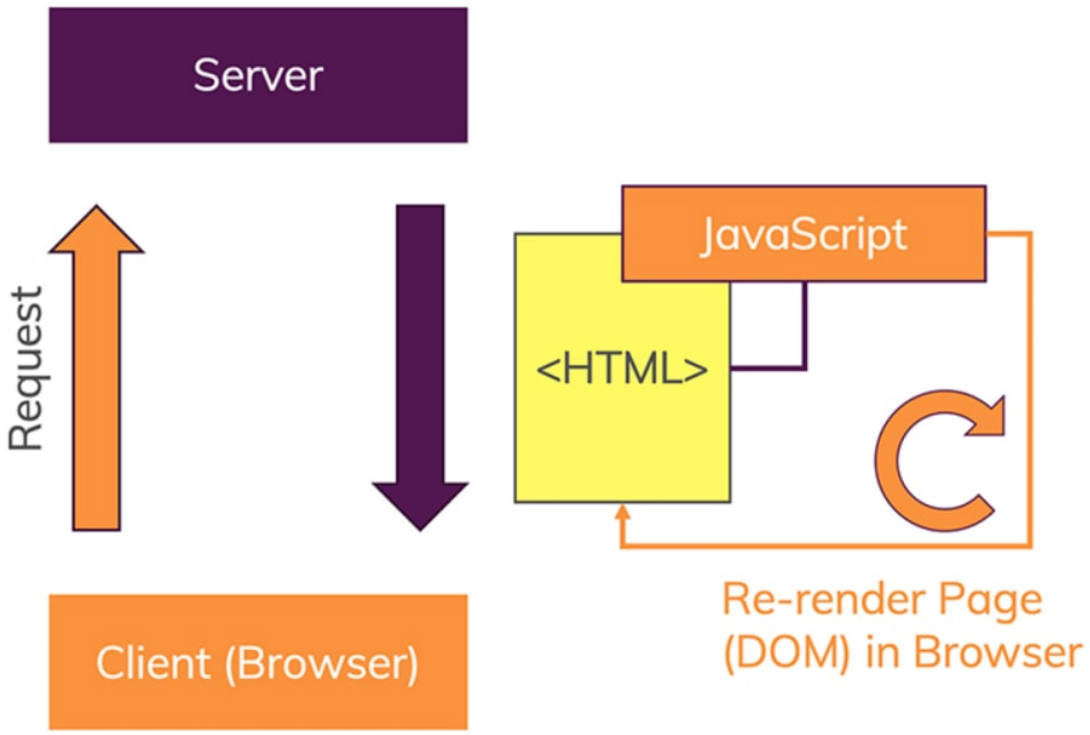
HTML pages are created dynamically, on the server - with the help of a **server-side programming language** and typically also a **"templating engine"**



Learn more

2. Single Page Applications

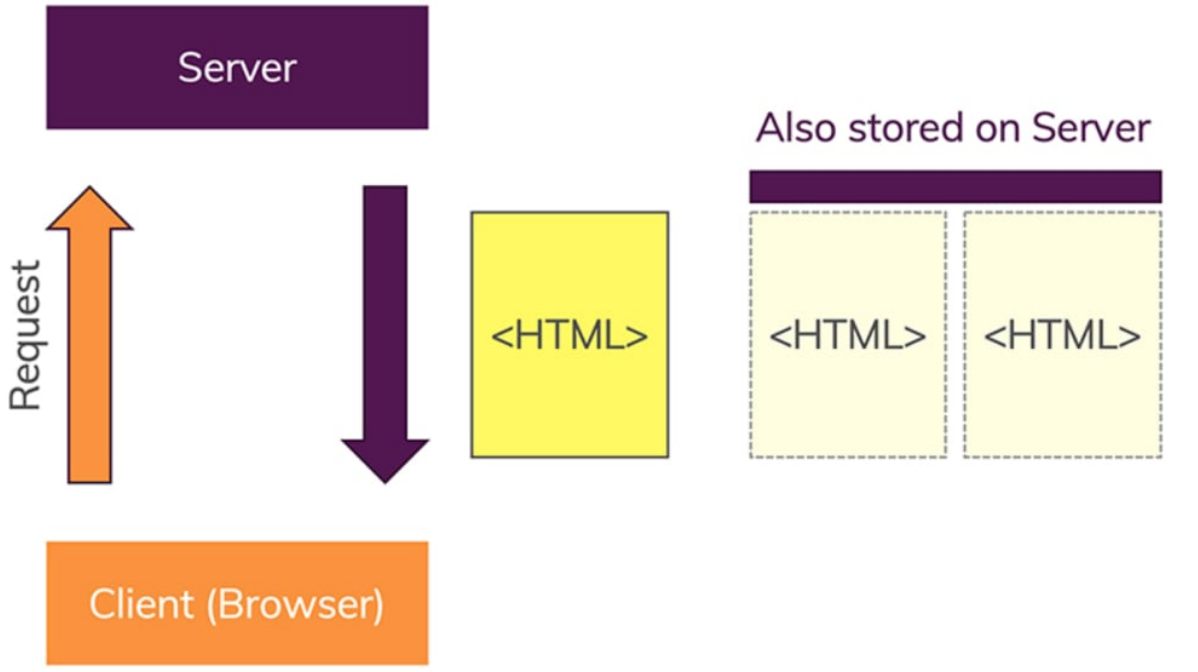
The server returns one single, pre-generated HTML page which in turn contains JavaScript code that changes the page dynamically in the browser (this approach is known as a **“Single Page Application”** or **“SPA”**)



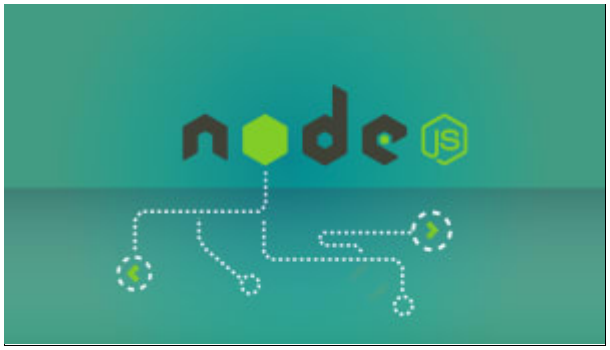
Learn more

3. Static Pages

HTML pages were pre-generated and are stored as such on the server - typically, those pages are not hand-written but also generated automatically (but before they're uploaded) with the help of a **static site generator**

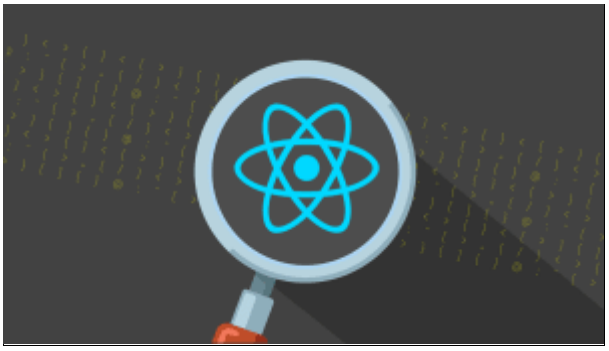


Learn more



Learn Node.js A - Z

Node.js is one of the most important programming languages you can learn right now. Don't miss it in 2019 and beyond!



Dive into React.js

Working on the UI side? React.js is a must-learn in 2019 and the years to come!

Dynamic Pages

Go back in time by ten years and this was the most common approach. It actually still is, but Static Pages and Single Page Applications are now rapidly growing and taking significant shares of the overall “website market”.

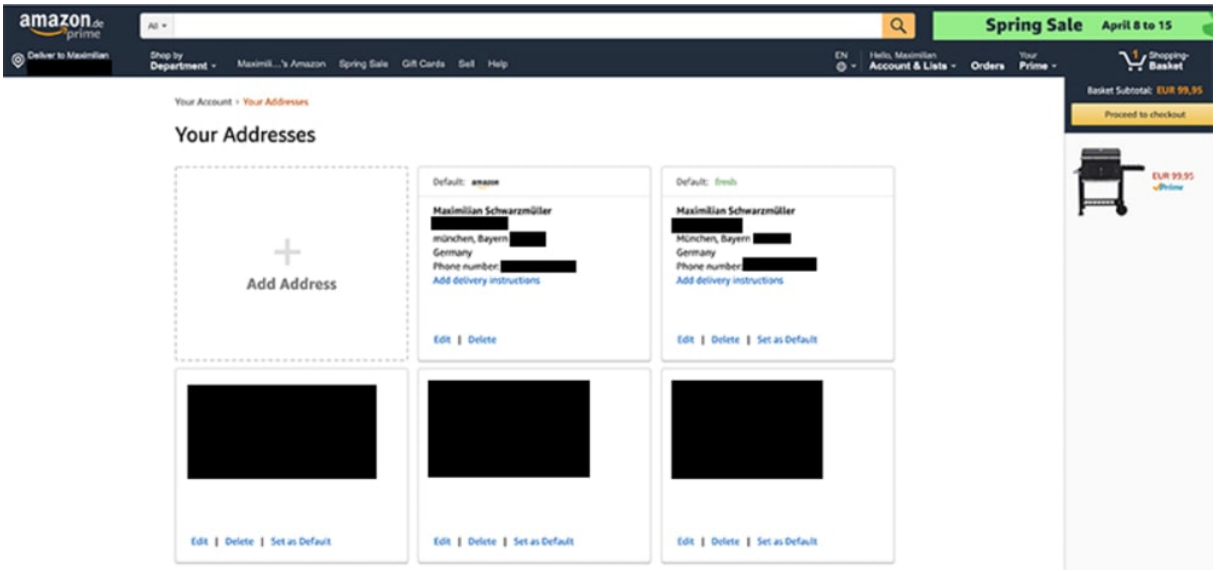
Especially modern websites and webservices (e.g. Netflix, Twitter) are embracing these newer approaches.

But let’s take a step back: What are “Dynamic websites”?

It’s not a reserved term or anything like that. It simply means that the website you’re viewing is dynamically created on the server. And with “website”, I of course mean the HTML code, including the information which styles and scripts to attach.

Consider your user profile on amazon.com.

It includes personal data like your name and order history. Obviously, Amazon does not hand-write the HTML code for that page. Still, if you inspect the HTML source code of that page, you’ll find your personal data in there.



This means that the HTML code was dynamically generated on the server, by the server-side language used there.

The incoming request was handled by the server as instructed by the language used there. It was parsed and required data (e.g. your name) was fetched from a database. The server-side language then used all that information to generate dynamic HTML code which was sent back in the response the browser awaited.

For this, server-side languages like Node.js could be used - typically in conjunction with specific extensions that make the dynamic creation of HTML code easier. These extensions are also referred to as “templating engines”.

In case you’re interested: Our complete Node.js course does not only teach Node from A - Z, it also **includes a section on templating engines** and how to render dynamic web pages.

The **advantage** of this approach is that the client (browser) receives the finished website with all the data that belongs to it. That’s especially important if the client is not a user but a search engine crawler - it always helps if it sees what the user would see (even though, that’s of course not your personal profile page - but the same concept applies to other pages, too).

Another advantage is, that all logic happens on the server, hence the client’s machine really only needs to render the HTML code. A task that typically isn’t too performance-intensive.

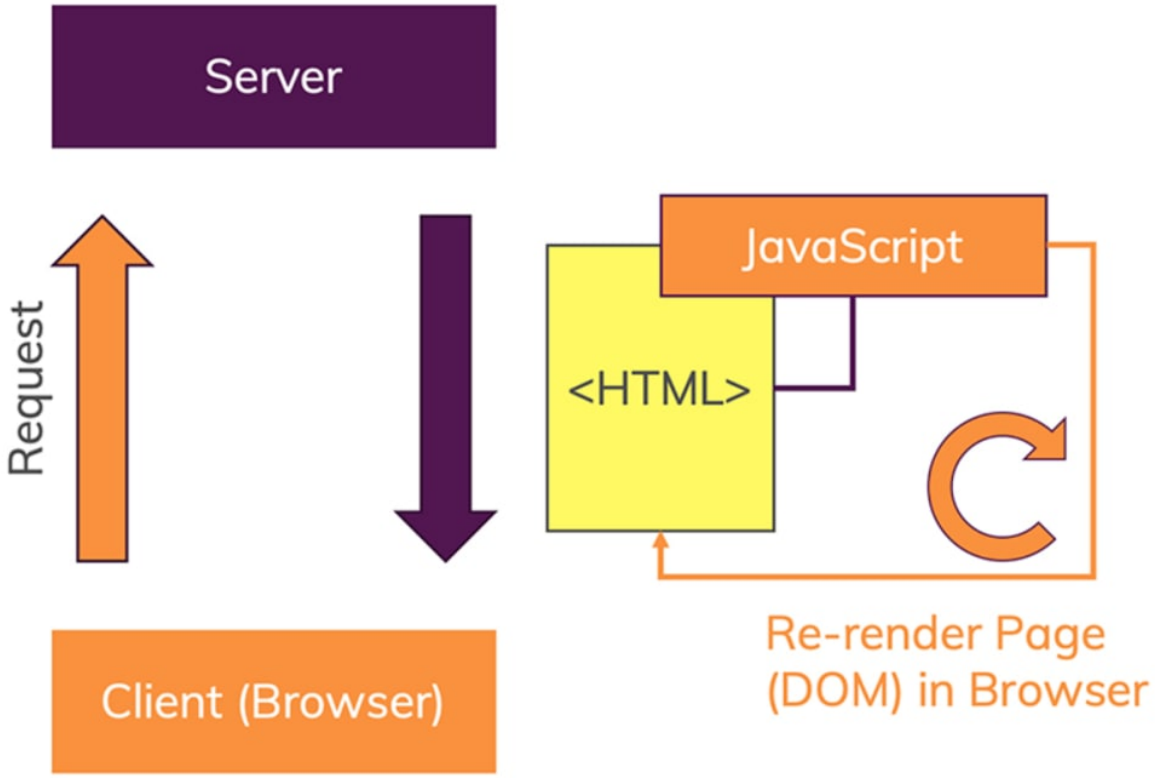
A **disadvantage** of this approach is that **every** page a user visits needs to be generated on the server. That means, that the user always has to wait for a new page to be sent. Modern servers and browsers are pretty good at that (e.g. they use caching to avoid unnecessary requests and roundtrips) but still, you have that extra work to be done. Even if just one detail on the page changes and the frame stays the same (e.g. your navigation bar and the footer), a new page has to be requested and rendered nonetheless.

In addition, **another disadvantage** could be, that developers building dynamic pages need both frontend (HTML + CSS + JS) **and** backend development knowledge. At least to a certain degree. Decoupling and splitting work is certainly possible but there is a slightly higher dependency.

Single Page Applications

Single Page Applications (SPAs) are the other extreme: There, all the HTML generation happens in the browser. The server only returns one basic HTMl page for **all incoming requests** (no matter the URL).

But that single HTML page contains a lot of JavaScript code (typically outsourced into separate files) which is responsible for changing the HTML code (technically, the DOM).



At first, this sounds strange, right?

Didn't the server fetch data from a database to then generate HTML code? Does the client (browser) now do that (connect to a database)?

It's important to understand, that with SPAs, there still is a server involved. Your client-side JavaScript code will **not** connect to a database. That would be highly insecure since access credentials would be exposed. You can't hide your frontend JavaScript code!.

But the entire UI-updating process will happen in the browser (via client-side JavaScript). This provides a mobile-app-like user experience since the user now never has to wait for a new page to load. Instead, updates and changes happen instantly.

Your app will probably still need some data from time to time but you can simply show a loading spinner on the UI until that data has been fetched behind the scenes (e.g. via AJAX).

Since updating the entire HTML page dynamically via client-side JavaScript code requires quite a lot of JavaScript instructions, you typically use a framework or library for the heavy lifting.

React.js, Angular or Vue.js are the most popular choices.

SPA development sounds interesting to you? Besides the three framework courses linked in the previous paragraph, you might also want to check out my detailed comparison of the three alternatives!

SPAs are on the rise because they provide such an amazing and fast user experience.

But handling all the UI updates and the content rendering in the browser has its downsides, too.

The **biggest disadvantage** is that the page you get sent over the wire (i.e. that initial single HTML page that kicks of the SPA) is **pretty empty**. Typically, there's nothing more than some entry HTML tag and a couple of script and style imports.

```
<body>
    <div id="portal"></div>
    <div id="___gatsby">
        <div style="outline:none" tabindex="-1" role="group"></div>
    </div>
    <script>
```

That's **not great for SEO** since the search engine crawler doesn't see all your beautiful content which will eventually be rendered - it just sees the empty page instead. Some crawlers may actually wait for JavaScript to run and render something but they'll never wait for asynchronously loaded data (i.e. for content that first has to be fetched via a behind-the-scenes HTTP request).

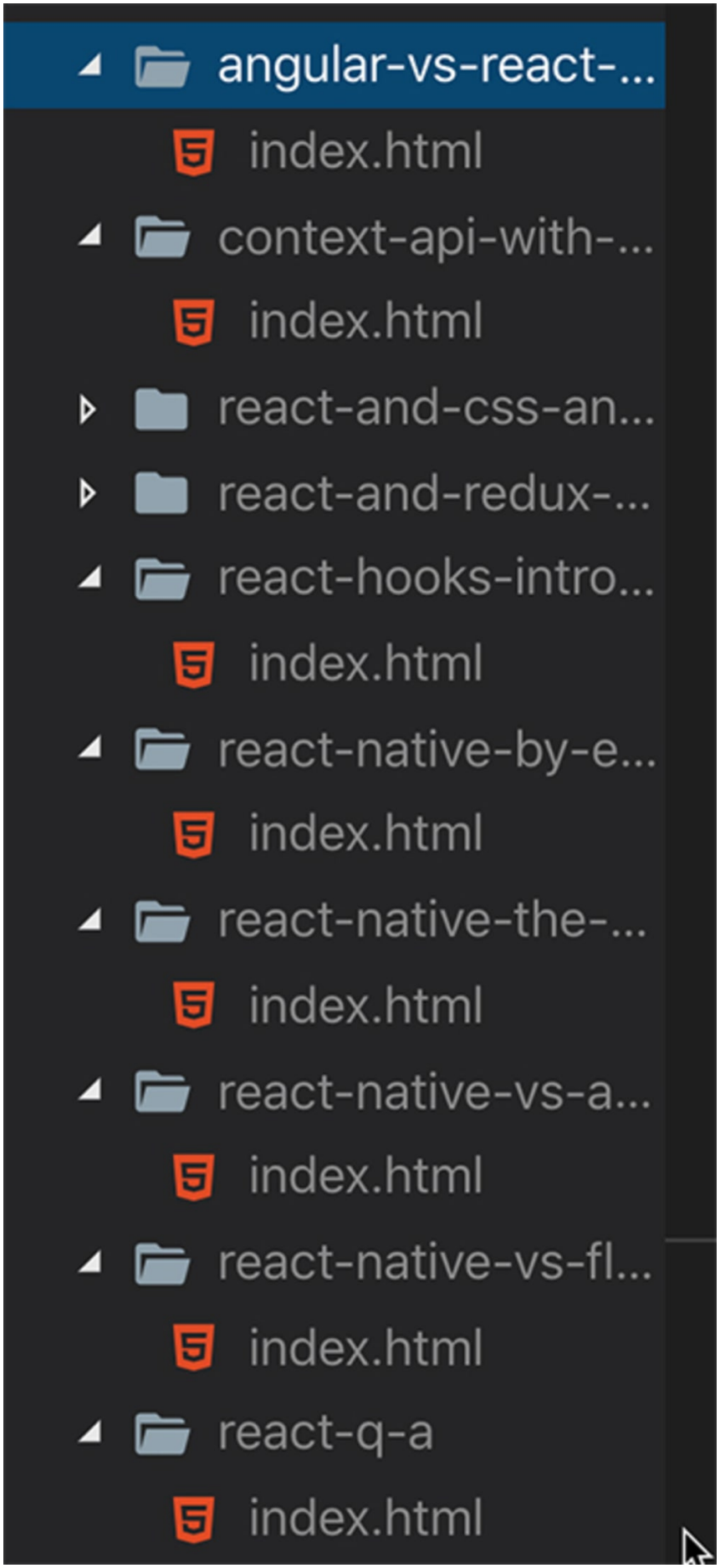
There are solutions to that problem - server-side rendering of SPA views for example. But this already sounds a bit more advanced - and indeed, it is. Thankfully, there are initiatives that make that process a bit easier and hence allow you to get the best of both worlds.

Besides the SEO "problem", it's also worth mentioning that more work in the browser also means that **more JavaScript code has to be downloaded**. And all that code has to be parsed + executed, which of course impacts the runtime performance of the website.

On slower devices and/ or slower networks, SPAs can therefore take very long to load or feel "laggy".

Static Pages

Static pages are the oldest form of web pages. After all, a static webpage simply means that all the HTML content has been created already. The finished pages are then uploaded onto a server and are waiting for requests to fetch them.



If you only need a very basic website where content doesn’t change often and where you also have no dynamic (e.g. user-specific) content, such a static page could be all you need.

But now comes the interesting part: Even pages **with dynamic content** can be converted into static pages. This page (academind.com) is an example - it's actually a static page.

How does that work? Do we write all the HTML code for all our pages manually? Do we copy and paste code from page to page?

No, that would not be feasible at all - leave alone that it would be extremely error-prone.

Instead, we use a static site generator, to be precise, we use Gatsby.js.

How does that work?

A static site generator allows you to define content in a simplified form - often as a markdown document.

For example, this is what a draft of the article you're currently reading looked like:

```
No, that would not be feasible at all - leave alone that it would be extremely error-prone.

Instead, we use a [static site generator](https://www.staticgen.com/), to be precise, we use [Gatsby.js]
/gatsbyjs.org).

## How does that work?

A static site generator allows you to define content in a simplified form - often as a [markdown documen
(https://en.wikipedia.org/wiki/Markdown).

For example, this is how a draft of the article you're currently reading looked like:
```

The static site generator then picks up that content and converts it into HTML + CSS code as instructed by you.

Every generator works differently, Gatsby.js actually uses a React app that it basically builds locally and where it then "visits" every page that would exist in a SPA. Snapshots of these pages then get stored as HTML files - and that's the final output. A bunch of HTML files as they would normally be generated by React in the browser.

We then simply upload these HTML files to our server.

Why all that hassle?

Compared to a normal SPA, the static page approach has a major advantage: The final page is shipped over the wire when requested. No JavaScript code needs to run to draw it onto the screen. Search engine crawlers therefore see what users see. And users don't need to wait for some initial JavaScript code to be parsed and executed.

But the best thing is: Some static pages (including this one), turn into a SPA once loaded.

Hence you get all the benefits of a SPA (fast updates, instant changes) as soon as the user has loaded the first page. But for that first page, you get the advantages of a static or dynamic web page. Pretty sweet! :)

Important: Not every static site generator gives you a SPA as well.

In addition, not every static page is created with the help of a static site generator. You can also write all HTML pages on your own and simply upload them. For small and static pages, this might be more than enough!

What's the Best Choice?

So … the static page (created with a static site generator that also yields a SPA) is best, right?

No, it’s not that easy!

There is no best or worst approach!

All approaches can be fine - it depends on what you’re building.

A **static page** - no matter if created with or without a static site generator - might be a poor choice for a website with content that changes multiple times per hour or minute. Even if the process is highly automated, the page would have to be re-generated and re-deployed constantly! A dynamic page or SPA might be a better choice here!

A **SPA** in general often is a decent choice - it’s especially great for web apps that don’t really need search engine crawling.

You could always create your landing pages (that should be found) in a SEO-friendly way (e.g. static pages) and create the rest of the app as a SPA.

The great reactivity and UX makes SPAs a great choice for in-browser apps like Google Docs, the Twitter feed or Netflix.

A **dynamic page** on the other hand can be perfect if you have fast-changing content and SEO matters a lot. An online shop for example.

As always in life, it doesn’t hurt to know all these alternatives - that allows you to pick the right tool for the job you’re working on.

Injecting Script Imports Into HTML Automatically

In the previous lecture, we manually adjusted the HTML files to import the generated JavaScript files.

For most projects, this is fine - you're probably not going to push out a new version of your scripts every few minutes.

But you could also automate this process if you wanted to - with the help of a special plugin for Webpack: The HtmlWebpackPlugin.
You can find a detailed documentation on how to use it on this page: https://github.com/jantimon/html-webpack-plugin

Git Tutorial for Beginners - Crash Course

Git is a **version control system** and allows you to manage your source code history efficiently. Whenever you work on your code, changes can be saved with Git and you can jump back to any previously saved versions. Without tools like Git, you would have to create manual copies of your code, which would be quite cumbersome and impossible to maintain once your application grows.

Just looking for the most important Git commands? Click here to jump to the end of the article!

CSS - The Complete Guide

Join this comprehensive 20h+ course to master CSS and learn how to create beautiful websites.

JavaScript - The Complete Guide

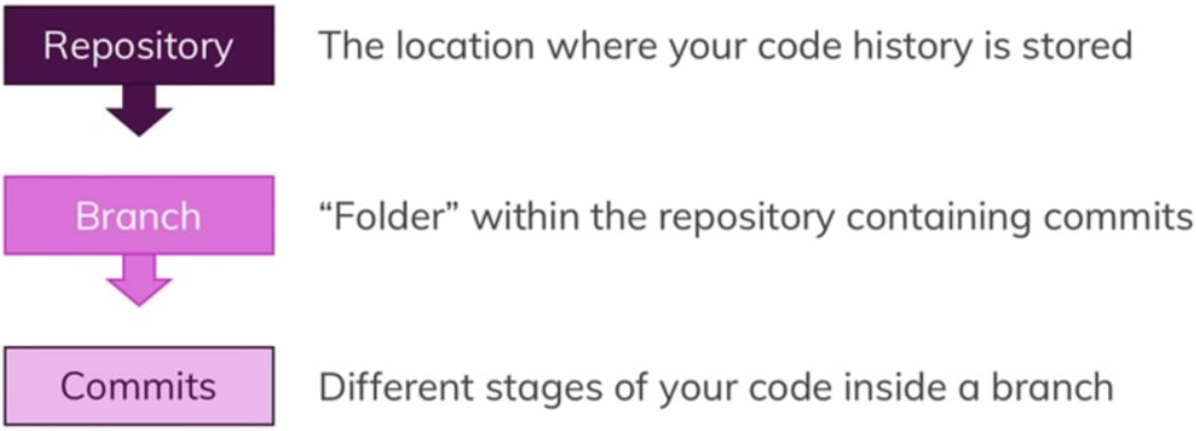
Learn JavaScript from scratch to build highly interactive and dynamic websites in this hands-on course!

Git vs Github

These terms can be confusing, especially for beginners. Git is a **version control system** which you can download to your computer, GitHub is a **hosting & collaboration provider** which hosts your Git projects. It is a web based solution allowing you to upload your Git Repositories (more on these in the next chapter) to it.

Uploading your code from your local Git environment to GitHub makes your code available to anybody who wants to contribute to your project. In this article we'll focus on Git though, if you want to dive deeper into GitHub, we also got you covered.

Repositories, Branches & Commits



A **Repository** is the location where your code is stored, so a folder on your machine containing your project code. After turning this folder into a Git Repository (we'll learn how this works soon), Git manages the project code version history.

The code is not directly stored in the Repository though. Inside our Repository we have “subfolders”, so-called **Branches**. After we added the first code to our Repository, a default “master” Branch is created. We are not limited to one Branch though, a Repository typically contains multiple Branches. So these are Repositories and Branches, but where is our code stored?

Inside our Branches! A Branch contains different versions of our code, our **Commits**. Each Commit is a snapshot of a specific version of the code.

These are the basic terms in short words, let's dive into Git now to see this in practice.

How to Install & Use Git

Git is for free and can be downloaded here.

Git runs in the terminal (MacOS) or the Command Prompt (Windows) only, so it doesn't come with a Graphical User Interface (GUI). Make sure to dive into dedicated resources on that in case these terms are totally new to you.

If you're working with an IDE, you can also use the integrated terminal. I will use Visual Studio Code in this video/ article.

You prefer videos? Don't miss the **Full Git Crash Course** video at the top of this page!

The Main Commands

Let's create a new project, a folder which contains an “index.html” file for example, and add some basic code to it:

```
<p>Hello</p>
<p>Some text</p>
```

Are you totally new to web development? Make sure to have a look at our How The Web Works video to easily get started!

git init

Let's write our first Git command. `git init` will turn our project into a project managed by Git. You should see the **Initialized empty Git Repository** info in your terminal, confirming that Git now monitors the project. But why is our project empty?

git status

`git status` provides information about the current state. We have untracked files, meaning that Git is aware of the files being located in our folder but we need to explicitly tell Git to track these files (i.e. to check if the files changed) and to save any changes in a new Commit.

git add

Tracking files either works with `git add filename` (“filename” should be the name of the file you want to track) or `git add .` which will track changes in all files inside the Repository. With `git status` we can see that we added a new file (“index.html” in our case) to the Repository, we didn't save any changes on the code up to this point though.

git commit

As a last step we need to commit or save the current code version in our Repository. `git commit -m` adds the latest version of the code to our Branch, `-m` should provide a meaningful Commit message. In our case we could use `git commit -m "added starting code"`.

Diving Deeper into Branches and Commits

Time for a quick summary before we dive deeper:

1) We turned our project folder into a Git Repository with `git init` 2) We told Git to track any changes in the `index.html` file with `git add .` 3) We saved the current version of our code in a commit with `git commit -m "added starting code"`

That was simple wasn't it? But what about Branches? I told you before that we automatically created our master Branch with our first Commit, but where can we see this Branch?

git branch

`git branch` lists all Branches in our Repository. Although we didn't add any name, the master Branch was automatically created with our first Commit:

```
Manuels-MBP:project LorenzM$ git branch
* master
```

We have one Branch only at the moment, the asterisk indicates that we are also currently working in this Branch.

Let's add more code to our index.html file:

```
<p>Hello</p>
<p>Some text</p>
<div>Another text</div>
```

With `git add .` and `git commit -m "div added"` we can add this new code as a second Commit to our Branch.

##git log

Entering `git log` in our terminal will display all Commits inside our Branch:

```
commit 114ca9893e7005fead7060bfd386225a97141d91 (HEAD -> master)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 11:05:32 2018 +0200

    div added

commit d1a1a697cd8778997af26040422515413ebdaa0a
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
```

Each Commit contains a unique ID, the author, the date and the Commit message (this `-m "your message"` part). "HEAD" just points to the latest Commit of our current Branch, more on that in the next chapter.

Scrolling down a bit will also show us our initial Commit named "starting code".

But how does Git create these Commits? Each Commit is not an updated copy of the previous Commit. After the first Commit, Git just tracks changes for each following Commit. So with `git add .`, Git checks our files for any changes, `git commit` saves these changes in a new Commit then.

Tracking changes like that makes Git fast and efficient as creating copies over and over again would simply bloat our Git folder, definitely something we want to avoid in any project!

#The HEAD

Time to have a closer look at the HEAD.

```
<p>Hello</p>
<p>Some text</p>
<div>Another text</div>

<div>Testing the head</div>
```

I added another "div" and created a new Commit (`git add .` and `git Commit -m "head testing"`), as explained earlier, this Commit became the HEAD as it is the latest Commit in our Branch:

```
commit 245865f7eec1590dafe3ae5cd59a7abd5078da21 (HEAD -> master)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 13:31:05 2018 +0200

    head testing

commit 114ca9893e7005fead7060bfd386225a97141d91
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
```

The latest Commit might not always be the one we want to work on though, we could either just want to have a quick look at a previous Commit or we might want to go back to an earlier Commit and delete later Commits.

##git checkout

`git checkout` is the required command here as it allows us to select a specific Branch (more on that later) or a specific Commit as required in our case.

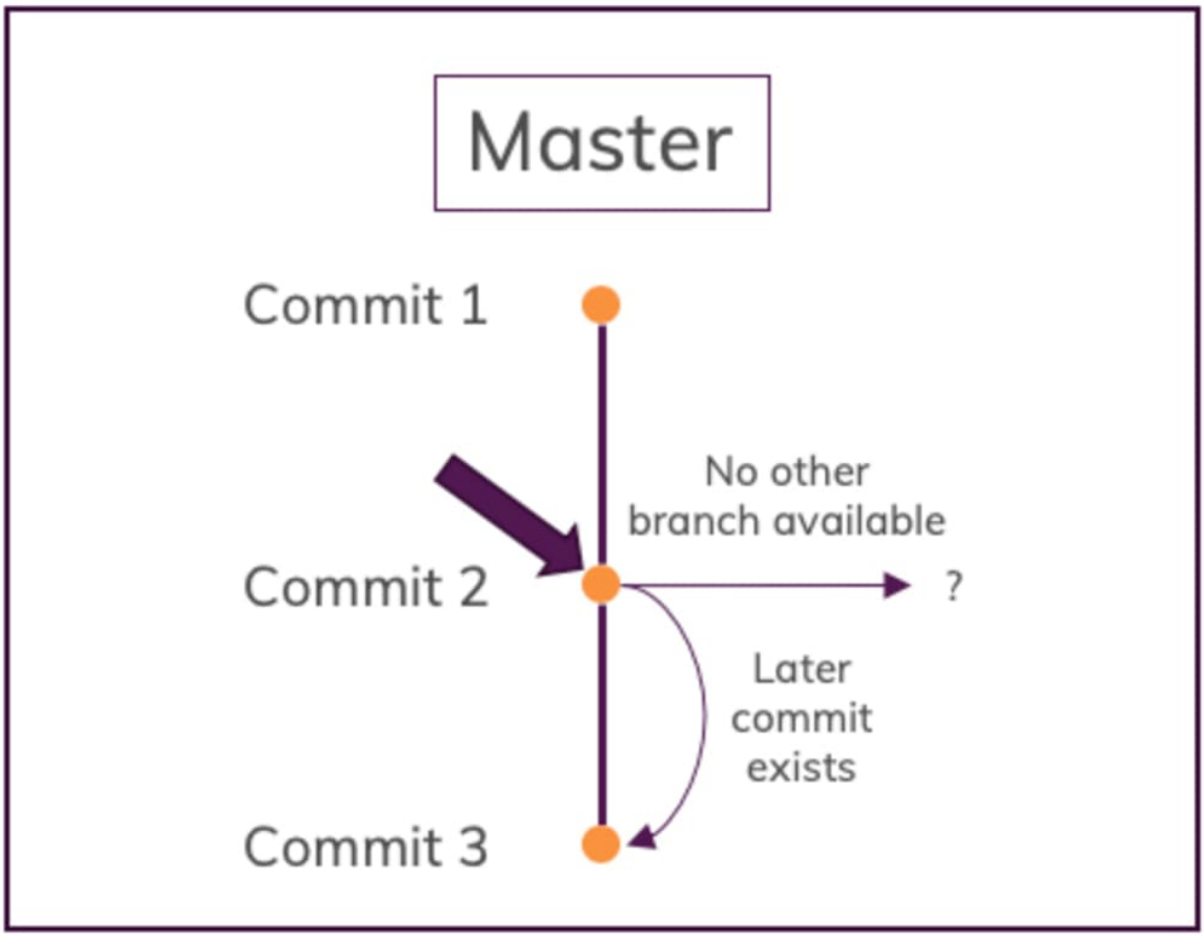
Here we can use the ID we previously had a look at with **`git log`**:

```
commit 114ca9893e7005fead7060bfd386225a97141d91
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 11:05:32 2018 +0200
```

Let's checkout the second Commit named "div added" with **`git checkout yourcommitsID`** (just copy and paste the Commit's ID to the "yourcommitsID" part).

Our HEAD is now pointing towards this Commit, but the Commit we just checked out is no longer part of our master Branch. In simple words it's kind of in the middle of nowhere at this stage, it doesn't belong to any Branch and therefore this approach just allows us to jump back and have quick look at the previous code.

If you really want to go back to this Commit, and therefore get rid of the later Commits, just checking it out like this won't work.



We first have to go back to our master Branch with `git checkout master` (or simply `git master`), now we're back in the latest Commit of the Branch (our HEAD).

Deleting our latest Commit is easy now. Copy the ID of the Commit you want to jump back to (NOT the ID of the Commit you want to delete) and use `git reset --hard ID` (ID should be the ID of the Commit). That's it, we now reset the HEAD and deleted all later Commits.

But be careful! After deleting the Commit we cannot undo this change, so think twice before using this command.

Reverting Unstaged Changes

Let's again add some code:

```
<p>Hello</p>
<p>Some text</p>
<div>Another text</div>

<div>Something I don't need</div>
```

Turns out that we actually don't need that code and we would like to delete it - the changes were not commited though. Besides manually deleting our code (which won't work that easily if you changed a lot of code in different places), `git checkout -- .` let's us jump back to our last Commit (which doesn't include these uncomitted changes of course).

Awesome, these were the basics about Commits, time to move on to Branches now.

Working with Branches

Up to this point we only worked in the master Branch. In a real world project, you typically have multiple Branches, for example:

- The master Branch which contains your currently stable and running version of your website
- The feature Branch where you're currently working on a new feature

The final goal would be to merge both Branches as soon as the new feature is finished and ready to go live.

git checkout -b

`git checkout -b` creates a new Branch based on the latest Commit of the Branch you're currently working in. As we are in the latest Commit in the master Branch which contains the "index.html" file, the new Branch created will therefore also include that code and all Commits inside that Branch - so we basically copy the entire Branch.

`git checkout -b new-feature` will create that new Branch named "new-feature":

```
Manuels-MBP:project LorenzM$ git checkout -b new-feature
Switched to a new branch 'new-feature'
Manuels-MBP:project LorenzM$ git branch
  master
* new-feature
```

After creating the Branch we're automatically working in the latest Commit of this Branch (you can always check that with `git branch` and `git log`) and we can add code to it, just a `styles.css` file to keep it simple.

`git add .` and `git commit -m "css added"` will create a new Commit with these changes in our "new-feature" Branch.

This also results in two different HEADS. In the "new-feature" Branch the Commit we just created is the HEAD, in the master Branch we have a different HEAD, our "div added" Commit - the second one on the screenshot:

```
commit 1b9a14add29f19bfe73505144daa4704fd0d933a (HEAD -> new-feature)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
Date:   Wed May 30 13:56:38 2018 +0200

    css added

commit 114ca9893e7005fead7060bfd386225a97141d91 (master)
Author: Manuel Lorenz <LorenzM@Manuels-MBP.fritz.box>
```

Merging Branches

Back in our master Branch with `git master`, we can now combine or merge our two Branches with `git merge new-feature`.

This will add the changes made in the "new-feature" Branch to our master Branch and therefore our last "css added" Commit is now the HEAD of both Branches.

Deleting Git Branches

As our new feature is implemented into the master Branch, we can get rid of the "new-feature" Branch with `git branch -D new-feature`.

Solving Merge Conflicts

Merging Branches is a great feature, but sometimes it can also cause conflicts.

We currently have this code in the latest Commit in our master Branch:

```
<p>Hello</p>
<p>Some text</p>
<div>Another text</div>
```

Let's create a new Branch with `git checkout -b new-feature` and let's change the code as follows:

```
<p>GOODBYE</p>
<p>Some text</p>
<div>Another text</div>
```

After adding and commiting the changes, let's switch back to the master and change the code like this:

```
<p>BYE</p>
<p>Some text</p>
<div>Another text</div>
```

If we add and commit this change too, we worked on the same code in two different Branches (the first `<p>`), so what happens if we try to merge the Branches now?

`git merge new-feature` will give us an error: "Automatic merge failed; fix conflicts and then commit the result".

The conflict can be solved by deleting the code you do not want to keep. After that, commit the changes as learned before, this resolves the conflict and merges the Branches successfully.

But again: Be careful whenever you delete code, Branches or Commits, after deleting information it is gone so you should know what you're doing here.

Useful Git Commands

Quickly need to find a Git command? Here is your little helper:

- **`git init`**: Initialize a Git repository in the current folder
- **`git status`**: Show the current status of your Git repository (the "working tree" status)
- **`git add .`**: Track changes of all files in your Repository
- **`git commit -m "your message"`**: Save updated code to a new Commit named "your message"
- **`git log`**: List all Commits inside your Branch
- **`git checkout branch-name`**: Jump to the last Commit of a Branch
- **`git checkout commitid`** Jump to a specific Commit of a Branch (commitid should be the ID of the Commit you want to checkout)
- **`git checkout -- .`**: Jump back to the last Commit and remove any untracked changes
- **`get reset --hard`**: Turn the selected Commit into the new HEAD
- **`git branch`**: List all Branches inside your Repository
- **`git checkout -b branch-name`**: Create a new Branch named branch-name
- **`git merge branch-name`**: Merge two Branches, branch-name is the Branch you merge with the Branch you're currently working in
- **`git branch -D branch-name`**: Delete the Branch named branch-name

What is “Performance”?

Performance can be a lot of things!

Startup time

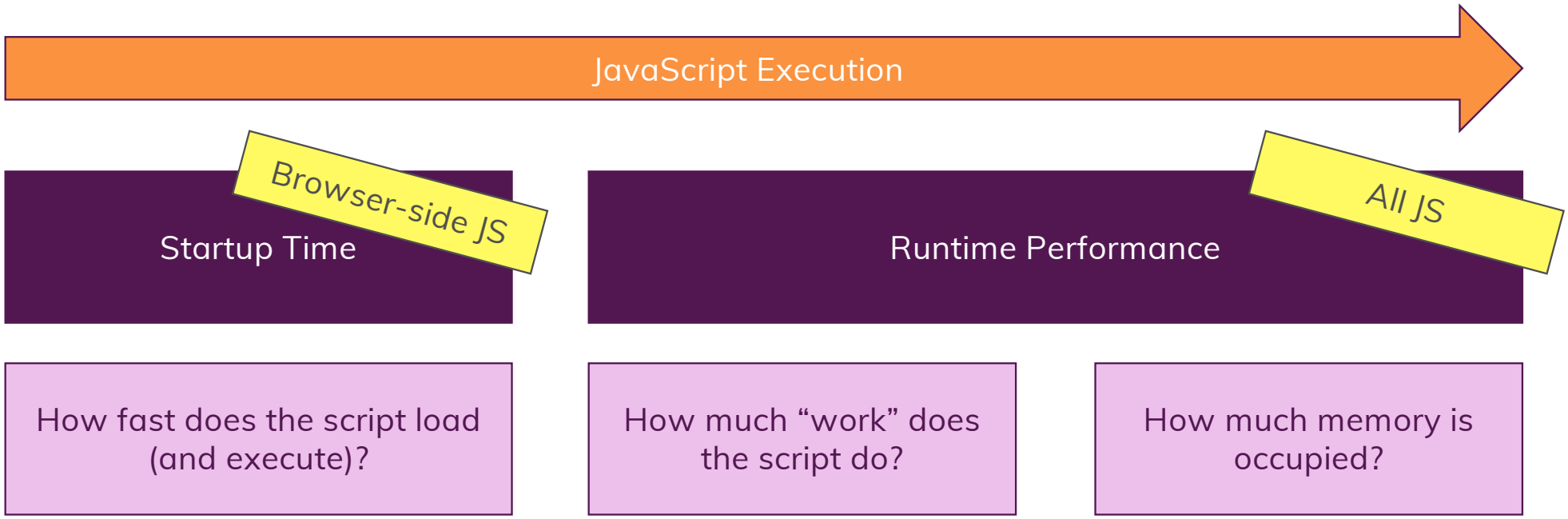
- How long does it take to “see something” on the screen?
- How quickly is a user able to interact with the page?

Runtime performance

- How smooth does the application run (are there freezes or lag)?
- How smooth do animations play, is there any “visual lag”?
- Are there any memory leaks, is the page getting slower over time?

Influenced by a lot of factors! CSS code, HTML code and of course also JavaScript code. Also influenced by server speed/ config.

What Influences Performance?



Different “Layers” of Performance Optimization

Startup Time

JS is not the end!

Runtime

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

Don’t guess!

Measure!

Production Code!

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

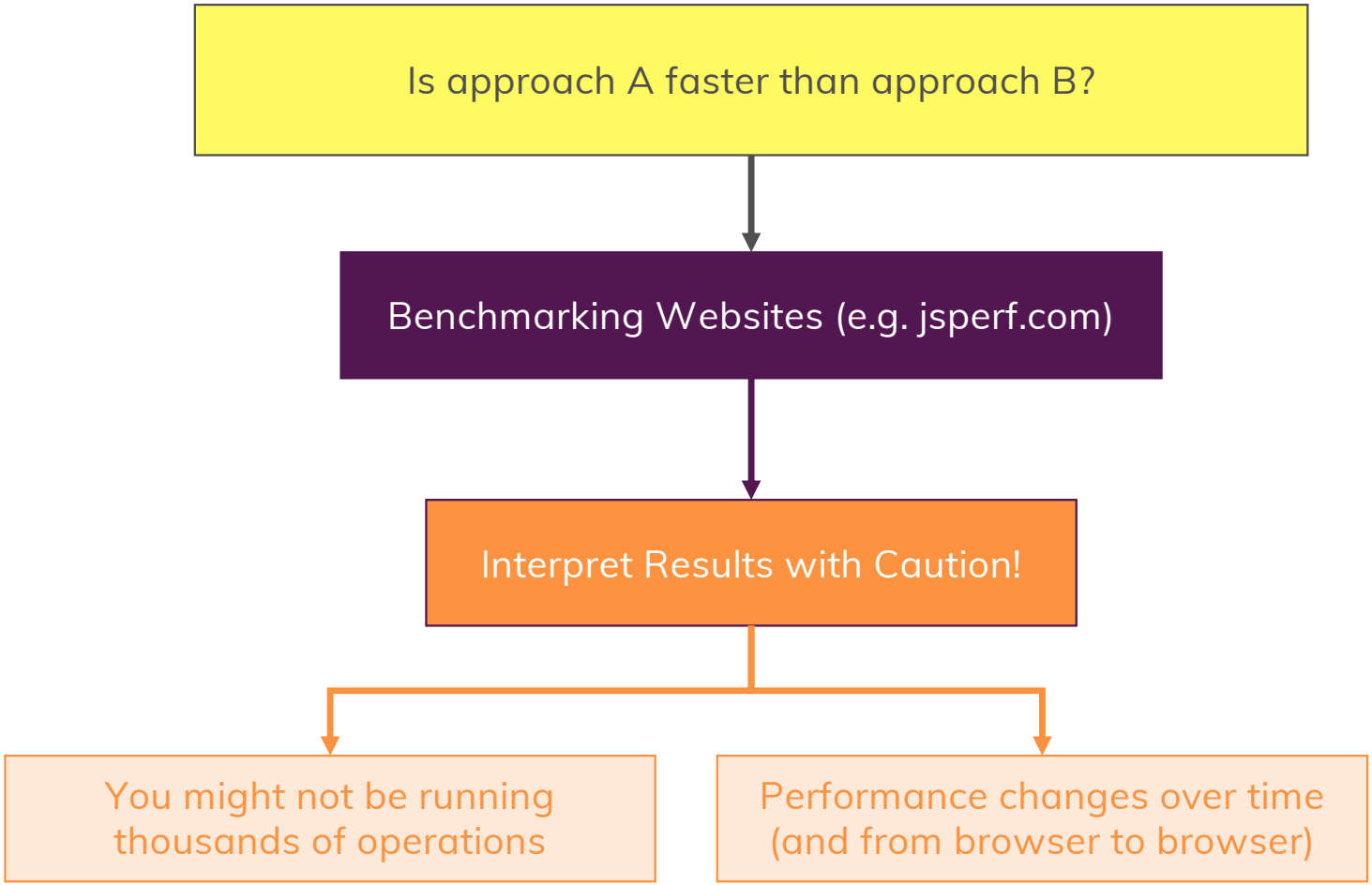
Measure performance with devtools

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



Server-side Performance Optimizations

In this course section we - of course - discussed JavaScript performance.

When it comes to the overall performance of a website, it's not just JavaScript that matters though. Besides other client-side optimizations (e.g. CSS, images
=> see: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency), it's also the server-side where you can improve performance.
Executing code on the server-side (e.g. NodeJS) is one thing but the server configuration is also important.

Specifically, there are three main areas of improvement which you might want to look into:

- Compression of served assets
- Caching (client-side and server-side)
- HTTP/2

Compression

Compression is about zipping static assets (CSS, JS, images) before serving them. Modern browsers know how to unzip such files and will automatically do so. Since zipped assets are transferred, less data is sent from server to client
=> Faster load time.

How you set up compression depends on which server/ service you're using. For example on Firebase, static assets are automatically compressed.

When having your own NodeJS server-side code, you would have to manually ensure that static assets are compressed (https://github.com/expressjs/compression).

Caching

Caching is a complex topic - it's about saving data or files for re-use and it can be done on different levels.

For example, browser automatically cache files (e.g. JS files) for you - based on the caching headers set by the serving host (https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control). So controlling these headers on the server-side config, allows you to control how browsers will cache such files. This can help you avoid unnecessary data transfer but of course you also have to make sure that visitors of your site don't miss out on important updates.
Server-side caching is all about storing data you work with on the server (e.g. fetched from a database) such that multiple requests requesting the same data can get that cached data.

You can learn more about caching here: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching
And here: https://wp-rocket.me/blog/different-types-of-caching/

HTTP/2

HTTP/2 is the latest "form" of the Http protocol and unlike HTTP 1, it allows for "server push". That means that servers can push required assets/ files actively to a client (instead of waiting for the client to request them).

You can learn more about it here: https://developers.google.com/web/fundamentals/performance/http2

JavaScript Testing Introduction

Introduction

Testing JavaScript code - or testing in general - is a hot topic of which a lot of developers shy away from.

You shouldn't!

It was also intimidating to me in the past but it turns out: With modern tooling, testing isn't that hard and it's definitely time to get started!

In this article and the above video, we'll dive into what testing is all about, why you would want to test and which kinds of tests you have. Of course we'll do all that by simply writing some tests!

Don't want to read? Watch the video which you find above the article, at the top of the page!

JavaScript - The Complete Guide

Learn JavaScript A to Z with this >50h completet guide course.

Master ES6 JavaScript

Focus just on ES6 and learn it in detail in a few hours!

What Is Testing?

So what is testing after all? Don't we test all the time when we just click around in our app?

Well, this is testing, you are right. But it's manual testing. We test by clicking around.

To some extent, this is a good thing to do. Actually, you will always be doing that because you want to experience your app on your own.

But writing automated tests - and that is what this article is mainly about - speeds up your development flow and gives you a way of quickly identifying issues, breaking changes and side effects.

```
 PASS  ./util.test.js
   ✓ should output name and age
   ✓ should generate a valid text output

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        0.044s, estimated 1s
Ran all test suites.
```

If you got a bunch of automated tests, you can quickly spot problems when you start working on your code and suddenly one or more of your tests fail.

```
 FAIL  ./util.test.js
   ✓ should output name and age
   × should generate a valid text output (4ms)

   ● should generate a valid text output

     expect(received).toBe(expected) // Object.is equality

     Expected: "Max (29 years old)"
     Received: "Max (30 years old)"

       8 | test('should generate a valid text output', () => {
       9 |   const text = checkAndGenerate('Max', 29);
     > 10 |   expect(text).toBe('Max (29 years old)')
         |                ^
      11 | });
      12 |

       at Object.toBe (util.test.js:10:16)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        0.264s, estimated 1s
Ran all test suites.
```

Automated tests, by the way, are really just code snippets that run your code and then check if the result of that code execution meets a certain expectation. But you'll see all that below.

So to sum it up, "testing" here really just means: Writing automated tests (i.e. code that runs your code) and defining expectations these tests/ code executions should meet.

Why Should You Test?

Now that it's clear WHAT testing is, let's focus on the WHY.

Why would it be a good idea to write automated tests?

The answer is trivial: It speeds up development because you don't have to test everything manually after every change.

Additionally, it's less error-prone. When testing code manually, it's easy to overlook a certain scenario and therefore to overlook a bug.

Of course you can also write bad automated tests, you can forget an important scenario there, too. But over the lifespan of your project development, you'll very likely encounter issues and add respective tests.

Additionally, if you write tests, you're forced to think about your app and potential issues harder. You have to come up with clever tests that will fail if something important changes.

You are also forced to write cleaner, more modular code because the more spaghetti your code becomes, the harder it will be to test.

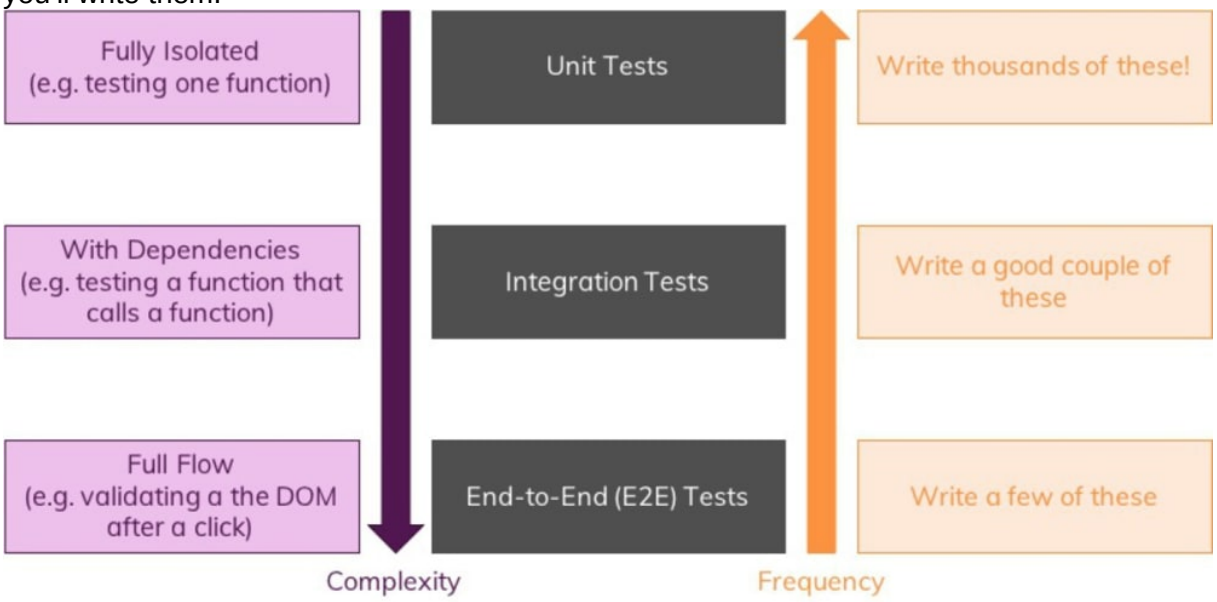
Less work, cleaner code, more thinking about your code - doesn't sound too bad, right?

Unit Tests, Integration Tests & End-to-End Tests

When talking about "tests" or "automated tests" (I'll use these terms interchangeably), we can differentiate between three kind of tests:

- **Unit tests** that test one isolated unit/ piece of code (e.g. a function)
- **Integration tests** which test the combination of features (e.g. a function calling another function)
- **End-to-End (e2e) or UI tests** which test a full interaction path in your app (e.g. the signup process)

These kind of tests have a different level of complexity to write them and a different frequency with which you'll write them.



Unit tests

Unit tests are the easiest tests to write because you have some input and can expect some result. There are no dependencies, no complex interactions.

Here's an example:

Consider this function which we use in our app - it'll take `name` and `age` as an input and return some text that contains these two parameters.

```
const generateText = (name, age) => {
  return `${name} (${age} years old)`
}
```

Here's a fitting unit test:

```
test('should output name and age', () => {
  const text = generateText('Max', 29)
  expect(text).toBe('Max (29 years old)')
})
```

This test will check whether the `generateText` function does return the expected text.

If we now change the `generateText` function, let's say like this:

```
const generateText = (name, age) => {
  return `${age} (${age} years old)`
}
```

Then our test will fail. Because this function would return `'29 (29 years old)'` instead of `'Max (29 years old)'`.

I'll come back to where the `test` and `expect` functions are coming from!

For that reason, you should split your app into a lot of small modules which you can test individually. This will lead to cleaner code as a nice side-effect.

For the same reason, you'll also write a lot of unit tests in a project. It'll be your most common form of tests. If you test all the individual units of your app, chances are high that the app as a whole will also work.

Integration tests

Integration tests are a bit more complex than unit tests because you now have to deal with some dependencies (e.g. another function that gets called). These dependencies of course also have an impact on the result of your test, hence it's important to write "good" tests which allow you to understand what kind of effect leads to which result.

It's also important to unit-test the dependencies of your integration test as this will help you narrow down issues.

You could also think that integration tests are redundant if you got unit tests for everything.

But that's not the case. Here's an example (which you also see in the above video):

```
exports.checkAndGenerate = (name, age) => {
  if (!validateInput(name, true, false) || !validateInput(age, false, true)) {
    return false
  }
  return generateText(name, age)
}
```

Here's a fitting integration test:

```
test('should generate a valid text output', () => {
  const text = checkAndGenerate('Max', 29)
  expect(text).toBe('Max (29 years old)')
})
```

There's no special syntax as you can see. It's a normal test. We just call it integration test because it tests something which does have dependencies.

The `checkAndGenerate` function returns the result of `generateText` in the end but before it does so, it also validates the input. It does all that by calling other functions - hence we got a dependency here.

On first look, you could think that this will only fail when either `validateInput` or `generateText` have a problem - which would of course be issues that should be detected by a unit test. So why should we test the `checkAndGenerate` function?

Well, here's the answer. Consider this change to the `checkAndGenerate` function:

```
exports.checkAndGenerate = (name, age) => {
  if (validateInput(name, true, false) || !validateInput(age, false, true)) {
    return false
  }
  return generateText(name, age)
}
```

What changed? I removed the `!` in front of the first `validateInput` call.

That will now break the logic of this function since we now handle the result of `validateInput` incorrectly. So neither `validateInput` nor `generateText` are broken and still `checkAndGenerate` would yield an invalid result.

That's why you need integration tests!

End-to-End (e2e)/ UI tests

These kinds of tests simulate a certain user interaction flow with your app. This involves clicking, entering text etc.

And since it involves this, we need a browser. Actually all tests run in the browser but they'll not load up your app. They just need a browser JavaScript environment (i.e. essentially an empty browser window that's loaded up behind the scenes).

For end-to-end/ UI testing, we need a browser that loads our app though. And we need to be able to control that browser via code (so that we can program certain user interactions and simulate them).

There are tools for that and I'll dive into them in the next section.

First of all, let me show you what a test would look like:

```
test('should create an element with text and correct class', async () => {
  const browser = await puppeteer.launch({
    headless: true,
  })
  const page = await browser.newPage()
  await page.goto('localhost:3000/your-page')
  await page.click('input#name')
  await page.type('input#name', 'Anna')
  await page.click('input#age')
  await page.type('input#age', '28')
  await page.click('#btnAddUser')
  const finalText = await page.$eval('.user-item', el => el.textContent)
  expect(finalText).toBe('Anna (28 years old)')
}, 10000)
```

You can see that we simply tell the browser what to do. And at the end, we can get access to the DOM elements and validate their state, values, CSS classes - whatever we want to do.

Tools for Testing

To run all the test code you saw above, we need certain tools.

Because the test code is **not** part of your app code!

We only write it to execute it during development, it will not be shipped together with your app code. It will never run in the browser of your application users. That's really important to understand!

Instead, we need some tools that allow us to execute our tests locally, define our expectations (and check them) and control the browser for e2e testing.

In short, we need three kinds of tools:

- A test runner that executes your tests (`test()`) and summarizes the results
- An assertion library (`expect()`) that allows you to define your expectations (and check them)
- An e2e testing tool, often you use a (headless) browser for this

Test runners

The test runner executes your tests and summarizes the results in the terminal. Mocha is very popular but Jest really is the new star in the JavaScript testing scene.

The code above and the project shown in the video also use Jest. It's easy to use, well-documented and very powerful.

You can install Jest into your project (assuming you're using npm for managing it) by running:

```
npm install --save-dev jest
```

Assertion library

You also need to be able to define your `expect`ations and check them. Assertion libraries like Chai help you with that.

But here, we also can use Jest! And that's the cool thing about it. Besides being powerful and all that, it's also not just a test runner but test runner + assertion library combined. Another reason for its popularity.

e2e-testing tools

For e2e-testing, Selenium is pretty popular but Puppeteer is even more popular.

Puppeteer is a headless version of the Google Chrome browser. And it's even developed by the Google Chrome team. It's meant to be used as a headless version of Chrome (though you can even run it with an UI attached) and it's great for automated testing.

Puppeteer - which I use in this article + video - can be installed with this command:

```
npm install --save-dev puppeteer
```

Writing Tests

With the "WHAT", "WHY", tooling and kinds of tests explained - it's time to write tests, right?

You saw snippets above and you can see the full project in the video that you find at the top of the page.

Tests are really all about defining code that should be executed by the test runner and checked via the assertion library.

With Jest, you can define a new test with the `test` function. It's globally available when running Jest. Jest will automatically execute files that end with `.spec.js` or `.test.js` hence you should place your tests in there.

Example:

```
// inside util.test.js - which should test functions defined in util.js
const { generateText } = require('./util')

test('should output name and age', () => {
  const text = generateText('Max', 29)
  expect(text).toBe('Max (29 years old)')
})
```

Here, we first of all import the functions we want to test. Jest supports these Node-style imports, it does **not** support ES module imports by default.

Thereafter, a test is defined via the `test` function.

As you can see, two arguments are passed to `test`. A description of the test which will show up in the test summary once you execute your tests. And an anonymous function that contains your test logic.

Inside the test logic/ code, you find the `expect` function.

This function is provided by the assertion library - Jest in our case.

The "expectable" object which you get then has a bunch of built-in methods that allow you to compare the result/ value you're checking with a result/ value you would have expected.

`toBe(5)` would check if a value is equal to `5` for example. But you can also check other things like:

- `toBeCalled` => Was a function executed during the test?
- `not.toBe(5)` => Is the value **not** 5?
- And much more, all to be found here.

You can then execute your tests by running Jest, e.g. by adding a script to your `package.json` file:

```
"scripts": {
  "test": "jest"
},
```

After that, just run `npm test` to run all your tests and get the result in the same terminal.

It's also possible to run tests in watch mode so that they automatically re-run whenever you change something:

```
"scripts": {
  "test": "jest --watch"
},
```

The test runner will gather errors/ unexpected values and mark the test as failed as soon as at least one expectation in a test (you can have multiple ones) fails.

As you saw earlier in the article, there also is no syntactical difference between unit tests and integration tests.

For e2e-tests, the general approach of using `test` and `expect` also doesn't change. But we have some additional code in our tests since we use external tools (like Puppeteer) to control the browser:

```
const puppeteer = require('puppeteer')

test('should create an element with text and correct class', async () => {
  const browser = await puppeteer.launch({
    headless: true,
  })
  const page = await browser.newPage()
  await page.goto('localhost:4000/your-page')
  await page.click('input#name')
  await page.type('input#name', 'Anna')
```

```
  await page.click('input#age')
  await page.type('input#age', '28')
  await page.click('#btnAddUser')
  const finalText = await page.$eval('.user-item', el => el.textContent)
  expect(finalText).toBe('Anna (28 years old)')
}, 10000)
```

We launch a browser (that can be controlled via the test) with **`puppeteer.launch`**. This browser object can then be used to create new pages (**`newPage()`**), navigate to different URLs (**`goto()`**) and interact with the page (e.g. **`click()`**).

Since all that code interacts with a real browser, we can then also use some built-in methods (e.g. **`$eval`**) to evaluate DOM elements. In the example, we extract the **`textContent`** of a created element. At the end, **`expect`** is used again to check whether the created element has a valid text or not.

Summary & Next Steps

Now that you know what testing is about, how it works, which tools you need and how to write tests, it's time to dive in deeper.

There are more complex testing scenarios (e.g. async code, Http requests) that you'll be facing in your projects.

The official docs and future content here on Academind should be helpful for mastering testing though.

ANNEXE - URLS

Visual Studio Code User and Workspace Settings
Operator precedence - JavaScript | MDN
Javascript: difference between a statement and an expression? - Stack Overflow
javascript - switch statement to compare values greater or less than a number - Stack Overflow
Control flow and error handling - JavaScript | MDN
Loops and iteration - JavaScript | MDN
Strict mode - JavaScript | MDN
Getting garbage collection for free · V8
Memory Management - JavaScript | MDN
Function.prototype.bind() - JavaScript | MDN
Functions - JavaScript | MDN
Node.nodeType - Web APIs | MDN
Introduction to the DOM - Web APIs | MDN
Array - JavaScript | MDN
this - JavaScript | MDN
Classes - JavaScript | MDN
Object prototypes - Learn web development | MDN
Working with objects - JavaScript | MDN
Element.getBoundingClientRect() - Web APIs | MDN
Location - Web APIs | MDN
Window - Web APIs | MDN
Navigator - Web APIs | MDN
Functions - JavaScript | MDN
Closures - JavaScript | MDN
Using XMLHttpRequest - Web APIs | MDN
Using files from web applications - Web APIs | MDN
Fetch API - Web APIs | MDN
JavaScript modules - JavaScript | MDN
Document.cookie - Web APIs | MDN
Using IndexedDB - Web APIs | MDN
jakearchibald/idb: IndexedDB, but with promises
Overview | Maps JavaScript API | Google Developers
Angular vs React vs Vue - My Thoughts
Proxy - JavaScript | MDN
Comparing Reflect and Object methods - JavaScript | MDN
Security of JavaScript Applications | by JavaScript UI Libraries — DHTMLX | Medium
Optimize JavaScript Execution | Web Fundamentals | Google Developers
Performance features reference - Chrome Developers
Fix memory problems - Chrome Developers
Stubs, Spies and Mocks in JavaScript · Harry's Blog
Getting Started · Jest
Custom properties (--*): CSS variables - CSS: Cascading Style Sheets | MDN
Using CSS transitions - CSS: Cascading Style Sheets | MDN
adrianmejia - Data Structures in JavaScript: Arrays, HashMaps, and Lists | Adrian Mejia Blog
adrianmejia - How you can change the world by learning Data Structures and Algorithms | Adrian Mejia Blog
adrianmejia - 8 time complexities that every programmer should know | Adrian Mejia Blog

[trekhleb/javascript-algorithms: 📝 Algorithms and data structures implemented in JavaScript with explanations and links to further readings]

https://code.visualstudio.com/docs/getstarted/settings
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence
https://stackoverflow.com/questions/12703214/javascript-difference-between-a-statement-and-an-expression
https://stackoverflow.com/questions/32576618/switch-statement-to-compare-values-greater-or-less-than-a-number/32576647
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Loops_and_iteration
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode#changes_in_strict_mode
https://v8.dev/blog/free-garbage-collection
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_objects/Function/bind
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions
https://developer.mozilla.org/en-US/docs/Web/API/Node/nodeType
https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes
https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects#using_a_constructor_function
https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect
https://developer.mozilla.org/en-US/docs/Web/API/Location
https://developer.mozilla.org/en-US/docs/Web/API/Window
https://developer.mozilla.org/en-US/docs/Web/API/Navigator
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions#recursion
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures
https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest
https://developer.mozilla.org/en-US/docs/Web/API/File/Using_files_from_web_applications
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules
https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie
https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API/Using_IndexedDB
https://github.com/jakearchibald/idb
https://developers.google.com/maps/documentation/javascript/overview
https://academind.com/tutorials/angular-vs-react-vs-vue-my-thoughts/
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy#a_complete_traps_list_example
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect/Comparing_Reflect_and_Object_methods
https://dhtmlx.medium.com/security-of-javascript-applications-1c95cd2ce533
https://developers.google.com/web/fundamentals/performance/rendering/optimize-javascript-execution
https://developer.chrome.com/docs/devtools/evaluate-performance/reference/
https://developer.chrome.com/docs/devtools/memory-problems/
https://www.harrymt.com/blog/2018/04/11/stubs-spies-and-mocks-in-js.html
https://jestjs.io/docs/getting-started
https://developer.mozilla.org/en-US/docs/Web/CSS/--*
https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Transitions/Using_CSS_transitions
https://adrianmejia.com/data-structures-time-complexity-for-beginners-arrays-hashmaps-linked-lists-stacks-queues-tutorial/
https://adrianmejia.com/how-you-can-change-the-world-learning-data-structures-algorithms-free-online-course-tutorial/
https://adrianmejia.com/most-popular-algorithms-time-complexity-every-programmer-should-know-free-online-tutorial-course/
https://github.com/trekhleb/javascript-algorithms